

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

Programmer's Reference Manual (PRM)

Volume 1: Graphics Core

January 2008

Revision 1.0a

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.

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, or life sustaining applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

The Intel® 965 Express Chipset Family and Intel® G35 Express Chipset may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

I2C is a two-wire communications bus/protocol developed by Philips. SMBus is a subset of the I2C bus/protocol and was developed by Intel. Implementations of the I2C bus/protocol may require licenses from various entities, including Philips Electronics N.V. and North American Philips Corporation.

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

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

Copyright © 2008, Intel Corporation. All rights reserved.



Contents

1	Introduction	18
1.1	Notations and Conventions	20
1.1.1	Reserved Bits and Software Compatibility	20
1.2	Terminology	20
2	Graphics Device Overview	32
2.1	Graphics Memory Controller Hub (GMCH).....	32
2.2	Graphics Processing Unit (GPU)	33
3	Graphics Processing Engine (GPE)	34
3.1	Introduction	34
3.2	Overview	34
3.2.1	Block Diagram	34
3.2.2	Command Stream (CS) Unit	35
3.2.3	3D Pipeline.....	35
3.2.4	Media Pipeline.....	36
3.2.5	GEN4 Subsystem	36
3.2.5.1	Execution Units (EUs).....	36
3.2.6	GPE Function IDs	36
3.3	Pipeline Selection	38
3.4	URB Allocation	38
3.4.1	URB_FENCE.....	39
3.5	Constant URB Entries (CURBEs).....	44
3.5.1	Overview	44
3.5.2	Multiple CURBE Allocation	44
3.5.3	CS_URB_STATE	45
3.5.4	CONSTANT_BUFFER.....	46
3.6	Memory Access Indirection	47
3.6.1	STATE_BASE_ADDRESS	49
3.7	Instruction and State Prefetch	53
3.7.1	STATE_PREFETCH	54
3.8	System Thread Configuration	55
3.8.1	STATE_SIP	55
3.9	Command Ordering Rules	56
3.9.1	PIPELINE_SELECT	56
3.9.2	PIPE_CONTROL	56
3.9.3	URB-Related State-Setting Commands	57
3.9.4	Common Pipeline State-Setting Commands.....	57
3.9.5	3D Pipeline-Specific State-Setting Commands	57
3.9.6	Media Pipeline-Specific State-Setting Commands	58
3.9.7	URB_FENCE (URB Fencing & Entry Allocation).....	58
3.9.8	CONSTANT_BUFFER (CURBE Load)	59
3.9.9	3DPRIMITIVE.....	59
3.9.10	MEDIA_OBJECT.....	59
4	Graphics Command Formats.....	60



4.1	Command Formats	60
4.1.1	Memory Interface Commands.....	61
4.1.2	2D Commands	61
4.1.3	3D/Media Commands	61
4.1.4	Video Codec Commands	61
4.1.5	Command Header	61
4.2	Command Map.....	64
4.2.1	Memory Interface Command Map	64
4.2.2	2D Command Map.....	66
4.2.3	3D/Media Command Map.....	67
5	Register Address Maps.....	70
5.1	Graphics Register Address Map.....	70
5.1.1	Memory and I/O Space Registers.....	70
5.1.2	PCI Configuration Space	72
5.1.3	Graphics Register Memory Address Map.....	73
5.2	VGA and Extended VGA Register Map	95
5.2.1	VGA and Extended VGA I/O and Memory Register Map	95
5.3	Indirect VGA and Extended VGA Register Indices	96
6	Memory Data Formats	100
6.1	Memory Object Overview.....	100
6.1.1	Memory Object Types	100
6.2	Channel Formats	101
6.2.1	Unsigned Normalized (UNORM)	101
6.2.2	Gamma Conversion (SRGB)	102
6.2.3	Signed Normalized (SNORM).....	102
6.2.4	Unsigned Integer (UINT/USCALED)	102
6.2.5	Signed Integer (SINT/SSCALED).....	102
6.2.6	Floating Point (FLOAT)	103
6.2.6.1	32-bit Floating Point.....	103
6.2.6.2	64-bit Floating Point.....	103
6.3	Non-Video Surface Formats	103
6.3.1	Surface Format Naming.....	103
6.3.2	Intensity Formats.....	104
6.3.3	Luminance Formats	104
6.3.4	P4A4_UNORM	105
6.3.5	A4P4_UNORM	105
6.4	Compressed Surface Formats.....	106
6.4.1	FXT Texture Formats	106
6.4.1.1	Overview of FXT1 Formats	106
6.4.1.2	FXT1 CC_HI Format	107
6.4.1.3	FXT1 CC_CHROMA Format	109
6.4.1.4	FXT1 CC_MIXED Format	111
6.4.1.5	FXT1 CC_ALPHA Format.....	116
6.4.2	BC Texture Formats.....	119
6.4.2.1	Opaque and One-bit Alpha Textures (BC1)	119
6.4.2.2	Opaque Textures (BC1_RGB).....	122
6.4.2.3	Compressed Textures with Alpha Channels (BC2-3)	122
6.5	Video Pixel/Texel Formats.....	124
6.5.1	Packed Memory Organization	124
6.5.2	Planar Memory Organization	125
6.6	Surface Memory Organizations	127



6.7	Graphics Translation Tables	127
6.8	Hardware Status Page.....	128
6.9	Instruction Ring Buffers.....	128
6.10	Instruction Batch Buffers	128
6.11	Display, Overlay, Cursor Surfaces	128
6.12	2D Render Surfaces	128
6.13	2D Monochrome Source	129
6.14	2D Color Pattern	129
6.15	3D Color Buffer (Destination) Surfaces	129
6.16	3D Depth Buffer Surfaces	130
6.17	Surface Layout.....	130
6.17.1	Buffers	130
6.17.2	1D Surfaces.....	131
6.17.3	2D Surfaces.....	131
6.17.3.1	Computing MIP level sizes.....	132
6.17.3.2	Base Address for LOD Calculation	132
6.17.3.3	Minimum Pitch	133
6.17.3.4	Alignment Unit Size.....	134
6.17.3.5	Cartesian to Linear Address Conversion.....	134
6.17.3.6	Compressed Mipmap Layout	134
6.17.3.7	Surface Arrays	135
6.17.4	Cube Surfaces	135
6.17.4.1	Hardware Cube Map Layout.....	135
6.17.4.2	Restrictions.....	136
6.17.5	3D Surfaces.....	136
6.17.5.1	Minimum Pitch	138
6.18	Surface Padding Requirements	139
6.18.1	Sampling Engine Surfaces	139
6.18.2	Render Target and Media Surfaces	139
6.19	Logical Context Data.....	140
6.19.1	Overall Context Layout.....	140
6.19.1.1	Per-Process GTT and Run Lists Disabled	140
6.19.2	Register/State Context.....	140
6.19.3	The Probe List.....	155
6.19.4	Pipelined State Page	155
6.19.5	Ring Buffer.....	155
6.19.6	The Per-Process Hardware Status Page.....	156
7	Device 2 Configuration Registers	158
7.1	Introduction	158
7.2	Device 2, Function 0	158
7.2.1	VID2 — Vendor Identification	160
7.2.2	DID2 — Device Identification	161
7.2.3	PCICMD2 — PCI Command	162
7.2.4	PCISTS2 — PCI Status	163
7.2.5	RID2 — Revision Identification	164
7.2.6	CC — Class Code.....	165
7.2.7	CLS — Cache Line Size.....	165
7.2.8	MLT2 — Master Latency Timer.....	166
7.2.9	HDR2 — Header Type	166
7.2.10	BIST — Built In Self Test	166
7.2.11	GTTMMADR — Graphics Translation Table Range Address.....	167



7.2.12	GMADR — Graphics Memory Range Address	168
7.2.13	IOBAR — I/O Base Address	169
7.2.14	SVID2 — Subsystem Vendor Identification	169
7.2.15	SID2 — Subsystem Identification	170
7.2.16	ROMADR — Video BIOS ROM Base Address	170
7.2.17	CAPPOINT — Capabilities Pointer	170
7.2.18	INTRLINE — Interrupt Line	171
7.2.19	INTRPIN — Interrupt Pin	171
7.2.20	MINGNT — Minimum Grant	171
7.2.21	MAXLAT — Maximum Latency	172
7.2.22	MCAPPTR — Capabilities Pointer (to Mirror of Dev0 CAPID)	172
7.2.23	MCAPID — Mirror of Dev 0 Capability Identification.	172
7.2.24	MGGC — Mirror of Dev0 GMCH Graphics Control	173
7.2.25	MDEVENdev0F0 — Mirror of Dev0 DEVEN	174
7.2.26	SSRW — Software Scratch Read Write	174
7.2.27	BSM — Base of Stolen Memory	174
7.2.28	HSRW — Hardware Scratch Read Write	175
7.2.29	MSAC — Multi Size Aperture Control	175
7.2.30	SCWBFC — Secondary CWB Flush Control ([DevBW] Only)	176
7.2.31	CAPL — Capabilities List Control	176
7.2.32	MSI_CAPID — Message Signaled Interrupts Capability ID	177
7.2.33	MC — Message Control	178
7.2.34	MA — Message Address	179
7.2.35	MD — Message Data	179
7.2.36	GDRST — Graphics Device Reset	180
7.2.37	GMBUSFREQ — GMBUS frequency binary encoding	181
7.2.38	PMCAPID — Power Management Capabilities ID	181
7.2.39	PMCAP — Power Management Capabilities	182
7.2.40	PMCS — Power Management Control/Status	183
7.2.41	SWSMI — Software SMI	184
7.2.42	ASLE — System Display Event Register	184
7.2.43	SWSCI — Software SCI	185
7.2.44	LBB — Legacy Backlight Brightness ([DevCL] Only)	186
7.2.45	MID2 — Manufacturing ID	187
7.2.46	ASLS — ASL Storage	187
7.3	Device 2, Function 1	188
7.3.1	VID2 — Vendor Identification	189
7.3.2	DID2 — Device Identification	189
7.3.3	PCICMD2 — PCI Command	190
7.3.4	PCISTS2 — PCI Status	191
7.3.5	RID2 — Revision Identification	192
7.3.6	CC — Class Code	193
7.3.7	CLS — Cache Line Size	193
7.3.8	MLT2 — Master Latency Timer	194
7.3.9	HDR2 — Header Type	194
7.3.10	BIST — Built In Self Test	194
7.3.11	MMADR — Memory Mapped Range Address	195
7.3.12	SVID2 — Subsystem Vendor Identification	195
7.3.13	SID2 — Subsystem Identification	196
7.3.14	ROMADR — Video BIOS ROM Base Address	196
7.3.15	CAPPOINT — Capabilities Pointer	196
7.3.16	MINGNT — Minimum Grant	197
7.3.17	MAXLAT — Maximum Latency	197
7.3.18	MCAPPTR — Capabilities Pointer (to Mirror of Dev0 CAPID)	197
7.3.19	MCAPID — Mirror of Dev 0 Capability Identification.	198



7.3.20	MGGC — Mirror of Dev0 GMCH Graphics Control	199
7.3.21	MDEVENdev0F0 — Mirror of Dev0 DEVEN.....	199
7.3.22	SSRW — Software Scratch Read Write.....	200
7.3.23	BSM — Base of Stolen Memory.....	200
7.3.24	HSRW — Hardware Scratch Read Write	200
7.3.25	MSAC — Multi Size Aperture Control	201
8	Memory Interface Registers.....	202
8.1	Introduction	202
8.2	Virtual Memory Control	202
8.2.1	Global Virtual Memory.....	202
8.2.1.1	PGTBL_CTL—Page Table Control Register	203
8.2.1.2	PGTBL_ER—Page Table Error Register (<i>Debug</i>).....	205
8.2.1.3	Graphics Translation Table (GTT) Range (GTTADR)	207
8.2.1.4	GTT Page Table Entries (PTEs).....	208
8.2.2	Single-Level (Flat) Per-Process Virtual Memory	209
8.2.2.1	PGTBL_CTL2— Per Process Page Table Control Register	209
8.2.2.2	PGTBL_STR2—Page Table Steer Register (Per Process).	211
8.2.3	TLB Read Interface	213
8.2.3.1	TLB_RD_EXT — TLB Read Extent.....	213
8.2.3.2	Instruction/State Cache (ISC).....	214
8.2.3.3	Vertex Fetch (VF)	215
8.2.3.4	Command Streamer (CS).....	216
8.2.3.5	Texture Cache (MT)	217
8.2.3.6	Render Cache (RC)	218
8.3	GFX_MODE – Graphics Mode Register.....	219
8.4	EXCC—Execute Condition Code Register	220
8.5	RINGBUF—Ring Buffer Registers.....	222
8.5.1	UHPTR — Pending Head Pointer Register.....	226
8.6	Debug Registers Control	227
8.6.1	HW_MEMRD—Memory Read Sync Register (Debug)	227
8.6.2	IPEIR—Instruction Parser Error Identification Register (Debug)	228
8.6.3	IPEHR—Instruction Parser Error Header Register (Debug)	229
8.6.4	INSTDONE—Instruction Stream Interface Done Register (Debug)	229
8.6.5	INSTPS—Instruction Parser State Register (Debug)	231
8.6.6	ACTHD — Active Head Pointer Register (Debug)	231
8.6.7	DMA_FADD_P — Primary DMA Engine Fetch Address (Debug)	232
8.6.8	INSTDONE_1 — Additional Instruction Stream Interface Done (Debug)	232
8.6.9	GFX_FLSH_CNTL — Graphics Flush Control	234
8.7	NOPID — NOP Identification Register	235
8.8	Interrupt Control Registers	236
8.8.1	HWS_PGA — Hardware Status Page Address Register	239
8.8.2	PWRCTXA — Power Context Register Address ([DevCL] Only)	240
8.8.3	HWSTAM — Hardware Status Mask Register	241
8.8.4	IER — Interrupt Enable Register	244
8.8.5	IIR — Interrupt Identity Register	245
8.8.6	IMR—Interrupt Mask Register.....	246
8.8.7	ISR — Interrupt Status Register	247
8.9	Hardware-Detected Error Bit Definitions (for EIR, EMR, ESR)	248
8.9.1	EIR — Error Identity Register	249
8.9.2	EMR—Error Mask Register.....	250
8.9.3	ESR—Error Status Register	251



8.10	Register Definitions for Context Save	252
8.10.1	INSTPM—Instruction Parser Mode Register	252
8.10.2	Cache_Mode_0— Cache Mode Register 0	254
8.10.3	Cache_Mode_1— Cache Mode Register 1	257
8.10.4	BB_ADDR—Batch Buffer Head Pointer Register	261
8.10.5	BB_STATE – Batch Buffer State Register.....	262
8.10.6	CTXT_SR_CTL – Context Save/Restore Control Register.....	263
8.11	Logical Context Support	264
8.11.1	CCID—Current Context ID Register	264
8.11.2	CXT_SIZE—Context Size with Extended State.....	266
8.11.3	CXT_SIZE_NOEXT—Context Size without the Extended State.....	266
8.12	Arbitration Control, and Scratch Bits	267
8.12.1	MI_DISPLAY_POWER_DOWN—Display Power Down ([DevCL] Only).....	267
8.12.2	MI_ARB_STATE—Memory Interface Arbitration State Register.....	268
8.12.3	MI_RDRET_STATE—Memory Interface Read Return State Register.....	271
8.12.4	MI_MODE – Mode Register for Software Interface	274
8.12.5	ECOSKPD—ECO Scratch Pad (DEBUG).....	278
8.13	Debug Registers.....	281
8.13.1	CSFLFSM – Flush FSM (Debug).....	281
8.13.2	CSFLFLAG – Flush FLAG (Debug)	283
8.13.3	CSFLTRK – Flush Track (Debug)	284
8.13.4	CSCMDOP – Instruction DWORD (Debug).....	284
8.13.5	CSCMDVLD – Instruction DWORD Valid (Debug)	285
8.13.6	CLKCMP – Compare count clock stop (Debug).....	285
8.13.7	VFDC—Set Value of Draw Count (DEBUG).....	286
8.13.8	VFSKPD—VF Scratch Pad (DEBUG).....	286
8.14	Software Visible Counter Registers.....	288
8.14.1	PS_DEPTH_COUNT – Reported Pixels Passing Depth Test Counter.....	288
8.14.2	TIMESTAMP – Reported Timestamp Count.....	289
8.15	MTCH_CID_RST – Matched Context ID Reset Register	290
8.16	Interrupt Control Registers	291
8.16.1.1	BCS_IPEIR—Instruction Parser Error Identification Register (Debug)	292
8.16.1.2	BCS_IPEHR—Instruction Parser Error Header Register (Debug)	292
8.16.1.3	BCS_ACTHD – Active Head Pointer Register (Debug).....	292
8.16.1.4	BCS_DMA_FADD –DMA Engine Fetch Address (Debug).....	293
8.16.1.5	BCS_HWS_PGA – Hardware Status Page Address Register.....	293
8.16.1.6	BCS_NOPID – NOP Identification Register	294
8.16.1.7	BCS_MI_MODE – Mode Register for Software Interface.....	294
8.16.1.8	BCS_INSTPM—Instruction Parser Mode Register	295
8.16.1.9	BCS_UHPTR – Pending Head Pointer Register.....	296
8.16.1.10	BCS_CNTR—Counter for the Bit Stream Decode Engine	296
8.16.1.11	BCS_THRSH—Threshold for the Counter of Bit Stream Decode Engine	296
8.16.1.12	BCS_BB_ADDR—Batch Buffer Head Pointer Register	297
8.16.1.13	BCS_RCCID—Ring Buffer Current Context ID Register ..	297
8.16.1.14	BCS_RNCID—Ring Buffer Next Context ID Register.....	298
8.17	Software Control Bit Definitions.....	298
8.18	Frame Buffer Compression Control ([DevCL] Only).....	299
8.18.1	FBC_CFB_BASE – Compressed Frame Buffer Base Address	299
8.18.2	FBC_LL_BASE – Compressed Frame Line Length Buffer Address ..	300
8.18.3	FBC_CONTROL – Frame Buffer Compression Control Register	301
8.18.4	FBC_COMMAND – Frame Buffer Compression Command Register	302



	8.18.5	FBC_STATUS — Frame Buffer Compression Status Register	303
	8.18.6	FBC_CONTROL2— Frame Buffer Compression 2 nd Control Register	305
	8.18.7	FBC_DISP_YOFF — FBC Fence Display Buffer Y Offset	306
	8.18.8	FBC_MOD_NUM— FBC Number of Modifications for Recompression	307
	8.18.9	FBC_TAG — Frame Buffer Compression TAG Interface (DEBUG) ...	308
	8.19	Fence Registers.....	310
	8.19.1	FENCE — Graphics Memory Fence Table Registers.....	310
	8.20	GFX MMIO – MCHBAR Aperture	312
9		Memory Interface Commands for Rendering Engine	314
	9.1	Introduction	314
	9.2	MI_ARB_CHECK	314
	9.3	MI_BATCH_BUFFER_END.....	315
	9.4	MI_BATCH_BUFFER_START.....	315
	9.5	MI_DISPLAY_FLIP.....	318
	9.6	MI_FLUSH	323
	9.7	MI_LOAD_REGISTER_IMM	324
	9.8	MI_LOAD_SCAN_LINES_EXCL	325
	9.9	MI_LOAD_SCAN_LINES_INCL.....	327
	9.10	MI_NOOP	328
	9.11	MI_OVERLAY_FLIP.....	329
	9.11.1	Turning the Overlay Off.....	331
	9.11.2	Valid Overlay Flip Sequences	331
	9.12	MI_REPORT_HEAD.....	332
	9.13	MI_SET_CONTEXT	332
	9.14	MI_STORE_DATA_IMM	335
	9.15	MI_STORE_DATA_INDEX.....	337
	9.16	MI_STORE_REGISTER_MEM.....	339
	9.17	MI_USER_INTERRUPT	341
	9.18	MI_WAIT_FOR_EVENT.....	341
10		Memory Interface Commands for Blitter Engine.....	346
	10.1	Introduction	346
	10.2	MI_LOAD_REGISTER_IMM	347
	10.3	MI_NOOP	348
	10.4	MI_STORE_DATA_IMM	349
	10.5	MI_STORE_DATA_INDEX.....	350
	10.6	MI_USER_INTERRUPT	352
	10.7	MI_WAIT_FOR_EVENT.....	352
11		Graphics Memory Interface Functions.....	354
	11.1	Introduction	354
	11.2	Graphics Memory Clients	354
	11.3	Graphics Memory Addressing Overview.....	355
	11.3.1	Graphics Address Path	355
	11.4	Graphics Memory Address Spaces.....	357
	11.5	Address Tiling Function	357
	11.5.1	Linear vs. Tiled Storage.....	357
	11.5.2	Tile Formats	360



11.5.3	Tiling Algorithm	362
11.5.4	Tiling Support	363
11.5.4.1	Tiled (Fenced) Regions	363
11.5.4.2	Tiled Surface Parameters	364
11.5.4.3	Tiled Surface Restrictions	364
11.5.5	Per-Stream Tile Format Support	367
11.6	Logical Memory Mapping	367
11.6.1	Logical Memory Space Mappings	368
11.7	Physical Graphics Memory	372
11.7.1	Physical Graphics Address Types	372
11.7.2	Main Memory	373
11.7.2.1	Optimizing Main Memory Allocation	373
11.7.2.2	Application of the Theory (Page Coloring)	373
12	Device Programming Environment	376
12.1	Programming Model	376
12.2	Graphics Device Register Programming	376
12.3	Graphics Device Command Streams	377
12.3.1	Command Use	377
12.3.2	Command Transport Overview	377
12.3.3	Command Parser	378
12.3.4	The Ring Buffer	378
12.3.4.1	The Ring Buffer (RB)	379
12.3.4.2	Ring Buffer Registers	379
12.3.4.3	Ring Buffer Placement	381
12.3.4.4	Ring Buffer Initialization	381
12.3.4.5	Ring Buffer Use	381
12.3.4.6	Ring Buffer Semaphore	382
12.3.5	Batch Buffers	382
12.3.5.1	Batch Buffer Chaining	382
12.3.5.2	Ending Batch Buffers	383
12.3.6	Indirect Data	383
12.3.6.1	Logical Contexts	383
12.3.7	Command Arbitration	383
12.3.7.1	Arbitration Policies and Rationale	383
12.3.7.2	Wait Commands	384
12.3.7.3	Wait Events/Conditions	384
12.3.7.4	Command Arbitration Points	385
12.3.7.5	Command Arbitration Rules	385
12.3.7.6	Batch Buffer Protection	385
12.3.8	Graphics Engine Synchronization	386
12.3.9	Graphics Memory Coherency	387
12.3.10	Graphics Cache Coherency	387
12.3.10.1	Rendering Cache	387
12.3.10.2	Sampler Cache	388
12.3.10.3	Instruction/State Cache	388
12.3.10.4	Vertex Cache	389
12.3.10.5	GTT TLBs	389
12.3.11	Command Synchronization	389
12.3.11.1	MI_FLUSH	390
12.3.11.2	Sync Flush	390
12.4	Hardware Status	391
12.4.1	Hardware-Detected Errors (Master Error bit)	392
12.4.2	Thermal Sensor Event	392



	12.4.3	Sync Status.....	392
	12.4.4	Display Plane A, B, Flip Pending.....	392
	12.4.5	Overlay Flip Pending	392
	12.4.6	Display Pipe A,B VBLANK.....	392
	12.4.7	User Interrupt.....	393
	12.4.8	PIPE_CONTROL Notify Interrupt.....	393
	12.4.9	Display Port Interrupt	393
	12.5	Hardware Status Writes.....	393
	12.6	Interrupts	393
	12.7	Errors.....	394
	12.7.1	Error Reporting	394
	12.7.2	Page Table Errors	395
	12.7.3	Clearing Errors.....	395
	12.8	Rendering Context Management	396
	12.8.1	Multiple Logical Rendering Contexts	396
	12.8.1.1	Current Context IDs	397
	12.8.1.2	Intra-Ring Context Switch	397
	12.8.1.3	Logical Rendering Context Creation and Initialization ...	398
	12.8.1.4	Context Save	398
	12.9	Reset State	399
13		Frame Buffer Compression ([DevCL] Only)	400
	13.1	Overview	400
	13.2	Programming Interface	401
	13.2.1	FBC unit programming interface	401
	13.2.2	Programming interface from Display Engine	402
	13.3	Operating Modes	403
	13.3.1	RLE-FBC Function Modes	403
	13.3.2	Compression Modes.....	404
	13.3.2.1	Single Compression Mode	404
	13.3.2.2	Periodic Compression Mode	404
	13.4	Usage Restrictions.....	405
	13.5	Power Management Interface	406
	13.6	Memory Data Structures.....	407
	13.6.1	RLE Pixel Runs	407
	13.6.2	RLE Pixel Run Sets	407
	13.6.3	RLE-Compressed Line	407
	13.6.4	RLE Compressed Frame and Line Length Buffers	408
	13.7	Tuning Parameters	409
	13.7.1	Stride	409
	13.7.2	Interval	409
	13.7.3	FBC Modification Counter.....	409
	13.8	Implementation (DEBUG)	410
	13.8.1	Tag Array.....	410
	13.8.1.1	Transitions.....	410
	13.8.2	Compressor.....	411
	13.8.3	Decompressor.....	412
	13.8.4	Frame Buffer Write Detector	412
	13.8.5	Coherency.....	413
14		BLT Engine.....	414
	14.1	Introduction	414
	14.2	Classical BLT Engine Functional Description	414



14.2.1	Basic BLT Functional Considerations	415
14.2.1.1	Color Depth Configuration and Color Expansion	415
14.2.1.2	Graphics Data Size Limitations	416
14.2.1.3	Bit-Wise Operations	416
14.2.1.4	Per-Pixel Write-Masking Operations	421
14.2.1.5	When the Source and Destination Locations Overlap	422
14.2.2	Basic Graphics Data Considerations	426
14.2.2.1	Contiguous vs. Discontinuous Graphics Data	426
14.2.2.2	Source Data	427
14.2.2.3	Monochrome Source Data	428
14.2.2.4	Pattern Data	429
14.2.2.5	Destination Data	431
14.2.3	BLT Programming Examples	432
14.2.3.1	Pattern Fill — A Very Simple BLT	432
14.2.3.2	Drawing Characters Using a Font Stored in System Memory	435
14.3	BLT Instruction Overview	438
14.4	BLT Engine State	438
14.5	Cacheable Memory Support	439
14.6	Device Cache Coherency: Render and Texture Caches	439
14.7	BLT Engine Instructions	440
14.7.1	Blt Programming Restrictions	440
14.8	Fill/Move Instructions	440
14.8.1	COLOR_BLT (Fill)	441
14.8.2	SRC_COPY_BLT (Move)	442
14.9	2D (X,Y) BLT Instructions	443
14.9.1	XY_SETUP_BLT	445
14.9.2	XY_SETUP_MONO_PATTERN_SL_BLT	446
14.9.3	XY_SETUP_CLIP_BLT	447
14.9.4	XY_PIXEL_BLT	447
14.9.5	XY_SCANLINES_BLT	448
14.9.6	XY_TEXT_BLT	449
14.9.7	XY_TEXT_IMMEDIATE_BLT	450
14.9.8	XY_COLOR_BLT	451
14.9.9	XY_PAT_BLT	452
14.9.10	XY_PAT_CHROMA_BLT	453
14.9.11	XY_PAT_BLT_IMMEDIATE	454
14.9.12	XY_PAT_CHROMA_BLT_IMMEDIATE	455
14.9.13	XY_MONO_PAT_BLT	456
14.9.14	XY_MONO_PAT_FIXED_BLT	458
14.9.14.1	Monochrome Pattern Memory Format	460
14.9.14.2	HS_HORIZONTAL 0	461
14.9.14.3	HS_VERTICAL 1	461
14.9.14.4	HS_FDIAGONAL 2	461
14.9.14.5	HS_BDIAGONAL 3	461
14.9.14.6	HS_CROSS 4	462
14.9.14.7	HS_DIAGCROSS 5	462
14.9.14.8	Screen Door 8	462
14.9.14.9	SD Wide 9	462
14.9.14.10	Walking Bit (One) A	463
14.9.14.11	Walking Zero B	463
14.9.15	XY_SRC_COPY_BLT	463
14.9.16	XY_SRC_COPY_CHROMA_BLT	465
14.9.17	XY_MONO_SRC_COPY_BLT	466



14.9.18	XY_MONO_SRC_COPY_IMMEDIATE_BLT	468
14.9.19	XY_FULL_BLT.....	470
14.9.20	XY_FULL_IMMEDIATE_PATTERN_BLT.....	472
14.9.21	XY_FULL_MONO_SRC_BLT.....	474
14.9.22	XY_FULL_MONO_SRC_IMMEDIATE_PATTERN_BLT	476
14.9.23	XY_FULL_MONO_PATTERN_BLT	478
14.9.24	XY_FULL_MONO_PATTERN_MONO_SRC_BLT	480
14.10	BLT Engine Instruction Field Definitions	482
14.10.1	BR00—BLT Opcode & Control	482
14.10.2	BR01—Setup BLT Raster OP, Control, and Destination Offset	485
14.10.3	BR05—Setup Expansion Background Color	487
14.10.4	BR06—Setup Expansion Foreground Color.....	488
14.10.5	BR07—Setup Color Pattern Address	489
14.10.6	BR09—Destination Address.....	490
14.10.7	BR11—BLT Source Pitch (Offset).....	491
14.10.8	BR12—Source Address	492
14.10.9	BR13—BLT Raster OP, Control, and Destination Pitch.....	492
14.10.10	BR14—Destination Width & Height	494
14.10.11	BR15—Color Pattern Address	495
14.10.12	BR16—Pattern Expansion Background & Solid Pattern Color.....	496
14.10.13	BR17—Pattern Expansion Foreground Color.....	496
14.10.14	BR18—Source Expansion Background, and Destination Color	497
14.10.15	BR19—Source Expansion Foreground Color	497



Figures

Figure 2-1. GMCH Block Diagram	32
Figure 2-2. Block Diagram of the GPU	33
Figure 3-1. The Graphics Processing Engine	34
Figure 3-2. GPE Diagram Showing Fixed/Shared Functions.....	35
Figure 3-3. URB Allocation – 3D Pipeline.....	39
Figure 3-4. URB Allocation – Media Pipeline	40
Figure 6-1. FXT1 Encoded Blocks.....	106
Figure 6-2. Memory Layout of Packed YUV 4:2:2 Formats	125
Figure 6-3. YUV 4:2:0 Format Memory Organization	126
Figure 6-4. YUV 4:1:0 Format Memory Organization	127
Figure 6-5. Volume Texture Map	136
Figure 11-1. Graphics Memory Paths.....	356
Figure 11-2. Rectangular Memory Operand Parameters	358
Figure 11-3. Linear Surface Layout	358
Figure 11-4. Memory Tile Dimensions	359
Figure 11-5. Tiled Surface Layout.....	360
Figure 11-6. Y-Major Tile Layout	361
Figure 11-7. Tiled Surface Placement	365
Figure 11-8. Global and Render GTT Mapping	369
Figure 11-9. GTT Re-mapping to Handle Differing Pitches	371
Figure 11-10. Logical-to-Physical Graphics Memory Mapping	371
Figure 11-11. Memory Interfaces	372
Figure 11-12. Memory Pages backing Color and Depth Buffers.....	374
Figure 12-1. Graphics Controller Command Interface	378
Figure 12-2. Ring Buffer.....	379
Figure 12-3. Batch Buffer Chaining	382
Figure 13-1. 32bpp Pixel Run.....	407
Figure 13-2. 16bpp Pixel Run.....	407
Figure 13-3. Pixel Run Set.....	407
Figure 13-4. RLE-Compression Buffers	408
Figure 14-1. Block Diagram and Data Paths of the BLT Engine	415
Figure 14-2. Block Diagram and Data Paths of the BLT Engine	421
Figure 14-3. Source Corruption in BLT with Overlapping Source and Destination Locations.....	423
Figure 14-4. Correctly Performed BLT with Overlapping Source and Destination Locations.....	424
Figure 14-5. Suggested Starting Points for Possible Source and Destination Overlap Situations	425
Figure 14-6. Representation of On-Screen Single 6-Pixel Line in the Frame Buffer ..	426
Figure 14-7. Representation of On-Screen 6x4 Array of Pixels in the Frame Buffer..	427
Figure 14-8. Pattern Data -- Always an 8x8 Array of Pixels.....	429
Figure 14-9. 8bpp Pattern Data -- Occupies 64 Bytes (8 quadwords)	430
Figure 14-10. 16bpp Pattern Data -- Occupies 128 Bytes (16 quadwords)	430
Figure 14-11. 32bpp Pattern Data -- Occupies 256 Bytes (32 quadwords)	430
Figure 14-12. On-Screen Destination for Example Pattern Fill BLT	432
Figure 14-13. Pattern Data for Example Pattern Fill BLT.....	433
Figure 14-14. Results of Example Pattern Fill BLT	434
Figure 14-15. On-Screen Destination for Example Character Drawing BLT	435



Figure 14-16. Source Data in System Memory for Example Character Drawing BLT	435
Figure 14-17. Results of Example Character Drawing BLT	437

Tables

Table 1-1. Supported Chipsets	18
Table 3-1. Gen4 Function IDs	37
Table 3-2. Base Address Utilization	48
Table 4-1. RCP Command Header Format	62
Table 4-2. VCCP Command Header Format	63
Table 4-3. Memory Interface Commands for RCP	64
Table 4-4. Memory Interface Commands for VCCP	65
Table 5-1. Graphics Controller Register Memory and I/O Map	71
Table 5-2. Memory-Mapped Registers	73
Table 5-3. I/O and Memory Register Map	95
Table 5-4. 2D Sequence Registers (3C4h / 3C5h)	96
Table 5-5. 2D Graphics Controller Registers (3CEh / 3CFh)	97
Table 5-6. 2D Attribute Controller Registers (3C0h / 3C1h)	97
Table 5-7. 2D CRT Controller Registers (3B4h / 3D4h / 3B5h / 3D5h)	98
Table 6-1. FXT1 Format Summary	106
Table 6-2. FXT CC_HI Block Encoding	107
Table 6-3. FXT CC_HI Decoded Colors	108
Table 6-4. FXT CC_HI Interpolated Color Table	108
Table 6-5. FXT CC_CHROMA Block Encoding	109
Table 6-6. FXT CC_CHROMA Decoded Colors	110
Table 6-7. FXT CC_CHROMA Interpolated Color Table	111
Table 6-8. FXT CC_MIXED Block Encoding	111
Table 6-9. FXT CC_MIXED (Alpha[0]=0) Decoded Colors	112
Table 6-10. FXT CC_MIXED Decoded Colors (Alpha[0] = 0)	112
Table 6-11. FXT CC_MIXED Interpolated Color Table (Alpha[0]=0, Texels 0-15)	113
Table 6-12. FXT CC_MIXED Interpolated Color Table (Alpha[0]=0, Texels 16-31)	114
Table 6-13. FXT CC_MIXED (Alpha[0]=0) Decoded Colors	114
Table 6-14. FXT CC_MIXED Decoded Colors (Alpha[0] = 1)	114
Table 6-15. FXT CC_MIXED Interpolated Color Table (Alpha[0]=1, Texels 0-15)	115
Table 6-16. FXT CC_MIXED Interpolated Color Table (Alpha[0]=1, Texels 16-31)	115
Table 6-17. FXT CC_ALPHA Block Encoding	116
Table 6-18. FXT CC_ALPHA Decoded Colors	117
Table 6-19. FXT CC_ALPHA Interpolated Color Table (LERP=0)	118
Table 6-20. FXT CC_ALPHA Interpolated Color Table (LERP=1, Texels 0-15)	118
Table 6-21. FXT CC_ALPHA Interpolated Color Table (LERP=1, Texels 16-31)	118
Table 6-22. Depth Buffer Formats	130
Table 6-23. Alignment Units for Texture Maps	134
Table 6-24. Context Setup that Cannot Use Defaults	150
Table 6-25. Initialization of Command State	151
Table 9-1. Bit Definition for Interrupt Control Registers	236
Table 9-2. Hardware-Detected Error Bits	248
Table 8-3. Bit Definition for Interrupt Control Registers	291
Table 11-1. Graphics Memory Clients	354
Table 11-2. Graphics Memory Address Types	357
Table 11-3. X-Major Tile Layout	361



Table 11-4. Physical Memory Address Types	372
Table 12-1. Ring Buffer Characteristics.....	380
Table 12-2. Graphics Memory Coherency.....	387
Table 12-3. Page Table Error Types	395
Table 14-1. Bit-Wise Operations and 8-Bit Codes (00-3F)	417
Table 14-2. Bit-Wise Operations and 8-bit Codes (40 - 7F).....	418
Table 14-3. Bit-Wise Operations and 8-bit Codes (80 - BF).....	419
Table 14-4. Bit-Wise Operations and 8-bit Codes (C0 - FF).....	420



Revision History

Document Number	Revision Number	Description	Revision Date
1	1.0a	Initial release.	January 2008



1 Introduction

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

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

Table 1-1. Supported Chipsets

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

NOTES:

1. Unless otherwise specified, the information in this document applies to all of the devices mentioned in Table 1-1. For Information that does not apply to all devices, the Device Tag is used.
2. Throughout the PRM, references to "All" in a project field refers 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 Chipset 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, WMV9, 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 <i>3D Pipeline</i> .
Adjacency	—	One can consider a single line object as existing in a strip of connected lines. The neighboring line objects are called “adjacent objects”, with the non-shared endpoints called the “adjacent vertices.” The same concept can be applied to a single triangle object, considering it as existing in a mesh of connected triangles. Each triangle shares edges with three other adjacent triangles, each defined by a non-shared adjacent vertex. Knowledge of these adjacent objects/vertices is required by some object processing algorithms (e.g., silhouette edge detection). See <i>3D Pipeline</i> .
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.



Term	Abbr.	Definition
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.
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 “perspective 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.



Term	Abbr.	Definition
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.
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 <i>Strips and Fans</i> .
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	—	The width of each of several data elements that may be processed by a single Gen4 SIMD instruction.
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.



Term	Abbr.	Definition
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.
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 <i>Graphics Processing Engine</i> .
Geometry Shader	GS	Fixed-function unit between the vertex shader and the clipper that (if enabled) dispatches "geometry shader" threads on its input primitives. See <i>Geometry Shader</i> .
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. It is required to support WMV9 Profile B video decoder acceleration.
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 both MPEG-2 and WMV.
Jitter	—	Just-in-time compiler.
Kernel	—	A sequence of Gen4 instructions that is logically part of the driver or generated by the jitter. Differentiated from a Shader which is an application supplied program that is translated by the jitter to Gen4 instructions.
Least Significant Bit	LSB	Least Significant Bit
MathBox	—	See Extended Math Unit
Media	—	Term for operations such as video decode and encode that are normally performed by the Media pipeline.



Term	Abbr.	Definition
Media Pipeline	—	Fixed function stages dedicated to media and “generic” processing, sometimes referred to as the generic pipeline.
Message	—	Messages are data packages transmitted from a thread to another thread, another shared function or another fixed function. Message passing is the primary communication mechanism of GEN4 architecture.
Message Gateway	—	Shared function that enables thread-to-thread message communication/synchronization used solely by the Media pipeline.
Message Register File	MRF	Write-only registers used by EUs to assemble messages prior to sending and as the operand of a send instruction.
Most Significant Bit	MSB	Most Significant Bit
Motion Compensation	MC	Part of the video decoding pipe.
Motion Picture Expert Group	MPEG	MPEG is the international standard body JTC1/SC29/WG11 under ISO/IEC that has defined audio and video compression standards such as MPEG-1, MPEG-2, and MPEG-4, etc.
Motion Vector Field Selection	MVFS	A four-bit field selecting reference fields for the motion vectors of the current macroblock.
Multi Render Targets	MRT	Multiple independent surfaces that may be the target of a sequence of 3D or Media commands that use the same surface state.
Normalized Device Coordinates	NDC	Clip Space Coordinates that have been divided by the Clip Space “W” component.
Object	—	A single triangle, line or point.
Out-of-loop De-Blocking Filter	OLDB	The de-blocking filter operation outside the decoding loop. It is required to support WMV9 Profile A video decoder acceleration.
Out-of-loop De-Ringing Filter	OLDR	The de-ringing filter operation outside the decoding loop. It is required to support WMV9 Profile A video decoder acceleration.
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.



Term	Abbr.	Definition
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.
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 discontinuous register data to be fetched and used as a single operand.
Render Cache	RC	Cache in which pixel color and depth information is written prior to being written to memory, and where prior pixel destination attributes are read in preparation for blending and Z test.
Render Target	RT	A destination surface in memory where render results are written.
Render Target Array Index	—	Selector of which of several render targets the current operation is targeting.
Root Thread	—	A root-node thread. A thread corresponds to a root-node in a thread generation hierarchy. It is a kind of thread associated with the media fixed function pipeline. A root thread is originated from the VFE unit and forwarded to the Thread Dispatcher by the TS unit. A root thread may or may not have child threads. A root thread may have scratch memory managed by TS. A root thread with children has its URB resource managed by the VFE.
Sampler	—	Shared function that samples textures and reads data from buffers on behalf of EU programs.
Scratch Space	—	Memory allocated to the subsystem that is used by EU threads for data storage that exceeds their register allocation, persistent storage, storage of mask stack entries beyond the first 16, etc.
Shader	—	A Gen4 program that is supplied by the application in an high level shader language, and translated to Gen4 instructions by the jitter.



Term	Abbr.	Definition
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.
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.
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, <i>r2</i> , is 256-bit wide, 256-bit aligned register. A sub-register, <i>r2.3:d</i> , is the fourth dword of GRF register <i>r2</i> .
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



Term	Abbr.	Definition
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.
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.
Vertical Stride	VertStride	The distance in element-sized units between 2 vertically-adjacent elements of a Gen4 region-based GRF access.



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

§§





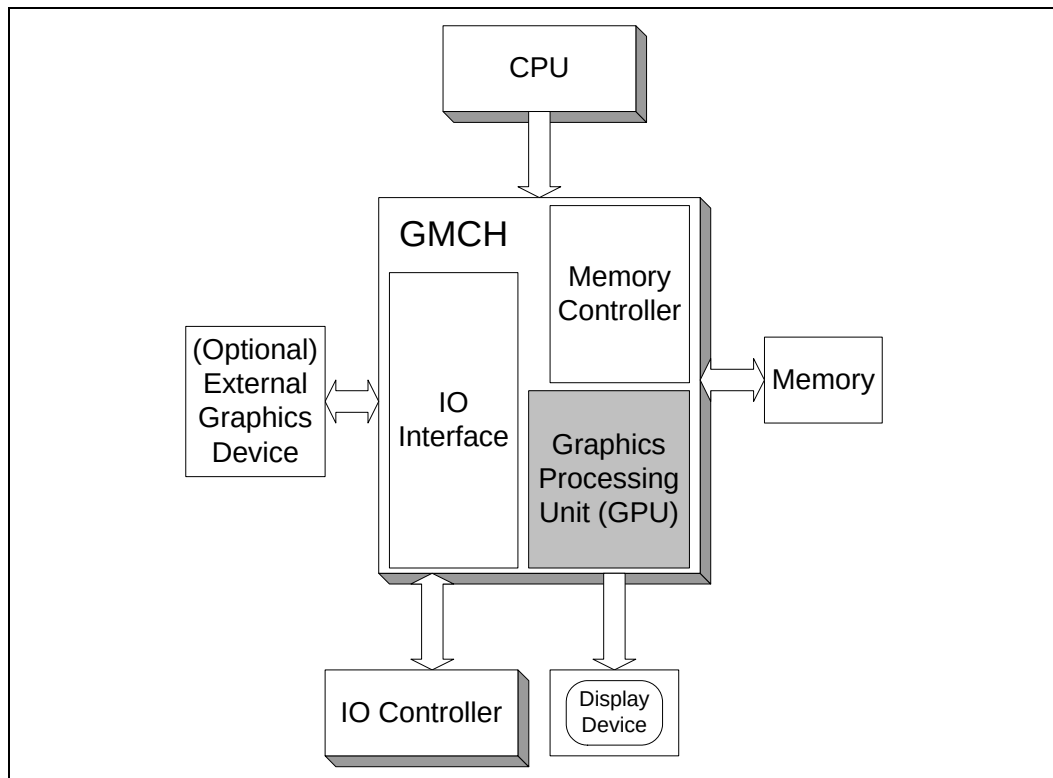
2 Graphics Device Overview

2.1 Graphics Memory Controller Hub (GMCH)

The GMCH is a system memory controller with an integrated graphics device. The integrated graphics device is sometimes referred to in this document as a Graphics Processing Unit (GPU). The GMCH connects to the CPU via a host bus and to system memory via a memory bus. The GMCH also contains some IO functionality to interface to an external graphics device and also to an IO controller. This document will not contain any further references to external graphics devices or IO controllers.

The graphics core, or GPU, resides within the GMCH, which also contains the memory interface, configuration registers, and other chipset functions. The GPU itself can be viewed as comprising the command streamer (CS) or command parser, the Memory Interface or MI, the display interface, and (by far the largest element of the Gen4 family GMCH) the 3D/Media engine. This latter piece is made up of the 3D and media “fixed function” (FF) pipelines, and the Gen4 subsystem, which these pipelines make use of to run “shaders” and kernels.

Figure 2-1. GMCH Block Diagram

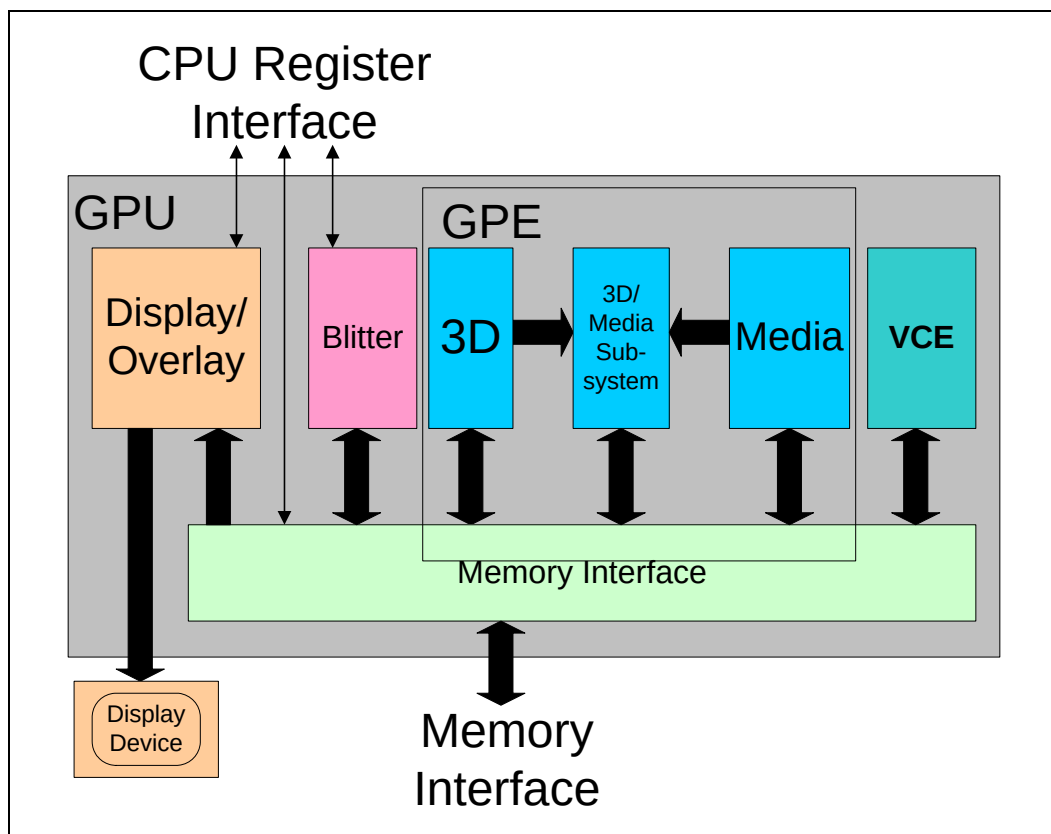


2.2 Graphics Processing Unit (GPU)

The Graphics Processing Unit is controlled by the CPU through a direct interface of memory-mapped IO registers, and indirectly by parsing commands that the CPU has placed in memory. The display interface and blitter (**block image transferr**er) are controlled primarily by direct CPU register addresses, while the 3D and Media pipelines and the parallel Video Codec Engine (VCE) are controlled primarily through instruction lists in memory.

The Gen4 subsystem contains an array of cores, or execution units, along with a number of "shared functions", which receive and process messages at the behest of programs running on the cores. The shared functions perform critical tasks such as sampling textures and updating the render target (usually the frame buffer). The cores themselves are described by an instruction set architecture, or ISA.

Figure 2-2. Block Diagram of the GPU





3 Graphics Processing Engine (GPE)

3.1 Introduction

This chapter serves two purposes: It provides a high-level description of the Graphics Processing Engine (GPE) of the GEN4 Graphics Processing Unit (GPU). It also specifies the programming and behaviors of the functions common to both pipelines (3D, Media) within the GPE. However, details specific to either pipeline are not addressed here.

3.2 Overview

The Graphics Processing Engine (GPE) performs the bulk of the graphics processing provided by the GEN4 GPU. It consists of the 3D and Media fixed-function pipelines, the Command Streamer (CS) unit that feeds them, and the GEN4 Subsystem that provides the bulk of the computations required by the pipelines.

3.2.1 Block Diagram

Figure 3-1. The Graphics Processing Engine

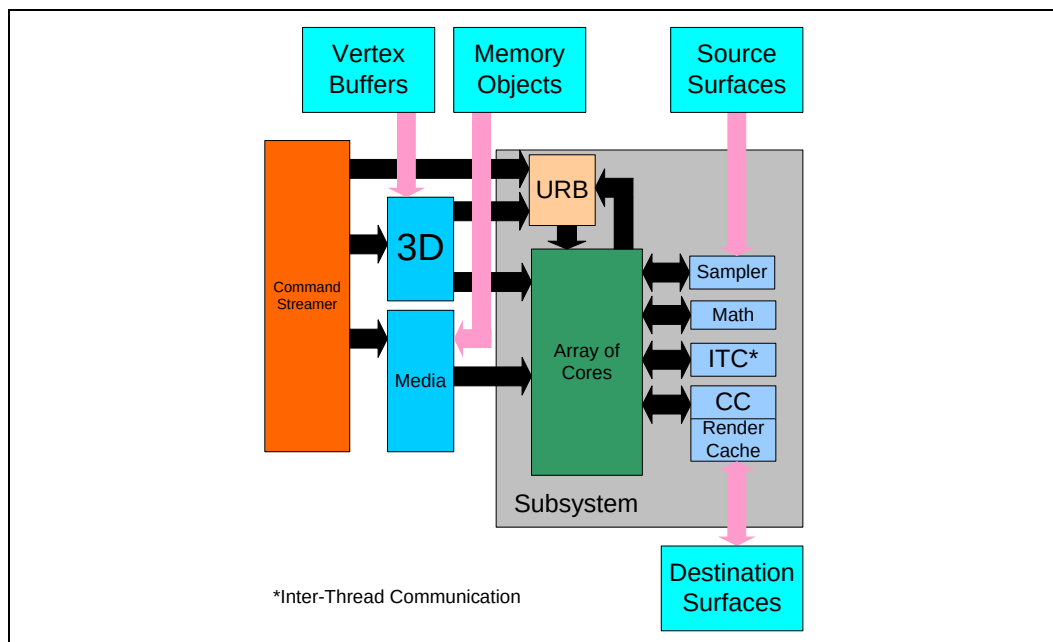
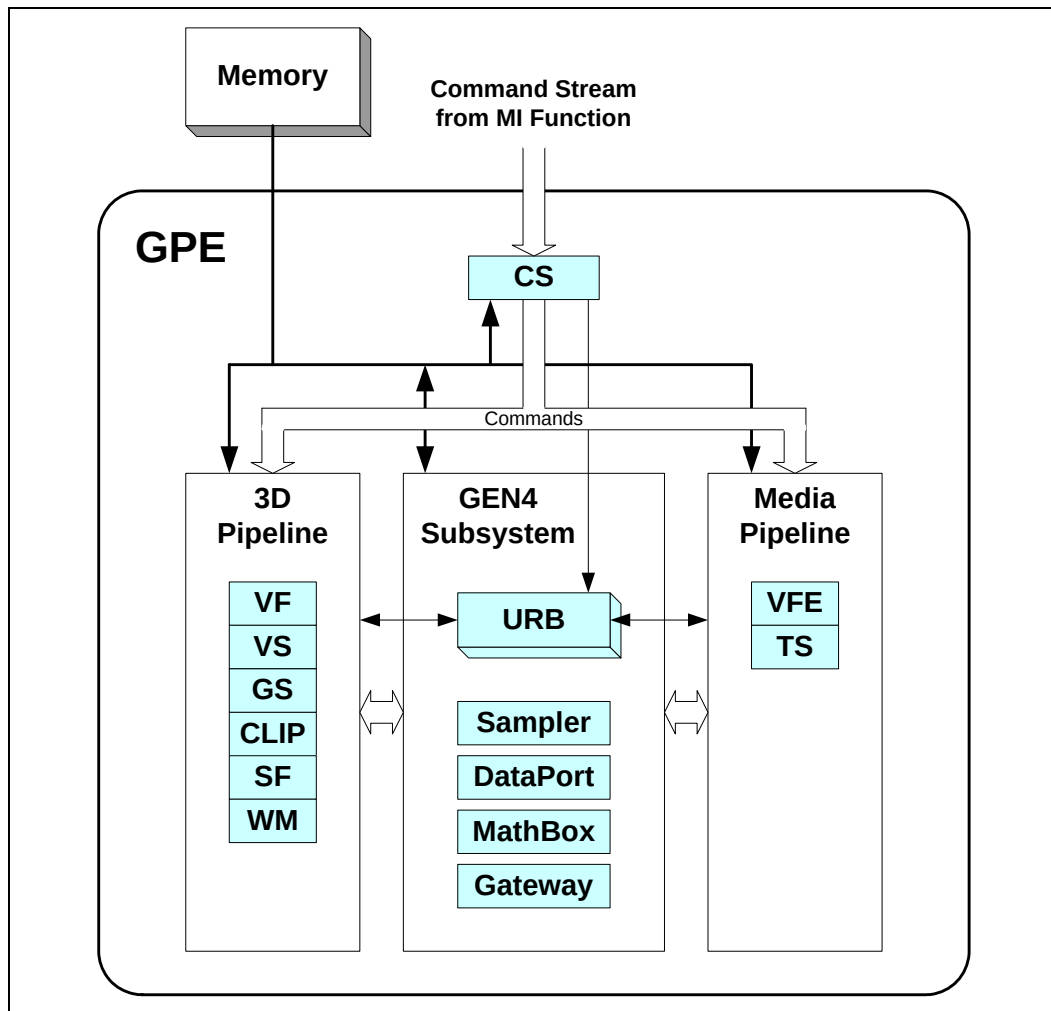


Figure 3-2. GPE Diagram Showing Fixed / Shared Functions



3.2.2 Command Stream (CS) Unit

The Command Stream (CS) unit manages the use of the 3D and Media pipelines, in that it performs switching between pipelines and forwarding command streams to the currently active pipeline. It manages allocation of the URB and helps support the Constant URB Entry (CURBE) function.

3.2.3 3D Pipeline

The 3D pipeline provides specialized 3D primitive processing functions. These functions are provided by a pipeline of “fixed function” stages (units) and GEN4 threads spawned by these units. See *3D Pipeline Overview*.



3.2.4 Media Pipeline

The Media pipeline provides both specialized media-related processing functions and the ability to perform more general (“generic”) functionality. These Media-specific functions are provided by a Video Front End (VFE) unit. A Thread Spawner (TS) unit is utilized to spawn GEN4 threads requested by the VFE unit or as required when the pipeline is used for general processing. See *Media Pipeline Overview*.

3.2.5 GEN4 Subsystem

The GEN4 Subsystem is the collective name for the GEN4 programmable cores, the Shared Functions accessed by them (including the Sampler, Extended Math Unit (“MathBox”), the DataPort, and the Inter-Thread Communication (ITC) Gateway), and the Dispatcher which manages threads running on the cores.

3.2.5.1 Execution Units (EUs)

While the number of EU cores in the GEN4 subsystem is almost entirely transparent to the programming model, there are a few areas where this parameter comes into play:

- The amount of scratch space required is a function of (#EUs * #Threads/EU)
- Debug registers (e.g., EU-enable bitmasks)

Device	# of EUs	#Threads/EU
All	8	4

3.2.6 GPE Function IDs

The following table lists the assignments (encodings) of the Shared Function and Fixed Function IDs used within the GPE. A Shared Function is a valid target of a message initiated via a ‘send’ instruction. A Fixed Function is an identifiable unit of the 3D or Media pipeline. Note that the Thread Spawner is both a Shared Function and Fixed Function.

Note: The initial intention was to combine these two ID namespaces, so that (theoretically) an agent (such as the Thread Spawner) that served both as a Shared Function and Fixed Function would have a single, unique 4-bit ID encoding. However, this is not a requirement of the architecture.



Table 3-1. Gen4 Function IDs

ID[3:0]	SFID	Shared Function	FFID	Fixed Function
0x0	SFID_NULL	Null	FFID_NULL	Null
0x1	SFID_MATH	Extended Math	Reserved	---
0x2	SFID_SAMPLER	Sampler	Reserved	---
0x3	SFID_GATEWAY	Message Gateway	Reserved	---
0x4	SFID_DP_READ	DataPort Read	Reserved	---
0x5	SFID_DP_WRITE	DataPort Write	Reserved	---
0x6	SFID_URB	URB	Reserved	---
0x7	SFID_SPAWNER	Thread Spawner	FFID_SPAWNER	Thread Spawner
0x8	Reserved	---	FFID_VFE	Video Front End
0x9	Reserved	---	FFID_VS	Vertex Shader
0xA	Reserved	---	FFID_CS	Command Stream
0xB	Reserved	---	FFID_VF	Vertex Fetch
0xC	Reserved	---	FFID_GS	Geometry Shader
0xD	Reserved	---	FFID_CLIP	Clipper Unit
0xE	Reserved	---	FFID_SF	Strip/Fan Unit
0xF	Reserved	---	FFID_WM	Windower/Masker Unit



3.3 Pipeline Selection

The PIPELINE_SELECT command is used to specify which GPE pipeline (3D or Media) is to be considered the “current” active pipeline. Issuing 3D-pipeline-specific commands when the Media pipeline is selected, or vice versa, is UNDEFINED.

This command causes the URB deallocation of the previously selected pipe. For example, switching from the 3D pipe to the Media pipe (either within or between contexts) will cause the CS to send a “Deallocating Flush” down the 3D pipe. This will cause each 3D FF to start a URB deallocation sequence after the current tasks are done. When the WM sees this, it will de-reference the current Constant URB Entry. Once this happens, all 3D URB entries will be deallocated (after some north bus delay). This allows the CS to set the URB fences for the media pipe. And vice versa for switching from media to 3D pipes.

Programming Restriction:

- Software must ensure the current pipeline is flushed via an MI_FLUSH prior to the execution of PIPELINE_SELECT.

DWord	Bit	Description
0	31:29	Instruction Type = GFXPIPE = 3h
	28:16	3D Instruction Opcode = PIPELINE_SELECT GFXPIPE[28:27 = 0h, 26:24 = 1h, 23:16 = 04h] (Non-pipelined)
	15:1	Reserved: MBZ
	0	Pipeline Select 0: 3D pipeline is selected 1: Media pipeline is selected

This one bit of **Pipeline Select** state is contained within the logical context.

Note: Implementation Note: Currently, this bit is only required for switching pipelines. The CS unit needs to know which pipeline (if any) has an outstanding CURBE reference pending. A switch away from that pipeline requires the CS unit to force any CURBE entries to be deallocated.

3.4 URB Allocation

Storage in the URB is divided among the various fixed functions in a programmable fashion using the URB_FENCE command (see following).

3.4.1 URB_FENCE

The URB_FENCE command is used to define the current URB allocation for those FF units that can own (write) URB entries. The FF units' allocations are specified via a set of 512-bit granular *fence pointers*, in a predefined order in the URB as shown in the diagram below. (In the discussion below, “previous” refers to the relative position in the list presented in Figure 3-3, not necessarily with respect to the order of fence pointers in the command or the order of FF units in the physical pipelines).

The URB_FENCE command is required in certain programming sequences (see programming notes below, as well as the Command Ordering Rules subsection below).

Each FF unit that can own URB entries is provided with a fence pointer that specifies the URB address immediately following that FF unit's allocated region (i.e., it identifies the end of the allocated region). The range allocated to a particular FF unit therefore starts at the previous FF unit's fence pointer and ends at its associated fence pointer. The starting fence pointer for the first (VS) fixed function is implied to be 0. URB locations starting at the fence pointer of the last FF unit in the list (CS) are effectively unusable. If a FF unit's fence pointer is identical to the previous FF unit's fence pointer, the FF unit has no URB storage allocated to it (and therefore the FF unit must either be disabled or otherwise programmed to not require its own URB entries).

The fencing and allocation of the URB is performed in a pipeline-dependent manner. The following diagrams show the layout of the URB fence regions for the 3D and Media pipelines (depending on which one is selected via PIPELINE_SELECT). In the URB_FENCE command, **Fence** values not associated with the currently selected pipeline will be ignored.

Figure 3-3. URB Allocation – 3D Pipeline

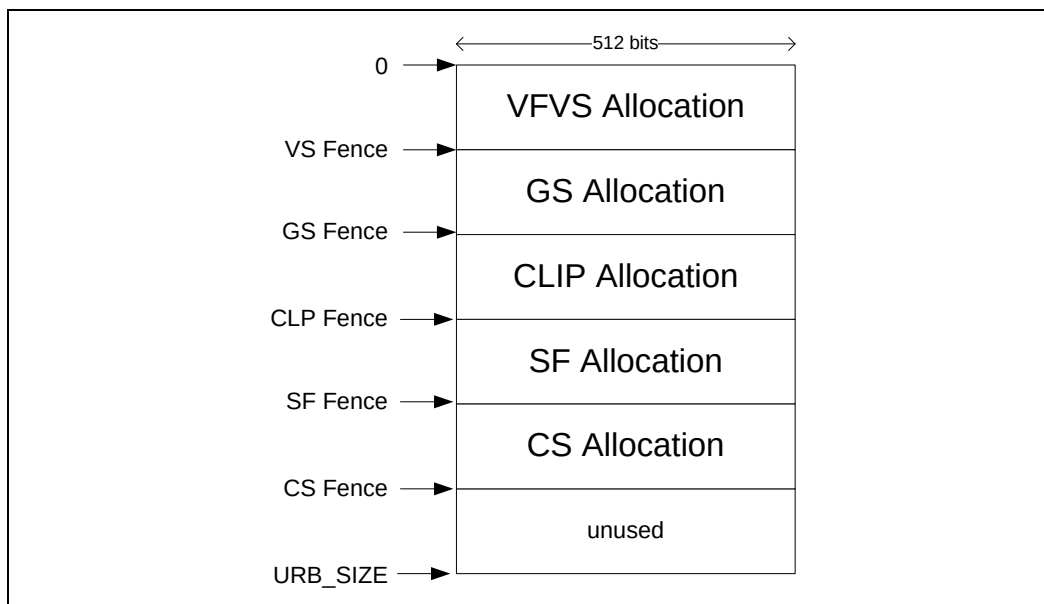
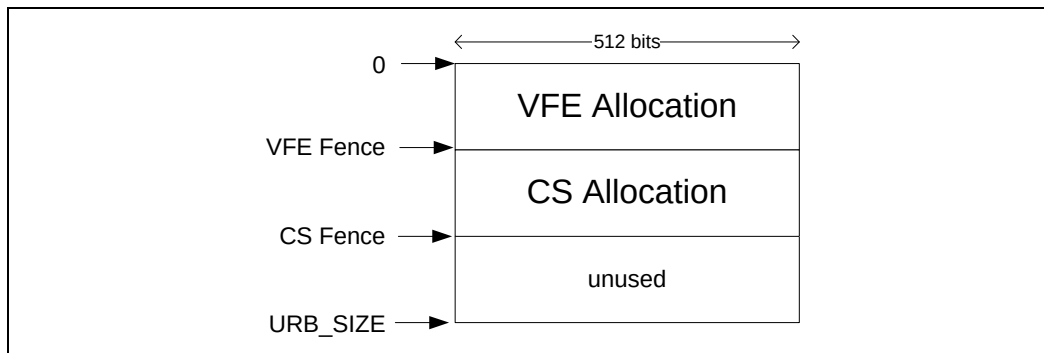




Figure 3-4 URB Allocation – Media Pipeline



Programming Notes:

1. URB Size
 - a. URB_SIZE is 16KB = 256 512-bit units
2. On a per-fixed-function basis, software must modify (via pipeline state pointer commands) any (active) fixed-function state which relies on the size of the fixed-function's fenced URB region. If a fixed-function's URB region is repositioned within the URB, but retains the same size, the previous state is still valid. Note that changing fence pointers via URB_FENCE only affects the location of the allocated region, not the contents – i.e., no data copy is performed.
3. A URB_FENCE command must be issued subsequent to any change to the value in the GS or CLIP unit's **Maximum Number of Threads** state (via PIPELINE_STATE_POINTERS) and before any subsequent pipeline processing (e.g., via 3DPRIMITIVE or CONSTANT_BUFFER).
4. A URB_FENCE command must be issued subsequent to any change to the value in any FF unit's **Number of URB Entries or URB Entry Allocation Size** state (via PIPELINE_STATE_POINTERS) and before any subsequent pipeline processing (e.g., via 3DPRIMITIVE or CONSTANT_BUFFER). Also see the Command Ordering Rules subsection below.
5. To workaround a silicon issue it is required that this instruction be programmed within a 64 byte cacheline aligned memory chunk (i.e., it must not cross a 64-byte cacheline boundary.)

URB_FENCE			
Project: All		Length Bias:	2
This command is used to set the fences between URB regions owned by the fixed functions.			
DWord	Bit	Description	
0	31:29	Command Type Default Value: 3h GFXPIPE Format: OpCode	



URB_FENCE		
28:27	Command SubType Default Value: 0h GFXPIPE_COMMON Format: OpCode	
26:24	3D Command Opcode Default Value: 0h GFXPIPE_PIPELINED Format: OpCode	
23:16	3D Command Sub Opcode Default Value: 00h URB_FENCE Format: OpCode	
15:14	Reserved Project: All Format: MBZ	
13	CS Unit URB Reallocation Request Project: All Format: Enable FormatDesc If set, the CS unit will perform a URB entry deallocation/reallocation action. Note: Modifying the CS URB allocation via URB_FENCE invalidates any previous CURBE entries. Therefore software must subsequently [re]issue a CONSTANT_BUFFER command before CURBE data can be used in the pipeline. (The following description applies to all URB Reallocation Request bits): A reallocation action is required if either (a) the region of the URB allocated to this unit changes location or size as defined by the bracketing Fence values, or (b) the Number of URB Entries or URB Entry Allocation Size state variables associated with this unit have been modified since the last reallocation action. Software is required to set this bit accordingly. Within the context's command stream, this is the only cause of a reallocation action --- a reallocation action is not performed as a side effect of a change to the formentioned state variables. Hardware will, however, take care of deallocation/reallocation resulting from context swtiches. Note that all Fence values provided in this command (and relevant to the selected pipeline) are considered valid and provided to the active pipeline, regardless of any reallocation requests. For example, if the 3D pipeline is selected and only the CS Fence is being changed, the CLIP, GS, VS and SF Fence values must be programmed to their correct (previous) values.	
12	VFE Unit URB Reallocation Request Project: All Format: Enable FormatDesc If set, the VFE unit will perform a URB entry deallocation/reallocation action. (See CS Unit URB Reallocation Request description)	
11	SF Unit URB Reallocation Request Project: All Format: Enable FormatDesc If set, the SF unit will perform a URB entry deallocation/reallocation action. (See CS Unit URB Reallocation Request description)	



URB_FENCE		
	10	CLIP Unit URB Reallocation Request Project: All Format: Enable FormatDesc If set, the CLIP unit will perform a URB entry deallocation/reallocation action. (See CS Unit URB Reallocation Request description)
	9	GS Unit URB Reallocation Request Project: All Format: Enable FormatDesc If set, the GS unit will perform a URB entry deallocation/reallocation action. (See CS Unit URB Reallocation Request description)
	8	VS Unit URB Reallocation Request Project: All Format: Enable FormatDesc If set, the VS unit will perform a URB entry deallocation/reallocation action. (See CS Unit URB Reallocation Request description)
	7:0	DWord Length Default Value: 1h Excludes DWord (0,1) Format: =n Total Length - 2 Project: All
1	31:30	Reserved Project: All Format: MBZ
	29:20	CLIP Fence Project: All Format: U10 representing the first 512-bit URB address beyond this unit's URB space FormatDesc Range [GS Fence,256] Indicates the URB fence value for the CLIP unit. This field is considered valid whenever the 3D pipeline is selected via PIPELINE_SELECT. Otherwise it is ignored.
	19:10	GS Fence Project: All Format: U10 representing the first 512-bit URB address beyond this unit's URB space FormatDesc Range [VS Fence,256] Indicates the URB fence value for the GS unit. This field is considered valid whenever the 3D pipeline is selected via PIPELINE_SELECT. Otherwise it is ignored.



URB_FENCE		
	9:0	VS Fence Project: All Format: U10 representing the first 512-bit URB address beyond this unit's URB space Range [0,256] FormatDesc Indicates the URB fence value for the VS unit. Note: When the 3D pipeline is used, the VS FF unit must be allocated URB space even if the VS function (i.e., "vertex shading") is disabled. The VF unit utilizes Vertex URB Entries (VUEs) allocated to the VS in order to input vertex data to the 3D pipeline even if vertex shading is not enabled. This field is considered valid whenever the 3D pipeline is selected via PIPELINE_SELECT. Otherwise it is ignored.
2	31	Reserved Project: All Format: MBZ
	30:20	CS Fence Project: All Format: U11 representing the first 512-bit URB address beyond this unit's URB space Range [VFE Fence,256] (Media) or [SF Fence,256] (3D Pipe) FormatDesc Indicates the URB fence value for the CS unit. This field is always considered valid, as it is relevant regardless of the currently selected pipeline.
	19:10	VFE Fence Project: All Format: U10 representing the first 512-bit URB address beyond this unit's URB space Range [0,256] FormatDesc Indicates the URB fence value for the VFE unit. This field is considered valid whenever the Media pipeline is selected via PIPELINE_SELECT. Otherwise it is ignored.
	9:0	SF Fence Project: All Format: U10 representing the first 512-bit URB address beyond this unit's URB space Range [CLIP Fence,256] FormatDesc Indicates the URB fence value for the SF unit. This field is considered valid whenever the 3D pipeline is selected via PIPELINE_SELECT. Otherwise it is ignored.



3.5 Constant URB Entries (CURBEs)

3.5.1 Overview

It is anticipated that threads will need to access some amount of non-immediate constant data, e.g., a matrix from a VS kernel. While the DataPort can be used to read (“pull”) this data from a memory buffer, doing so may incur a performance penalty due to the latency of the access. In order to provide a higher-performance path, both pipelines are provided with the ability to preload (“push”) data from a memory buffer into the URB and have portions of that data automatically included in subsequent thread payloads. These pushed constants will then be immediately available for use by the thread (at the expense of increased GRF allocation, dispatch latency, etc.).

The mechanism to push constants into thread payloads is the *Constant URB Entry* (CURBE). The CURBE is a special URB entry (owned by the CS unit) used to store the constant data. Software can issue the `CONSTANT_BUFFER` command to specify the source Constant Buffer in memory. Upon receipt of that command, the CS unit will read the Constant Buffer data from memory and write the data into the CURBE. Fixed functions of the pipeline can be programmed to include their subset of the CURBE data in thread payloads.

3.5.2 Multiple CURBE Allocation

There is only one “current” CURBE state provided by the architecture. Portions of the current CURBE is available to the various fixed-function stages of the pipelines. However, in order to avoid having to flush the pipeline prior to modifying the contents of the current CURBE, the GPE is supplied with the ability to pipeline changes to the current CURBE. This support comes in the form of a set of CURBEs that can be maintained in the URB. A region of the URB can be allocated to the CS unit (see `URB_FENCE` command) to hold this set of CURBEs. Within that region, software can define a set of up to 4 *Constant URB Entries* (CURBEs) – (see `CS_URB_STATE` command).

When a `CONSTANT_BUFFER` command is received, an attempt is made to find an unused CURBE within the set. If one is found, it is used as the destination of the memory read, and the handle of that CURBE is passed down the pipeline without incurring a pipeline flush performance penalty. Fixed functions will switch to using the new CURBE as the handle travels down the pipeline. When the handle reaches the end of the pipeline, the previous CURBE is marked as unused.

If a `CONSTANT_BUFFER` command is encountered and there is only one CURBE allocated and it is in use, the CS unit will implicitly wait for the pipeline to drain and the CURBE to become available to be overwritten. Due to the performance impact of modifying the CURBE when only a single CURBE is allocated, it is recommended that software operate with a single CURBE allocation only if (a) the CURBE is large enough to make multiple allocations undesirable, and/or (b) it is anticipated that the constant data will remain static for long processing periods (thus amortizing the impact of modifying it).



3.5.3 CS_URB_STATE

CS_URB_STATE			
Project: All		Length Bias:	2
The CS_URB_STATE packet is used to define the number and size of CURBEs contained within the CS unit's allocated URB region.			
DWord	Bit	Description	
0	31:29	Command Type Default Value: 3h GFXPIPE Format: OpCode	
	28:27	Command SubType Default Value: 0h GFXPIPE_COMMON Format: OpCode	
	26:24	3D Command Opcode Default Value: 0h GFXPIPE_PIPELINED Format: OpCode	
	23:16	3D Command Sub Opcode Default Value: 01h CS_URB_STATE 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	31:9	Reserved Project: All Format: MBZ	
	8:4	URB Entry Allocation Size Project: All Format: U5 count (of 512-bit units) – 1 FormatDesc Range [0,31] = [1,32] 512-bit units = [2,64] 256-bit URB rows Specifies the length of each URB entry owned by the CS unit.	
	3	Reserved Project: All Format: MBZ	
	2:0	Number of URB Entries Project: All Format: U3 count of entries FormatDesc Range [0,4] Specifies the number of URB entries that are used by the CS unit.	



3.5.4 CONSTANT_BUFFER

CONSTANT_BUFFER		
Project:	All	Length Bias: 2
The CONSTANT_BUFFER packet is used to define the memory address of data that will be read by the CS unit and stored into the current CURBE entry. Programming Notes: - Issuing a CONSTANT_BUFFER packet with Valid set when the CS unit does not have any CURBE entries allocated in the URB results in UNDEFINED behavior. - Modifying the CS URB allocation via URB_FENCE invalidates any previous CURBE entries. Therefore software must subsequently [re]issue a CONSTANT_BUFFER command before CURBE data can be used in the pipeline.		
DWord	Bit	Description
0	31:29	Command Type Default Value: 3h GFXPIPE Format: OpCode
	28:27	Command SubType Default Value: 0h GFXPIPE_COMMON Format: OpCode
	26:24	3D Command Opcode Default Value: 0h GFXPIPE_PIPELINED Format: OpCode
	23:16	3D Command Sub Opcode Default Value: 02h CONSTANT_BUFFER Format: OpCode
	15:9	Reserved Project: All Format: MBZ
	8	Valid Project: All Format: Enable FormatDesc If TRUE, a Constant Buffer will be defined and possibly used in the pipeline (depending on FF unit state programming). The Buffer Starting Address and Buffer Length fields are valid. If FALSE, the Constant Buffer becomes undefined and unused. The Buffer Starting Address and Buffer Length fields are ignored. The FF unit state descriptors must not specify the use of CURBE data, or behavior is UNDEFINED.
	7:0	DWord Length Default Value: 0h Excludes DWord (0,1) Format: =n Total Length - 2 Project: All



CONSTANT_BUFFER		
1	31:6	Buffer Starting Address Project: All Format: GeneralStateOffset[31:6] or GraphicsAddress[31:6] (see below) FormatDesc If Valid is set and INSTPM<CONSTANT_BUFFER Address Offset Disable> is clear (enabled), this field defines the location of the memory-resident constant data via a 64Byte-granular offset from the General State Base Address. If Valid is set and INSTPM<CONSTANT_BUFFER Address Offset Disable> is set (disabled), this field defines the location of the memory-resident constant data via a 64Byte-granular Graphics Address (not offset). Programming Notes Constant Buffers can only be allocated in linear (not tiled) graphics memory Constant Buffers can only be mapped to Main Memory (UC)
	5:0	Buffer Length Project: All Format: U6 Count-1 in 512-bit units FormatDesc If Valid is set, this field specifies the length of the constant data to be loaded from memory into the CURBE in 512-bit units (minus one). The length must be less than or equal to the URB Entry Allocation Size specified via the CS_URB_STATE command.

3.6 Memory Access Indirection

The GPE supports the indirection of certain graphics (GTT-mapped) memory accesses. This support comes in the form of two *base address* state variables used in certain memory address computations with the GPE.

The intent of this functionality is to support the dynamic relocation of certain driver-generated memory structures after command buffers have been generated but prior to their submittal for execution. For example, as the driver builds the command stream it could append pipeline state descriptors, kernel binaries, etc. to a general state buffer. References to the individual items would be inserting in the command buffers as offsets from the base address of the state buffer. The state buffer could then be freely relocated prior to command buffer execution, with the driver only needing to specify the final base address of the state buffer. Two base addresses are provided to permit surface-related state (binding tables, surface state tables) to be maintained in a state buffer separate from the general state buffer.

While the use of these base addresses is unconditional, the indirection can be effectively disabled by setting the base addresses to zero. The following table lists the various GPE memory access paths and which base address (if any) is relevant.



Table 3-2. Base Address Utilization

Base Address Used	Memory Accesses
General State Base Address	CS unit reads from CURBE Constant Buffers via CONSTANT_BUFFER when INSTPM < CONSTANT_BUFFER Address Offset Disable > is clear (enabled).
	3D Pipeline FF state read by the 3D FF units, as referenced by state pointers passed via 3DSTATE_PIPELINE_POINTERS.
	Media pipeline FF state , as referenced by state pointers passed via MEDIA_PIPELINE_POINTERS.
	DataPort memory accesses resulting from 'stateless' DataPort Read / Write requests . See <i>DataPort</i> for a definition of the 'stateless' form of requests.
General State Base Address	Sampler reads of Sampler State data and associated Default Color State data
	Viewport states used by CLIP, SF, and WM/CC
	COLOR_CALC_STATE
General State Base Address	Normal EU instruction stream (non-system routine)
	System routine EU instruction stream (starting address = SIP)
Surface State Base Address	Sampler and DataPort reads of Binding Table data, as referenced by BT pointers passed via 3DSTATE_BINDING_TABLE_POINTERS
	Sampler and DataPort reads of Surface State data
Indirect Object Base Address	MEDIA_OBJECT Indirect Data accessed by the CS unit .
None	CS unit reads from Ring Buffers, Batch Buffers
	CS unit reads from CURBE Constant Buffers via CONSTANT_BUFFER when INSTPM < CONSTANT_BUFFER Address Offset Disable > is set (disabled).
	CS writes resulting from 3D_CONTROL
	All VF unit memory accesses (Index Buffers, Vertex Buffers)
	All Sampler Surface Memory Data accesses (texture fetch, etc.)
	All DataPort memory accesses <u>except 'stateless' DataPort Read/Write requests</u> (e.g., RT accesses.) See <i>DataPort</i> for a definition of the 'stateless' form of requests.
	Memory reads resulting from STATE_PREFETCH commands
	Any physical memory access by the device
	GTT-mapped accesses not included above (i.e., default)



The following notation is used in the PRM to distinguish between addresses and offsets:

Notation	Definition
PhysicalAddress[n:m]	Corresponding bits of a physical graphics memory byte address (not mapped by a GTT)
GraphicsAddress[n:m]	Corresponding bits of an absolute, virtual graphics memory byte address (mapped by a GTT)
GeneralStateOffset[n:m]	Corresponding bits of a relative byte offset added to the General State Base Address value, the result of which is interpreted as a virtual graphics memory byte address (mapped by a GTT)
SurfaceStateOffset[n:m]	Corresponding bits of a relative byte offset added to the Surface State Base Address value, the result of which is interpreted as a virtual graphics memory byte address (mapped by a GTT)

3.6.1 STATE_BASE_ADDRESS

The STATE_BASE_ADDRESS command sets the base pointers for subsequent state, instruction, and media indirect object accesses by the GPE. (See Table 3-2. Base Address Utilization for details)

Programming Notes:

- The following commands must be reissued following any change to the base addresses:
 - 3DSTATE_PIPELINE_POINTERS
 - 3DSTATE_BINDING_TABLE_POINTERS
 - MEDIA_STATE_POINTERS.
- Execution of this command causes a full pipeline flush, thus its use should be minimized for higher performance.



STATE_BASE_ADDRESS			
Project:	All	Length Bias:	2
The STATE_BASE_ADDRESS command sets the base pointers for subsequent state, instruction, and media indirect object accesses by the GPE. (See Table 3-2. Base Address Utilization for details) Programming Notes: - The following commands must be reissued following any change to the base addresses: — 3DSTATE_PIPELINE_POINTERS — 3DSTATE_BINDING_TABLE_POINTERS — MEDIA_STATE_POINTERS. - Execution of this command causes a full pipeline flush, thus its use should be minimized for higher performance.			
DWord	Bit	Description	
0	31:29	Command Type Default Value: 3h GFXPIPE Format: OpCode	
	28:27	Command SubType Default Value: 0h GFXPIPE_COMMON Format: OpCode	
	26:24	3D Command Opcode Default Value: 1h GFXPIPE_NONPIPELINED Format: OpCode	
	23:16	3D Command Sub Opcode Default Value: 01h STATE_BASE_ADDRESS 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	



STATE_BASE_ADDRESS															
1	31:12	General State Base Address Project: All Format: GraphicsAddress[31:12] FormatDesc Specifies the 4K-byte aligned base address for general state accesses. See Table 3-2 for details on where this base address is used.													
	11:1	Reserved Project: All Format: MBZ Modify Enable Project: All Format: Enable FormatDesc The address in this dword is updated only when this bit is set.													
	0	<table> <thead> <tr> <th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr> </thead> <tbody> <tr> <td>0h</td><td>Disable</td><td>Ignore the updated address</td><td>All</td></tr> <tr> <td>1h</td><td>Enable</td><td>Modify the address</td><td>All</td></tr> </tbody> </table>		Value	Name	Description	Project	0h	Disable	Ignore the updated address	All	1h	Enable	Modify the address	All
Value	Name	Description	Project												
0h	Disable	Ignore the updated address	All												
1h	Enable	Modify the address	All												
2	31:12	Surface State Base Address Project: All Format: GraphicsAddress[31:12] FormatDesc Specifies the 4K-byte aligned base address for binding table and surface state accesses. See Table 3-2 for details on where this base address is used.													
	11:1	Reserved Project: All Format: MBZ Modify Enable Project: All Format: Enable FormatDesc The address in this dword is updated only when this bit is set.													
	0	<table> <thead> <tr> <th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr> </thead> <tbody> <tr> <td>0h</td><td>Disable</td><td>Ignore the updated address</td><td>All</td></tr> <tr> <td>1h</td><td>Enable</td><td>Modify the address</td><td>All</td></tr> </tbody> </table>		Value	Name	Description	Project	0h	Disable	Ignore the updated address	All	1h	Enable	Modify the address	All
Value	Name	Description	Project												
0h	Disable	Ignore the updated address	All												
1h	Enable	Modify the address	All												



STATE_BASE_ADDRESS															
3	31:12	Indirect Object Base Address Project: All Format: GraphicsAddress[31:12] FormatDesc Specifies the 4K-byte aligned base address for indirect object load in MEDIA_OBJECT command. See Table 3-2 for details on where this base address is used.													
	11:1	Reserved Project: All Format: MBZ													
	0	Modify Enable Project: All Format: Enable FormatDesc The address in this dword is updated only when this bit is set.													
		<table> <tr> <th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr> <tr> <td>0h</td><td>Disable</td><td>Ignore the updated address</td><td>All</td></tr> <tr> <td>1h</td><td>Enable</td><td>Modify the address</td><td>All</td></tr> </table>	Value	Name	Description	Project	0h	Disable	Ignore the updated address	All	1h	Enable	Modify the address	All	
Value	Name	Description	Project												
0h	Disable	Ignore the updated address	All												
1h	Enable	Modify the address	All												
4	31:12	General State Access Upper Bound Project: All Format: GraphicsAddress[31:12] FormatDesc Specifies the 4K-byte aligned (exclusive) maximum Graphics Memory address for general state accesses. This includes all accesses that are offset from General State Base Address (see Table 3-2). Read accesses from this address and beyond will return UNDEFINED values. Data port writes to this address and beyond will be “dropped on the floor” (all data channels will be disabled so no writes occur). Setting this field to 0 will cause this range check to be ignored. If non-zero, this address must be greater than the General State Base Address .													
	11:1	Reserved Project: All Format: MBZ													
	0	Modify Enable Project: All Format: Enable FormatDesc The bound in this dword is updated only when this bit is set.													
		<table> <tr> <th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr> <tr> <td>0h</td><td>Disable</td><td>Ignore the updated bound</td><td>All</td></tr> <tr> <td>1h</td><td>Enable</td><td>Modify the bound</td><td>All</td></tr> </table>	Value	Name	Description	Project	0h	Disable	Ignore the updated bound	All	1h	Enable	Modify the bound	All	
Value	Name	Description	Project												
0h	Disable	Ignore the updated bound	All												
1h	Enable	Modify the bound	All												



STATE_BASE_ADDRESS			
5	31:12	Indirect Object Access Upper Bound	
		Project:	All
		Format:	GraphicsAddress[31:12]

3.7 Instruction and State Prefetch

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



3.7.1 STATE_PREFETCH

STATE_PREFETCH			
Project: All		Length Bias:	2
(This command is provided strictly for performance optimization opportunities, and likely requires some experimentation to evaluate the overall impact of additional prefetching.) The STATE_PREFETCH command causes the GPE to attempt to prefetch a sequence of 64-byte cache lines into the GPE-internal cache ("L2 ISC") used to access EU kernel instructions and fixed/shared function indirect state data. While state descriptors, surface state, and sampler state are <u>automatically</u> prefetched by the GPE, this command may be used to prefetch data not automatically prefetched, such as: 3D viewport state; Media pipeline Interface Descriptors; EU kernel instructions.			
DWord	Bit	Description	
0	31:29	Command Type Default Value: 3h GFXPIPE Format: OpCode	
	28:27	Command SubType Default Value: 0h GFXPIPE_COMMON Format: OpCode	
	26:24	3D Command Opcode Default Value: 0h GFXPIPE_PIPELINED Format: OpCode	
	23:16	3D Command Sub Opcode Default Value: 03h STATE_PREFETCH 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	31:6	Prefetch Pointer Project: All Format: GraphicsAddress[31:6] FormatDesc Specifies the 64-byte aligned address to start the prefetch from. This pointer is an absolute virtual address, it is <i>not</i> relative to any base pointer.	
	5:3	Reserved Project: All Format: MBZ	
	2:0	Prefetch Count Project: All Format: U3 count of cache lines (minus one) FormatDesc Range [0,7] indicating a count of [1,8] Indicates the number of contiguous 64-byte cache lines that will be prefetched.	



3.8 System Thread Configuration

3.8.1 STATE_SIP

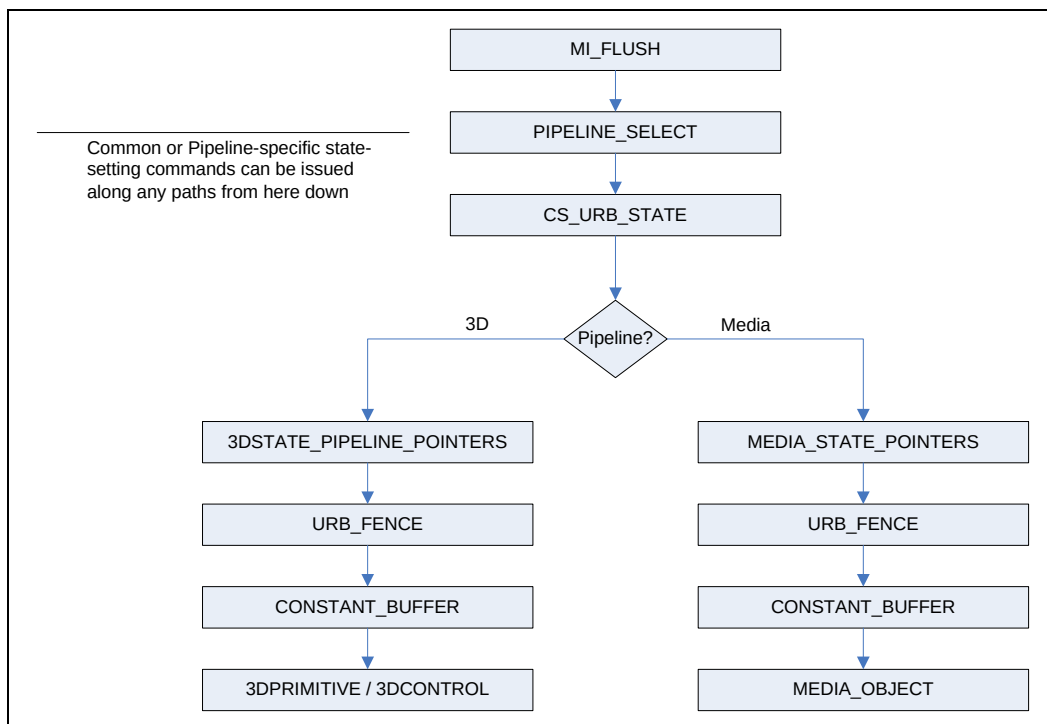
STATE_SIP									
Project: All		Length Bias:	2						
The STATE_SIP command specifies the starting instruction location of the System Routine that is shared by all threads in execution.									
DWord	Bit	Description							
0	31:29	Command Type Default Value: 3h GFXPIPE Format: OpCode							
	28:27	Command SubType Default Value: 0h GFXPIPE_COMMON Format: OpCode							
	26:24	3D Command Opcode Default Value: 1h GFXPIPE_NONPIPELINED Format: OpCode							
	23:16	3D Command Sub Opcode Default Value: 02h STATE_SIP 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	31:4	System Instruction Pointer (SIP) Project: All Format: GeneralStateOffset[31:4] FormatDesc Specifies the instruction address of the system routine associated with the current context as a 128-bit granular offset from the General State Base Address . SIP is shared by all threads in execution. The address specifies the double quadword aligned instruction location. <table><tr><th>Errata</th><th>Description</th><th>Project</th></tr><tr><td>BWT007</td><td>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.)</td><td>[DevBW-A]</td></tr></table>		Errata	Description	Project	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.)	[DevBW-A]
	Errata	Description	Project						
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.)	[DevBW-A]							
3:0	Reserved Project: All Format: MBZ								



3.9 Command Ordering Rules

There are several restrictions regarding the ordering of commands issued to the GPE. This subsection describes these restrictions along with some explanation of why they exist. Refer to the various command descriptions for additional information.

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



3.9.1 PIPELINE_SELECT

The previously-active pipeline needs to be flushed via the MI_FLUSH command immediately before switching to a different pipeline via use of the PIPELINE_SELECT command. Refer to Section 3.3 for details on the PIPELINE_SELECT command.

3.9.2 PIPE_CONTROL

The PIPE_CONTROL command does not require URB fencing/allocation to have been performed, nor does it rely on any other pipeline state. It is intended to be used on both the 3D pipe and the Media pipe. It has special optimizations to support the pipelining capability in the 3D pipe which do not apply to the Media pipe.



3.9.3 URB-Related State-Setting Commands

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

CS_URB_STATE (only) specifies these state variables for the common CS FF unit. 3DSTATE_PIPELINED_POINTERS sets the state variables for FF units in the 3D pipeline, and MEDIA_STATE_POINTERS sets them for the Media pipeline. Depending on which pipeline is currently active, only one of these commands needs to be used. Note that these commands can also be reissued at a later time to change other state variables, though if a change is made to (a) any **Number of URB Entries** and the **URB Entry Allocation Size** state variables or (b) the **Maximum Number of Threads** state for the GS or CLIP FF units, a URB_FENCE command must follow.

3.9.4 Common Pipeline State-Setting Commands

The following commands are used to set state common to both the 3D and Media pipelines. This state is comprised of CS FF unit state, non-pipelined global state (EU, etc.), and Sampler shared-function state.

- STATE_BASE_ADDRESS
- STATE_SIP
- 3DSTATE_SAMPLER_PALETTE_LOAD
- 3DSTATE_CHROMA_KEY

The state variables associated with these commands must be set appropriately prior to initiating activity within a pipeline (i.e., 3DPRIMITIVE or MEDIA_OBJECT).

3.9.5 3D Pipeline-Specific State-Setting Commands

The following commands are used to set state specific to the 3D pipeline.

- 3DSTATE_PIPELINED_POINTERS
- 3DSTATE_BINDING_TABLE_POINTERS
- 3DSTATE_VERTEX_BUFFERS
- 3DSTATE_VERTEX_ELEMENTS
- 3DSTATE_INDEX_BUFFERS
- 3DSTATE_DRAWING_RECTANGLE
- 3DSTATE_CONSTANT_COLOR
- 3DSTATE_DEPTH_BUFFER
- 3DSTATE_POLY_STIPPLE_OFFSET
- 3DSTATE_POLY_STIPPLE_PATTERN
- 3DSTATE_LINE_STIPPLE
- 3DSTATE_GLOBAL_DEPTH_OFFSET

The state variables associated with these commands must be set appropriately prior to issuing 3DPRIMITIVE.



3.9.6 Media Pipeline-Specific State-Setting Commands

The following commands are used to set state specific to the Media pipeline.

- MEDIA_STATE_POINTERS

The state variables associated with this command must be set appropriately prior to issuing MEDIA_OBJECT.

3.9.7 URB_FENCE (URB Fencing & Entry Allocation)

URB_FENCE command is used to initiate URB entry deallocation/allocation processes within pipeline FF units. The URB_FENCE command is first processed by the CS FF unit, and is then directed down the currently selected pipeline to the FF units comprising that pipeline.

As the FF units receive the URB_FENCE command, a URB entry deallocation/allocation process will be initiated if (a) the FF unit is currently enabled (note that some cannot be disabled) and (b) the **ModifyEnable** bit associated with that FF unit's **Fence** value is set. If these conditions are met, the deallocation of the FF unit's currently-allocated URB entries (if any) commences. (Implementation Note: For better performance, this deallocation proceeds in parallel with allocation of new handles).

Modifying the CS URB allocation via URB_FENCE invalidates any previous CURBE entries. Therefore software must subsequently [re]issue a CONSTANT_BUFFER command before CURBE data can be used in the pipeline.

The allocation of new handles (if any) for the FF unit then commences. The parameters used to perform this allocation come from (a) the URB_FENCE **Fence** values, and (b) the relevant URB entry state associated with the FF unit: specifically, the **Number of URB Entries** and the **URB Entry Allocation Size**. For the CS unit, this state is programmed via CS_URB_STATE, while the other FF units receive this state indirectly via PIPELINED_STATE_POINTERS or MEDIA_STATE_POINTERS commands.

Although a FF unit's allocation process relies on its URB **Fence** as well as the relevant FF unit pipelined state, only the URB_FENCE command initiates URB entry deallocation/allocation. This imposes the following restriction: If a change is made to (a) the **Number of URB Entries** or **URB Entry Allocation Size** state for a given FF unit or (b) the **Maximum Number of Threads** state for the GS or CLIP FF units, a URB_FENCE command specifying a valid URB Fence state for that FF unit must be subsequently issued – at some point prior to the next CONSTANT_BUFFER, 3DPRIMITIVE (if using the 3D pipeline) or MEDIA_OBJECT (if using the Media pipeline). It is invalid to change **Number of URB Entries** or **URB Entry Allocation Size** state for enabled FF units without also issuing a subsequent URB_FENCE command specifying a valid **Fence** valid for that FF unit.

It is valid to change a FF unit's Fence value without specifying a change to its **Number of URB Entries** or **URB Entry Allocation Size** state, though the values must be self-consistent.



3.9.8 CONSTANT_BUFFER (CURBE Load)

The CONSTANT_BUFFER command is used to load constant data into the CURBE URB entries owned by the CS unit. In order to write into the URB, CS URB fencing and allocation must have been established. Therefore, CONSTANT_BUFFER can only be issued after CS_URB_STATE and URB_FENCE commands have been issued, and prior to any other pipeline processing (i.e., 3DPRIMITIVE or MEDIA_OBJECT). See the definition of CONSTANT_BUFFER for more details.

Modifying the CS URB allocation via URB_FENCE invalidates any previous CURBE entries. Therefore software must subsequently [re]issue a CONSTANT_BUFFER command before CURBE data can be used in the pipeline.

3.9.9 3DPRIMITIVE

Before issuing a 3DPRIMITIVE command, all state (with the exception of MEDIA_STATE_POINTERS) needs to be valid. Therefore the commands used to set this state need to have been issued at some point prior to the issue of 3DPRIMITIVE.

3.9.10 MEDIA_OBJECT

Before issuing a MEDIA_OBJECT command, all state (with the exception of 3D-pipeline-specific state) needs to be valid. Therefore the commands used to set this state need to have been issued at some point prior to the issue of MEDIA_OBJECT.



4 Graphics Command Formats

4.1 Command Formats

This section describes the general format of the graphics device commands.

Graphics commands are defined with various formats. The first DWord of all commands is called the *header* DWord. The header contains the only field common to all commands -- the *client* field that determines the device unit that will process the command data. The Command Parser examines the client field of each command to condition the further processing of the command and route the command data accordingly.

Some Gen4 Devices include two Command Parsers, each controlling an independent processing engine. These will be referred to in this document as the Render Command Parser (RCP) and the Video Codec Command Parser (VCCP).

Valid client values for the Render Command Parser are:

Client #	Client
0	Memory Interface (MI_XXX)
1	Miscellaneous (includes Trusted Ops)
2	2D Rendering (XXX_BLT_XXX)
3	Graphics Pipeline (3D and Media)
4-7	Reserved

Graphics commands vary in length, though are always multiples of DWords. The length of a command is either:

- Implied by the client/opcode
- Fixed by the client/opcode yet included in a header field (so the Command Parser explicitly knows how much data to copy/process)
- Variable, with a field in the header indicating the total length of the command

Note that command *sequences* require QWord alignment and padding to QWord length to be placed in Ring and Batch Buffers.

The following subsections provide a brief overview of the graphics commands by client type provides a diagram of the formats of the header DWords for all commands. Following that is a list of command mnemonics by client type.



4.1.1 Memory Interface Commands

Memory Interface (MI) commands are basically those commands which do not require processing by the 2D or 3D Rendering/Mapping engines. The functions performed by these commands include:

- Control of the command stream (e.g., Batch Buffer commands, breakpoints, ARB On/Off, etc.)
- Hardware synchronization (e.g., flush, wait-for-event)
- Software synchronization (e.g., Store DWORD, report head)
- Graphics buffer definition (e.g., Display buffer, Overlay buffer)
- Miscellaneous functions

Refer to the *Memory Interface Commands* chapter for a description of these commands.

4.1.2 2D Commands

The 2D commands include various flavors of Blt operations, along with commands to set up Blt engine state without actually performing a Blt. Most commands are of fixed length, though there are a few commands that include a variable amount of "inline" data at the end of the command.

Refer to the *2D Commands* chapter for a description of these commands.

4.1.3 3D / Media Commands

The 3D/Media commands are used to program the graphics pipelines for 3D or media operations.

Refer to the *3D* chapter for a description of the 3D state and primitive commands and the *Media* chapter for a description of the media-related state and object commands.

4.1.4 Video Codec Commands

4.1.5 Command Header

The Command Headers are shown in the following tables.



Table 4-1. RCP Command Header Format

Bits							
TYPE	31:29	28:24		23	22	21:0	
Memory Interface (MI)	000	Opcode 00h – NOP 0Xh – Single DWord Commands 1Xh – Two+ DWord Commands 2Xh – Store Data Commands 3Xh – Ring/Batch Buffer Cmds			Identification No./DWord Count Command Dependent Data 5:0 – DWord Count 5:0 – DWord Count 5:0 – DWord Count		
Reserved	001	Opcode – 11111		23:19 Sub Opcode 00h – 01h		18:16 Re- served 15:0 DWord Count	
2D	010	Opcode				Command Dependent Data 4:0 – DWord Count	
TYPE	31:29	28:27	26:24	23:16		15:8	7:0
Common	011	00	Opcode – 000	Sub Opcode		Data	DWord Count
Common (NP)	011	00	Opcode – 001	Sub Opcode		Data	DWord Count
Reserved	011	00	Opcode – 010 – 111				
Single Dword Command	011	01	Opcode – 000 – 001	Sub Opcode			N/A
Reserved	011	01	Opcode – 010 – 111				
Media State	011	10	Opcode – 000	Sub Opcode			Dword Count
Media Object	011	10	Opcode – 001 – 010	Sub Opcode		Dword Count	
Reserved	011	10	Opcode – 011 – 111				
3DState	011	11	Opcode – 000	Sub Opcode		Data	DWord Count
3DState (NP)	011	11	Opcode – 001	Sub Opcode		Data	DWord Count
PIPE_Control	011	11	Opcode – 010			Data	DWord Count
3DPrimitive	011	11	Opcode – 011			Data	DWord Count
Reserved	011	11	Opcode – 100 – 111				
Reserved	1XX	XX					

NOTES:

1. The qualifier “NP” indicates that the state variable is non-pipelined and the render pipe is flushed before such a state variable is updated. The other state variables are pipelined (default).



Table 4-2. VCCP Command Header Format

Bits						
TYPE	31:29	28:24		23	22	21:0
Memory Interface (MI)	000	Opcode 00h – NOP 0Xh – Single DWord Commands 1Xh – Reserved 2Xh – Store Data Commands 3Xh – Ring/Batch Buffer Cmds			Identification No./DWord Count Command Dependent Data 5:0 – DWord Count 5:0 – DWord Count 5:0 – DWord Count	
TYPE	31:29	28:27	26:24	23:16		15:0
Reserved	011	0X	XXX	XX		
Reserved	011	10	0XX			
AVC State	011	10	100	Opcode: 0h – 4h		DWord Count
AVC Object	011	10	100	Opcode: 8h		DWord Count
VC1 State	011	10	101	Opcode: 0h – 4h		DWord Count
VC1 Object	011	10	101	Opcode: 8h		DWord Count
Reserved	011	10	110	Opcode: 0h – 1h		DWord Count
Reserved	011	10	110	Opcode: 8h		DWord Count
Reserved	011	10	11X			
Reserved	011	11	XXX			
TYPE	31:29	28:27	26:24	23:21	20:16	15:0
MFX Common	011	10	000	000	subopcode	DWord Count
Reserved	011	10	000	001-111	subopcode	DWord Count
AVC Common	011	10	001	000	subopcode	DWord Count
AVC Dec	011	10	001	001	subopcode	DWord Count
AVC Enc	011	10	001	010	subopcode	DWord Count
Reserved	011	10	001	011-111	subopcode	DWord Count
Reserved (for VC1 Common)	011	10	010	000	subopcode	DWord Count
VC1 Dec	011	10	010	001	subopcode	DWord Count
Reserved (for VC1 Enc)	011	10	010	010	subopcode	DWord Count
Reserved	011	10	010	011-111	subopcode	DWord Count
Reserved (MPEG2 Common)	011	10	011	000	subopcode	DWord Count
MPEG2 Dec	011	10	011	001	subopcode	DWord Count
Reserved (for MPEG2 Enc)	011	10	011	010	subopcode	DWord Count
Reserved	011	10	011	011-111	subopcode	DWord Count
Reserved	011	10	100-111	XXX		



4.2 Command Map

This section provides a map of the graphics command opcodes.

4.2.1 Memory Interface Command Map

All the following commands are defined in *Memory Interface Commands*.

Table 4-3. Memory Interface Commands for RCP

Opcode (28:23)	Command	Comments
1-DWord		
00h	MI_NOOP	
01h	Reserved	
02h	MI_USER_INTERRUPT	
03h	MI_WAIT_FOR_EVENT	
04h	MI_FLUSH	
05h	MI_ARB_CHECK	
06h	Reserved	
07h	MI_REPORT_HEAD	
08-09h	Reserved	
0Ah	MI_BATCH_BUFFER_END	
0Bh-0Fh	Reserved	
2+ DWord		
10h	Reserved	
11h	MI_OVERLAY_FLIP	
12h	MI_LOAD_SCAN_LINES_INCL	
13h	MI_LOAD_SCAN_LINES_EXCL	
14h	MI_DISPLAY_BUFFER_INFO	
15h	Reserved	
16h	Reserved	
17h	Reserved	
18h	MI_SET_CONTEXT	
19h-1Fh	Reserved	
Store Data		
20h	MI_STORE_DATA_IMM	
21h	MI_STORE_DATA_INDEX	
22h	MI_LOAD_REGISTER_IMM	
23h	Reserved	
24h	MI_STORE_REGISTER_MEM	



Opcode (28:23)	Command	Comments
25h	Reserved	
26h	Reserved	
27h–2Fh	Reserved	
Ring / Batch Buffer		
30h	Reserved	
31h	MI_BATCH_BUFFER_START	
32h–3Fh	Reserved	

Table 4-4. Memory Interface Commands for VCCP

Opcode (28:23)	Command	Comments
1-DWord		
00h	MI_NOOP	
01h	Reserved	
02h	MI_USER_INTERRUPT	
03h	Reserved	
04h	MI_FLUSH	
05h	MI_ARB_CHECK	
06-09h	Reserved	
0Ah	MI_BATCH_BUFFER_END	
0Bh–0Fh	Reserved	
2- DWord		
10h–1Fh	Reserved	
Store Data		
20h	MI_STORE_DATA_IMM	
21h	MI_STORE_DATA_INDEX	
22h–2Fh	Reserved	
Ring / Batch Buffer		
30h	Reserved	
31h	MI_BATCH_BUFFER_START	
32h–3Fh	Reserved	



4.2.2 2D Command Map

All the following commands are defined in *Blitter Instructions*.

Opcode (28:22)	Command	Comments
00h	Reserved	
01h	XY_SETUP_BLT	
02h	Reserved	
03h	XY_SETUP_CLIP_BLT	
04h–10h	Reserved	
11h	XY_SETUP_MONO_PATTERN_SL_BLT	
12h–23h	Reserved	
24h	XY_PIXEL_BLT	
25h	XY_SCANLINES_BLT	
26h	XY_TEXT_BLT	
27h–30h	Reserved	
31h	XY_TEXT_IMMEDIATE_BLT	
32h–3Fh	Reserved	
40h	COLOR_BLT	
41h–42h	Reserved	
43h	SRC_COPY_BLT	
44h–4Fh	Reserved	
50h	XY_COLOR_BLT	
51h	XY_PAT_BLT	
52h	XY_MONO_PAT_BLT	
53h	XY_SRC_COPY_BLT	
54h	XY_MONO_SRC_COPY_BLT	
55h	XY_FULL_BLT	
56h	XY_FULL_MONO_SRC_BLT	
57h	XY_FULL_MONO_PATTERN_BLT	
58h	XY_FULL_MONO_PATTERN_MONO_SRC_BLT	
59h	XY_MONO_PAT_FIXED_BLT	
5Ah–70h	Reserved	
71h	XY_MONO_SRC_COPY_IMMEDIATE_BLT	
72h	XY_PAT_BLT_IMMEDIATE	
73h	XY_SRC_COPY_CHROMA_BLT	
74h	XY_FULL_IMMEDIATE_PATTERN_BLT	
75h	XY_FULL_MONO_SRC_IMMEDIATE_PATTERN_BLT	
76h	XY_PAT_CHROMA_BLT	



Opcode (28:22)	Command	Comments
77h	XY_PAT_CHROMA_BLT_IMMEDIATE	
78h–7Fh	Reserved	

4.2.3 3D / Media Command Map

Pipeline Type (28:27)	Opcode	Sub Opcode	Command	Definition Chapter
Common (pipelined)	Bits 26:24	Bits 23:16		
0h	0h	00h	URB_FENCE	Graphics Processing Engine
0h	0h	01h	CS_URB_STATE	Graphics Processing Engine
0h	0h	02h	CONSTANT_BUFFER	Graphics Processing Engine
0h	0h	03h	STATE_PREFETCH	Graphics Processing Engine
0h	0h	04h-FFh	Reserved	
Common (non-pipelined)	Bits 26:24	Bits 23:16		
0h	1h	00h	Reserved	n/a
0h	1h	01h	STATE_BASE_ADDRESS	Graphics Processing Engine
0h	1h	02h	STATE_SIP	Graphics Processing Engine
0h	1h	03h-FFh	Reserved	n/a
Reserved	Bits 26:24	Bits 23:16		
0h	2h-7h	XX	Reserved	n/a



Pipeline Type (28:27)	Opcode	Sub Opcode	Command	Definition Chapter
Single DW	Opcode (26:24)	Bits 23:16		
1h	0h	00h-01h	Reserved	n/a
1h	0h	02h	Reserved	n/a
1h	0h	03h-0Ah	Reserved	n/a
1h	0h	0Bh	Reserved	n/a
1h	0h	0Ch-FFh	Reserved	n/a
1h	1h	00h-03h	Reserved	n/a
1h	1h	04h	PIPELINE_SELECT	Graphics Processing Engine
1h	1h	05h-FFh	Reserved	n/a
1h	2h-7h	XX	Reserved	n/a
Media	Opcode (26:24)	Bits 23:16		
2h	0h	00h	MEDIA_STATE_POINTERS	Media
2h	1h	00h	MEDIA_OBJECT	Media
2h	1h	01h	MEDIA_OBJECT_EX	Media
2h	1h	02h	MEDIA_OBJECT_PRT	Media
2h	2h-7h	XX	Reserved	n/a

Pipeline Type (28:27)	Opcode	Sub Opcode	Command	Definition Chapter
3D State (Pipelined)	Bits 26:24	Bits 23:16		
3h	0h	00h	3DSTATE_PIPELINED_POINTERS	3D Pipeline
3h	0h	01h	3DSTATE_BINDING_TABLE_POINTERS	3D Pipeline
3h	0h	02h	Reserved	
3h	0h	03h-04h	Reserved	n/a
3h	0h	05h	3DSTATE_URB	3D Pipeline
3h	0h	06h-07h	Reserved	n/a
3h	0h	08h	3DSTATE_VERTEX_BUFFERS	Vertex Fetch
3h	0h	09h	3DSTATE_VERTEX_ELEMENTS	Vertex Fetch



Pipeline Type (28:27)	Opcode	Sub Opcode	Command	Definition Chapter
3h	0h	0Ah	3DSTATE_INDEX_BUFFER	Vertex Fetch
3h	0h	0Bh	Reserved	n/a
3h	0h	0Ch	Reserved	n/a
3h	0h	0Dh	3DSTATE_VIEWPORT_STATE_POINTERS	3D Pipeline
3h	0h	0Eh–FFh	Reserved	n/a
3D State (Non-Pipelined)	Bits 26:24	Bits 23:16		
3h	1h	00h	3DSTATE_DRAWING_RECTANGLE	Strips & Fans
3h	1h	01h	3DSTATE_CONSTANT_COLOR	Color Calculator
3h	1h	02h	3DSTATE_SAMPLER_PALETTE_LOAD0	Sampling Engine
3h	1h	03h	Reserved	
3h	1h	04h	3DSTATE_CHROMA_KEY	Sampling Engine
3h	1h	05h	3DSTATE_DEPTH_BUFFER	Windower
3h	1h	06h	3DSTATE_POLY_STIPPLE_OFFSET	Windower
3h	1h	07h	3DSTATE_POLY_STIPPLE_PATTERN	Windower
3h	1h	08h	3DSTATE_LINE_STIPPLE	Windower
3h	1h	09h	3DSTATE_GLOBAL_DEPTH_OFFSET_CLAMP	Windower
3h	1h	0Ah–FFh	Reserved	Windower
3D (Control)	Bits 26:24	Bits 23:16		
3h	2h	00h	PIPE_CONTROL	3D Pipeline
3h	2h	01h–FFh	Reserved	n/a
3D (Primitive)	Bits 26:24	Bits 23:16		
3h	3h	00h	3DPRIMITIVE	Vertex Fetch
3h	3h	01h–FFh	Reserved	n/a
3h	4h–7h	00h–FFh	Reserved	n/a



5 *Register Address Maps*

5.1 Graphics Register Address Map

This chapter provides address maps of the graphics controllers I/O and memory-mapped registers. Individual register bit field descriptions are provided in the following chapters. PCI configuration address maps and register bit descriptions are provided in the following chapter.

5.1.1 Memory and I/O Space Registers

This section provides a high-level register map (register groupings per function). The memory and I/O maps for the graphics device registers are shown in the following table, except PCI Configuration registers that are described in the following chapter.

Note: The VGA and Extended VGA registers can be accessed via standard VGA I/O locations as well as via memory-mapped locations.

Note: All graphics MMIO registers can also be accessed via CPU I/O.

The memory space address listed for each register is an offset from the base memory address programmed into the MMADR register (PCI configuration offset 14h).



Table 5-1. Graphics Controller Register Memory and I/O Map

Start Offset	End Offset	Description
00000h	00FFFh	VGA and Extended VGA Control Registers. These registers are located in both I/O space and memory space. The VGA and Extended VGA registers contain the following register sets: General Control/Status, Sequencer (SRxx), Graphics Controller (GRxx), Attribute Controller (ARxx), VGA Color Palette, and CRT Controller (CRxx) registers. Detailed bit descriptions are provided in the <i>VGA and Extended VGA Register</i> Chapter. The registers within a set are accessed using an indirect addressing mechanism as described at the beginning of each section. Note that some of the register description sections have additional operational information at the beginning of the section
01000h	01FFFh	Reserved
02000h	02FFFh	Instruction, Memory, and Interrupt Control Registers: Instruction Control Registers Ring Buffer registers and page table control registers are located in this address range. Various instruction status, error, and operating registers are located in this group of registers. Graphics Memory Fence Registers. The Graphics Memory Fence registers are used for memory tiling capabilities. Interrupt Control/Status Registers. This register set provides interrupt control/status for various GC functions. Display Interface Control Register. This register controls the FIFO watermark and provides burst length control. Logical Context Registers Software Visible Counters
03000h	031FFh	FENCE & Per Process GTT Control registers
03200h	03FFFh	Frame Buffer Compression Registers
04000h	043FFh	Reserved.
04400h	04FFFh	Reserved.
05000h	05FFFh	I/O Control Registers
06000h	06FFFh	Clock Control Registers. This memory address space is the location of the GC clock control and power management registers
07000h	073FFh	3D Internal Debug Registers
07400h	088FFh	GPE Debug Registers (3D / Media Fixed Functions)
08900h	08FFFh	Reserved for Subsystem Debug Registers
09000h	09FFFh	Reserved
0A000h	0AFFFh	Display Palette Registers
0B000h	0FFFFh	Reserved
10000h	13FFFh	MMIO MCHBAR. Alias through which the graphics driver can access registers in the MCHBAR accessed through device 0.
14000h	2FFFFh	Reserved



Start Offset	End Offset	Description
30000h	3FFFFh	Overlay Registers. These registers provide control of the overlay engine. The overlay registers are double-buffered with one register buffer located in graphics memory and the other on the device. On-chip registers are not directly writeable. To update the on-chip registers software writes to the register buffer area in graphics memory and instructs the device to update the on-chip registers.
40000h	5FFFFh	Reserved
60000h	6FFFFh	Display Engine Pipeline Registers
70000h	72FFFh	Display and Cursor Registers
73000h	73FFFh	Performance Counters
74000h	7FFFFh	Reserved

5.1.2 PCI Configuration Space

See the relevant EDS for details on accessing PCI configuration space, PCI address map tables, and register descriptions.



5.1.3 Graphics Register Memory Address Map

All graphics device registers are directly accessible via memory-mapped I/O and indirectly accessible via the MMIO_INDEX and MMIO_DATA I/O registers. In addition, the VGA and Extended VGA registers are I/O mapped.

Table 5-2. Memory-Mapped Registers

Address Offset	Symbol	Register Name	Access
00000h–00FFFh	—	VGA and VGA Extended Registers These registers are both memory and I/O mapped and are listed in the following table. Note that the I/O address and memory offset address are the same value for each register.	—
Reserved (1000h–1FFFh)			
01000h–01FFFh	—	Reserved	—
Primary CS Instruction and Interrupt Control Registers (02000h–02FFFh)			
02000h–0201Fh	—	Reserved	—
02020h–02023h	PGTBL_CTL	Page Table Control Register	R/W
02024h–02027h	PGTBL_ER	Page Table Error Register (<i>DEBUG</i>)	RO
02028h–0202Bh	EXCC	Execute Condition Code Register	R/W,RO
0202Ch–0202Fh	—	Reserved	—
02030h–02033h	PRB0_TAIL	Primary Ring Buffer 0 Tail Register	R/W
02034h–02037h	PRB0_HEAD	Primary Ring Buffer 0 Head Register	R/W
02038h–0203Bh	PRB0_STARTsted	Primary Ring Buffer 0 Start Register	R/W
0203Ch–0203Fh	PRB0_CTL	Primary Ring Buffer 0 Control Register	R/W
02040h–0205Fh	—	Reserved	—
02060h–02063h	HW_MEMRD	Memory Read Sync Register (<i>DEBUG</i>)	RO
02064h–02067h	IPEIR	Instruction Parser Error Identification Register (<i>DEBUG</i>)	RO
02068h–0206Bh	IPEHR	Instruction Parser Error Header Register (<i>DEBUG</i>)	RO
0206Ch–0206Fh	INSTDONE	Instruction Stream Interface Done Register (<i>DEBUG</i>)	RO
02070h–02073h	INSTPS	Instruction Parser State Register (<i>DEBUG</i>)	RO
02074h–02077h	ACTHD	Active Head Pointer Register (<i>DEBUG</i>)	RO
02078h–0207Bh	DMA_FADD_P	Primary DMA Engine Fetch Address Register (<i>DEBUG</i>)	RO
0207Ch–0207Fh	INSTDONE_1	Instruction Stream Interface Done 1 (Debug)	RO



Table 5-2. Memory-Mapped Registers

Address Offset	Symbol	Register Name	Access
02080h–02083h	HWS_PGA	Hardware Status Page Address Register	R/W
02084h–02087h	—	Reserved	—
02088h–0208Ch	PWRCTXA	Power Context Register Address ([DevCL])	R/W
0208Dh–02093h	—	Reserved	—
02094h–02097h	NOPID	NOP Identification Register	RO
02098h–0209Bh	HWSTAM	Hardware Status Mask Register	R/W
0209Ch–0209Fh	MI_MODE	Mode Register for Software Interface	R/W
020A0h–020A3h	IER	Interrupt Enable Register	R/W
020A4h–020A7h	IIR	Interrupt Identity Register	R/WC
020A8h–020ABh	IMR	Interrupt Mask Register	R/W
020ACh–020AFh	ISR	Interrupt Status Register	RO
020B0h–020B3h	EIR	Error Identity Register	R/WC
020B4h–020B7h	EMR	Error Mask Register	R/W
020B8h–020BBh	ESR	Error Status Register	RO
020BCh–020BFh	—	Reserved	—
020C0h–020C3h	INSTPM	Instruction Parser Mode Register (SAVED/RESTORED)	R/W
020C4h–020C7h	PGTBL_CTL2	Per-process Page Table Control 0	R/W
020C8h–020CBh	PGTBL_STR2	Page Table Steer Register (Per Process)	R/W
020CCh–020DFh	—	Reserved	—
020E0h–020E3h	MI_DISPLAY_POWER_DOWN	Display Power Down Enable ([DevCL] Only)	R/W
	MI_RDRET_STATE	Memory Interface Read Return State Register ([DevBW] Only)	R/W
020E4h–020E7h	MI_ARB_STATE	Memory Interface Arbitration State Register (SAVED/RESTORED)	R/W
020E8h–020FBh	—	Reserved	—
020FCh–020FFh	MI_RDRET_STATE	Memory Interface Read Return State Register ([DevCL] Only)	R/W
02100h–0211Fh	—	Reserved	—
02120h–02123h	CACHE_MODE_0	Cache Mode Register 0 (DEBUG) (SAVED/RESTORED)	R/W
02124h–02127h	CACHE_MODE_1	Cache Mode Register 1 (DEBUG) (SAVED/RESTORED)	R/W



Table 5-2. Memory-Mapped Registers

Address Offset	Symbol	Register Name	Access
02128h–02133h	—	Reserved	—
02134h–02137h	UHPTR	Pending Head Pointer Register	R/W
02138h–0213Fh	—	Reserved	—
02140h–02147h	BB_ADDR	Batch Buffer Current Address	RO
02148h–0214Bh	BB_STATE	Batch Buffer State Register	R/W
0214Ch–0216Fh	—	Reserved	—
02170h–02177h	GFX_FLSH_CNTL	Graphics Flush Control	R/W
02178h–0217Bh	—	Reserved	—
0217Ch–0217Fh	PR_CTR_THRSH	Reserved	—
02180h–02183h	CCID0	Current Context ID 0 (assoc w/ PRB0)	R/W
02184h–0218Fh	—	Reserved	—
02190h–02193h	—	Reserved	—
02194h–0219Fh	—	Reserved	—
021A0h–021A3h	CXT_SIZE	Context Size (<i>DEBUG</i>)	R/W
021A4h–021A7h	CXT_SIZE_NOEXT	Context Size without Ext. State (<i>DEBUG</i>)	R/W
021A8h–021CFh	—	Reserved	—
021D0h–021D3h	ECOSKPD	ECO Scratch Pad (<i>DEBUG</i>)	R/W
021D4h–021FFh	—	Reserved	—
02200h–02303h	CSFLFSM	Flush FSM (Debug)	R/W
02204h–02207h	CSFLFLAG	Flush FLAG (Debug)	R/W
02208h–0220Bh	CSFLTRK	Flush Track (Debug)	R/W
0220Ch–0220Fh	CSCMDOP	Instruction DWORD (Debug)	R/W
02210h–02213h	CSCMDVLD	Instruction DWORD Valid (Debug)	R/W
02214h–0230Fh	—	Reserved	—
02310h–0234Fh	—	Reserved	—
02350h–02357h	PS_DEPTH_COUNT	Reported Pixels Passing Depth Test Counter	R/W
02358–0235Fh	TIMESTAMP	Reported Timestamp Count	R/W
02360–02367h	CLKCMP	Compare Count Clock Stop (Debug)	—
02368h–0236Fh	—	Reserved	—
02370h–02377h	—	Reserved	—
02378h–0237Fh	—	Reserved	—



Table 5-2. Memory-Mapped Registers

Address Offset	Symbol	Register Name	Access
02380h–02387h	—	Reserved	—
02388h–0244Fh	—	Reserved	—
02450h–02453h	VFDC	Set Value of Draw Count (<i>DEBUG</i>)	R/W
02454h–0246Fh	—	Reserved	—
02470h–02473h	VFSKPD	VF Scratch Pad (<i>DEBUG</i>)	R/W
02474h–024FFh	—	Reserved	—
Per-Process GTT Control (02500h–025FFh)			
02500h–0251Fh	—	Reserved	—
02520h–02520	GFX_MODE	Graphics Mode	R/W
Probe List Control (02600h–026FFh) : Reserved			
02600h–026FFh	—	Reserved	—
Run List Control (02700h–027FFh) : Reserved			
02700h–02FFFh	—	Reserved	—
FENCE & Per-Process GTT Control (03000h–031FFh)			
03000h–03007h	FENCE[0]	Graphics Memory Fence Table Register [0]	R/W
...	...	...	
0307Ch–0307Fh	FENCE[15]	Graphics Memory Fence Table Register [15]	R/W
Frame Buffer Compression Control (03200h–03FFFh) ([DevCL] Only)			
03200h–03203h	FBC_CFB_BASE	Compressed Frame Buffer Base Address	R/W
03204h–03207h	FBC_LL_BASE	Compressed Frame Line Length Buffer Address	R/W
03208h–0320Bh	FBC_CONTROL	Frame Buffer Compression Control Register	R/W
0320Ch–0320Fh	FBC_COMMAND	Frame Buffer Compression Command Register	R/W
03210h–03213h	FBC_STATUS	Frame Buffer Compression Status Register	R/W
03214h–03217h	FBC_CONTROL2	Frame Buffer Compression 2 nd Control Register	R/W
0321Bh–0321Eh	FBC_DISPYOFF	Frame Buffer Compression Display Y Offset	R/W
03220h–03223h	FBC_MOD_NUM	Frame Buffer Compression Num of Modifications	R/W
03214h–032FFh	—	Reserved	—
03300h–033C3h	FBC_TAG	Frame Buffer Compression Tag Interface (Debug)	R/W
03400h–03FFFh	—	Reserved	—
Frame Buffer Compression Control (03200h–03FFFh) : Reserved			
03200h–03FFFh	DPFC_CB_BASE	DPFC Compressed Buffer Base Address	R/W



Table 5-2. Memory-Mapped Registers

Address Offset	Symbol	Register Name	Access
BCS Instruction and Interrupt Control Registers (04000h–043FFh)			
04000h–043FFh	—	Reserved	—
04064h–04067h	BCS_IPEIR	Instruction Parser Error Identification Register (Debug)	RO
04068h–0406Bh	BCS_IPEHR	Instruction Parser Error Header Register (Debug)	RO
04074h–04077h	BCS_ACTHD	Active Head Pointer Register (Debug)	RO
04078h – 0407Bh	BCS_DMA_FADD	DMA Engine Fetch Address (Debug)	RO
04080h–04083h	BCS_HWS_PGA	Hardware Status Page Address Register	R/W
04084h–04093h	—	Reserved	—
04094h–04097h	BCS_NOPID	NOP Identification Register	RO
04097h–0409B	—	Reserved	—
0409Ch–0409Fh	BCS_MI_MODE	Mode Register for Software Interface	R/W
040A0h–040BFh	—	Reserved	—
040C0h–040C3h	BCS_INSTPM	Instruction Parser Mode Register	R/W
040C4h–04133h	—	Reserved	—
04134h–04137h	BCS_UHPTR	Pending Head Pointer	R/W
04138h–04177h	—	Reserved	—
04178h–0417Bh	BCS_CNTR	Counter for the Bit Stream Decode Engine	R/W
0417Ch–0417Fh	BCS_THRSH	Threshold for the Counter of Bit Stream Decode Engine	R/W
04180h–0413Fh	—	Reserved	—
04140h–04147h	BCS_BB_ADDR	Batch Buffer Head Pointer Register	RO
04148h–0418Fh	—	Reserved	—
04190h–04193h	BCS_RCCID	Ring Buffer Current Context ID	R/W
04194h–04197h	BCS_RNCID	Ring Buffer Next Context ID	R/W
04198h–043FFh	—	Reserved	—
(04400h–044FFh) : Reserved			
04400h–044FFh	—	Reserved	—



Table 5-2. Memory-Mapped Registers

Address Offset	Symbol	Register Name	Access
MFC Status Registers (04500h–04FFFh) : Reserved			
04500h–04FFFh	—	Reserved	—
I/O Control Registers (05000h–05FFFh)			
05000h–0500Fh	—	Reserved	—
05010h–05013h	GPIO_CTL0	General Purpose I/O Control Register [0]	R/W
05014h–05017h	GPIO_CTL1	General Purpose I/O Control Register [1]	R/W
05018h–0501Bh	GPIO_CTL2	General Purpose I/O Control Register [2]	R/W
0501Ch–0501Fh	GPIO_CTL3	General Purpose I/O Control Register [3]	R/W
05020h–05023h	GPIO_CTL4	General Purpose I/O Control Register [4]	R/W
05024h–05027h	GPIO_CTL5	General Purpose I/O Control Register [5]	R/W
05028h–0502Bh	GPIO_CTL6	General Purpose I/O Control Register [6]	R/W
0502Ch–0502Fh	GPIO_CTL7	General Purpose I/O Control Register [7]	R/W
05030h–050FFh	—	Reserved	—
05100h–05103h	GMBUS0	GMBUS Clock Select/Device Select	R/W
05104h–05107h	GMBUS1	GMBUS Command/Status	R/W
05108h–0510Bh	GMBUS2	GMBUS Status	R/W
0510Ch–0510Fh	GMBUS3	GMBUS Data Buffer	R/W
05110h–05113h	GMBUS4	GMBUS Interrupt Mask	R/W
05114h–0511Fh	—	Reserved	—
05120h–05123h	GMBUS5	GMBUS 2-Byte Index Register	R/W
05124h–05FFFh	—	Reserved	—
VSC Registers (05000h – 05FFFh) : Reserved			
05000h–0506Fh	—	Reserved	—
VSC Registers : Reserved			
05070h–05083h		Reserved	
Clock Control and Power Management Registers (06000h–06FFFh)			
06000h–06003h	VGA0	VGA 0 Divisor	R/W
06004h–06007h	VGA1	VGA 1 Divisor	R/W
06008h–0600Fh		Reserved	
06010h–06013h	VGA_PD	VGA Post Divisor Select	R/W
06014h–06017h	DPLLA_CTRL	Display PLL A Control	R/W



Table 5-2. Memory-Mapped Registers

Address Offset	Symbol	Register Name	Access
06018h–0601Bh	DPLLB_CTRL	Display PLL B Control	R/W
0601Ch–0601Fh	DPLLAMD	Display PLL A SDVO/UDI Multiplier/Divisor	R/W
06020h–06023h	DPLLBMD	Display PLL B SDVO/UDI Multiplier/Divisor	R/W
06024h–0603Fh	—	Reserved	—
06040h–06043h	FPA0	DPLL A Divisor 0	R/W
06044h–06047h	FPA1	DPLL A Divisor 1	R/W
06048h–0604Bh	FPB0	DPLL B Divisor 0	R/W
0604Ch–0604Fh	FPB1	DPLL B Divisor 1	R/W
06050h–0606Bh	—	Reserved	—
0606Ch–0606Fh	DPLL_TEST	DPLLA and DPLLB Test Register	R/W
06070h–06103h	—	Reserved	—
06104h–06107h	D_STATE	D State Function Control	R/W
06108h–061FFh	—	Reserved	—
06200h–06203h	DSPCLK_GATE_D	Clock Gating Disable for Display Register	R/W
06204h–06207h	RENCLK_GATE_D1	Clock Gating Disable for Render Register I	R/W
06208h–0620Bh	RENDCLK_GATE_D2	Clock Gating Disable for Render Register II	—
0620Ch–0620Fh	—	Reserved	—
06210h–06213h	RAMCLK_GATE_D	GFX RAM Clock Gating Disable Register ([DevCL] Only)	R/W
06214h–06125h	DEUC	Dynamic EU Control	R/W/L
06216h–06FFFh	—	Reserved	—
3D-Internal Debug Registers (07000h–073FFh) Reserved			
07000h–073FFh	—	Reserved	—
GPE Debug Registers (07400h–088FFh, DEBUG ONLY, Subject to Change)			
07400h–07403h	SVG_CTL	Debug Control	R/W
07404h–07407h	SVG_RDATA	Debug Return Data	RO
07408h–0740Bh	SVG_WORK_CTL	Debug Workaround Control	R/W
0740Ch–074FFh	—	Reserved	—
07500h–07503h	VF_CTL	Debug Control	R/W
07504h–07507h	VF_STRG_VAL	Debug Snapshot Trigger Value	R/W
07508h–0750Bh	VF_STR_VL_OVR	Debug Start Vertex Location Override	R/W
0750Ch–0750Fh	VF_VC_OVR	Debug Vertex Count Override	R/W



Table 5-2. Memory-Mapped Registers

Address Offset	Symbol	Register Name	Access
07510h–07513h	VF_STR_PSKIP	Debug Starting Primitives Skipped	RO
07514h–07517h	VF_MAX_PRIM	Debug Max Primitives	R/W
07518h–0751Bh	VF_RDATA	Debug Return Data	RO
0751Ch–075FFh	—	Reserved	—
07600h–07603h	VS_CTL	Debug Control	R/W
07604h–07607h	VS_STRG_VAL	Debug Snapshot Trigger Value	R/W
07608h–0760Bh	VS_RDATA	Debug Return Data	RO
0760Ch–078FFh	—	Reserved	—
07900h–07903h	GS_CTL	Debug Control	R/W
07904h–07907h	GS_STRG_VAL	Debug Snapshot Trigger Value	R/W
07908h–0790Bh	GS_RDATA	Debug Return Data	RO
0790Ch–079FFh	—	Reserved	—
07A00h–07A03h	CL_CTL	Debug Control	R/W
07A04h–07A07h	CL_STRG_VAL	Debug Snapshot Trigger Value	R/W
07A08h–07A0Bh	CL_RDATA	Debug Return Data	RO
07A0Ch–07AFFh	—	Reserved	—
07B00h–07B03h	SF_CTL	Debug Control	R/W
07B04h–07B07h	SF_STRG_VAL	Debug Snapshot Trigger Value	R/W
07B08h–07B0Bh	SF_MIN_PR_IND	Debug Minimum Primitive Index	R/W
07B0Ch–07B0Fh	SF_MAX_PR_IND	Debug Maximum Primitive Index	R/W
07B10h–07B13h	SF_CLIP_RMIN	Debug Clip Rectangle Minimum Coordinates	R/W
07B14h–07B17h	SF_CLIP_RMAX	Debug Clip Rectangle Maximum Coordinates	R/W
07B18h–07B1Bh	SF_RDATA	Debug Return Data	RO
07B1Ch–07BFFh	—	Reserved	—
07C00h–07C03h	WIZ_CTL	Debug Control	R/W
07C04h–07C07h	WIZ_STRG_VAL	Debug Snapshot Trigger Value	R/W
07C08h–07C0Bh	WIZ_RDATA	Debug Return Data	RO
07C0Ch–07CFFh	—	Reserved	—
07D00h–07D03h	VFE_CTL	Debug Control	R/W
07D04h–07D07h	VFE_STRG_VAL	Debug Snapshot Trigger Value	R/W
07D08h–07D0Bh	VFE_RDATA	Debug Return Data	RO
07D0Ch–07DFFh	—	Reserved	—



Table 5-2. Memory-Mapped Registers

Address Offset	Symbol	Register Name	Access
07E00h–07E03h	TS_CTL	Debug Control	R/W
07E04h–07E07h	TS_STRG_0-6VAL	Debug Snapshot Trigger R0.6 Value	R/W
07E08h–07E0Bh	TS_STRG_0-7VAL	Debug Snapshot Trigger R0.7 Value	R/W
07E0Ch–07E0Fh	TS_RDATA	Debug Return Data	RO
07E10h–07FFFh	—	Reserved	—
08000h–08003h	TD_CTL	Debug Control	R/W
08004h–08007h	TD_CTL2	Debug Control 2	R/W
08008h–0800Bh	TD_VF_VS_EMSK	Debug VF/VS Execution Mask	R/W
0800Ch–0800Fh	TD_GS_EMSK	Debug GS Execution Mask	R/W
08010h–08013h	TD_CLIP_EMSK	Debug Clipper Execution Mask	R/W
08014h–08017h	TD_SF_EMSK	Debug SF Execution Mask	R/W
08018h–0801Bh	TD_WIZ_EMSK	Debug WIZ Execution Mask	R/W
0801Ch–0801Fh	TD_0-6_EHTRG_VAL	Debug R0.6 External Halt Trigger Value	R/W
08020h–08023h	TD_0-7_EHTRG_VAL	Debug R0.7 External Halt Trigger Value	R/W
08024h–08027h	TD_0-6_EHTRG_MSK	Debug R0.6 External Halt Trigger Mask	R/W
08028h–0802Bh	TD_0-7_EHTRG_MSK	Debug R0.7 External Halt Trigger Mask	R/W
0802Ch–0802Fh	TD_RDATA	Debug Return Data	RO
08030h–08033h	TD_TS_EMSK	Debug TS Execution Mask	—
08034h–080FFh	—	Reserved	—
08100h–08103h	MATH_CTL	Math Debug Control	R/W
08104h–08107h	MATH_RDATA	Math Debug Return Data	RO
08108h–081FFh	—	Reserved	—
08200h–08203h	ISC_CTL	Instruction / State Debug Control	R/W
08204h–0827FFh	—	Reserved	—
08280h–08283h	ISC_L1CA_CTR	Instruction L1 Cache Debug Control	RO
08284h–08287h	ISC_L1CA_RDATA	Instruction L1 Cache Debug Return Data	
08288h–0828Bh	ISC_L1CA_BP_ADR1	Instruction L1 Cache Breakpoint Address 1 Control	
0828Ch–0828Fh	—	Reserved	—
08290h–08293h	ISC_L1CA_BP_ADR2	Instruction L1 Cache Breakpoint Address 2 Control	
08294h–08297h	ISC_L1CA_BP_OPC1	Instruction L1 Cache Breakpoint Opcode 1 Control	
08298h–0829Bh	ISC_L1CA_BP_OPC2	Instruction L1 Cache Breakpoint Opcode 2 Control	
0829Ch–082FFh	—	Reserved	—



Table 5-2. Memory-Mapped Registers

Address Offset	Symbol	Register Name	Access
08300h–08303h	MA_DEBUG_1	Message Arbiter Debug Control	R/W
08304h–083FFh	—	Reserved	—
08400h–08403h	SAMPLER_CTL	Sampler Debug Control	R/W
08404h–08407h	SAMPLER_RDATA	Sampler Debug Return Data	RO
08408h–084FFh	—	Reserved	—
08500h–08503h	DP_CTL	Data Port Debug Control	R/W
08504h–08507h	DP_RDATA	Data Port Debug Return Data	RO
08508h–085FFh	—	Reserved	—
08600h–08603h	RC_CTL	Debug Control	R/W
08604h–08607h	RC_DEF_CLR	Debug Force Default Color	R/W
08608h–0860Bh	RC_RDATA	Debug Return Data	RO
0860Ch–086FFh	—	Reserved	—
08700h–08703h	URB_CTL	Debug Control	R/W
08704h–08707h	—	Reserved	—
08708h–0870Bh	URB_RDATA	Debug Return Data	RO
0870Ch–087FFh	—	Reserved	—
08800h–08803h	EU_CTL	Debug Control	R/W
08804h–0880Fh	—	Reserved	—
08810h–08817h	EU_ATT	Debug Attention	RO
08818h–0881Fh	—	Reserved	—
08820h–08827h	EU_ATT_DATA	EU Debug Attention Data	RO
08828h–0882Fh	—	Reserved	—
08830h–08837h	EU_ATT_CLR	Debug Attention Clear	WO
08838h–0883Fh	—	Reserved	—
08840h–08843h	EU_RDATA	Debug Return Data	RO
08844h–088FFh	—	Reserved	—
Reserved for Debug (08900h–09FFFh)			
08900h–08FFFh	—	Reserved for Subsystem Debug	—
09000h–09FFFh	—	Reserved	—
Display Palette (0A000h–0AFFh)			
0A000h–0A3FFh	DPALETTE_A	Display Pipe A Palette	R/W
0A400h–0A7FFh	—	Reserved	—



Table 5-2. Memory-Mapped Registers

Address Offset	Symbol	Register Name	Access
0A800h–0ABFFh	DPALETTE_B	Display Pipe B Palette	R/W
0AC00h–0AFFh	—	Reserved	—
TLB Read Range (0B000h–0BFFFh) : Reserved			
0B000h–0BFFFh	—	Reserved	—
AVC Video Decode (0C000h–0CFFFh) : Reserved			
0C000h–0CFFFh	--	Reserved	--
0D000h–0FFFFh	—	Reserved	—
GFX MMIO – MCHBAR Aperture (10000h–13FFFh)			
10000h–13FFFh	—	MCHBAR Aperture	R/W
Reserved (14000h–2FFFFh)			
14000h–2FFFFh	—	Reserved	—
Overlay Registers (30000h–03FFFFh) (For additional address offsets in the double-buffering scheme, see Overlay Chapter)			
30000h–30003h	OVADD	Overlay Register Update Address	R/W
30004h–30007h	OTEST	Overlay Test Register	R/W
30008h–3000Bh	DOVSTA	Display/Overlay Status	RO
3000Ch–3000Fh	DOVSTAEX	Display/Overlay Extended Status	RO
30010h–30013h	OVR_GAMMA5	Overlay Gamma Correction [5]	R/W
30014h–30017h	OVR_GAMMA4	Overlay Gamma Correction [4]	R/W
30018h–3001Bh	OVR_GAMMA3	Overlay Gamma Correction [3]	R/W
3001Ch–3001Fh	OVR_GAMMA2	Overlay Gamma Correction [2]	R/W
30020h–30023h	OVR_GAMMA1	Overlay Gamma Correction [1]	R/W
30024h–30027h	OVR_GAMMA0	Overlay Gamma Correction [0]	R/W
30028h–30057h	—	Reserved	—
30058h–3005Bh	SYNCPH0	Overlay Flip Sync Lock Phase 0	RO
3005Ch–3005Fh	SYNCPH1	Overlay Flip Sync Lock Phase 1	RO
30060h–30063h	SYNCPH2	Overlay Flip Sync Lock Phase 2	RO
30064h–30067h	SYNCPH3	Overlay Flip Sync Lock Phase 3	RO
30068h–300FFh	—	Reserved	—
30100h–30103	OBUF_0Y	Overlay Buffer 0 Y Pointer	RO
30104h–30107h	OBUF_1Y	Overlay Buffer 1 Y Pointer	RO
30108h–3010Bh	OBUF_0U	Overlay Buffer 0 U Pointer	RO



Table 5-2. Memory-Mapped Registers

Address Offset	Symbol	Register Name	Access
3010Ch–3010Fh	OBUF_0V	Overlay Buffer 0 V Pointer	RO
30110h–30113h	OBUF_1U	Overlay Buffer 1 U Pointer	RO
30114h–30117h	OBUF_1V	Overlay Buffer 1 V Pointer	RO
30118h–3011Bh	OSTRIDE	Overlay Stride	RO
3011Ch–3011Fh	YRGB_VPH	Y/RGB Vertical Phase	RO
30120h–30123h	UV_VPH	UV Vertical Phase	RO
30124h–30127h	HORZ_PH	Horizontal Phase	RO
30128h–3012Bh	INIT_PHS	Initial Phase	RO
3012Ch–3012Fh	DWINPOS	Destination Window Position	RO
30130h–30133h	DWINSZ	Destination Window Size	RO
30134h–30137h	SWIDTH	Source Width	RO
30138h–3013Bh	SWIDTHSW	Source Width in Swords	RO
3013Ch–3013Fh	SHEIGHT	Source Height	RO
30140h–30143h	YRGBSCALE	Y/RGB Scale Factor	RO
30144h–30147h	UVSCALE	U V Scale Factor	RO
30148h–3014Bh	OVCLRC0	Overlay Color Correction 0	RO
3014Ch–3014Fh	OVCLRC1	Overlay Color Correction 1	RO
30150h–30153h	DCLRKV	Destination Color Key Value	RO
30154h–30157h	DCLRKM	Destination Color Key Mask	RO
30158h–3015Bh	SCHRKVH	Source Chroma Key Value High	RO
3015Ch–3015Fh	SCHRKVL	Source Chroma Key Value Low	RO
30160h–30163h	SCHRKEN	Source Chroma Key Enable	RO
30164h–30167h	OCONFIG	Overlay Configuration	RO
30168h–3016Bh	OCMD	Overlay Command	RO
3016Ch–3016Fh	—	Reserved	—
30170h–30173h	OSTART_0Y	Overlay Surface Y 0 Base Address Register	RO
30174h–30177h	OSTART_1Y	Overlay Surface Y 1 Base Address Register	RO
30178h–3017Bh	OSTART_0U	Overlay Surface U 0 Base Address Register	RO
3017Ch–3017Fh	OSTART_0V	Overlay Surface V 0 Base Address Register	RO
30180h–30183h	OSTART_1U	Overlay Surface U 1 Base Address Register	RO
30184h–30187h	OSTART_1V	Overlay Surface V 1 Base Address Register	RO
30188h–3018Bh	OTILEOFF_0Y	Overlay Surface Y 0 Base Address Register	RO



Table 5-2. Memory-Mapped Registers

Address Offset	Symbol	Register Name	Access
3018Ch–3018Fh	OTILEOFF _1Y	Overlay Surface Y 1 Base Address Register	RO
30190h–30193h	OTILEOFF _0U	Overlay Surface U 0 Base Address Register	RO
30194h–30197h	OTILEOFF _0V	Overlay Surface V 0 Base Address Register	RO
30198h–3019Bh	OTILEOFF _1U	Overlay Surface U 1 Base Address Register	RO
3019Ch–3019Fh	OTILEOFF _1V	Overlay Surface V 1 Base Address Register	RO
301A0h–301A3h	—	Reserved	—
301A4h–301A7h	UVSCALEV	UV Vertical Downscale Integer Register	RO
301A8h–302FFh	—	Reserved	—
30300h–303FFh	Y_VCOEFS	Overlay Y Vertical Filter Coefficients	RO
30368h–303FFh	—	Reserved	—
30400h–305FFh	Y_HCOEFS	Overlay Y Horizontal Filter Coefficient	RO
304ACh–305FFh	—	Reserved	—
30600h–306FFh	UV_VCOEFS	Overlay UV Vertical Filter Coefficients	RO
30668h–306FFh	—	Reserved	—
30700h–307FFh	UV_HCOEFS	Overlay UV Horizontal Filter Coefficients	RO
30768h–3FFFFh	—	Reserved	—
Reserved (40000h–5FFFFh)			
40000h–5FFFFh	—	Reserved	—
Display Engine Pipeline Registers (60000h–6FFFFh)			
Display Pipeline A			
60000h–60003h	HTOTAL_A	Pipe A Horizontal Total	R/W
60004h–60007h	HBLANK_A	Pipe A Horizontal Blank	R/W
60008h–6000Bh	HSYNC_A	Pipe A Horizontal Sync	R/W
6000Ch–6000Fh	VTOTAL_A	Pipe A Vertical Total	R/W
60010h–60013h	VBLANK_A	Pipe A Vertical Blank	R/W
60014h–60017h	VSYNC_A	Pipe A Vertical Sync	R/W
60018h–6001Bh	—	Reserved	R/W
6001Ch–6001Fh	PIPEASRC	Pipe A Source Image Size	R/W
60020h–60023h	BCLRPAT_A	Pipe A Border Color Pattern	R/W
60024h–60027h	—	Reserved	—
60028h–6002Bh	VSYNCSHIFT_A	Vertical Sync Shift Register A	—
6002Ch–6004Fh	—	Reserved	—



Table 5-2. Memory-Mapped Registers

Address Offset	Symbol	Register Name	Access
60050h–60053h	CRCCTRLREDA	Pipe A CRC Red Control	R/W
60054h–60057h	CRCCTRLGREENA	Pipe A CRC Green Control	R/W
60058h–6005Bh	CRCCTRLBLUEA	Pipe A CRC Blue Control	R/W
6005Ch–6005Fh	CRCCTRLRESA	Pipe A CRC Residual Control Register	R/W
60060h–60063h	CRCRESREDA	Pipe A CRC Red Result	RO
60064h–60067h	CRCRESGREENA	Pipe A CRC Green Result	RO
60068h–6006Bh	CRCRESBLUEA	Pipe A CRC Blue Result	RO
6006Ch–6006Fh	CRCRESRESA	Pipe A CRC Residual Result	RO
60070h–60FFFh	—	Reserved	—
Display Pipeline B			
61000h–61003h	HTOTAL_B	Pipe B Horizontal Total	R/W
61004h–61007h	HBLANK_B	Pipe B Horizontal Blank	R/W
61008h–6100Bh	HSYNC_B	Pipe B Horizontal Sync	R/W
6100Ch–6100Fh	VTOTAL_B	Pipe B Vertical Total	R/W
61010h–61013h	VBLANK_B	Pipe B Vertical Blank	R/W
61014h–61017h	VSYNC_B	Pipe B Vertical Sync	R/W
61018h–6101Bh	—	Reserved	—
6101Ch–6101Fh	PIPEBSRC	Pipe B Source Image Size	R/W
61020h–61023h	BCLRPAT_B	Pipe B Border Color Pattern	R/W
61024h–61027h	—	Reserved	—
61028h–6102Bh	VSYNCSHIFT_B	Vertical Sync Shift Register B	—
6102Ch–6104Fh	—	Reserved	—
61050h–61053h	CRCCTRLREDB	Pipe B CRC Red Control	R/W
61054h–61057h	CRCCTRLGREENB	Pipe B CRC Green Control	R/W
61058h–6105Bh	CRCCTRLBLUEB	Pipe B CRC Blue Control	R/W
6105Ch–6105Fh	CRCCTRLRESB	Pipe B CRC Residual Control Register	R/W
61060h–61063h	CRCRESREDB	Pipe B CRC Red Result	RO
61064h–61067h	CRCRESGREENB	Pipe B CRC Green Result	RO
61068h–6106Bh	CRCRESBLUEB	Pipe B CRC Blue Result	RO
6106Ch–6106Fh	CRCRESRESB	Pipe B CRC Residual Result	RO
61070h–610FFh	—	Reserved	—
61100h–61103h	ADPA	Analog Display Port A Control	R/W



Table 5-2. Memory-Mapped Registers

Address Offset	Symbol	Register Name	Access
61104h–6110Fh	—	Reserved	—
61110h–61113h	PORT_HOTPLU_EN	Port HotPlug Enable	R/W
61114h–61117h	PORT_HOTPLU_STAT	Port HotPlug Status	R/W
61118h–61113Fh	—	Reserved	—
61140h–61143h	sDVO/HDMIB	Digital Display Port B Control Register	R/W
61144h–6114Fh	—	Reserved	—
61150h–61153h	sDVO/DP	Digital Display Port DFT Register	R/W
61154h–61157h	sDVO/DP	Digital Display Port DFT Register 2	R/W
61158h–6115Fh	—	Reserved	—
61160h–61163h	sDVO/HDMIC	Digital Display Port C I	R/W
61164h–6116Bh	—	Reserved	—
6116Ch–6116Fh	—	Reserved	—
61170h–61173h	VIDEO_DIP_CTL	Video DIP Control	R/W
61174h–61177h	—	Reserved	—
61178h–6117Bh	VIDEO_DIP_DATA	Video Data Island Packet Data	R/W
6117Ch–61177h	—	Reserved	—
LVDS ([DevCL] Only)			
61180h–61183h	LVDS	Digital Display Port Control ([DevCL])	R/W
61184h–611FFh	—	Reserved	—
Panel Power Sequencing ([DevCL] Only)			
61200h–61203h	PP_STATUS	Panel Power Status	RO
61204h–61207h	PP_CONTROL	Panel Power Control	R/W
61208h–6120Bh	PP_ON_DELAYS	Panel Power On Sequencing Delays	R/W
6120Ch–6120Fh	PP_OFF_DELAYS	Panel Power Off Sequencing Delays	R/W
61210h–61213h	PP_DIVISOR	Panel Power Cycle Delay and Reference Divisor	R/W
61214h–6122Fh	—	Reserved	—
Panel Fitting ([DevCL] Only)			
61230h–61233h	PFIT_CONTROL	Panel Fitting Control	R/W
61234h–61237h	PFIT_PGM_RATIOS	Programmed Panel Fitting Ratios	R/W
61238h–6124Fh	—	Reserved	—



Table 5-2. Memory-Mapped Registers

Address Offset	Symbol	Register Name	Access
Backlight Control and Modulation ([DevCL] Only)			
61250h–61253h	BLC_PWM_CTL2	Backlight PWM Control Register 2	R/W
61254h–61257h	BLC_PWM_CTL	Backlight PWM Control	R/W
61258h–6125Fh	—	Reserved	—
61260h–61263h	BLM_HIST_CTL	Image BLM Histogram Control Register	R/W
61264h–61267h		Image Enhancement Bin Data Register	RO, R/W
61268h–6126Bh		Histogram Threshold Guardband Register	R/W
6126Ch–61FFFh	—	Reserved	—
High Definition Audio Registers (62000h–62FFFh)			
62000h–62003h	AUD_CONFIG	Audio Configuration	R/W
62004h–6200Fh	—	Reserved	—
62010h–62013h	AUD_DEBUG	Audio Debug	RO
62014h–6201Fh	—	Reserved	—
62020h–62023h	AUD_VID_DID	Audio Vendor ID / Device ID	RO
62024h–62027h	AUD_RID	Audio Revision ID	RO
62028h–6202Bh	AUD_SUBN_CNT	Audio Subordinate Node Count	RO
6202Ch–6203Fh	—	Reserved	—
62040h–62043h	AUD_FUNC_GRP	Audio Function Group Type	RO
62044h–62047h	AUD_FUNC_SUBN_CNT	Audio Function Subordinate Node Count	RO
62048h–6204Bh	AUD_GRP_CAP	Audio Function Group Capabilities	RO
6204Ch–6204Fh	AUD_PWRST	Audio Power State	RO
62050h–62053h	AUD_SUPPWR	Audio Supported Power State	RO
62054h–62057h	AUD_SID	Audio Root Node Subsystem ID	RO
62058h–6206Fh	—	Reserved	—
62070h–62073h	AUD_OUT_CWCAP	Audio Output Converter Widget Capabilities	RO
62074h–62077h	AUD_OUT_PCMSIZE	Audio PCM Size and Rates	R/W
62078h–6207Bh	AUD_OUT_STR	Audio Stream Formats	R/W
6207Ch–6207Fh	AUD_OUT_DIG_CNVT	Audio Digital Converter	R/W
62080h–62083h	AUD_OUT_CH_STR	Audio Channel ID and Stream ID	RO
62084h–62087h	AUD_OUT_STR_DESC	Audio Stream Descriptor Format	RO
62088h–6209Fh	—	Reserved	—
620A0h–620A3h	AUD_PINW_CAP	Audio Pin Complex Widget Capabilities	RO



Table 5-2. Memory-Mapped Registers

Address Offset	Symbol	Register Name	Access
620A4h–620A7h	AUD_PIN_CAP	Audio Pin Capabilities	RO
620A8h–620ABh	AUD_PINW_CONNLNG	Audio Connection List Length	RO
620ACh–620AFh	AUD_PINW_CONNLST	Audio Connection List Entry	RO
620B0h–620B3h	AUD_PINW_CNTR	Audio Pin Widget Control	RO
620B4h–620B7h	AUD_CNTL_ST	Audio Control State	RO
620B8h–620BBh	AUD_PINW_UNRESOLVED	Audio Unsolicited Response Enable	RO
620BCh–620BFh	AUD_PINW_CONFIG	Audio Configuration Default	RO
620C0h–620D3h	—	Reserved	—
620D4h–620D7h	AUD_HDMIW_STATUS	Audio HDMI Status	R/W
620D8h–6210Bh	—	Reserved	—
6210Ch–62117h	AUD_HDMIW_HDMI_EDID	HDMI Data EDID Block	R/W
62118h–62127h	AUD_HDMIW_INFOFR	Audio HDMI Widget Data Island Packet	R/W
62128h–67FFFh	—	Reserved	—
TV Out Control Registers (68000h–6FFFFh)			
68000h–68003h	TV_CTL	TV Out Control	R/W
68004h–68007h	TV_DAC	TV DAC Control/Status	R/W, RO
68008h–6800Fh	—	Reserved	—
68010h–68013h	TV_CSC_Y	Color Space Convert Y	R/W
68014h–68017h	TV_CSC_Y2	Color Space Convert Y2	R/W
68018h–6801Bh	TV_CSC_U	Color Space Convert U	R/W
6801Ch–6801Fh	TV_CSC_U2	Color Space Convert U2	R/W
68020h–68023h	TV_CSC_V	Color Space Convert V	R/W
68024h–68027h	TV_CSC_V2	Color Space Convert V2	R/W
68028h–6802Bh	TV_CLR_KNOBS	Color Knobs	R/W
6802Ch–6802Fh	TV_CLR_LEVEL	Color Level Control	R/W
68030h–68033h	TV_H_CTL_1	H Control 1	R/W
68034h–68037h	TV_H_CTL_2	H Control 2	R/W
68038h–6803Bh	TV_H_CTL_3	H Control 3	R/W
6803Ch–6803Fh	TV_V_CTL_1	V Control 1	R/W
68040h–68043h	TV_V_CTL_2	V Control 2	R/W
68044h–68047h	TV_V_CTL_3	V Control 3	R/W
68048h–6804Bh	TV_V_CTL_4	V Control 4	R/W
6804Ch–6804Fh	TV_V_CTL_5	V Control 5	R/W



Table 5-2. Memory-Mapped Registers

Address Offset	Symbol	Register Name	Access
68050h–68053h	TV_V_CTL_6	V Control 6	R/W
68054h–68057h	TV_V_CTL_7	V Control 7	R/W
68058h–6805Fh	—	Reserved	—
68060h–68063h	TV_SC_CTL_1	SubCarrier Control 1	R/W
68064h–68067h	TV_SC_CTL_2	SubCarrier Control 2	R/W
68068h–6806Bh	TV_SC_CTL_3	SubCarrier Control 3	R/W
6806Ch–6806Fh	—	Reserved	—
68070h–68073h	TV_WIN_POS	Window Position	R/W
68074h–68077h	TV_WIN_SIZE	Window Size	R/W
68078h–6807Fh	—	Reserved	—
68080h–68083h	TV_FILTER_CTL_1	Filter Control 1	R/W
68084h–68087h	TV_FILTER_CTL_2	Filter Control 2	R/W
68088h–6808Bh	TV_FILTER_CTL_3	Filter Control 3	R/W
6808Ch–6808Fh	SIN_ROM	Sine ROM	—
68090h–68093h	TV_CC_CTL	Closed Caption Control	R/W
68094h–68097h	TV_CC_DATA1	Closed Caption Data Field 1	R/W
68098h–6809Bh	TV_CC_DATA2	Closed Caption Data Field 2	R/W
6809Ch–680AFh	—	Reserved	—
680B0h–680B3h	TV_WSS_CTL	WSS Control	R/W
680B4h–680B7h	TV_WSS_DATA	WSS Data	R/W
68100h–681EFh	TV_H_LUMA	H Filter Luma Coefficients	R/W
681F0h–681FFh	—	Reserved	—
68200h–682EFh	TV_H_CHROMA	H Filter Chroma Coefficients	R/W
682F0h–682FFh	—	Reserved	—
68300h–683ABh	TV_V_LUMA	V Filter Luma Coefficients	R/W
683ACh–683FFh	—	Reserved	—
68400h–684ABh	TV_V_CHROMA	V Filter Chroma Coefficients	R/W
684ACh–6FFFFh	—	Reserved	—
Display and Cursor Control Registers (70000h–77FFFh)			
Display Pipeline A Control			
70000h–70003h	PIPEA_DSL	Pipe A Display Scan Line Count	RO
70004h–70007h	PIPEA_SLC	Pipe A Display Scan Line Count Range Compare	RO
70008h–7000Bh	PIPEACONF	Pipe A Configuration	R/W
7000Ch–7000Fh	—	Reserved	—
70010h–70013h	PIPEAGCMAXRED	Pipe A Gamma Correction Max Red	R/W
70014h–70017h	PIPEAGCMAXGRN	Pipe A Gamma Correction Max Green	R/W



Table 5-2. Memory-Mapped Registers

Address Offset	Symbol	Register Name	Access
70018h–7001Bh	PIPEAGCMAXBLU	Pipe A Gamma Correction Max Blue	R/W
7001Ch–70023h	—	Reserved	—
70024h–70027h	PIPEASTAT	Pipe A Display Status Select	R/W
70028h–7002Fh	—	Reserved	—
70030h–70033h	DSPARB	Display Arbitration Control	R/W
70034h–70037h	FW1	Display FIFO Watermark Control 1	R/W
70038h–7003Bh	FW2	Display FIFO Watermark Control 2	
7003Ch–7003Fh	FW3	Display FIFO Watermark Control 3	R/W
70040h–70043h	PIPEAFRAMEH	Pipe A Frame Count High	RO
70044h–70047h	PIPEAFRAMEPIX	Pipe A Frame Count Low and Pixel Count	RO
70048h–7007Fh	—	Reserved	—
Cursor A and B Registers			
70080h–70083h	CURACNTR	Cursor A Control	R/W
70084h–70087h	CURABASE	Cursor A Base Address	R/W
70088h–7008Bh	CURAPOS	Cursor A Position	R/W
7008Ch–7008Fh	—	Reserved	—
70090h–7009Fh	CURAPALET[0:3]	Cursor A Palette 0:3	R/W
700A0h–700BFh	—	Reserved	—
700C0h–700C3h	CURBCNTR	Cursor B Control	R/W
700C4h–700C7h	CURBBASE	Cursor B Base Address	R/W
700C8h–700CBh	CURBPOS	Cursor B Position	R/W
700CCh–700CFh	—	Reserved	—
700D0h–700DFh	CURBPALET[0:3]	Cursor B Palette 0:3	R/W
700E0h–7017Fh	—	Reserved	—
Display A Control			
70180h–70183h	DSPACNTR	Display A Plane Control	R/W
70184h–70187h	DSPALINOFF	Display A Linear Offset Register	R/W
70188h–7018Bh	DSPASTRIDE	Display A Stride	R/W
7018Ch–7018Fh	—	Reserved	—
70190h–70193h	DSPARESV (RSVD)	Display A Reserved	R/W
70194h–70197h	DSPAKEYVAL	Sprite Color Key Value	R/W
70198h–7019Bh	DSPAKEYMSK	Sprite Color Key Mask Value	R/W
7019Ch–7019Fh	DSPASURF	Display A Surface Base Address Register	R/W
701A0h–701A3h	—	Reserved	—
701A4h–701A7h	DSPATILEOFF	Display A Tiled Offset Register	R/W
701A8h–701FFh	—	Reserved	—
70200h–70203h	DSPAFLPQSTAT	Flip Queue Status Register	R/W



Table 5-2. Memory-Mapped Registers

Address Offset	Symbol	Register Name	Access
70204h–703FFh	—	Reserved	—
VBIOS Software Flags 0-6			
70400h–70403h	—	Reserved	—
70404h–7040Fh	—	Reserved	—
70410h–7044Fh	SWFxx	Software Flag 00:0F	R/W
70450h–70FFFh	—	Reserved	—
Display Pipeline B Control			
71000h–71003h	PIPEB_DSL	Pipe B Display Scan Line Count	RO
71004h–71007h	PIPEB_SLC	Pipe B Display Scan Line Range Compare	RO
71008h–7100Bh	PIPEBCONF	Pipe B Configuration	R/W
7100Ch–7100Fh	—	Reserved	—
71010h–71013h	PIPEBGCMAXRED	Pipe B Gamma Correction Max Red	R/W
71014h–71017h	PIPEBGCMAXGRN	Pipe B Gamma Correction Max Green	R/W
71018h–7101Bh	PIPEBGCMAXBLU	Pipe B Gamma Correction Max Blue	R/W
71024h–71027h	PIPEBSTAT	Pipe B Status	R/W
71028h–7103Fh	—	Reserved	—
71040h–71043h	PIPEBFRAMEH	Pipe B Frame Count High	RO
71044h–71047h	PIPEBFRAMEPIX	Pipe B Frame Count Low and Pixel Count	RO
71048h–7117Fh	—	Reserved	—
Display B / Sprite Control			
71180h–71183h	DSPBCNTR	Display B / Sprite Control	R/W
71184h–71187h	DSPBLINOFFSET	Display B / Sprite Linear Offset	R/W
71188h–7118Bh	DSPBSTRIDE	Display B / Sprite Stride	R/W
7118Ch–71193h	—	Reserved	—
71194h–71197h	DSPBKEYVAL	Display B / Sprite Color Key Value	R/W
71198h–7119Bh	DSPBKEYMSK	Display B / Sprite Color Key Mask	R/W
7119Ch–7119Fh	DSPBSURF	Display B Surface Base Address Register	R/W
711A0h–711A3h	—	Reserved	—
711A4h–711A7h	DSPBTILEOFF	Display B Tiled Offset Register	R/W
711A8h–711FFh	—	Reserved	—
71200h–71203h	DSPBFLPQSTAT	Flip Queue Status Register	R/W
71204h–713FFh	—	Reserved	—



Table 5-2. Memory-Mapped Registers

Address Offset	Symbol	Register Name	Access
Video BIOS Registers			
71400h–71403h	VGACNTRL	VGA Display Plane Control	R/W
71404h–7140Fh	—	Reserved	—
VBIOS Software Flags 10-1F			
71410h–7144Fh	SWF[10-1F]	Software Flag 10 – 1F	R/W
71450h–71FFFh	—	Reserved	—
Display C / Sprite Control			
72000h–7217Fh	—	Reserved	—
72180h–72183h	DSPCCNTR	Display C / Sprite Control	R/W
72184h–72187h	DSPCLINOFF	Display C / Sprite Linear Offset Register	R/W
72188h–7218Bh	DSPCSTRIDE	Display C / Sprite Stride	R/W
7218Ch–7218Fh	DSPCPOS	Display C / Sprite Position	R/W
72190h–72193h	DSPCSIZE	Display C / Sprite Height and Width	R/W
72194h–72197h	DSPCKEYMINVAL	Display C / Sprite Color Key Min Value	R/W
72198h–7219Bh	DSPCKEYMSK	Display C / Sprite Color Key Mask	R/W
7219Ch–7219Fh	DSPCSURF	Display C Surface Address Register	R/W
721A0h–721A3h	DSPCKEYMAXVAL	Display C / Sprite Color Key Max Value	R/W
721A4h–721A7h	DSPCTILEOFF	Display C Tiled Offset Register	R/W
721A8h–721FFh	—	Reserved	—
72200h–72203h	DSPCFLPQSTAT	Flip Queue Status Register	R/W
72204h–721CFh	—	Reserved	—
721D0h–721D3h	DCLRC0	Display C Color Correction 0	R/W
721D4h–721D7h	DCLRC1	Display C Color Correction 1	R/W
721D8h–721DFh	—	Reserved	—
721E0h–721E3h	GAMC5	Display C Gamma Correction Register 5	R/W
721E4h–721E7h	GAMC4	Display C Gamma Correction Register 4	R/W
721E8h–721EBh	GAMC3	Display C Gamma Correction Register 3	R/W
721ECh–721EFh	GAMC2	Display C Gamma Correction Register 2	R/W
721F0h–721F3h	GAMC1	Display C Gamma Correction Register 1	R/W
721F4h–721F7h	GAMC0	Display C Gamma Correction Register 0	R/W
721F8h–723FFh	—	Reserved	—



Table 5-2. Memory-Mapped Registers

Address Offset	Symbol	Register Name	Access
Video Sprite A Control : Reserved			
72000h–723FFh	—	Reserved	—
VBIOS Software Flags 30-32			
72400h–72413h	—	Reserved	—
72414h–72417h	SWF[30]	Software Flag 30	R/W
72418h–7241Bh	SWF[31]	Software Flag 31	R/W
7241Ch–7241Fh	SWF[32]	Software Flag 32	R/W
72420h–72FFFh	—	Reserved	—
Performance Counters (73000h-73FFFh)			
73000h–73003h	PCSRC	Performance Counter Source Register	R/W
73004h–73007h	PCSTAT	Performance Counter Status Register	RO
73008h–7317Fh	—	Reserved	—
Video Sprite B Control : Reserved			
73180h–733FFh	—	Reserved	—
Reserved (74000h-7FFFFh)			
74000h–7FFFFh	—	Reserved	—



5.2 VGA and Extended VGA Register Map

For I/O locations, the value in the address column represents the register I/O address. For memory mapped locations, this address is an offset from the base address programmed in the MMADR register.

5.2.1 VGA and Extended VGA I/O and Memory Register Map

Table 5-3. I/O and Memory Register Map

Address	Register Name (Read)	Register Name (Write)
2D Registers		
3B0h–3B3h	Reserved	Reserved
3B4h	VGA CRTC Index (CRX) (monochrome)	VGA CRTC Index (CRX) (monochrome)
3B5h	VGA CRTC Data (monochrome)	VGA CRTC Data (monochrome)
3B6h–3B9h	Reserved	Reserved
3Bah	VGA Status Register (ST01)	VGA Feature Control Register (FCR)
3BBh–3BFh	Reserved	Reserved
3C0h	VGA Attribute Controller Index (ARX)	VGA Attribute Controller Index (ARX)/ VGA Attribute Controller Data (alternating writes select ARX or write ARxx Data)
3C1h	VGA Attribute Controller Data (read ARxx data)	Reserved
3C2h	VGA Feature Read Register (ST00)	VGA Miscellaneous Output Register (MSR)
3C3h	Reserved	Reserved
3C4h	VGA Sequencer Index (SRX)	VGA Sequencer Index (SRX)
3C5h	VGA Sequencer Data (SRxx)	VGA Sequencer Data (SRxx)
3C6h	VGA Color Palette Mask (DACMASK)	VGA Color Palette Mask (DACMASK)
3C7h	VGA Color Palette State (DACSTATE)	VGA Color Palette Read Mode Index (DACRX)
3C8h	VGA Color Palette Write Mode Index (DACWX)	VGA Color Palette Write Mode Index (DACWX)
3C9h	VGA Color Palette Data (DACDATA)	VGA Color Palette Data (DACDATA)
3CAh	VGA Feature Control Register (FCR)	Reserved
3CBh	Reserved	Reserved
3CCh	VGA Miscellaneous Output Register (MSR)	Reserved



Address	Register Name (Read)	Register Name (Write)
3CDh	Reserved	Reserved
3CEh	VGA Graphics Controller Index (GRX)	VGA Graphics Controller Index (GRX)
3CFh	VGA Graphics Controller Data (GRxx)	VGA Graphics Controller Data (GRxx)
3D0h–3D1h	Reserved	Reserved
2D Registers		
3D4h	VGA CRTC Index (CRX)	VGA CRTC Index (CRX)
3D5h	VGA CRTC Data (CRxx)	VGA CRTC Data (CRxx)
System Configuration Registers		
3D6h	GFX/2D Configurations Extensions Index (XRX)	GFX/2D Configurations Extensions Index (XRX)
3D7h	GFX/2D Configurations Extensions Data (XRxx)	GFX/2D Configurations Extensions Data (XRxx)
2D Registers		
3D8h–3D9h	Reserved	Reserved
3DAh	VGA Status Register (ST01)	VGA Feature Control Register (FCR)
3DBh–3DFh	Reserved	Reserved

5.3 Indirect VGA and Extended VGA Register Indices

The registers listed in this section are indirectly accessed by programming an index value into the appropriate SRX, GRX, ARX, or CRX register. The index and data register address locations are listed in the previous section. Additional details concerning the indirect access mechanism are provided in the *VGA and Extended VGA Register Description* Chapter (see SRxx, GRxx, ARxx or CRxx sections).

Table 5-4. 2D Sequence Registers (3C4h / 3C5h)

Index	Sym	Description
00h	SR00	Sequencer Reset
01h	SR01	Clocking Mode
02h	SR02	Plane / Map Mask
03h	SR03	Character Font
04h	SR04	Memory Mode
07h	SR07	Horizontal Character Counter Reset



Table 5-5. 2D Graphics Controller Registers (3CEh / 3CFh)

Index	Sym	Register Name
00h	GR00	Set / Reset
01h	GR01	Enable Set / Reset
02h	GR02	Color Compare
03h	GR03	Data Rotate
04h	GR04	Read Plane Select
05h	GR05	Graphics Mode
06h	GR06	Miscellaneous
07h	GR07	Color Don't Care
08h	GR08	Bit Mask
10h	GR10	Address Mapping
11h	GR11	Page Selector
18h	GR18	Software Flags

Table 5-6. 2D Attribute Controller Registers (3C0h / 3C1h)

Index	Sym	Register Name
00h	AR00	Palette Register 0
01h	AR01	Palette Register 1
02h	AR02	Palette Register 2
03h	AR03	Palette Register 3
04h	AR04	Palette Register 4
05h	AR05	Palette Register 5
06h	AR06	Palette Register 6
07h	AR07	Palette Register 7
08h	AR08	Palette Register 8
09h	AR09	Palette Register 9
0Ah	AR0A	Palette Register A
0Bh	AR0B	Palette Register B
0Ch	AR0C	Palette Register C
0Dh	AR0D	Palette Register D
0Eh	AR0E	Palette Register E
0Fh	AR0F	Palette Register F
10h	AR10	Mode Control
11h	AR11	Overscan Color
12h	AR12	Memory Plane Enable
13h	AR13	Horizontal Pixel Panning
14h	AR14	Color Select



Table 5-7. 2D CRT Controller Registers (3B4h / 3D4h / 3B5h / 3D5h)

Index	Sym	Register Name
00h	CR00	Horizontal Total
01h	CR01	Horizontal Display Enable End
02h	CR02	Horizontal Blanking Start
03h	CR03	Horizontal Blanking End
04h	CR04	Horizontal Sync Start
05h	CR05	Horizontal Sync End
06h	CR06	Vertical Total
07h	CR07	Overflow
08h	CR08	Preset Row Scan
09h	CR09	Maximum Scan Line
0Ah	CR0A	Text Cursor Start
0Bh	CR0B	Text Cursor End
0Ch	CR0C	Start Address High
0Dh	CR0D	Start Address Low
0Eh	CR0E	Text Cursor Location High
0Fh	CR0F	Text Cursor Location Low
10h	CR10	Vertical Sync Start
11h	CR11	Vertical Sync End
12h	CR12	Vertical Display Enable End
13h	CR13	Offset
14h	CR14	Underline Location
15h	CR15	Vertical Blanking Start
16h	CR16	Vertical Blanking End
17h	CR17	CRT Mode
18h	CR18	Line Compare
22h	CR22	Memory Read Latch Data
24h	CR24	Test Register for Toggle State of Attribute Control Register





6 Memory Data Formats

This chapter describes the attributes associated with the memory-resident data objects operated on by the graphics pipeline. This includes object types, pixel formats, memory layouts, and rules/restrictions placed on the dimensions, physical memory location, pitch, alignment, etc. with respect to the specific operations performed on the objects.

6.1 Memory Object Overview

Any memory data accessed by the device is considered part of a *memory object* of some memory object type.

6.1.1 Memory Object Types

The following table lists the various memory objects types and an indication of their role in the system.

Memory Object Type	Role
Graphics Translation Table (GTT)	Contains PTEs used to translate “graphics addresses” into physical memory addresses.
Hardware Status Page	Cached page of sysmem used to provide fast driver synchronization.
Logical Context Buffer	Memory areas used to store (save/restore) images of hardware rendering contexts. Logical contexts are referenced via a pointer to the corresponding Logical Context Buffer.
Ring Buffers	Buffers used to transfer (DMA) instruction data to the device. Primary means of controlling rendering operations.
Batch Buffers	Buffers of instructions invoked indirectly from Ring Buffers.
State Descriptors	Contains state information in a prescribed layout format to be read by hardware. Many different state descriptor formats are supported.
Vertex Buffers	Buffers of 3D vertex data indirectly referenced through “indexed” 3D primitive instructions.
VGA Buffer (Must be mapped UC on PCI)	Graphics memory buffer used to drive the display output while in legacy VGA mode.
Display Surface	Memory buffer used to display images on display devices.
Overlay Surface	Memory buffer used to display overlaid images on display devices.
Overlay Register, Filter Coefficients Buffer	Memory area used to provide double-buffer for Overlay register and filter coefficient loading.
Cursor Surface	Hardware cursor pattern in memory.



Memory Object Type	Role
2D Render Source	Surface used as primary input to 2D rendering operations.
2D Render R-M-W Destination	2D rendering output surface that is read in order to be combined in the rendering function. Destination surfaces that accessed via this Read-Modify-Write mode have somewhat different restrictions than Write-Only Destination surfaces.
2D Render Write-Only Destination	2D rendering output surface that is written but not read by the 2D rendering function. Destination surfaces that accessed via a Write-Only mode have somewhat different restrictions than Read-Modify-Write Destination surfaces.
2D Monochrome Source	1 bpp surfaces used as inputs to 2D rendering after being converted to foreground/background colors.
2D Color Pattern	8x8 pixel array used to supply the “pattern” input to 2D rendering functions.
DIB	“Device Independent Bitmap” surface containing “logical” pixel values that are converted (via LUTs) to physical colors.
3D Color Buffer	Surface receiving color output of 3D rendering operations. May also be accessed via R-M-W (aka blending). Also referred to as a Render Target.
3D Depth Buffer	Surface used to hold per-pixel depth and stencil values used in 3D rendering operations. Accessed via RMW.
3D Texture Map	Color surface (or collection of surfaces) which provide texture data in 3D rendering operations.
“Non-3D” Texture	Surface read by Texture Samplers, though not in normal 3D rendering operations (e.g., in video color conversion functions).
Motion Comp Surfaces	These are the Motion Comp reference pictures.
Motion Comp Correction Data Buffer	This is Motion Comp intra-coded or inter-coded correction data.

6.2 Channel Formats

6.2.1 Unsigned Normalized (UNORM)

An unsigned normalized value with n bits is interpreted as a value between 0.0 and 1.0. The minimum value (all 0's) is interpreted as 0.0, the maximum value (all 1's) is interpreted as 1.0. Values inbetween are equally spaced. For example, a 2-bit UNORM value would have the four values 0, 1/3, 2/3, and 1.

If the incoming value is interpreted as an n -bit integer, the interpreted value can be calculated by dividing the integer by $2^n - 1$.



6.2.2 Gamma Conversion (SRGB)

Gamma conversion is only supported on UNORM formats. If this flag is included in the surface format name, it indicates that a reverse gamma conversion is to be done after the source surface is read, and a forward gamma conversion is to be done before the destination surface is written.

6.2.3 Signed Normalized (SNORM)

A signed normalized value with n bits is interpreted as a value between -1.0 and +1.0. If the incoming value is interpreted as a 2's-complement n -bit signed integer, the interpreted value can be calculated by dividing the integer by $2^{n-1}-1$. Note that the most negative value of -2^{n-1} will result in a value slightly smaller than -1.0. This value is clamped to -1.0, thus there are two representations of -1.0 in SNORM format.

6.2.4 Unsigned Integer (UINT/USCALED)

The UINT and USCALED formats interpret the source as an unsigned integer value with n bits with a range of 0 to 2^n-1 .

The UINT formats copy the source value to the destination (zero-extending if required), keeping the value as an integer.

The USCALED formats convert the integer into the corresponding floating point value (e.g., 0x03 --> 3.0f). For 32-bit sources, the value is rounded to nearest even.

6.2.5 Signed Integer (SINT/SSCALED)

A signed integer value with n bits is interpreted as a 2's complement integer with a range of -2^{n-1} to $+2^{n-1}-1$.

The SINT formats copy the source value to the destination (sign-extending if required), keeping the value as an integer.

The SSCALED formats convert the integer into the corresponding floating point value (e.g., 0xFFFFD --> -3.0f). For 32-bit sources, the value is rounded to nearest even.



6.2.6 Floating Point (FLOAT)

Refer to IEEE Standard 754 for Binary Floating-Point Arithmetic. The *IA-32 Intel® Architecture Software Developer's Manual* also describes floating point data types (though GEN4 deviates slightly from those behaviors).

6.2.6.1 32-bit Floating Point

Bit	Description
31	Sign (s)
30:23	Exponent (e) Biased Exponent
22:0	Fraction (f) Does not include “hidden one”

The value of this data type is derived as:

- if $e == 255$ and $f != 0$, then v is NaN regardless of s
- if $e == 255$ and $f == 0$, then $v = (-1)^s * \text{infinity}$ (signed infinity)
- if $0 < e < 255$, then $v = (-1)^s * 2^{(e-127)} * (1.f)$
- if $e == 0$ and $f != 0$, then $v = (-1)^s * 2^{(e-126)} * (0.f)$ (denormalized numbers)
- if $e == 0$ and $f == 0$, then $v = (-1)^s * 0$ (signed zero)

6.2.6.2 64-bit Floating Point

Bit	Description
63	Sign (s)
62:52	Exponent (e) Biased Exponent
51:0	Fraction (f) Does not include “hidden one”

The value of this data type is derived as:

- if $e == b'11..11'$ and $f != 0$, then v is NaN regardless of s
- if $e == b'11..11'$ and $f == 0$, then $v = (-1)^s * \text{infinity}$ (signed infinity)
- if $0 < e < b'11..11'$, then $v = (-1)^s * 2^{(e-1023)} * (1.f)$
- if $e == 0$ and $f != 0$, then $v = (-1)^s * 2^{(e-1022)} * (0.f)$ (denormalized numbers)
- if $e == 0$ and $f == 0$, then $v = (-1)^s * 0$ (signed zero)

6.3 Non-Video Surface Formats

This section describes the lowest-level organization of a surfaces containing discrete “pixel” oriented data (e.g., discrete pixel (RGB,YUV) colors, subsampled video data, 3D depth/stencil buffer pixel formats, bump map values etc. Many of these pixel formats are common to the various pixel-oriented memory object types.

6.3.1 Surface Format Naming

Unless indicated otherwise, all pixels are **stored** in “**little endian**” byte order. I.e., pixel bits 7:0 are stored in byte n , pixel bits 15:8 are stored in byte $n+1$, and so on.



The format labels include color components in little endian order (e.g., R8G8B8A8 format is physically stored as R, G, B, A).

The name of most of the surface formats specifies its format. Channels are listed in little endian order (LSB channel on the left, MSB channel on the right), with the channel format specified following the channels with that format. For example, R5G5_SNORM_B6_UNORM contains, from LSB to MSB, 5 bits of red in SNORM format, 5 bits of green in SNORM format, and 6 bits of blue in UNORM format.

6.3.2 Intensity Formats

All surface formats containing "I" include an intensity value. When used as a source surface for the sampling engine, the intensity value is replicated to all four channels (R,G,B,A) before being filtered. Intensity surfaces are not supported as destinations.

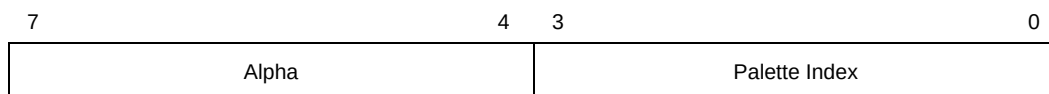
6.3.3 Luminance Formats

All surface formats containing "L" include a luminance value. When used as a source surface for the sampling engine, the luminance value is replicated to the three color channels (R,G,B) before being filtered. The alpha channel is provided either from another field or receives a default value. Luminance surfaces are not supported as destinations.



6.3.4 P4A4_UNORM

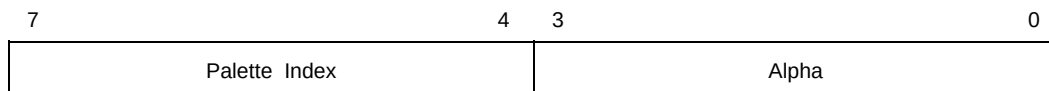
This texel format contains a 4-bit Alpha value (in the high nibble) and a 4-bit Palette Index value (in the low nibble).



Bit	Description
7:4	Alpha Alpha value which will be replicated to both the high and low nibble of an 8-bit value, and then divided by 255 to yield a [0.0,1.0] Alpha value. Format: U4
3:0	Palette Index A 4-bit index which is used to lookup a 24-bit (RGB) value in the texture palette (loaded via 3DSTATE_SAMPLER_PALETTE_LOAD) Format: U4

6.3.5 A4P4_UNORM

This texel format contains a 4-bit Alpha value (in the low nibble) and a 4-bit Color Index value (in the high nibble).



Bit	Description
7:4	Palette Index A 4-bit color index which is used to lookup a 24-bit RGB value in the texture palette. Format: U4
3:0	Alpha Alpha value which will be replicated to both the high and low nibble of an 8-bit value, and then divided by 255 to yield a [0.0,1.0] alpha value. Format: U4



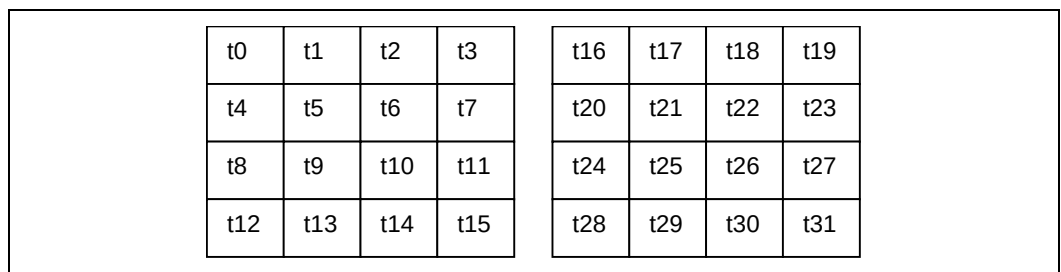
6.4 Compressed Surface Formats

This section contains information on the internal organization of compressed surface formats.

6.4.1 FXT Texture Formats

There are four different FXT1 compressed texture formats. Each of the formats compress two 4x4 texel blocks into 128 bits. In each compression format, the 32 texels in the two 4x4 blocks are arranged according to the following diagram:

Figure 6-1. FXT1 Encoded Blocks



6.4.1.1 Overview of FXT1 Formats

During the compression phase, the encoder selects one of the four formats for each block based on which encoding scheme results in best overall visual quality. The following table lists the four different modes and their encodings:

Table 6-1. FXT1 Format Summary

Bit 127	Bit 126	Bit 125	Block Compression Mode	Summary Description
0	0	X	CC_HI	2 R5G5B5 colors supplied. Single LUT with 7 interpolated color values and transparent black
0	1	0	CC_CHROMA	4 R5G5B5 colors used directly as 4-entry LUT.
0	1	1	CC_ALPHA	3 A5R5G5B5 colors supplied. LERP bit selects between 1 LUT with 3 discrete colors + transparent black and 2 LUTs using interpolated values of Color 0,1 (t0-15) and Color 1,2 (t16-31).
1	x	x	CC_MIXED	4 R5G5B5 colors supplied, where Color0,1 LUT is used for t0-t15, and Color2,3 LUT used for t16-31. Alpha bit selects between LUTs with 4 interpolated colors or 3 interpolated colors + transparent black.



6.4.1.2 FXT1 CC_HI Format

In the CC_HI encoding format, two base 15-bit R5G5B5 colors (Color 0, Color 1) are included in the encoded block. These base colors are then expanded (using high-order bit replication) to 24-bit RGB colors, and used to define an 8-entry lookup table of interpolated color values (the 8th entry is transparent black). The encoded block contains a 3-bit index value per texel that is used to lookup a color from the table.

6.4.1.2.1 CC_HI Block Encoding

The following table describes the encoding of the 128-bit (DQWord) CC_HI block format:

Table 6-2. FXT CC_HI Block Encoding

Bit	Description
127:126	Mode = '00'b (CC_HI)
125:121	Color 1 Red
120:116	Color 1 Green
115:111	Color 1 Blue
110:106	Color 0 Red
105:101	Color 0 Green
100:96	Color 0 Blue
95:93	Texel 31 Select
...	...
50:48	Texel 16 Select
47:45	Texel 15 Select
...	...
2:0	Texel 0 Select



6.4.1.2.2 CC_HI Block Decoding

The two base colors, Color 0 and Color 1 are converted from R5G5B5 to R8G8B8 by replicating the 3 MSBs into the 3 LSBs, as shown in the following table:

Table 6-3. FXT CC_HI Decoded Colors

Expanded Color Bit	Expanded Channel Bit	Encoded Block Source Bit
Color 1 [23:19]	Color 1 Red [7:3]	[125:121]
Color 1 [18:16]	Color 1 Red [2:0]	[125:123]
Color 1 [15:11]	Color 1 Green [7:3]	[120:116]
Color 1 [10:08]	Color 1 Green [2:0]	[120:118]
Color 1 [07:03]	Color 1 Blue [7:3]	[115:111]
Color 1 [02:00]	Color 1 Blue [2:0]	[115:113]
Color 0 [23:19]	Color 0 Red [7:3]	[110:106]
Color 0 [18:16]	Color 0 Red [2:0]	[110:108]
Color 0 [15:11]	Color 0 Green [7:3]	[105:101]
Color 0 [10:08]	Color 0 Green [2:0]	[105:103]
Color 0 [07:03]	Color 0 Blue [7:3]	[100:96]
Color 0 [02:00]	Color 0 Blue [2:0]	[100:98]

These two 24-bit colors (Color 0, Color 1) are then used to create a table of seven interpolated colors (with Alpha = 0FFh), along with an eight entry equal to RGBA = 0,0,0,0, as shown in the following table:

Table 6-4. FXT CC_HI Interpolated Color Table

Interpolated Color	Color RGB	Alpha
0	Color0.RGB	0FFh
1	$(5 * \text{Color0.RGB} + 1 * \text{Color1.RGB} + 3) / 6$	0FFh
2	$(4 * \text{Color0.RGB} + 2 * \text{Color1.RGB} + 3) / 6$	0FFh
3	$(3 * \text{Color0.RGB} + 3 * \text{Color1.RGB} + 3) / 6$	0FFh
4	$(2 * \text{Color0.RGB} + 4 * \text{Color1.RGB} + 3) / 6$	0FFh
5	$(1 * \text{Color0.RGB} + 5 * \text{Color1.RGB} + 3) / 6$	0FFh
6	Color1.RGB	0FFh
7	RGB = 0,0,0	0

This table is then used as an 8-entry Lookup Table, where each 3-bit Texel n Select field of the encoded CC_HI block is used to index into a 32-bit A8R8G8B8 color from the table completing the decode of the CC_HI block.



6.4.1.3 FXT1 CC_CHROMA Format

In the CC_CHROMA encoding format, four 15-bit R5B5G5 colors are included in the encoded block. These colors are then expanded (using high-order bit replication) to form a 4-entry table of 24-bit RGB colors. The encoded block contains a 2-bit index value per texel that is used to lookup a 24-bit RGB color from the table. The Alpha component defaults to fully opaque (0FFh).

6.4.1.3.1 CC_CHROMA Block Encoding

The following table describes the encoding of the 128-bit (DQWord) CC_CHROMA block format:

Table 6-5. FXT CC_CHROMA Block Encoding

Bit	Description
127:125	Mode = '010'b (CC_CHROMA)
124	Unused
123:119	Color 3 Red
118:114	Color 3 Green
113:109	Color 3 Blue
108:104	Color 2 Red
103:99	Color 2 Green
98:94	Color 2 Blue
93:89	Color 1 Red
88:84	Color 1 Green
83:79	Color 1 Blue
78:74	Color 0 Red
73:69	Color 0 Green
68:64	Color 0 Blue
63:62	Texel 31 Select
...	
33:32	Texel 16 Select
31:30	Texel 15 Select
...	
1:0	Texel 0 Select



6.4.1.3.2 CC_CHROMA Block Decoding

The four colors (Color 0-3) are converted from R5G5B5 to R8G8B8 by replicating the 3 MSBs into the 3 LSBs, as shown in the following tables:

Table 6-6. FXT CC_CHROMA Decoded Colors

Expanded Color Bit	Expanded Channel Bit	Encoded Block Source Bit
Color 3 [23:17]	Color 3 Red [7:3]	[123:119]
Color 3 [18:16]	Color 3 Red [2:0]	[123:121]
Color 3 [15:11]	Color 3 Green [7:3]	[118:114]
Color 3 [10:08]	Color 3 Green [2:0]	[118:116]
Color 3 [07:03]	Color 3 Blue [7:3]	[113:109]
Color 3 [02:00]	Color 3 Blue [2:0]	[113:111]
Color 2 [23:17]	Color 2 Red [7:3]	[108:104]
Color 2 [18:16]	Color 2 Red [2:0]	[108:106]
Color 2 [15:11]	Color 2 Green [7:3]	[103:99]
Color 2 [10:08]	Color 2 Green [2:0]	[103:101]
Color 2 [07:03]	Color 2 Blue [7:3]	[98:94]
Color 2 [02:00]	Color 2 Blue [2:0]	[98:96]
Color 1 [23:17]	Color 1 Red [7:3]	[93:89]
Color 1 [18:16]	Color 1 Red [2:0]	[93:91]
Color 1 [15:11]	Color 1 Green [7:3]	[88:84]
Color 1 [10:08]	Color 1 Green [2:0]	[88:86]
Color 1 [07:03]	Color 1 Blue [7:3]	[83:79]
Color 1 [02:00]	Color 1 Blue [2:0]	[83:81]
Color 0 [23:17]	Color 0 Red [7:3]	[78:74]
Color 0 [18:16]	Color 0 Red [2:0]	[78:76]
Color 0 [15:11]	Color 0 Green [7:3]	[73:69]
Color 0 [10:08]	Color 0 Green [2:0]	[73:71]
Color 0 [07:03]	Color 0 Blue [7:3]	[68:64]
Color 0 [02:00]	Color 0 Blue [2:0]	[68:66]

This table is then used as a 4-entry Lookup Table, where each 2-bit Texel n Select field of the encoded CC_CHROMA block is used to index into a 32-bit A8R8G8B8 color from the table (Alpha defaults to 0FFh) completing the decode of the CC_CHROMA block.

Table 6-7. FXT CC_CHROMA Interpolated Color Table

Texel Select	Color ARGB
0	Color0.ARGB
1	Color1.ARGB
2	Color2.ARGB
3	Color3.ARGB

6.4.1.4 FXT1 CC_MIXED Format

In the CC_MIXED encoding format, four 15-bit R5G5B5 colors are included in the encoded block: Color 0 and Color 1 are used for Texels 0-15, and Color 2 and Color 3 are used for Texels 16-31.

Each pair of colors are then expanded (using high-order bit replication) to form 4-entry tables of 24-bit RGB colors. The encoded block contains a 2-bit index value per texel that is used to lookup a 24-bit RGB color from the table. The Alpha component defaults to fully opaque (0FFh).

6.4.1.4.1 CC_MIXED Block Encoding

The following table describes the encoding of the 128-bit (DQWord) CC_MIXED block format:

Table 6-8. FXT CC_MIXED Block Encoding

Bit	Description
127	Mode = '1'b (CC_MIXED)
126	Color 3 Green [0]
125	Color 1 Green [0]
124	Alpha [0]
123:119	Color 3 Red
118:114	Color 3 Green
113:109	Color 3 Blue
108:104	Color 2 Red
103:99	Color 2 Green
98:94	Color 2 Blue
93:89	Color 1 Red
88:84	Color 1 Green
83:79	Color 1 Blue
78:74	Color 0 Red



Bit	Description
73:69	Color 0 Green
68:64	Color 0 Blue
63:62	Texel 31 Select
...	...
33:32	Texel 16 Select
31:30	Texel 15 Select
...	...
1:0	Texel 0 Select

6.4.1.4.2 CC_MIXED Block Decoding

The decode of the CC_MIXED block is modified by Bit 124 (Alpha [0]) of the encoded block.

Alpha[0] = 0 Decoding

When Alpha[0] = 0 the four colors are encoded as 16-bit R5G6B5 values, with the Green LSB defined as per the following table:

Table 6-9. FXT CC_MIXED (Alpha[0]=0) Decoded Colors

Encoded Color Bit	Definition
Color 3 Green [0]	Encoded Bit [126]
Color 2 Green [0]	Encoded Bit [33] XOR Encoded Bit [126]
Color 1 Green [0]	Encoded Bit [125]
Color 0 Green [0]	Encoded Bit [1] XOR Encoded Bit [125]

The four colors (Color 0-3) are then converted from R5G5B6 to R8G8B8 by replicating the 3 MSBs into the 3 LSBs, as shown in the following table:

Table 6-10. FXT CC_MIXED Decoded Colors (Alpha[0] = 0)

Expanded Color Bit	Expanded Channel Bit	Encoded Block Source Bit
Color 3 [23:17]	Color 3 Red [7:3]	[123:119]
Color 3 [18:16]	Color 3 Red [2:0]	[123:121]
Color 3 [15:11]	Color 3 Green [7:3]	[118:114]
Color 3 [10]	Color 3 Green [2]	[126]
Color 3 [09:08]	Color 3 Green [1:0]	[118:117]
Color 3 [07:03]	Color 3 Blue [7:3]	[113:109]



Expanded Color Bit	Expanded Channel Bit	Encoded Block Source Bit
Color 3 [02:00]	Color 3 Blue [2:0]	[113:111]
Color 2 [23:17]	Color 2 Red [7:3]	[108:104]
Color 2 [18:16]	Color 2 Red [2:0]	[108:106]
Color 2 [15:11]	Color 2 Green [7:3]	[103:99]
Color 2 [10]	Color 2 Green [2]	[33] XOR [126]
Color 2 [09:08]	Color 2 Green [1:0]	[103:100]
Color 2 [07:03]	Color 2 Blue [7:3]	[98:94]
Color 2 [02:00]	Color 2 Blue [2:0]	[98:96]
Color 1 [23:17]	Color 1 Red [7:3]	[93:89]
Color 1 [18:16]	Color 1 Red [2:0]	[93:91]
Color 1 [15:11]	Color 1 Green [7:3]	[88:84]
Color 1 [10]	Color 1 Green [2]	[125]
Color 1 [09:08]	Color 1 Green [1:0]	[88:86]
Color 1 [07:03]	Color 1 Blue [7:3]	[83:79]
Color 1 [02:00]	Color 1 Blue [2:0]	[83:81]
Color 0 [23:17]	Color 0 Red [7:3]	[78:74]
Color 0 [18:16]	Color 0 Red [2:0]	[78:76]
Color 0 [15:11]	Color 0 Green [7:3]	[73:69]
Color 0 [10]	Color 0 Green [2]	[1] XOR [125]
Color 0 [09:08]	Color 0 Green [1:0]	[73:71]
Color 0 [07:03]	Color 0 Blue [7:3]	[68:64]
Color 0 [02:00]	Color 0 Blue [2:0]	[68:66]

The two sets of 24-bit colors (Color 0,1 and Color 2,3) are then used to create two tables of four interpolated colors (with Alpha = 0FFh). The Color0,1 table is used as a lookup table for texel 0-15 indices, and the Color2,3 table used for texels 16-31 indices, as shown in the following figures:

Table 6-11. FXT CC_MIXED Interpolated Color Table (Alpha[0]=0, Texels 0-15)

Texel 0-15 Select	Color RGB	Alpha
0	Color0.RGB	0FFh
1	$(2 * \text{Color0.RGB} + \text{Color1.RGB} + 1) / 3$	0FFh
2	$(\text{Color0.RGB} + 2 * \text{Color1.RGB} + 1) / 3$	0FFh
3	Color1.RGB	0FFh



Table 6-12. FXT CC_MIXED Interpolated Color Table (Alpha[0]=0, Texels 16-31)

Texel 16-31 Select	Color RGB	Alpha
0	Color2.RGB	0FFh
1	$(2/3) * \text{Color2.RGB} + (1/3) * \text{Color3.RGB}$	0FFh
2	$(1/3) * \text{Color2.RGB} + (2/3) * \text{Color3.RGB}$	0FFh
3	Color3.RGB	0FFh

Alpha[0] = 1 Decoding

When Alpha[0] = 1, Color0 and Color2 are encoded as 15-bit R5G5B5 values. Color1 and Color3 are encoded as RGB565 colors, with the Green LSB obtained as shown in the following table:

Table 6-13. FXT CC_MIXED (Alpha[0]=0) Decoded Colors

Encoded Color Bit	Definition
Color 3 Green [0]	Encoded Bit [126]
Color 1 Green [0]	Encoded Bit [125]

All four colors are then expanded to 24-bit R8G8B8 colors by bit replication, as show in the following diagram.

Table 6-14. FXT CC_MIXED Decoded Colors (Alpha[0] = 1)

Expanded Color Bit	Expanded Channel Bit	Encoded Block Source Bit
Color 3 [23:17]	Color 3 Red [7:3]	[123:119]
Color 3 [18:16]	Color 3 Red [2:0]	[123:121]
Color 3 [15:11]	Color 3 Green [7:3]	[118:114]
Color 3 [10]	Color 3 Green [2]	[126]
Color 3 [09:08]	Color 3 Green [1:0]	[118:117]
Color 3 [07:03]	Color 3 Blue [7:3]	[113:109]
Color 3 [02:00]	Color 3 Blue [2:0]	[113:111]
Color 2 [23:19]	Color 2 Red [7:3]	[108:104]
Color 2 [18:16]	Color 2 Red [2:0]	[108:106]
Color 2 [15:11]	Color 2 Green [7:3]	[103:99]
Color 2 [10:08]	Color 2 Green [2:0]	[103:101]
Color 2 [07:03]	Color 2 Blue [7:3]	[98:94]
Color 2 [02:00]	Color 2 Blue [2:0]	[98:96]
Color 1 [23:17]	Color 1 Red [7:3]	[93:89]
Color 1 [18:16]	Color 1 Red [2:0]	[93:91]
Color 1 [15:11]	Color 1 Green [7:3]	[88:84]
Color 1 [10]	Color 1 Green [2]	[125]



Expanded Color Bit	Expanded Channel Bit	Encoded Block Source Bit
Color 1 [09:08]	Color 1 Green [1:0]	[88:87]
Color 1 [07:03]	Color 1 Blue [7:3]	[83:79]
Color 1 [02:00]	Color 1 Blue [2:0]	[83:81]
Color 0 [23:19]	Color 0 Red [7:3]	[78:74]
Color 0 [18:16]	Color 0 Red [2:0]	[78:76]
Color 0 [15:11]	Color 0 Green [7:3]	[73:69]
Color 0 [10:08]	Color 0 Green [2:0]	[73:71]
Color 0 [07:03]	Color 0 Blue [7:3]	[68:64]
Color 0 [02:00]	Color 0 Blue [2:0]	[68:66]

The two sets of 24-bit colors (Color 0,1 and Color 2,3) are then used to create two tables of four colors. The Color0,1 table is used as a lookup table for texel 0-15 indices, and the Color2,3 table used for texels 16-31 indices. The color at index 1 is the linear interpolation of the base colors, while the color at index 3 is defined as Black (0,0,0) with Alpha = 0, as shown in the following figures:

Table 6-15. FXT CC_MIXED Interpolated Color Table (Alpha[0]=1, Texels 0-15)

Texel 0-15 Select	Color RGB	Alpha
0	Color0.RGB	0FFh
1	$(\text{Color0.RGB} + \text{Color1.RGB}) / 2$	0FFh
2	Color1.RGB	0FFh
3	Black (0,0,0)	0

Table 6-16. FXT CC_MIXED Interpolated Color Table (Alpha[0]=1, Texels 16-31)

Texel 16-31 Select	Color RGB	Alpha
0	Color2.RGB	0FFh
1	$(\text{Color2.RGB} + \text{Color3.RGB}) / 2$	0FFh
2	Color3.RGB	0FFh
3	Black (0,0,0)	0

These tables are then used as a 4-entry Lookup Table, where each 2-bit Texel n Select field of the encoded CC_MIXED block is used to index into the appropriate 32-bit A8R8G8B8 color from the table, completing the decode of the CC_CMIXED block.



6.4.1.5 FXT1 CC_ALPHA Format

In the CC_ALPHA encoding format, three A5R5G5B5 colors are provided in the encoded block. A control bit (LERP) is used to define the lookup table (or tables) used to dereference the 2-bit Texel Selects.

6.4.1.5.1 CC_ALPHA Block Encoding

The following table describes the encoding of the 128-bit (DQWord) CC_ALPHA block format:

Table 6-17. FXT CC_ALPHA Block Encoding

Bit	Description
127:125	Mode = '011'b (CC_ALPHA)
124	LERP
123:119	Color 2 Alpha
118:114	Color 1 Alpha
113:109	Color 0 Alpha
108:104	Color 2 Red
103:99	Color 2 Green
98:94	Color 2 Blue
93:89	Color 1 Red
88:84	Color 1 Green
83:79	Color 1 Blue
78:74	Color 0 Red
73:69	Color 0 Green
68:64	Color 0 Blue
63:62	Texel 31 Select
...	...
33:32	Texel 16 Select
31:30	Texel 15 Select
...	...
1:0	Texel 0 Select



6.4.1.5.2 CC_ALPHA Block Decoding

Each of the three colors (Color 0-2) are converted from A5R5G5B5 to A8R8G8B8 by replicating the 3 MSBs into the 3 LSBs, as shown in the following tables:

Table 6-18. FXT CC_ALPHA Decoded Colors

Expanded Color Bit	Expanded Channel Bit	Encoded Block Source Bit
Color 2 [31:27]	Color 2 Alpha [7:3]	[123:119]
Color 2 [26:24]	Color 2 Alpha [2:0]	[123:121]
Color 2 [23:17]	Color 2 Red [7:3]	[108:104]
Color 2 [18:16]	Color 2 Red [2:0]	[108:106]
Color 2 [15:11]	Color 2 Green [7:3]	[103:99]
Color 2 [10:08]	Color 2 Green [2:0]	[103:101]
Color 2 [07:03]	Color 2 Blue [7:3]	[98:94]
Color 2 [02:00]	Color 2 Blue [2:0]	[98:96]
Color 1 [31:27]	Color 1 Alpha [7:3]	[118:114]
Color 1 [26:24]	Color 1 Alpha [2:0]	[118:116]
Color 1 [23:17]	Color 1 Red [7:3]	[93:89]
Color 1 [18:16]	Color 1 Red [2:0]	[93:91]
Color 1 [15:11]	Color 1 Green [7:3]	[88:84]
Color 1 [10:08]	Color 1 Green [2:0]	[88:86]
Color 1 [07:03]	Color 1 Blue [7:3]	[83:79]
Color 1 [02:00]	Color 1 Blue [2:0]	[83:81]
Color 0 [31:27]	Color 0 Alpha [7:3]	[113:109]
Color 0 [26:24]	Color 0 Alpha [2:0]	[113:111]
Color 0 [23:17]	Color 0 Red [7:3]	[78:74]
Color 0 [18:16]	Color 0 Red [2:0]	[78:76]
Color 0 [15:11]	Color 0 Green [7:3]	[73:69]
Color 0 [10:08]	Color 0 Green [2:0]	[73:71]
Color 0 [07:03]	Color 0 Blue [7:3]	[68:64]
Color 0 [02:00]	Color 0 Blue [2:0]	[68:66]



LERP = 0 Decoding

When LERP = 0, a single 4-entry lookup table is formed using the three expanded colors, with the 4th entry defined as transparent black (ARGB=0,0,0,0). Each 2-bit Texel n Select field of the encoded CC_ALPHA block is used to index into a 32-bit A8R8G8B8 color from the table completing the decode of the CC_ALPHA block.

Table 6-19. FXT CC_ALPHA Interpolated Color Table (LERP=0)

Texel Select	Color	Alpha
0	Color0.RGB	Color0.Alpha
1	Color1.RGB	Color1.Alpha
2	Color2.RGB	Color2.Alpha
3	Black (RGB=0,0,0)	0

LERP = 1 Decoding

When LERP = 1, the three expanded colors are used to create two tables of four interpolated colors. The Color0,1 table is used as a lookup table for texel 0-15 indices, and the Color1,2 table used for texels 16-31 indices, as shown in the following figures:

Table 6-20. FXT CC_ALPHA Interpolated Color Table (LERP=1, Texels 0-15)

Texel 0-15 Select	Color ARGB
0	Color0.ARGB
1	$(2 * \text{Color0.ARGB} + \text{Color1.ARGB} + 1) / 3$
2	$(\text{Color0.ARGB} + 2 * \text{Color1.ARGB} + 1) / 3$
3	Color1.ARGB

Table 6-21. FXT CC_ALPHA Interpolated Color Table (LERP=1, Texels 16-31)

Texel 16-31 Select	Color ARGB
0	Color2.ARGB
1	$(2 * \text{Color2.ARGB} + \text{Color1.ARGB} + 1) / 3$
2	$(\text{Color2.ARGB} + 2 * \text{Color1.ARGB} + 1) / 3$
3	Color1.ARGB



6.4.2 BC Texture Formats

The hardware supports three “BCn” surface formats that divide surfaces (texture maps) into independent 4x4 texel blocks and stores compressed versions of these blocks in 1 or 2 QWord units. Note that non-power-of-2 dimensioned maps may require the surface to be padded out to the next multiple of four texels – here the pad texels are not referenced by the device.

An 8-byte (QWord) block encoding can be used if the source texture contains no transparency (is opaque) or if the transparency can be specified by a one-bit alpha. A 16-byte (DQWord) block encoding can be used to support source textures that require more than one-bit alpha: here the 1st QWord is used to encode the texel alpha values, and the 2nd QWord is used to encode the texel color values.

These three types of format are discussed in the following sections:

- Opaque and One-bit Alpha Textures (BC1)
- Opaque Textures (BC1_RGB)
- Textures with Alpha Channels (BC2-3)

Notes:

- Any single texture must specify that its data is stored as 64 or 128 bits per group of 16 texels. If 64-bit blocks—that is, format BC1—are used for the texture, it is possible to mix the opaque and one-bit alpha formats on a per-block basis within the same texture. In other words, the comparison of the unsigned integer magnitude of color_0 and color_1 is performed uniquely for each block of 16 texels.
- When 128-bit blocks are used, then the alpha channel must be specified in either explicit (format BC2) or interpolated mode (format BC3) for the entire texture. Note that as with color, once interpolated mode is selected then either 8 interpolated alphas or 6 interpolated alphas mode can be used on a block-by-block basis. Again the magnitude comparison of alpha_0 and alpha_1 is done uniquely on a block-by-block basis.

6.4.2.1 Opaque and One-bit Alpha Textures (BC1)

Texture format BC1 is for textures that are opaque or have a single transparent color. For each opaque or one-bit alpha block, two 16-bit R5G6B5 values and a 4x4 bitmap with 2-bits-per-pixel are stored. This totals 64 bits (1 QWord) for 16 texels, or 4-bits-per-texel.

In the block bitmap, there are two bits per texel to select between the four colors, two of which are stored in the encoded data. The other two colors are derived from these stored colors by linear interpolation.

The one-bit alpha format is distinguished from the opaque format by comparing the two 16-bit color values stored in the block. They are treated as unsigned integers. If the first color is greater than the second, it implies that only opaque texels are defined. This means four colors will be used to represent the texels. In four-color encoding, there are two derived colors and all four colors are equally distributed in



RGB color space. This format is analogous to R5G6B5 format. Otherwise, for one-bit alpha transparency, three colors are used and the fourth is reserved to represent transparent texels. Note that the color blocks in BC2-3 formats strictly use four colors, as the alpha values are obtained from the alpha block (the DX7 Direct3D reference rasterizer had a known bug that erroneously allowed 3-color BC2-3 color blocks).

In three-color encoding, there is one derived color and the fourth two-bit code is reserved to indicate a transparent texel (alpha information). This format is analogous to A1R5G5B5, where the final bit is used for encoding the alpha mask.

The following piece of pseudo-code illustrates the algorithm for deciding whether three- or four-color encoding is selected:

```
if (color_0 > color_1)
{
    // Four-color block: derive the other two colors.
    // 00 = color_0, 01 = color_1, 10 = color_2, 11 = color_3
    // These two bit codes correspond to the 2-bit fields
    // stored in the 64-bit block.
    color_2 = (2 * color_0 + color_1) / 3;
    color_3 = (color_0 + 2 * color_1) / 3;
}
else
{
    // Three-color block: derive the other color.
    // 00 = color_0, 01 = color_1, 10 = color_2,
    // 11 = transparent.
    // These two bit codes correspond to the 2-bit fields
    // stored in the 64-bit block.
    color_2 = (color_0 + color_1) / 2;
    color_3 = transparent;
}
```

The following tables show the memory layout for the 8-byte block. It is assumed that the first index corresponds to the y-coordinate and the second corresponds to the x-coordinate. For example, Texel[1][2] refers to the texture map pixel at (x,y) = (2,1).

Here is the memory layout for the 8-byte (64-bit) block:

Word Address	16-bit Word
0	Color_0
1	Color_1
2	Bitmap Word_0
3	Bitmap Word_1

Color_0 and Color_1 (colors at the two extremes) are laid out as follows:

Bits	Color
15:11	Red color component
10:5	Green color component
4:0	Blue color component



Bitmap Word_0 is laid out as follows:

Bits	Texel
1:0 (LSB)	Texel[0][0]
3:2	Texel[0][1]
5:4	Texel[0][2]
7:6	Texel[0][3]
9:8	Texel[1][0]
11:10	Texel[1][1]
13:12	Texel[1][2]
15:14	Texel[1][3]

Bitmap Word_1 is laid out as follows:

Bits	Texel
1:0 (LSB)	Texel[2][0]
3:2	Texel[2][1]
5:4	Texel[2][2]
7:6	Texel[2][3]
9:8	Texel[3][0]
11:10	Texel[3][1]
13:12	Texel[3][2]
15:14 (MSB)	Texel[3][3]

Example of Opaque Color Encoding

As an example of opaque encoding, we will assume that the colors red and black are at the extremes. We will call red color_0 and black color_1. There will be four interpolated colors that form the uniformly distributed gradient between them. To determine the values for the 4x4 bitmap, the following calculations are used:

```
00 ? color_0
01 ? color_1
10 ? 2/3 color_0 + 1/3 color_1
11 ? 1/3 color_0 + 2/3 color_1
```

Example of One-bit Alpha Encoding

This format is selected when the unsigned 16-bit integer, color_0, is less than the unsigned 16-bit integer, color_1. An example of where this format could be used is leaves on a tree to be shown against a blue sky. Some texels could be marked as transparent while three shades of green are still available for the leaves. Two of these colors fix the extremes, and the third color is an interpolated color.



The bitmap encoding for the colors and the transparency is determined using the following calculations:

```
00 ? color_0
01 ? color_1
10 ? 1/2 color_0 + 1/2 color_1
11 ? Transparent
```

6.4.2.2 Opaque Textures (BC1_RGB)

Texture format BC1_RGB is identical to BC1, with the exception that the One-bit Alpha encoding is removed. Color 0 and Color 1 are not compared, and the resulting texel color is derived strictly from the Opaque Color Encoding. The alpha channel defaults to 1.0.

6.4.2.3 Compressed Textures with Alpha Channels (BC2-3)

There are two ways to encode texture maps that exhibit more complex transparency. In each case, a block that describes the transparency precedes the 64-bit block already described. The transparency is either represented as a 4x4 bitmap with four bits per pixel (explicit encoding), or with fewer bits and linear interpolation analogous to what is used for color encoding.

The transparency block and the color block are laid out as follows:

Word Address	64-bit Block
3:0	Transparency block
7:4	Previously described 64-bit block

Explicit Texture Encoding

For explicit texture encoding (BC2 formats), the alpha components of the texels that describe transparency are encoded in a 4x4 bitmap with 4 bits per texel. These 4 bits can be achieved through a variety of means such as dithering or by simply using the 4 most significant bits of the alpha data. However they are produced, they are used just as they are, without any form of interpolation.

Note:

DirectDraw's compression method uses the 4 most significant bits.

The following tables illustrate how the alpha information is laid out in memory, for each 16-bit word.

This is the layout for Word 0:

Bits	Alpha
3:0 (LSB)	[0][0]
7:4	[0][1]
11:8	[0][2]
15:12 (MSB)	[0][3]



This is the layout for Word 1:

Bits	Alpha
3:0 (LSB)	[1][0]
7:4	[1][1]
11:8	[1][2]
15:12 (MSB)	[1][3]

This is the layout for Word 2:

Bits	Alpha
3:0 (LSB)	[2][0]
7:4	[2][1]
11:8	[2][2]
15:12 (MSB)	[2][3]

This is the layout for Word 3:

Bits	Alpha
3:0 (LSB)	[3][0]
7:4	[3][1]
11:8	[3][2]
15:12 (MSB)	[3][3]

Three-Bit Linear Alpha Interpolation

The encoding of transparency for the BC3 formats is based on a concept similar to the linear encoding used for color. Two 8-bit alpha values and a 4x4 bitmap with three bits per pixel are stored in the first eight bytes of the block. The representative alpha values are used to interpolate intermediate alpha values. Additional information is available in the way the two alpha values are stored. If alpha_0 is greater than alpha_1, then six intermediate alpha values are created by the interpolation. Otherwise, four intermediate alpha values are interpolated between the specified alpha extremes. The two additional implicit alpha values are 0 (fully transparent) and 255 (fully opaque).

The following pseudo-code illustrates this algorithm:

```
// 8-alpha or 6-alpha block?
if (alpha_0 > alpha_1) {
    // 8-alpha block: derive the other 6 alphas.
    // 000 = alpha_0, 001 = alpha_1, others are interpolated
    alpha_2 = (6 * alpha_0 + alpha_1) / 7; // bit code 010
    alpha_3 = (5 * alpha_0 + 2 * alpha_1) / 7; // Bit code 011
    alpha_4 = (4 * alpha_0 + 3 * alpha_1) / 7; // Bit code 100
    alpha_5 = (3 * alpha_0 + 4 * alpha_1) / 7; // Bit code 101
    alpha_6 = (2 * alpha_0 + 5 * alpha_1) / 7; // Bit code 110
}
```



```

    alpha_7 = (alpha_0 + 6 * alpha_1) / 7;          // Bit code 111
}
else { // 6-alpha block: derive the other alphas.
    // 000 = alpha_0, 001 = alpha_1, others are interpolated
    alpha_2 = (4 * alpha_0 + alpha_1) / 5;          // Bit code 010
    alpha_3 = (3 * alpha_0 + 2 * alpha_1) / 5;      // Bit code 011
    alpha_4 = (2 * alpha_0 + 3 * alpha_1) / 5;      // Bit code 100
    alpha_5 = (alpha_0 + 4 * alpha_1) / 5;          // Bit code 101
    alpha_6 = 0;                                    // Bit code 110
    alpha_7 = 255;                                  // Bit code 111
}

```

The memory layout of the alpha block is as follows:

Byte	Alpha
0	Alpha_0
1	Alpha_1
2	[0][2] (2 LSBs), [0][1], [0][0]
3	[1][1] (1 LSB), [1][0], [0][3], [0][2] (1 MSB)
4	[1][3], [1][2], [1][1] (2 MSBs)
5	[2][2] (2 LSBs), [2][1], [2][0]
6	[3][1] (1 LSB), [3][0], [2][3], [2][2] (1 MSB)
7	[3][3], [3][2], [3][1] (2 MSBs)

6.5 Video Pixel/Texel Formats

This section describes the “video” pixel/texel formats with respect to memory layout. See the Overlay chapter for a description of how the Y, U, V components are sampled.

6.5.1 Packed Memory Organization

Color components are all 8 bits in size for YUV formats. For YUV 4:2:2 formats each DWord will contain two pixels and only the byte order affects the memory organization.

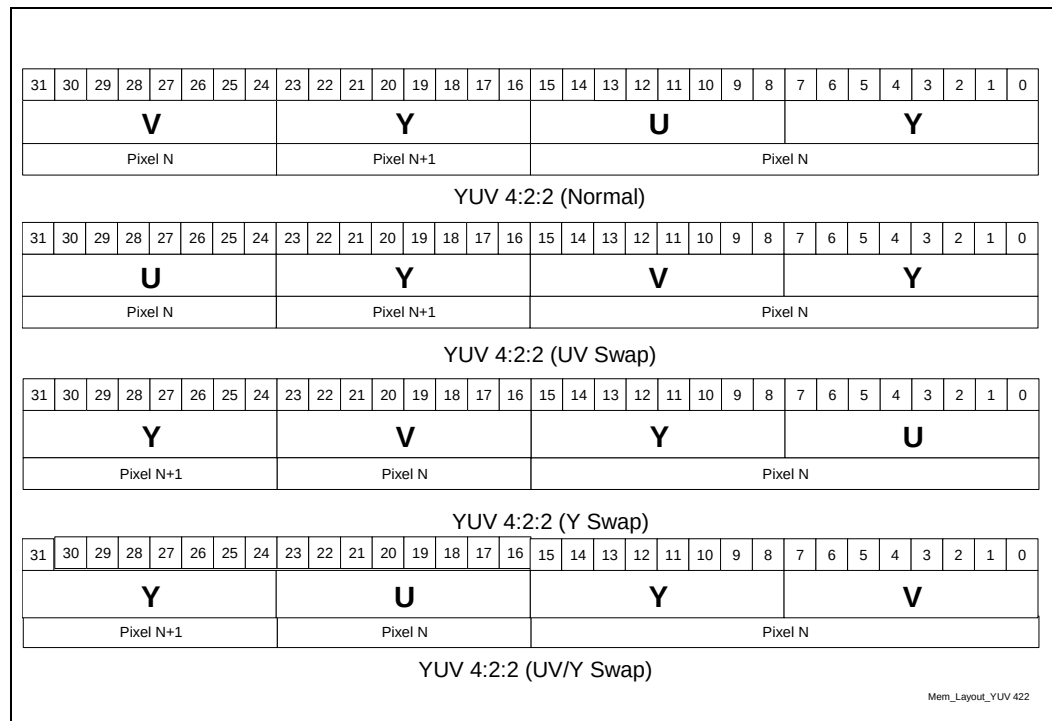
The following four YUV 4:2:2 surface formats are supported, listed with alternate names:

- YCRCB_NORMAL (UYVY) (R8G8_B8G8_UNORM)
- YCRCB_SWAPUVY (YUY2) (G8R8_G8B8_UNORM)
- YCRCB_SWAPUV
- YCRCB_SWAPY

The channels are mapped as follows:

Cr (V) Red
Y Green
Cb (U) Blue

Figure 6-2. Memory Layout of Packed YUV 4:2:2 Formats



6.5.2 Planar Memory Organization

Planar formats use what could be thought of as separate buffers for the three color components. Because there is a separate stride for the Y and U/V data buffers, several memory footprints can be supported.

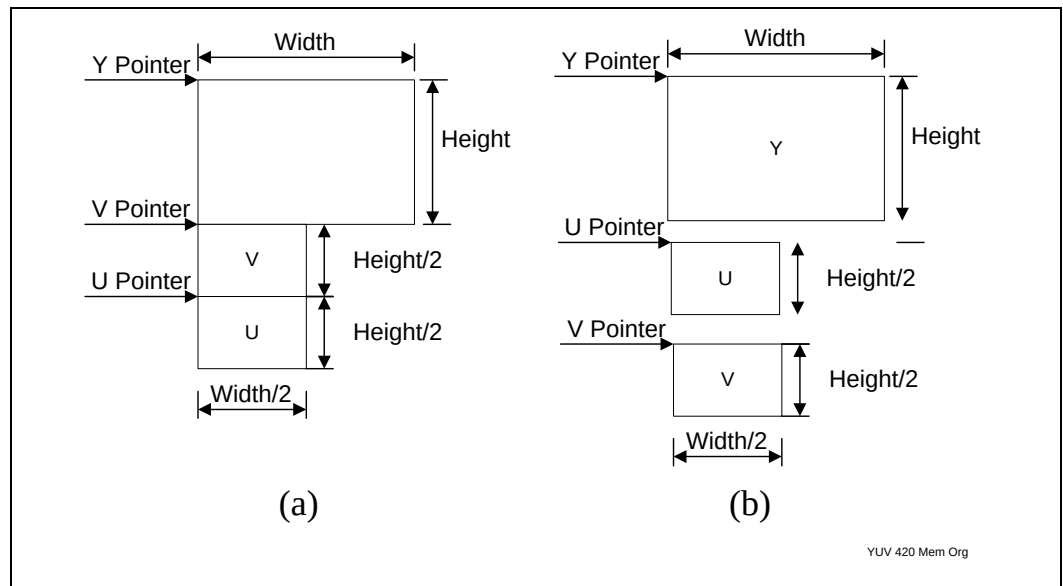
Note: There is no direct support for use of planar video surfaces as textures. The sampling engine can be used to operate on each of the 8bpp buffers separately (via a single-channel 8-bit format such as I8_UNORM). The U and V buffers can be written concurrently by using multiple render targets from the pixel shader. The Y buffer must be written in a separate pass due to its different size.

The following figure shows two types of memory organization for the YUV 4:2:0 planar video data:

1. The memory organization of the common YV12 data, where all three planes are contiguous and the strides of U and V components are half of that of the Y component.
2. An alternative memory structure that the addresses of the three planes are independent but satisfy certain alignment restrictions.

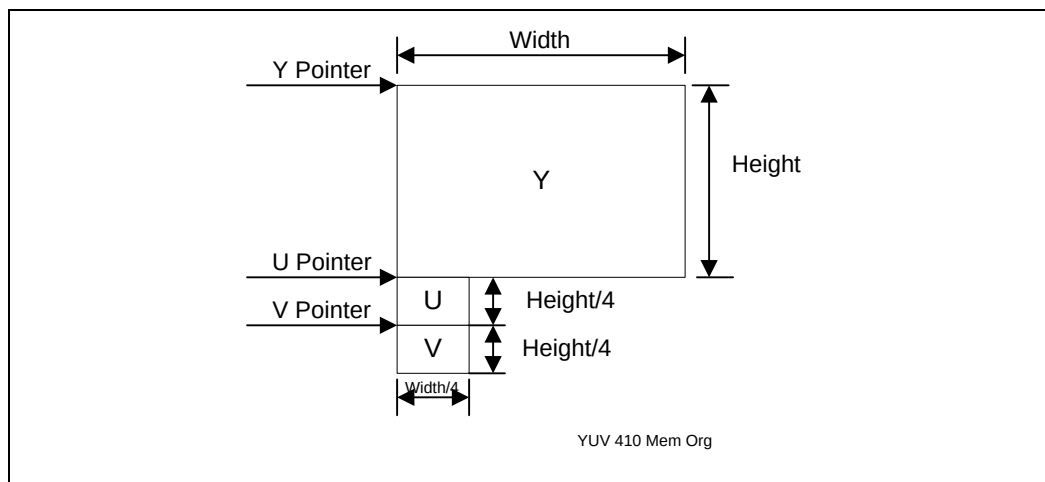


Figure 6-3. YUV 4:2:0 Format Memory Organization



The following figure shows memory organization of the planar YUV 4:1:0 format where the planes are contiguous. The stride of the U and V planes is a quarter of that of the Y plane.

Figure 6-4. YUV 4:1:0 Format Memory Organization



6.6 Surface Memory Organizations

See *Memory Interface Functions* chapter for a discussion of tiled vs. linear surface formats.

6.7 Graphics Translation Tables

The Graphics Translation Tables GTT (Graphics Translation Table, sometimes known as the global GTT) and PPGTT (Per-Process Graphics Translation Table) are memory-resident page tables containing an array of DWord Page Translation Entries (PTEs) used in mapping logical Graphics Memory addresses to physical memory addresses, and sometimes snooped system memory "PCI" addresses.

The graphics translation tables must reside in (unsnooped) system memory.

The base address (MM offset) of the GTT and the PPGTT are programmed via the PGTBL_CTL and PGTBL_CTL2 MI registers, respectively. The translation table base addresses must be 4KB aligned. The GTT size can be either 128KB, 256KB or 512KB (mapping to 128MB, 256MB, and 512MB aperture sizes respectively) and is physically contiguous. The global GTT should only be programmed via the range defined by GTTADR. The PPGTT is programmed directly in memory. The per-process GTT (PPGTT) size is controlled by the PGTBL_CTL2 register. The PPGTT can, in addition to the above sizes, also be 64KB in size (corresponding to a 64MB aperture). Refer to the GTT Range chapter for a bit definition of the PTE entries.



6.8 Hardware Status Page

The hardware status page is a naturally-aligned 4KB page residing in snoopable system memory. This page exists primarily to allow the device to report status via PCI master writes – thereby allowing the driver to read/poll WB memory instead of UC reads of device registers or UC memory.

The address of this page is programmed via the HWS_PGA MI register. The definition of that register (in *Memory Interface Registers*) includes a description of the layout of the Hardware Status Page.

6.9 Instruction Ring Buffers

Instruction ring buffers are the memory areas used to pass instructions to the device. Refer to the Programming Interface chapter for a description of how these buffers are used to transport instructions.

The RINGBUF register sets (defined in Memory Interface Registers) are used to specify the ring buffer memory areas. The ring buffer must start on a 4KB boundary and be allocated in linear memory. The length of any one ring buffer is limited to 2MB.

Note that “indirect” 3D primitive instructions (those that access vertex buffers) must reside in the same memory space as the vertex buffers.

6.10 Instruction Batch Buffers

Instruction batch buffers are contiguous streams of instructions referenced via an MI_BATCH_BUFFER_START and related instructions (see Memory Interface Instructions, Programming Interface). They are used to transport instructions external to ring buffers.

Note that batch buffers should not be mapped to snoopable SM (PCI) addresses. The device will treat these as MainMemory (MM) address, and therefore not snoop the CPU cache.

The batch buffer must be QWord aligned and a multiple of QWords in length. The ending address is the address of the last valid QWord in the buffer. The length of any single batch buffer is “virtually unlimited” (i.e., could theoretically be 4GB in length).

6.11 Display, Overlay, Cursor Surfaces

These surfaces are memory image buffers (planes) used to refresh a display device in non-VGA mode. See the Display chapter for specifics on how these surfaces are defined/used.

6.12 2D Render Surfaces

These surfaces are used as general source and/or destination operands in 2D Blt operations.



Note that the device provides no coherency between 2D render surfaces and the texture cache – i.e., the texture cache must be explicitly invalidated prior to the use of a texture that has been modified via the Blt engine.

See the 2D Instruction and 2D Rendering chapters for specifics on how these surfaces are used, restrictions on their size, placement, etc.

6.13 2D Monochrome Source

These 1bpp surfaces are used as source operands to certain 2D Blt operations, where the Blt engine expands the 1bpp source into the required color depth.

The device uses the texture cache to store monochrome sources. There is no mechanism to maintain coherency between 2D render surfaces and (texture)-cached monochrome sources, software is required to explicitly invalidate the texture cache before using a memory-based monochrome source that has been modified via the Blt engine. (Here the assumption is that SW enforces memory-based monochrome source surfaces as read-only surfaces).

See the 2D Instruction and 2D Rendering chapters for specifics on how these surfaces are used, restrictions on their size, placement, coherency rules, etc.

6.14 2D Color Pattern

Color pattern surfaces are used as special pattern operands in 2D Blt operations.

The device uses the texture cache to store color patterns. There is no mechanism to maintain coherency between 2D render surfaces and (texture)-cached color patterns, software is required to explicitly invalidate the texture cache before using a memory-based color pattern that has been modified via the Blt engine. (Here the assumption is that SW enforces memory-based color pattern surfaces as read-only surfaces).

See the *2D Instruction* and *2D Rendering* chapters for specifics on how these surfaces are used, restrictions on their size, placement, etc.

6.15 3D Color Buffer (Destination) Surfaces

3D Color buffer surfaces are used to hold per-pixel color values for use in the 3D pipeline. Note that the 3D pipeline always requires a Color buffer to be defined.

Refer to Non-Video Pixel/Texel Formats section in this chapter for details on the Color buffer pixel formats. Refer to the 3D Instruction and 3D Rendering chapters for details on the usage of the Color Buffer.

The Color buffer is defined as the BUFFERID_COLOR_BACK memory buffer via the 3DSTATE_BUFFER_INFO instruction. That buffer can be mapped to LM, SM (snooped or unsnooped) and can be linear or tiled. When both the Depth and Color buffers are tiled, the respective Tile Walk directions must match.

When a linear Color and a linear Depth buffers are used together:

1. They may have different pitches, though both pitches must be a multiple of 32 bytes.
2. They must be co-aligned with a 32-byte region.



6.16 3D Depth Buffer Surfaces

Depth buffer surfaces are used to hold per-pixel depth values and per-pixel stencil values for use in the 3D pipeline. Note that the 3D pipeline does not require a Depth buffer to be allocated, though a Depth buffer is required to perform (non-trivial) Depth Test and Stencil Test operations.

The following table summarizes the possible formats of the Depth buffer. Refer to Depth Buffer Formats section in this chapter for details on the pixel formats. Refer to the *Windower* and *DataPort* chapters for details on the usage of the Depth Buffer.

Table 6-22. Depth Buffer Formats

DepthBufferFormat / DepthComponent	bpp	Description
D32_FLOAT_S8X24_UINT	64	32-bit floating point Z depth value in first DWord, 8-bit stencil in upper byte of second DWord
D32_FLOAT	32	32-bit floating point Z depth value
D24_UNORM_S8_UINT	32	24-bit fixed point Z depth value in lower 3 bytes, 8-bit stencil value in upper byte
D16_UNORM	16	16-bit fixed point Z depth value

The Depth buffer is specified via the 3DSTATE_DEPTH_BUFFER command. See the description of that instruction in *Windower* for restrictions.

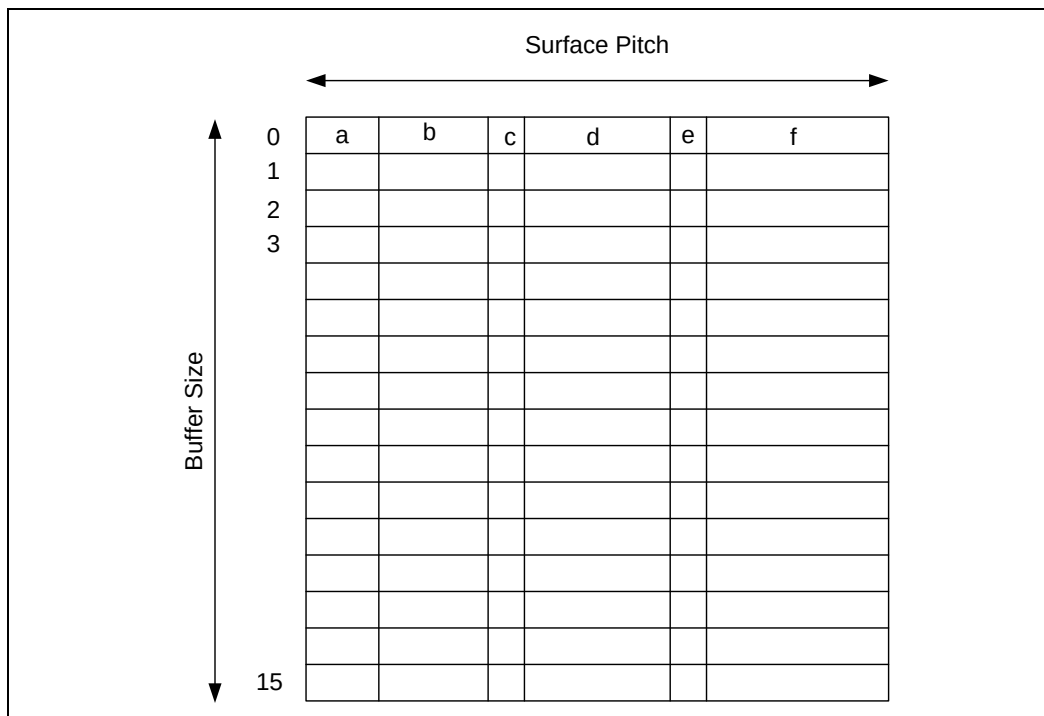
6.17 Surface Layout

This section describes the formats of surfaces and data within the surfaces.

6.17.1 Buffers

A buffer is an array of structures. Each structure contains up to 2048 bytes of elements. Each element is a single surface format using one of the supported surface formats depending on how the surface is being accessed. The surface pitch state for the surface specifies the size of each structure in bytes.

The buffer is stored in memory contiguously with each element in the structure packed together, and the first element in the next structure immediately following the last element of the previous structure. Buffers are supported only in linear memory.



6.17.2 1D Surfaces

One-dimensional surfaces are identical to 2D surfaces with height of one. Arrays of 1D surfaces are also supported. Please refer to the 2D Surfaces section for details on how these surfaces are stored.

6.17.3 2D Surfaces

Surfaces that comprise texture mip-maps are stored in a fixed “monolithic” format and referenced by a single base address. The base map and associated mipmaps are located within a single rectangular area of memory identified by the base address of the upper left corner and a pitch. The base address references the upper left corner of the base map. The pitch must be specified at least as large as the widest mip-map. In some cases it must be wider; see the section on Minimum Pitch below.

These surfaces may be overlapped in memory and must adhere to the following memory organization rules:

- For non-compressed texture formats, each mipmap must start on an even row within the monolithic rectangular area. For 1-texel-high mipmaps, this may require a row of padding below the previous mipmap. This restriction does not apply to any compressed texture formats: i.e., each subsequent (lower-res) compressed mipmap is positioned directly below the previous mipmap.
- Vertical alignment restrictions vary with memory tiling type: 1 DWord for linear, 16-byte (DQWord) for tiled. (Note that tiled mipmaps are *not* required to start at the left edge of a tile row).



6.17.3.1 Computing MIP level sizes

Map width and height specify the size of the largest MIP level (LOD 0). Less detailed LOD level ($i+1$) sizes are determined by dividing the width and height of the current (i) LOD level by 2 and truncating to an integer (floor). This is equivalent to shifting the width/height by 1 bit to the right and discarding the bit shifted off. The map height and width are clamped on the low side at 1.

In equations, the width and height of an LOD “ L ” can be expressed as:

$$W_L = ((width \gg L) > 0 ? width \gg L : 1)$$

$$H_L = ((height \gg L) > 0 ? height \gg L : 1)$$

6.17.3.2 Base Address for LOD Calculation

It is conceptually easier to think of the space that the map uses in Cartesian space (x, y), where x and y are in units of texels, with the upper left corner of the base map at $(0, 0)$. The final step is to convert from Cartesian coordinates to linear addresses as documented at the bottom of this section.

It is useful to think of the concept of “stepping” when considering where the next MIP level will be stored in rectangular memory space. We either step down or step right when moving to the next higher LOD.

- for MIPLAYOUT_RIGHT maps:
 - step right when moving from LOD 0 to LOD 1
 - step down for all of the other MIPs
- for MIPLAYOUT_BELOW maps:
 - step down when moving from LOD 0 to LOD 1
 - step right when moving from LOD 1 to LOD 2
 - step down for all of the other MIPs

To account for the cache line alignment required, we define i and j as the width and height, respectively, of an *alignment unit*. This alignment unit is defined below. We then define lower-case w_L and h_L as the padded width and height of LOD “ L ” as follows:

$$w_L = i * \text{ceil}\left(\frac{W_L}{i}\right)$$

$$h_L = j * \text{ceil}\left(\frac{H_L}{j}\right)$$

Equations to compute the upper left corner of each MIP level are then as follows:



for *MIPLAYOUT_RIGHT* maps:

$$LOD_0 = (0,0)$$

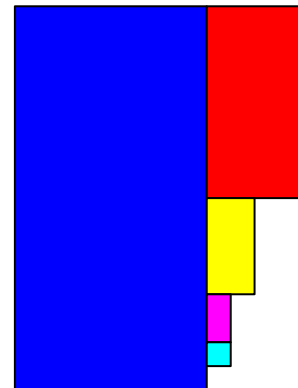
$$LOD_1 = (w_0,0)$$

$$LOD_2 = (w_0, h_1)$$

$$LOD_3 = (w_0, h_1 + h_2)$$

$$LOD_4 = (w_0, h_1 + h_2 + h_3)$$

...



for *MIPLAYOUT_BELOW* maps:

$$LOD_0 = (0,0)$$

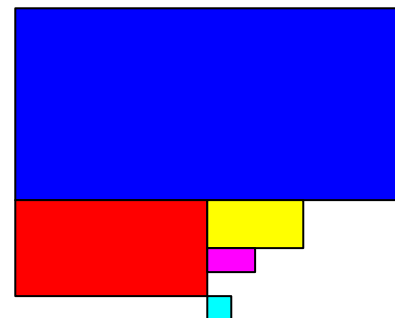
$$LOD_1 = (0, h_0)$$

$$LOD_2 = (w_1, h_0)$$

$$LOD_3 = (w_1, h_0 + h_2)$$

$$LOD_4 = (w_1, h_0 + h_2 + h_3)$$

...



6.17.3.3 Minimum Pitch

For *MIPLAYOUT_RIGHT* maps, the minimum pitch must be calculated before choosing a fence to place the map within. This is approximately equal to 1.5x the pitch required by the base map, with possible adjustments made for cache line alignment. For *MIPLAYOUT_BELOW* and *MIPLAYOUT_LEGACY* maps, the minimum pitch required is equal to that required by the base (LOD 0) map.

A safe but simple calculation of minimum pitch is equal to 2x the pitch required by the base map for *MIPLAYOUT_RIGHT* maps. This ensures that enough pitch is available, and since it is restricted to *MIPLAYOUT_RIGHT* maps, not much memory is wasted. It is up to the driver (hardware independent) whether to use this simple determination of pitch or a more complex one.



6.17.3.4 Alignment Unit Size

The following table indicates the *i* and *j* values that should be used for each map format. Note that the compressed formats are padded to a full compression cell.

Table 6-23. Alignment Units for Texture Maps

map format	alignment unit width “ <i>j</i> ”	alignment unit height “ <i>j</i> ”
YUV 4:2:2 formats	4	2
BC1-5	4	4
FXT1	8	4
all other formats	4	2

6.17.3.5 Cartesian to Linear Address Conversion

A set of variables are defined in addition to the *i* and *j* defined above.

- *b* = bytes per texel of the native map format (0.5 for BC1, FXT1, and 4-bit surface format, 2.0 for YUV 4:2:2, others aligned to surface format)
- *t* = texel rows / memory row (4 for BC1-3 and FXT1, 1 for all other formats)
- *p* = pitch in bytes (equal to pitch in dwords * 4)
- *B* = base address in bytes (address of texel 0,0 of the base map)
- *x*, *y* = cartesian coordinates from the above calculations in units of texels (assumed that *x* is always a multiple of *i* and *y* is a multiple of *j*)
- *A* = linear address in bytes

$$A = B + \frac{yp}{t} + xbt$$

This calculation gives the linear address in bytes for a given MIP level (taking into account L1 cache line alignment requirements).

6.17.3.6 Compressed Mipmap Layout

Mipmaps of textures using compressed (BCn, FXT) texel formats are also stored in a monolithic format. The compressed mipmaps are stored in a similar fashion to uncompressed mipmaps, with each block of source (uncompressed) texels represented by a 1 or 2 QWord compressed block. The compressed blocks occupy the same logical positions as the texels they represent, where each row of compressed blocks represent a 4-high row of uncompressed texels. The format of the blocks is preserved, i.e., there is no “intermediate” format as required on some other devices.

The following exceptions apply to the layout of compressed (vs. uncompressed) mipmaps:

- Mipmaps are not required to start on even rows, therefore each successive mip level is located on the texel row immediately below the last row of the previous mip level. Pad rows are neither required nor allowed.
- The dimensions of the mip maps are first determined by applying the sizing algorithm presented in Non-Power-of-Two Mipmaps above. Then, if necessary, they are padded out to compression block boundaries.



6.17.3.7 Surface Arrays

Both 1D and 2D surfaces can be specified as an array. The only difference in the surface state is the presence of a depth value greater than one, indicating multiple array “slices”.

A value *QPitch* is defined which indicates the worst-case size for one slice in the texture array. This *QPitch* is multiplied by the array index to and added to the surface base address to determine the base address for that slice. Within the slice, the map is stored identically to a MIPLAYOUT_BELOW 2D surface. *MIPLAYOUT_BELOW is the only format supported by 1D non-arrays and both 2D and 1D arrays, the programming of the MIP Map Layout Mode state variable is ignored when using a TextureArray.*

The following equation is used for surface formats other than compressed textures:

$$QPitch = (h_0 + h_1 + 11j) * Pitch$$

The input variables in this equation are defined in sections above.

The equation for compressed textures (BC* and FXT1 surface formats) follows:

$$QPitch = \frac{(h_0 + h_1 + 11j)}{4} * Pitch$$

6.17.4 Cube Surfaces

The 3D pipeline supports *cubic environment maps*, conceptually arranged as a cube surrounding the origin of a 3D coordinate system aligned to the cube faces. These maps can be used to supply texel (color/alpha) data of the environment in any direction from the enclosed origin, where the direction is supplied as a 3D “vector” texture coordinate. These cube maps can also be mipmapped.

Each texture map level is represented as a group of six, square *cube face* texture surfaces. The faces are identified by their relationship to the 3D texture coordinate system. The subsections below describe the cube maps as described at the API as well as the memory layout dictated by the hardware.

6.17.4.1 Hardware Cube Map Layout

The cube face textures are stored in the same way as 3D surfaces are stored (see section 6.17.5 for details). For cube surfaces, however, the depth is equal to the number of faces (always 6) and is not reduced for each MIP. The equation for D_L is replaced with the following for cube surfaces:

$$D_L = 6$$

The “q” coordinate is replaced with the face identifier as follows:



"q" coordinate	face
0	+x
1	-x
2	+y
3	-y
4	+z
5	-z

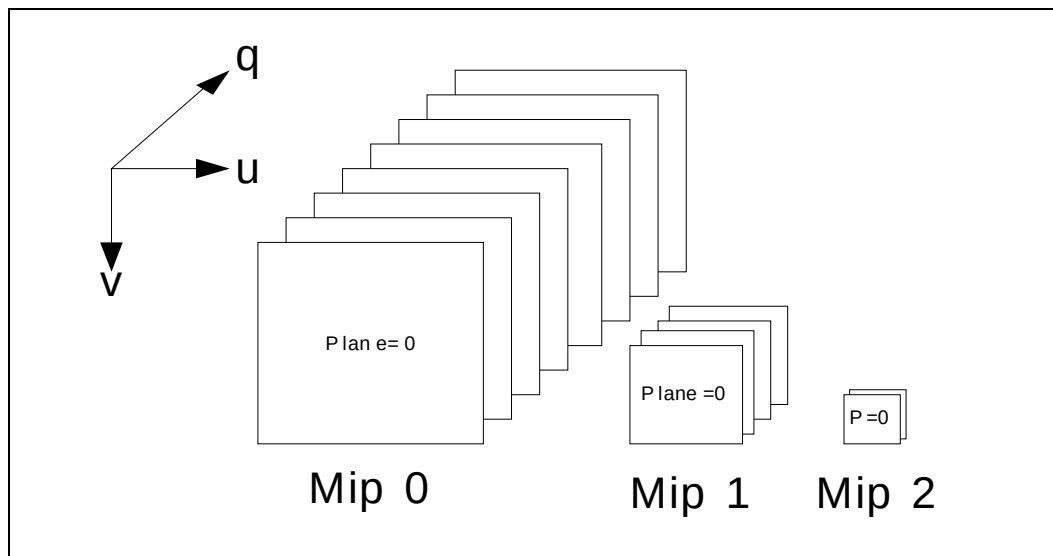
6.17.4.2 Restrictions

- The cube map memory layout is the same whether or not the cube map is mip-mapped, and whether or not all six faces are "enabled", though the memory backing disabled faces or non-supplied levels can be used by software for other purposes.
- The cube map faces all share the same **Surface Format**

6.17.5 3D Surfaces

Multiple texture map surfaces (and their respective mipmap chains) can be arranged into a structure known as a Texture3D (volume) texture. A volume texture map consists of many *planes* of 2D texture maps. See *Sampler* for a description of how volume textures are used.

Figure 6-5. Volume Texture Map





Note that the number of planes defined at each successive mip level is halved. Volumetric texture maps are stored as follows. All of the LOD=0 q-planes are stacked vertically, then below that, the LOD=1 q-planes are stacked two-wide, then the LOD=2 q-planes are stacked four-wide below that, and so on.

The width, height, and depth of LOD “L” are as follows:

$$W_L = ((width \gg L) > 0 ? width \gg L : 1)$$

$$H_L = ((height \gg L) > 0 ? height \gg L : 1)$$

This is the same as for a regular texture. For volume textures we add:

$$D_L = ((depth \gg L) > 0 ? depth \gg L : 1)$$

Cache-line aligned width and height are as follows, with i and j being a function of the map format as shown in Table 6-23.

$$w_L = i * \text{ceil}\left(\frac{W_L}{i}\right)$$

$$h_L = j * \text{ceil}\left(\frac{H_L}{j}\right)$$

Note that it is not necessary to cache-line align in the “depth” dimension (i.e. lower case “d”).

The following equations for $LOD_{L,q}$ give the base address Cartesian coordinates for the map at LOD L and depth q.

$$LOD_{0,q} = (0, q * h_0)$$

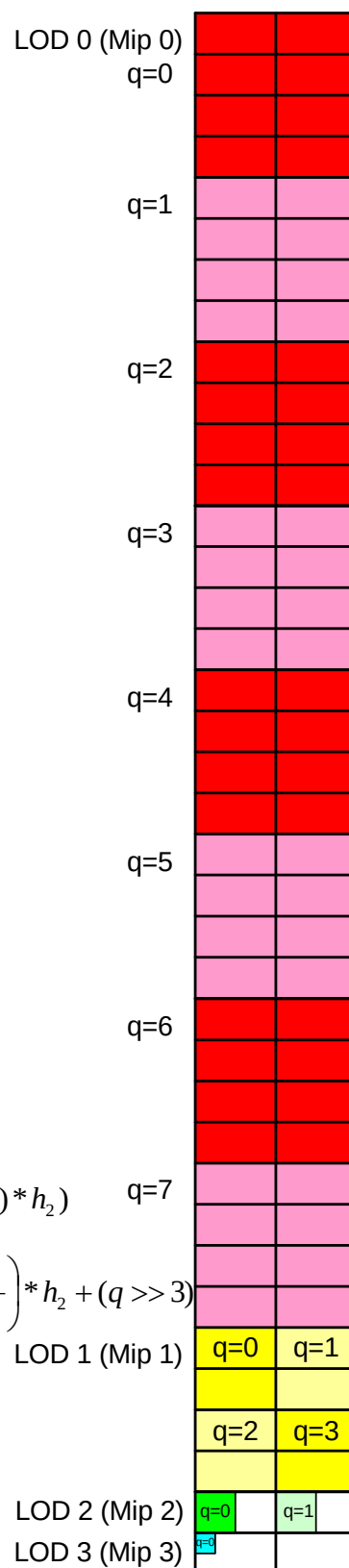
$$LOD_{1,q} = ((q \% 2) * w_1, D_0 * h_0 + (q >> 1) * h_1)$$

$$LOD_{2,q} = ((q \% 4) * w_2, D_0 * h_0 + \text{ceil}\left(\frac{D_1}{2}\right) * h_1 + (q >> 2) * h_2)$$

$$LOD_{3,q} = ((q \% 8) * w_3, D_0 * h_0 + \text{ceil}\left(\frac{D_1}{2}\right) * h_1 + \text{ceil}\left(\frac{D_2}{4}\right) * h_2 + (q >> 3) * h_3)$$

...

These values are then used as “base addresses” and the 2D MIP Map equations are used to compute the location within each LOD/q map.





6.17.5.1 Minimum Pitch

The minimum pitch required to store the 3D map may in some cases be greater than the minimum pitch required by the LOD=0 map. This is due to cache line alignment requirements that may impact some of the MIP levels requiring additional spacing in the horizontal direction.



6.18 Surface Padding Requirements

6.18.1 Sampling Engine Surfaces

The sampling engine accesses texels outside of the surface if they are contained in the same cache line as texels that are within the surface. These texels will not participate in any calculation performed by the sampling engine and will not affect the result of any sampling engine operation, however if these texels lie outside of defined pages in the GTT, a GTT error will result when the cache line is accessed. In order to avoid these GTT errors, “padding” at the bottom and right side of a sampling engine surface is sometimes necessary.

It is possible that a cache line will straddle a page boundary if the base address or pitch is not aligned. All pages included in the cache lines that are part of the surface must map to valid GTT entries to avoid errors. To determine the necessary padding on the bottom and right side of the surface, refer to the table in Section 6.17.3.4 for the *i* and *j* parameters for the surface format in use. The surface must then be extended to the next multiple of the alignment unit size in each dimension, and all texels contained in this extended surface must have valid GTT entries.

For example, suppose the surface size is 15 texels by 10 texels and the alignment parameters are *i*=4 and *j*=2. In this case, the extended surface would be 16 by 10. Note that these calculations are done in texels, and must be converted to bytes based on the surface format being used to determine whether additional pages need to be defined.

For buffers, which have no inherent “height,” padding requirements are different. A buffer must be padded to the next multiple of 256 array elements, with an additional 16 bytes added beyond that to account for the L1 cache line.

For cube surfaces, an additional two rows of padding are required at the bottom of the surface. This must be ensured regardless of whether the surface is stored tiled or linear. This is due to the potential rotation of cache line orientation from memory to cache.

For compressed textures (BC* and FXT1 surface formats), padding at the bottom of the surface is to an even compressed row, which is equal to a multiple of 8 uncompressed texel rows. Thus, for padding purposes, these surfaces behave as if *j* = 8 only for surface padding purposes. The value of 4 for *j* still applies for mip level alignment and QPitch calculation.

6.18.2 Render Target and Media Surfaces

The data port accesses data (pixels) outside of the surface if they are contained in the same cache request as pixels that are within the surface. These pixels will not be returned by the requesting message, however if these pixels lie outside of defined pages in the GTT, a GTT error will result when the cache request is processed. In order to avoid these GTT errors, “padding” at the bottom of the surface is sometimes necessary.

If the surface contains an odd number of rows of data, a final row below the surface must be allocated. If the surface will be accessed in field mode (**Vertical Stride** = 1), enough additional rows below the surface must be allocated to make the extended surface height (including the padding) a multiple of 4.



6.19 Logical Context Data

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

Logical Contexts are aligned to 256-byte boundaries.

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

6.19.1 Overall Context Layout

6.19.1.1 Per-Process GTT and Run Lists Disabled

For this case (which is the only case for [DevBW] and [DevCL]), the entire context image consists of the *Register/State Context*, including the pipelined state section.

6.19.2 Register / State Context

The following table describes the *device-dependent* layout of a logical context in memory.

DWord	Bits	State Field
MEMORY INTERFACE STATE		
00h	31:0	MI_Noop
01h	31:29	Instruction Type = MI_INSTRUCTION = 0h
	28:23	MI Instruction Opcode = MI_LOAD_REGISTER_IMM = 22h
	22:12	Reserved: MBZ
	11:8	Byte Write Disables: This field specifies which bytes of the Data DWord are not to be written to the DWord offset specified in <i>Register Offset</i> . Format = Enable[4] (bit 8 corresponds to Data DWord [7:0]). Range = Must specify a valid register write operation <i>This field will always be written as Fh on context saves.</i>
	7:6	Reserved: MBZ
	5:0	DWord Length (Excludes DWord 0,1) = 2bh (dec_44)
02h	31:0	CACHE_MODE_0 Address
03h	31:0	CACHE_MODE_0 Data
04h	31:0	CACHE_MODE_1 Address
05h	31:0	CACHE_MODE_1 Data
06h	31:0	MI_ARB_STATE Address
07h	31:0	MI_ARB_STATE Data
08h	31:0	INSTPM Address
09h	31:0	INSTPM Data
0Ah-29h	31:0	Reserved
2Ah	31:0	PS_DEPTH_COUNT Lower Address
2Bh	31:0	PS_DEPTH_COUNT Lower Data



DWord	Bits	State Field
2Ch	31:0	PS_DEPTH_COUNT Upper Address
2Dh	31:0	PS_DEPTH_COUNT Upper Data
2Eh	31:0	MI_Noop
2Fh	31:0	MI_Noop
PIPELINE_SELECT		
30h	31:29	Instruction Type = GFXPIPE = 3h
	28:23	3D Instruction Opcode = PIPELINE_SELECT GFXPIPE[28:27 = 0h, 26:24 = 1h, 23:16 = 04h] (Non-pipelined)
	22:1	Reserved: MBZ
	0	0: 3D pipeline is selected 1: Media pipeline is selected
CS_URB_STATE		
31h	31:29	Command Type = GFXPIPE = 3h
	28:16	3D Command Opcode = CS_URB_STATE GFXPIPE[28:27 = 0h, 26:24 = 0h, 23:16 = 01h] (Pipelined)
	15:8	Reserved : MBZ
	7:0	DWord Length (excludes DWords 0,1) = 0
32h	31:9	Reserved : MBZ
	8:4	URB Entry Allocation Size
	3	Reserved: MBZ
	2:0	Number of URB Entries
URB_FENCE		
33h	31:29	Instruction Type = GFXPIPE = 3h
	28:16	3D Instruction Opcode = URB_FENCE GFXPIPE[28:27 = 0h, 26:24 = 0h, 23:16 = 00h] (Pipelined)
	15:14	Reserved : MBZ
	13	ModifyEnable(CS Fence)
	12	ModifyEnable(VFE Fence)
	11	ModifyEnable(SF Fence)
	10	ModifyEnable(CLIP Fence)
	9	ModifyEnable(GS Fence)
	8	ModifyEnable(VS Fence)
	7:0	DWord Length (Excludes DWords 0,1) = 1
34h	31:30	Reserved : MBZ
	29:20	CLP Fence
	19:10	GS Fence
	9:0	VS Fence
35h	31:30	Reserved : MBZ
	29:20	CS Fence
	19:10	VFE Fence
	9:0	SF Fence
CONSTANT_BUFFER		
36h	31:29	Command Type = GFXPIPE = 3h
	28:16	3D Command Opcode = CONSTANT_BUFFER GFXPIPE[28:27 = 0h, 26:24 = 0h, 23:16 = 02h] (Pipelined)



DWord	Bits	State Field
	15:9	Reserved : MBZ
	8	Valid (Saved as <i>clear</i> since CONSTANT_BUFFER is saved later)
	7:0	DWord Length (excludes DWords 0,1) = 0
37h	31:6	Buffer Starting Address
	5:0	Buffer Length
STATE_BASE_ADDRESS		
38h	31:29	Command Type = GFXPIPE = 3h
	28:16	3D Command Opcode = STATE_BASE_ADDRESS GFXPIPE[28:27 = 0h, 26:24 = 1h, 23:16 = 01h] (Nonpipelined)
	15:8	Reserved : MBZ
	7:0	DWord Length (Excludes DWords 0,1) = 4
39h	31:12	General State Base Address
	11:1	Reserved : MBZ
	0	Modify Enable
3Ah	31:12	Surface State Base Address
	11:1	Reserved : MBZ
	0	Modify Enable
3Bh	31:12	Indirect Object Base Address
	11:1	Reserved : MBZ
	0	Modify Enable
3Ch	31:12	General State Access Upper Bound
	11:1	Reserved : MBZ
	0	Modify Enable
3Dh	31:12	Indirect Object Access Upper Bound
	11:1	Reserved: MBZ
	0	Modify Enable
STATE_SIP		
3Eh	31:29	Command Type = GFXPIPE = 3h
	28:16	Command Opcode = STATE_SIP GFXPIPE[28:27 = 0h, 26:24 = 1h, 23:16 = 02h] (Non-Pipelined)
	15:8	Reserved : MBZ
	7:0	DWord Length (Excludes DWords 0,1) = 0
3Fh	31:4	System Instruction Pointer (SIP)
	3:0	Reserved : MBZ
3DSTATE_DRAWING_RECTANGLE		
40h	31:29	Instruction Type = GFXPIPE = 3h
	28:16	3D Instruction Opcode = 3DSTATE_DRAWING_RECTANGLE GFXPIPE[28:27 = 3h, 26:24 = 1h, 23:16 = 00h] (Non-Pipelined)
	15:0	DWord Length (Excludes DWord 0,1) = 2



DWord	Bits	State Field
41h	31:16	Clipped Drawing Rectangle Y Min
	15:0	Clipped Drawing Rectangle X Min
42h	31:16	Clipped Drawing Rectangle Y Max
	15:0	Clipped Drawing Rectangle X Max
43h	31:16	Drawing Rectangle Origin Y
	15:0	Drawing Rectangle Origin X
3DSTATE_DEPTH_BUFFER		
44h	31:29	Instruction Type = GFXPIPE = 3h
	28:16	3D Instruction Opcode = 3DSTATE_DEPTH_BUFFER GFXPIPE[28:27 = 3h, 26:24 = 1h, 23:16 = 05h] (Non-Pipelined)
	15:0	DWord Length (excl. DWord 0,1) = 3
45h	31:29	Surface Type
	28	Reserved: MBZ
	27	Tiled Surface
	26	Tile Walk
	25	Depth Buffer Coordinate Offset Disable
	24:21	Reserved : MBZ
	20:18	Surface Format
	17:0	Surface Pitch
46h	31:0	Surface Base Address
47h	31:19	Height
	18:6	Width
	5:2	LOD
	1	MIP Map Layout Mode
	0	Reserved : MBZ
48h	31:21	Depth
	20:12	Minimum Array Element
	11:0	Reserved : MBZ
3DSTATE_CHROMA_KEY (INDEX_0)		
49h	31:29	Instruction Type = 3D_INSTRUCTION = 3h
	28:16	3D Instruction Opcode = 3DSTATE_CHROMA_KEY GFXPIPE[28:27 = 3h, 26:24 = 1h; 23:16 = 04h] (Non-Pipelined)
	15:0	DWord Length (Excludes DWords 0,1) = 2
4Ah	31:30	ChromaKey Table Index = 0
	29:0	Reserved: MBZ
4Bh	31:0	ChromaKey Low Value
4Ch	31:0	ChromaKey High Value
3DSTATE_CHROMA_KEY (INDEX_1)		
4Dh	31:29	Instruction Type = 3D_INSTRUCTION = 3h
	28:16	3D Instruction Opcode = 3DSTATE_CHROMA_KEY GFXPIPE[28:27 = 3h, 26:24 = 1h; 23:16 = 04h] (Non-Pipelined)
	15:0	DWord Length (Excludes DWords 0,1) = 2
4Eh	31:30	ChromaKey Table Index = 1
	29:0	Reserved: MBZ
4Fh	31:0	ChromaKey Low Value
50h	31:0	ChromaKey High Value
3DSTATE_CHROMA_KEY (INDEX_2)		
51h	31:29	Instruction Type = 3D_INSTRUCTION = 3h
	28:16	3D Instruction Opcode = 3DSTATE_CHROMA_KEY GFXPIPE[28:27 = 3h, 26:24 = 1h; 23:16 = 04h] (Non-Pipelined)
	15:0	DWord Length (Excludes DWords 0,1) = 2
52h	31:30	ChromaKey Table Index = 2
	29:0	Reserved: MBZ



DWord	Bits	State Field
53h	31:0	ChromaKey Low Value
54h	31:0	ChromaKey High Value
3DSTATE_CHROMA_KEY (INDEX_3)		
55h	31:29	Instruction Type = 3D_INSTRUCTION = 3h
	28:16	3D Instruction Opcode = 3DSTATE_CHROMA_KEY GFXPIPE[28:27 = 3h, 26:24 = 1h, 23:16 = 04h] (Non-Pipelined)
	15:0	DWord Length (Excludes DWords 0,1) = 2
56h	31:30	ChromaKey Table Index = 3
	29:0	Reserved: MBZ
57h	31:0	ChromaKey Low Value
58h	31:0	ChromaKey High Value
3D State Constant Color		
59h	31:29	Instruction Type = GFXPIPE = 3h
	28:16	3D Instruction Opcode = 3DSTATE_CONSTANT_COLOR GFXPIPE[28:27 = 3h, 26:24 = 1h, 23:16 = 01h] (Non-Pipelined)
	15:0	DWord Length (excl. DWord 0,1) = 3
5Ah	31:0	Blend Constant Color Red
5Bh	31:0	Blend Constant Color Blue
5Ch	31:0	Blend Constant Color Green
5Dh	31:0	Blend Constant Color Alpha
3DSTATE_GLOBAL_DEPTH_OFFSET_CLAMP		
5Eh	31:29	Instruction Type = 3D_INSTRUCTION = 3h
	28:16	3D Instruction Opcode = 3DSTATE_GLOABL_DEPTH_OFFSET_CLAMP GFXPIPE[28:27 = 3h, 26:24 = 1h, 23:16 = 09h] (Non-Pipelined)
	15:0	DWord Length (excl. DWord 0,1) = 0
5Fh	31:0	Global Depth Offset Clamp
3DSTATE_POLY_STIPPLE_OFFSET		
60h	31:29	Instruction Type = 3D_INSTRUCTION = 3h
	28:16	3D Instruction Opcode = 3DSTATE_POLY_STIPPLE_OFFSET GFXPIPE[28:27 = 3h, 26:24 = 1h, 23:16 = 06h] (Non-Pipelined)
	15:0	DWord Length (excl. DWord 0,1) = 0
61h	31:13	Reserved: MBZ
	12:8	Polygon Stipple X Offset
	7:5	Reserved: MBZ
	4:0	Polygon Stipple Y Offset
3DSTATE_LINE_STIPPLE		
62h	31:29	Instruction Type = 3D_INSTRUCTION = 3h
	28:16	3D Instruction Opcode = 3DSTATE_LINE_STIPPLE GFXPIPE[28:27 = 3h, 26:24 = 1h, 23:16 = 08h] (Non-Pipelined)
	15:0	DWord Length (excl. DWord 0,1) = 1
63h	31	Modify Enable (Current Repeat Counter , Current Stipple Index)
	30	Reserved: MBZ
	29:21	Current Repeat Counter This field sets the HW-internal repeat counter state. Format = U9
	20	Reserved: MBZ
	19:16	Current Stipple Index This field sets the HW-internal stipple pattern index. Format = U4
	15:0	Line Stipple Pattern Specifies a pattern used to mask out bit specific pixels while rendering lines. Format = 16 bit mask. Bit 15 = most significant bit, Bit 0 = least significant bit
64h	31:16	Line Stipple Inverse Repeat Count
	15:9	Reserved: MBZ



DWord	Bits	State Field
	8:0	Line Stipple Repeat Count
SVGunit Context Data (Media)		
MEDIA_STATE_POINTERS		
Note: Dwords 65h – 67h will be saved as MI_NOOP (opcode 00h) unless MEDIA_STATE_POINTERS has been initialized (issued at least once).		
65h	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
66h	31:5	Pointer to VLD_STATE
	4:1	Reserved : MBZ
	0	VLD Enable
67h	31:5	Pointer to VFE_STATE
	4:0	Reserved : MBZ
SVGunit Context Data (3D)		
3DSTATE_PIPELINE_POINTERS		
Note: Dwords 68h – 6Eh will be saved as MI_NOOP (opcode 00h) unless 3DSTATE_PIPELINE_POINTERS has been initialized (issued at least once).		
68h	31:29	Command Type = GFXPIPE = 3h
	28:16	3D Command Opcode = 3DSTATE_PIPELINED_POINTERS GFXPIPE[28:27 = 3h, 26:24 = 0h, 23:16 = 00h] (Pipelined)
	15:8	Reserved : MBZ
	7:0	DWord Length (Excludes DWords 0,1) = 5
69h	31:5	Pointer to VS_STATE
	4:0	Reserved : MBZ
6Ah	31:5	Pointer to GS_STATE
	4:1	Reserved : MBZ
	0	GS Enable
6Bh	31:5	Pointer to CLP_STATE
	4:1	Reserved : MBZ
	0	CLP Enable
6Ch	31:5	Pointer to SF_STATE
	4:0	Reserved : MBZ
6Dh	31:5	Pointer to WINDOWER_STATE
	4:0	Reserved : MBZ
6Eh	31:6	Pointer to COLOR_CALC_STATE
	5:0	Reserved : MBZ
3DSTATE_BINDING_TABLE_POINTERS		
Note: Dwords 6Fh – 74h will be saved as MI_NOOP (opcode 00h) unless 3DSTATE_BINDING_TABLE_POINTERS has been initialized (issued at least once).		
6Fh	31:29	Command Type = GFXPIPE = 3h
	28:16	3D Command Opcode = 3DSTATE_BINDING_TABLE_POINTERS GFXPIPE[28:27 = 3h, 26:24 = 0h, 23:16 = 01h] (Pipelined)
	15:8	Reserved : MBZ
	7:0	DWord Length (Excludes DWords 0,1) = 4
70h	31:5	Pointer to VS Binding Table
	4:0	Reserved : MBZ
71h	31:5	Pointer to GS Binding Table
	4:0	Reserved : MBZ
72h	31:5	Pointer to CLP Binding Table
	4:0	Reserved : MBZ
73h	31:5	Pointer to SF Binding Table
	4:0	Reserved : MBZ
74h	31:5	Pointer to PS Binding Table



DWord	Bits	State Field
	4:0	Reserved : MBZ
CONSTANT_BUFFER		
Note: Dwords 75h – 76h will be saved as MI_NOOP (opcode 00h) unless CONSTANT_BUFFER has been initialized (issued at least once).		
75h	31:29	Command Type = GFXPIPE = 3h
	28:16	3D Command Opcode = CONSTANT_BUFFER GFXPIPE[28:27 = 0h, 26:24 = 0h, 23:16 = 02h] (Pipelined)
	15:9	Reserved : MBZ
	8	Valid (Will be <u>set</u> if CONSTANT_BUFFER was issued in the context to be saved)
	7:0	DWord Length (excludes DWords 0,1) = 0
76h	31:6	Buffer Starting Address
	5:0	Buffer Length
77h	31:0	MI_Noop
(This region was formerly Blitter Related Context Data)		
78 – 87h	31:0	Reserved Should be treated as garbage data when inspecting a saved context.
VFunit Related Context Data		
3DSTATE_INDEX_BUFFER		
88h	31:29	Command Type = GFXPIPE = 3h
	28:16	GFXPIPE Opcode = 3DSTATE_INDEX_BUFFER GFXPIPE[28:27 = 3h, 26:24 = 0h, 23:16 = 0Ah] (Pipelined)
	15:11	Reserved : MBZ
	10	Cut Index Enable
	9:8	Index Format
	7:0	DWord Length (excludes DWords 0,1) = 1
89h	31:0	Buffer Starting Address
8Ah	31:0	Buffer Ending Address
3DSTATE_VERTEX_BUFFER		
8Bh	31:29	Command Type = GFXPIPE = 3h
	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)
8C-8Fh		Vertex Buffer 0 State
90-93		Vertex Buffer 1 State
		...
CC-CFh		Vertex Buffer 16 State
3DSTATE_VERTEX_ELEMENT (71 - 93h)		
D0h	31:29	Command Type = GFXPIPE = 3h
	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)
D1 – D2h	[1-2] dw	Element[0]
D3 – D4h	[3-4] dw	Element[1]
...	...	...
F3 – F4h	[37-38] dw	Element[17]
F5h	31:0	Reserved



DWord	Bits	State Field
F6-FFh		MI_NOOP
DMunit Related Context Data		
3DSTATE_SAMPLER_PALETTE_LOAD (ONLY on Extended SAVE Mode)		
100h	31:29	Instruction Type = GFXPIPE = 3h
	28:16	3D Instruction Opcode = 3DSTATE_SAMPLER_PALETTE_LOAD GFXPIPE[28:27 = 3h, 26:24 = 1h; 23:16 = 02h] (Non-Pipelined)
	15:4	Reserved: MBZ
	3:0	DWord Length (excludes DWords 0,1)
101-110h	31:24	Reserved
	23:0	Palette Color[0:N-1]
111-117h	31:0	MI_NOOP
WIZunit Related Context Data		
3DSTATE_POLY_STIPPLE_PATTERN (ONLY on Extended SAVE Mode)		
118h	31:29	Instruction Type = 3D_INSTRUCTION = 3h
	28:16	3D Instruction Opcode = 3DSTATE_POLY_STIPPLE_PATTERN GFXPIPE[28:27 = 3h, 26:24 = 1h, 23:16 = 07h] (Non-Pipelined)
	15:0	DWord Length (excl. DWord 0,1) = 31
119h	31:0	Polygon Stipple Pattern Row 1 (top most)
11Ah	31:0	Polygon Stipple Pattern Row 2
11Bh – 138h	31:0	Polygon Stipple Pattern Rows 3 through 32 (bottom-most)
139-13Fh	31:0	MI_Noop

6.19.2.1.1 Power Context Memory Layout ([DevCL] Only)

Additional context data is required if a reset occurs (if power is lost, for example) between the save and restore of a context. A mobile-only feature provides for saving and restoring the following context state/registers in this event. Note that the context below includes a pointer (in an MI_SET_CONTEXT command) to the usual logical rendering context which is considered a subset of the power context when power context is saved/restored. See the device EDS for further details.

DWord	Bits	State Field
MEMORY INTERFACE STATE		
00h	31:0	MI_NOOP
01h	31:29	Instruction Type = MI_INSTRUCTION = 0h
	28:23	MI Instruction Opcode = MI_LOAD_REGISTER_IMM = 22h
	22:12	Reserved: MBZ
	11:8	Byte Write Disables = Fh (all enabled)
	7:6	Reserved: MBZ
	5:0	DWord Length (Excludes DWord 0,1) = Ah
02h	31:0	Scratch Pad Register Address Offset
03h	31:0	Scratch Pad Register Data
04h	31:0	EXCC Register Address Offset



DWord	Bits	State Field
05h	31:21	Reserved: MBZ.
	20:16	Bit Write Masks for Bits 4:0: Written as 1Fh (all enabled)
	15:5	Reserved: MBZ
	4:0	User Defined Condition Codes
06h	31:0	Ring Buffer Tail Pointer Register Offset
07h	31:21	Reserved: MBZ
	20:3	Tail Offset (Never Saved on Context Switch)
	2:1	Reserved: MBZ
	0	In Use (Always saved as 0)
08h	31:0	Ring Buffer Starting Address Register Offset
09h	31:12	Starting Address
	11:0	Reserved: MBZ
0Ah	31:0	Ring Buffer Head Pointer Register Offset <i>Note: The Head reg is restored <u>after</u> the Address reg, as restoring the Address reg resets the Head.</i>
0Bh	31:21	Wrap Count
	20:2	Head Offset
	1:0	Reserved: MBZ
0Ch	31:0	Ring Buffer Length Register Offset
0Dh	31:21	Reserved: MBZ
	20:12	Buffer Length
	11	RB Wait
	10	RB Arb off
	9	RB in time slice
	8	Disable Register Accesses
	7:3	Reserved: MBZ
	2:1	Automatic Report Head Pointer
	0	Ring Buffer Enable
0Eh	31:29	Instruction Type = MI_INSTRUCTION = 0h
	28:23	MI Instruction Opcode = MI_SET_CONTEXT = 18h
	22:6	Reserved: MBZ
	5:0	DWord Length (Excludes Dword 0,1) = 0
0Fh	31:11	Logical Context Address
	10:4	Reserved: MBZ
	8	Memory Space Select
	7:4	Physical Start Address Extension
	3	Extended State Save Enable
	2	Extended State Restore Enable
	1	Force Restore



DWord	Bits	State Field
	0	Restore Inhibit



6.19.2.1.2 Logical Context Initialization

Each logical context should initialize all device state before beginning operations so that any context switches that occur subsequently will save and restore coherent device state. See *Memory Interface Functions* for more information. Table 6-25 provides values that should be used to initialize any state that the context does not require for its operations. Note that these state variables will need to be set to something more intelligent for a context that intends to perform operations that depend on them. The values of these state variables are saved (and subsequently restored) on any context switch, with the exception of the 3DSTATE_SAMPLER_PALETTE_LOAD and 3DSTATE_POLY_STIPPLE_PATTERN which are only saved from and restored to contexts that have the **Extended State Save Enable** and **Extended State Restore Enable**, respectively, set in the MI_SET_CONTEXT command that triggers the context switch. See *Memory Interface Commands* for details of this command.

Note that 3D/Media *pipelined* state cannot be initialized; it is not stored internally to the device but is accessed from state blocks in memory as required by rendering operations. Any context that will issue 3DPRIMITIVE or MEDIA_OBJECT_LOAD commands must first place valid state structures in memory and send down the corresponding command (3DSTATE_PIPELINED_POINTERS or MEDIA_STATE_POINTERS) to point to it. There are no defaults for this state. The following table (Table 6-24) summarizes state that **MUST BE** properly set up for a given context. Please refer to the *Graphics Processing Engine (GPE)*, *3D Pipeline* and *Media* chapters for details on these commands.

Table 6-24. Context Setup that Cannot Use Defaults

Context	Required Setup	Notes
3D	PIPELINE_SELECT	3D Pipeline must be selected
	CS_URB_STATE	Must allocate sufficient URB space for constants that will be used.
	3DSTATE_PIPELINED_POINTERS	Pointers for all enabled FF units (when offset from base address) must point to valid state in memory.
	3DSTATE_BINDING_TABLE_POINTERS	Pointers for all enabled FF units (when offset from base address) must point to valid binding tables in memory.
	STATE_BASE_ADDRESS	Must be properly initialized so that pointers above point to valid state blocks.
	URB_FENCE	Enabled FF units must be allocated sufficient URB space to avoid deadlock. Note that most FF units <i>cannot</i> be disabled. Only VS and CLIP can be disabled.
	CONSTANT_BUFFER	Must point to a valid constant buffer if constants will be used.
	STATE_SIP	Must point to a valid exception handler if any threads will be dispatched with any exceptions enabled.



Context	Required Setup	Notes
Media	PIPELINE_SELECT	Media Pipeline must be selected
	CS_URB_STATE	Same as above
	MEDIA_STATE_POINTERS	Pointers for one, or both if enabled, Media FF units (when offset from base address) must point to valid state in memory.
	STATE_BASE_ADDRESS	Must be properly initialized so that pointers above point to valid state blocks.
	URB_FENCE	Enabled FF units must be allocated sufficient URB space to avoid deadlock. Note that the VFE FF unit <i>cannot</i> be disabled.
	CONSTANT_BUFFER	Must point to a valid constant buffer if constants will be used.
	STATE_SIP	Must point to a valid exception handler if any threads will be dispatched with any exceptions enabled.

Table 6-25. Initialization of Command State

Instruction / Field	Value
PIPELINE_SELECT	
Pipeline Select	0 = 3D pipeline is selected
CS_URB_STATE	
URB Entry Allocation Size	0
Number of URB Entries	0
URB_FENCE	
CS Unit URB Reallocation Request	0
VFE Fence Unit URB Reallocation Request	0
SF Unit URB Reallocation Request	0
CLIP Unit URB Reallocation Request	0
GS Unit URB Reallocation Request	0
VS Unit URB Reallocation Request	0
CLP Fence	192
GS Fence	128
VS Fence	64
CS Fence	256
VFE Fence	0
SF Fence	252
CONSTANT_BUFFER	
Valid	0
Buffer Starting Address	0
Buffer Length	0
STATE_BASE_ADDRESS	
General State Base Address	0



Instruction /Field	Value
Surface State Base Address	0
Indirect Object Base Address	0
General State Access Upper Bound	0
Indirect Object Access Upper Bound	0
STATE_SIP	
System Instruction Pointer	0
3DSTATE_DRAWING_RECTANGLE	
Clipped Drawing Rectangle Y Min	0
Clipped Drawing Rectangle X Min	0
Clipped Drawing Rectangle Y Max	8191
Clipped Drawing Rectangle X Max	8191
Drawing Rectangle Origin Y	0
Drawing Rectangle Origin X	0
3DSTATE_DEPTH_BUFFER	
Surface Type	7 (SURFTYPE_NULL)
Tiled Surface	0
Tile Walk	1 = Y
Depth Buffer Coordinate Offset Disable	0
Surface Format	0
Surface Pitch	0
Surface Base Address	0
Height	0
Width	0
LOD	0
MIP Map Layout Mode	0 = MIPLAYOUT_BELOW
Depth	0
Minimum Array Element	0
3DSTATE_CHROMA_KEY (INDEX_0)	
ChromaKey Table Index	0
ChromaKey Low Value	0
ChromaKey High Value	0
3DSTATE_CHROMA_KEY (INDEX_1)	
ChromaKey Table Index	1
ChromaKey Low Value	0
ChromaKey High Value	0
3DSTATE_CHROMA_KEY (INDEX_2)	
ChromaKey Table Index	2
ChromaKey Low Value	0
ChromaKey High Value	0
3DSTATE_CHROMA_KEY (INDEX_3)	
ChromaKey Table Index	3
ChromaKey Low Value	0
ChromaKey High Value	0
3DSTATE_CONSTANT_COLOR	
Blend Constant Color Red	1.0
Blend Constant Color Blue	1.0



Instruction /Field	Value
Blend Constant Color Green	1.0
Blend Constant Color Alpha	1.0
3DSTATE_GLOBAL_DEPTH_OFFSET_CLAMP	
Global Depth Offset Clamp	0.0
3DSTATE_POLY_STIPPLE_OFFSET	
Polygon Stipple X Offset	0
Polygon Stipple Y Offset	0
3DSTATE_LINE_STIPPLE	
Modify Enable	0
Current Repeat Counter	0
Current Stipple Index	0
Line Stipple Pattern	0
Line Stipple Inverse Repeat Count	0
Line Stipple Repeat Count	0
MEDIA_STATE_POINTERS	
Pointer to VLD_STATE	0
VLD Enable	0
Pointer to VFE_STATE	0
3DSTATE_PIPELINE_POINTERS	
Pointer to VS_STATE	0
Pointer to GS_STATE	0
GS Enable	0
Pointer to CLP_STATE	0
CLP Enable	0
Pointer to SF_STATE	0
Pointer to WINDOWER_STATE	0
Pointer to COLOR_CALC_STATE	0
3DSTATE_BINDING_TABLE_POINTERS	
Pointer to VS Binding Table	0
Pointer to GS Binding Table	0
Pointer to CLP Binding Table	0
Pointer to SF Binding Table	0
Pointer to PS Binding Table	0
3DSTATE_INDEX_BUFFER	
Cut Index Enable	0
Index Format	0
Buffer Starting Address	0
Buffer Ending Address	0
3DSTATE_VERTEX_BUFFER (0 - 16)	
DWord Length (excludes DWords 0,1)	50 (32h)
Vertex Buffer Index	0
Buffer Access Type	0 = VERTEXDATA
Buffer Pitch	0
Buffer Starting Address	0
Max Index	0
... values repeated for all 17 Vertex Buffers	...



Instruction / Field	Value
3DSTATE_VERTEX_ELEMENT (0 – 17)	
DWord Length (excludes DWords 0,1)	35 (23h)
Vertex Buffer Index	0
Valid	0
Source Element Format	0
Source Element Offset	0
Component 0 Control	2 = VFCOMP_STORE_0
Component 1 Control	0 = VFCOMP_NOSTORE
Component 2 Control	0 = VFCOMP_NOSTORE
Component 3 Control	0 = VFCOMP_NOSTORE
Destination Element Offset	0
<i>... values repeated for all 18 Vertex Elements</i>	...
3DSTATE_SAMPLER_PALETTE_LOAD (Required to be initialized only if context uses extended save)	
DWord Length (excludes DWords 0,1)	15
Palette Color 0	0
Palette Color 1	0
...	0
Palette Color 15	0
3DSTATE_POLY_STIPPLE_PATTERN (Required to be initialized only if context uses extended save)	
DWord Length (excl. DWord 0,1)	31
Polygon Stipple Pattern Row 1 (top most)	0
Polygon Stipple Pattern Row 2	0
...	0
Polygon Stipple Pattern Row 32 (bottom-most)	0



6.19.3 The Probe List

The Probe List consists of 1024 slots. Each slot can hold a probe list entry. Each entry is one Dword and has the following format:

Bit	Description
31:12	Surface Page Base Address. Format = PerProcessGraphicsVirtualAddress[31:12]
11:1	Reserved. MBZ
0	Fault. This bit is set by HW if this probe faults (either on context restore or when executing MI_PROBE.) This bit is ignored when this probe entry is read in order to be re-checked as part of a context restore operation.

SW must clear the **Fault** bit in a probe list entry for which it has successfully serviced a surface fault. When restoring a context, **Fault** bits are only set for new faults. They are not cleared for reprobes which do not fault.

6.19.4 Pipelined State Page

This page is used a scratch area for the pipeline to store pipelined state that is not referenced indirectly. Under no circumstances should SW read from or write to this page.

6.19.5 Ring Buffer

This page is used a scratch area for the pipeline to store ring buffer commands that need to be reissued. Under no circumstances should SW read from or write to this page.



6.19.6 The Per-Process Hardware Status Page

The following table defines the layout of the Per-process Hardware Status Page:

DWord Offset	Description
(3FFh – 020h)	These locations can be used for general purpose via the MI_STORE_DATA_INDEX or MI_STORE_DATA_IMM instructions.
1F:1A	Reserved.
19	Context Save Finished Timestamp
18	Context Restore Complete Timestamp
17	Pre-empt Request Received Timestamp
16	Last Switch Timestamp
15:12	Reserved.
11:10	Probe List Slot Fault Register (2 DWs)
F:5	Reserved.
4	Ring Head Pointer Storage: The contents of the Ring Buffer Head Pointer register (register DWord 1) are written to this location either as result of an MI_REPORT_HEAD instruction or as the result of an “automatic report” (see RINGBUF registers).
3:0	Reserved.

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

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

§§





7 Device 2 Configuration Registers

7.1 Introduction

PCI Configuration Device 2 is the Internal Graphics Device (IGD). The common subset of these registers is thus documented in this specification. For all other configuration register devices, please see the EDS for the particular device concerned.

Note that only a subset of the Device 2 Configuration registers is documented here. Registers that are not documented here are available for use (and many are already used) for product-specific control registers that relate to Device 2. Please see the EDS for the complete set of Device 2 registers for a given product.

All registers documented herein are common between all products in the Gen4 family except for the minor exceptions noted. Any changes to the registers documented here must be presented to the common graphics core change control board.

7.2 Device 2, Function 0

Register Name	Register Symbol	Register Start	Register End	Default Value	Access
Vendor Identification	VID2	0	1	8086h	RO;
Device Identification	DID2	2	3	[Device Specific]	RO;
PCI Command	PCICMD2	4	5	0000h	RO; R/W;
PCI Status	PCISTS2	6	7	0090h	RO;
Revision Identification	RID2	8	8	00h	RO;
Class Code	CC	9	B	030000h	RO;
Cache Line Size	CLS	C	C	00h	RO;
Master Latency Timer	MLT2	D	D	00h	RO;
Header Type	HDR2	E	E	80h	RO;
Built In Self Test	BIST	F	F	00h	RO;
Graphics Translation Table Range Address	GTTMMADR	10	17	000000000000 0004h	RO; R/W;
Graphics Memory Range Address	GMADR	18	1F	000000000000 000Ch	RO; R/W; R/W/L;
I/O Base Address	IOBAR	20	23	00000001h	RO; R/W;



Register Name	Register Symbol	Register Start	Register End	Default Value	Access
Subsystem Vendor Identification	SVID2	2C	2D	0000h	R/WO;
Subsystem Identification	SID2	2E	2F	0000h	R/WO;
Video BIOS ROM Base Address	ROMADR	30	33	00000000h	RO;
Capabilities Pointer	CAPPOINT	34	34	90h	RO;
Interrupt Line	INTRLINE	3C	3C	00h	R/W;
Interrupt Pin	INTRPIN	3D	3D	01h	RO;
Minimum Grant	MINGNT	3E	3E	00h	RO;
Maximum Latency	MAXLAT	3F	3F	00h	RO;
Capabilities Pointer (to Mirror of Dev0 CAPID)	MCAPPTR	44	44	48h	RO;
Mirror of Dev 0 Capability Identification	MCAPID	48	51	[Device Specific]	RO;
Mirror of Dev0 GMCH Graphics Control	MGGC	52	53	0030h	RO;
Mirror of Dev0 DEVEN	MDEVENdev0F0	54	57	[Device Specific]	RO;
Software Scratch Read Write	SSRW	58	5B	00000000h	R/W;
Base of Stolen Memory	BSM	5C	5F	[Device Specific]	RO;
Hardware Scratch Read Write	HSRW	60	61	0000h	R/W;
Multi Size Aperture Control	MSAC	62	62	02h	RO; R/W; R/W/L;
VTD Status	VTDS	63	63	02h or 00h	RO;
Secondary CWB Flush Control [DevBW Only]	SCWBFC	68	6F	000000000000 0000h	RO
Capabilities List Control	CAPL	7F	7F	00h	RO; R/W;
Message Signaled Interrupts Capability ID	MSI_CAPID	90	91	D005h	RO;
Message Control	MC	92	93	0000h	RO; R/W;
Message Address	MA	94	97	00000000h	R/W; RO;
Message Data	MD	98	99	0000h	R/W;
FLR Capability ID	FLRCAPID	A4	A5	0009h	RO;
FLR Length and Version	FLRLENVER	A6	A7	2006h	RO;
FLR Control	FLRCNTL	A8	A9	0000h	RO; R/W;
FLR Status	FLRSTAT	AA	AA	00h	RO
Graphics Device Reset	GDRST	C0	C0	00h	RO; R/W;



Register Name	Register Symbol	Register Start	Register End	Default Value	Access
GMBUS frequency binary encoding	GMBUSFREQ	CC	CD	0000h	R/W; RO;
Power Management Capabilities ID	PMCAPID	D0	D1	0001h	RO;
Power Management Capabilities	PMCAP	D2	D3	0022h or 0023h	RO;
Power Management Control/Status	PMCS	D4	D5	0000h	RO; R/W;
Software SMI	SWSMI	E0	E1	0000h	R/W;
System Display Event Register	ASLE	E4	E7	00000000h	R/W;
Software SCI	SWSCI	E8	E9	0000h	RO; R/W;
Legacy Backlight Brightness	LBB	F4	F7	00000000h	R/W;
Manufacturing ID	MID2	F8	FB	[Device Specific]	RO;
ASL Storage	ASLS	FC	FF	00000000h	R/W;

7.2.1 VID2 — Vendor Identification

B/D/F/Type: 0/2/0/PCI
 Address Offset: 0-1h
 Default Value: 8086h
 Access: RO;
 Size: 16 bits

This register combined with the Device Identification register uniquely identifies any PCI device.

Bit	Access	Default Value	Description
15:0	RO	8086h	Vendor Identification Number (VID): PCI standard identification for Intel.



7.2.2 DID2 — Device Identification

B/D/F/Type: 0/2/0/PCI
Address Offset: 2-3h
Default Value: [Device Specific]
Access: RO;
Size: 16 bits

This register combined with the Vendor Identification register uniquely identifies any PCI device.

Bit	Access	Default Value	Description
15:0	RO	--	Device Identification Number (DID): Identifier assigned to the GMCH core/primary PCI device. Intel Reserved Text: Some bits of this field are actually determined by fuses, which allows unique Device IDs to be used for different product SKUs.



7.2.3 PCICMD2 — PCI Command

B/D/F/Type: 0/2/0/PCI
 Address Offset: 4-5h
 Default Value: 0000h
 Access: RO; R/W;
 Size: 16 bits

This 16-bit register provides basic control over the IGDs ability to respond to PCI cycles. The PCICMD Register in the IGD disables the IGD PCI compliant master accesses to main memory.

Bit	Access	Default Value	Description
15:11	RO	00h	Reserved
10	R/W	0b	Interrupt Disable: This bit disables the device from asserting INTx#. 0: Enable the assertion of this device's INTx# signal. 1: Disable the assertion of this device's INTx# signal. DO_INTx messages will not be sent to DMI.
9	RO	0b	Fast Back-to-Back (FB2B): Not Implemented. Hardwired to 0.
8	RO	0b	SERR Enable (SERRE): Not Implemented. Hardwired to 0.
7	RO	0b	Address/Data Stepping Enable (ADSTEP): Not Implemented. Hardwired to 0.
6	RO	0b	Parity Error Enable (PERRE): Not Implemented. Hardwired to 0. Since the IGD belongs to the category of devices that does not corrupt programs or data in system memory or hard drives, the IGD ignores any parity error that it detects and continues with normal operation.
5	RO	0b	Video Palette Snooping (VPS): This bit is hardwired to 0 to disable snooping.
4	RO	0b	Memory Write and Invalidate Enable (MWIE): Hardwired to 0. The IGD does not support memory write and invalidate commands.
3	RO	0b	Special Cycle Enable (SCE): This bit is hardwired to 0. The IGD ignores Special cycles.
2	R/W	0b	Bus Master Enable (BME): 0: Disable IGD bus mastering. 1: Enable the IGD to function as a PCI compliant master.
1	R/W	0b	Memory Access Enable (MAE): This bit controls the IGDs response to memory space accesses. 0: Disable. 1: Enable.
0	R/W	0b	I/O Access Enable (IOAE): This bit controls the IGDs response to I/O space accesses. 0: Disable. 1: Enable.



7.2.4 PCISTS2 — PCI Status

B/D/F/Type: 0/2/0/PCI
Address Offset: 6-7h
Default Value: 0090h
Access: RO;
Size: 16 bits

PCISTS is a 16-bit status register that reports the occurrence of a PCI compliant master abort and PCI compliant target abort. PCISTS also indicates the DEVSEL# timing that has been set by the IGD.

Bit	Access	Default Value	Description
15	RO	0b	Detected Parity Error (DPE): Since the IGD does not detect parity, this bit is always hardwired to 0.
14	RO	0b	Signaled System Error (SSE): The IGD never asserts SERR#, therefore this bit is hardwired to 0.
13	RO	0b	Received Master Abort Status (RMAS): The IGD never gets a Master Abort, therefore this bit is hardwired to 0.
12	RO	0b	Received Target Abort Status (RTAS): The IGD never gets a Target Abort, therefore this bit is hardwired to 0.
11	RO	0b	Signaled Target Abort Status (STAS): Hardwired to 0. The IGD does not use target abort semantics.
10:9	RO	00b	DEVSEL Timing (DEVT): N/A. These bits are hardwired to "00".
8	RO	0b	Master Data Parity Error Detected (DPD): Since Parity Error Response is hardwired to disabled (and the IGD does not do any parity detection), this bit is hardwired to 0.
7	RO	1b	Fast Back-to-Back (FB2B): Hardwired to 1. The IGD accepts fast back-to-back when the transactions are not to the same agent.
6	RO	0b	User Defined Format (UDF): Hardwired to 0.
5	RO	0b	66 MHz PCI Capable (66C): N/A - Hardwired to 0.
4	RO	1b	Capability List (CLIST): This bit is set to 1 to indicate that the register at 34h provides an offset into the function's PCI Configuration Space containing a pointer to the location of the first item in the list.
3	RO	0b	Interrupt Status: This bit reflects the state of the interrupt in the device. Only when the Interrupt Disable bit in the command register is a 0 and this Interrupt Status bit is a 1, will the device's INTx# signal be asserted. Setting the Interrupt Disable bit to a 1 has no effect on the state of this bit.
2:0	RO	000b	Reserved.:



7.2.5 RID2 — Revision Identification

B/D/F/Type: 0/2/0/PCI
Address Offset: 8h
Default Value: 00h
Access: RO;
Size: 8 bits

Compatible Revision ID (CRID):

An 8 bit hardwired value assigned by the ID Council. Normally, the value assigned as the CRID will be identical to the SRID value of a previous stepping of the product with which the new product is deemed "compatible". Note that CRID is not an addressable PCI register. The CRID value is simply reflected through the RID register when appropriately selected. Lower 4 bits of the CRID are driven by Fuses. The CRID fuses are programmed based on the SKU.

Stepping Revision ID (SRID):

An 8 bit hardwired value assigned by the ID Council. The values assigned as the SRID of a product's steppings will be selectively incremented based on the degree of change to that stepping. It is suggested that the first stepping of any given product have an SRID value = 01h simply to avoid the "reserved register" value of 00h. Note that SRID is not an addressable PCI register. The SRID value is simply reflected through the RID register when appropriately selected.

RID Select Key Value:

This is hardwired value (69h). If the latched value written to the RID register address matches this RID Select Key Value, the CRID value be presented for reading from the RID register.

RID Definition:

This register contains the revision number of the GMCH Device #0. Following PCI Reset the SRID value is selected to be read. When a write occurs to this register the write data is compared to the hardwired RID Select Key Value which is 69h. If the data matches this key a flag is set that enables the CRID value to be read through this register.

Note that the flag is a "write once". Therefore once the CRID is selected to be read, the only way to again select the SRID is to PCI Reset the component. Also if any value other than the key (69h) is written to the RID register, the flag is locked such that the SRID is selected until the component is PCI Reset. Note that the RID register itself is not directly write-able.

This register contains the revision number for Device #2 Functions 0 and 1.

Bit	Access	Default Value	Description
7:0	RO	00h	Revision Identification Number (RID): This is an 8-bit value that indicates the revision identification number for the GMCH.



7.2.6 CC — Class Code

B/D/F/Type: 0/2/0/PCI
Address Offset: 9-Bh
Default Value: 030000h
Access: RO;
Size: 24 bits

This register contains the device programming interface information related to the Sub-Class Code and Base Class Code definition for the IGD. This register also contains the Base Class Code and the function sub-class in relation to the Base Class Code.

Bit	Access	Default Value	Description
23:16	RO	03h	Base Class Code (BCC): This is an 8-bit value that indicates the base class code for the GMCH. This code has the value 03h, indicating a Display Controller.
15:8	RO	00h	Sub-Class Code (SUBCC): Based on Device #0 GGC-GMS bits and GGC-IVD bits. 00h: VGA compatible 80h: Non VGA (GMS = "000" or IVD = "1")
7:0	RO	00h	Programming Interface (PI): 00h: Hardwired as a Display controller.

7.2.7 CLS — Cache Line Size

B/D/F/Type: 0/2/0/PCI
Address Offset: Ch
Default Value: 00h
Access: RO;
Size: 8 bits

The IGD does not support this register as a PCI slave.

Bit	Access	Default Value	Description
7:0	RO	00h	Cache Line Size (CLS): This field is hardwired to 0s. The IGD as a PCI compliant master does not use the Memory Write and Invalidate command and, in general, does not perform operations based on cache line size.



7.2.8 MLT2 — Master Latency Timer

B/D/F/Type: 0/2/0/PCI
Address Offset: Dh
Default Value: 00h
Access: RO;
Size: 8 bits

The IGD does not support the programmability of the master latency timer because it does not perform bursts.

Bit	Access	Default Value	Description
7:0	RO	00h	Master Latency Timer Count Value: Hardwired to 0s.

7.2.9 HDR2 — Header Type

B/D/F/Type: 0/2/0/PCI
Address Offset: Eh
Default Value: 80h
Access: RO;
Size: 8 bits

This register contains the Header Type of the IGD.

Bit	Access	Default Value	Description
7	RO	1b	Multi Function Status (MFunc): Indicates if the device is a Multi-Function Device. The Value of this register is determined by Device #0, offset 54h, DEVEN[4]. If Device #0 DEVEN[4] is set, the Mfunc bit is also set.
6:0	RO	00h	Header Code (H): This is a 7-bit value that indicates the Header Code for the IGD. This code has the value 00h, indicating a type 0 configuration space format.

7.2.10 BIST — Built In Self Test

B/D/F/Type: 0/2/0/PCI
Address Offset: Fh
Default Value: 00h
Access: RO;
Size: 8 bits

This register is used for control and status of Built In Self Test (BIST).

Bit	Access	Default Value	Description
7	RO	0b	BIST Supported: BIST is not supported. This bit is hardwired to 0.
6:0	RO	00h	Reserved



7.2.11 GTTMADR — Graphics Translation Table Range Address

B/D/F/Type: 0/2/0/PCI
 Address Offset: 10-17h
 Default Value: 0000000000000004h
 Access: RO; R/W;
 Size: 64 bits

This register requests allocation for combined Graphics Translation Table and Memory Mapped Range. The allocation is split evenly between GTTADDR and MMIO, with MMIO coming first (lowest address) in the space.

For the Global GTT, GTTADDR is defined as part of a memory BAR in graphics device config space as an alias with which software writes values (PTEs) into the global Graphics Translation Table (GTT). Writing PTEs directly into the global GTT memory area is allowed.

Device	Total Allocation	GTTADDR Size	# GTT Entries	Total Aperture Size	Base Address Bits
All	1 MB	512K	128K	512M	35:20

The device snoops writes to GTTADDR space in order to invalidate any cached translations within the various TLB's implemented on-chip. There are some exceptions to this – see GTT-TLB in the Programming Interface chapter.

The Global GTT base address is programmed in the PGTB_CNTL register. The Global GTT resides in Main Memory

The Global GTT is required to be 4KB aligned, with each entry being DWord aligned.

Bit	Access	Default Value	Description
63:36	R/W	0000000h	Must be set to 0 since addressing above 64GB is not supported.
35:21	R/W	0000h	Memory Base Address: Set by the OS, these bits correspond to address signals [35:21].
20	R/W		R/W, Memory Base Address[20]. 0 indicates at least 2MB address range.
19:4	RO	0000h	Reserved: Hardwired to 0's to indicate at least 1MB address range.
3	RO	0b	Prefetchable Memory: Hardwired to 0 to prevent prefetching.
2:1	RO	10b	Memory Type () 00 : To indicate 32 bit base address 01: Reserved 10 : To indicate 64 bit base address 11: Reserved
0	RO	0b	Memory / IO Space: Hardwired to 0 to indicate memory space.



7.2.12 GMADR — Graphics Memory Range Address

B/D/F/Type: 0/2/0/PCI
 Address Offset: 18-1Fh
 Default Value: 000000000000000Ch
 Access: RO; R/W; R/W/L;
 Size: 64 bits

IGD graphics memory base address is specified in this register.

Bit	Access	Default Value	Description
63:36	RO	0000000h	Reserved
35:29	R/W	00h	Memory Base Address: Set by the OS, these bits correspond to address signals [35:29].
28	R/W/L	0b	512 MB Address Mask: This bit is either part of the Memory Base Address (R/W) or part of the Address Mask (RO), depending on the value of MSAC[1:0]. See MSAC (Dev 2, Func 0, offset 62h) for details.
27	R/W/L	0b	256 MB Address Mask: This bit is either part of the Memory Base Address (R/W) or part of the Address Mask (RO), depending on the value of MSAC[1:0]. See MSAC (Dev 2, Func 0, offset 62h) for details.
26:4	RO	000000h	Address Mask: Hardwired to 0s to indicate at least 128MB address range.
3	RO	1b	Prefetchable Memory: Hardwired to 1 to enable prefetching.
2:1	RO	10b	Memory Type () 00 : To indicate 32 bit base address 01: Reserved 10 : To indicate 64 bit base address 11: Reserved
0	RO	0b	Memory / IO Space: Hardwired to 0 to indicate memory space.



7.2.13 IOBAR — I/O Base Address

B/D/F/Type: 0/2/0/PCI
Address Offset: 20-23h
Default Value: 00000001h
Access: RO; R/W;
Size: 32 bits

This register provides the Base offset of the I/O registers within Device #2. Bits 15:3 are programmable allowing the I/O Base to be located anywhere in 16bit I/O Address Space. Bits 2:1 are fixed and return zero, bit 0 is hardwired to a one indicating that 8 bytes of I/O space are decoded.

Access to the 8Bs of IO space is allowed in PM state D0 when IO Enable (PCICMD bit 0) set. Access is disallowed in PM states D1-D3 or if IO Enable is clear or if Device #2 is turned off or if internal graphics is disabled thru the fuse or fuse override mechanisms. Note that access to this IO BAR is independent of VGA functionality within Device #2. Also note that this mechanism is available only through function 0 of Device#2 and is not duplicated in function #1.

If accesses to this IO bar are allowed then the GMCH claims all 8, 16 or 32 bit IO cycles from the CPU that falls within the 8B claimed.

Bit	Access	Default Value	Description
31:16	RO	0000h	Reserved Read as 0's, these bits correspond to address signals [31:16].
15:3	R/W	0000h	IO Base Address: Set by the OS, these bits correspond to address signals [15:3].
2:1	RO	00b	Memory Type: Hardwired to 0s to indicate 32-bit address.
0	RO	1b	Memory / IO Space: Hardwired to 1 to indicate IO space.

7.2.14 SVID2 — Subsystem Vendor Identification

B/D/F/Type: 0/2/0/PCI
Address Offset: 2C-2Dh
Default Value: 0000h
Access: R/WO;
Size: 16 bits

Bit	Access	Default Value	Description
15:0	R/WO	0000h	Subsystem Vendor ID: This value is used to identify the vendor of the subsystem. This register should be programmed by BIOS during boot-up. Once written, this register becomes Read-Only. This register can only be cleared by a Reset.



7.2.15 SID2 — Subsystem Identification

B/D/F/Type: 0/2/0/PCI
Address Offset: 2E-2Fh
Default Value: 0000h
Access: R/WO;
Size: 16 bits

Bit	Access	Default Value	Description
15:0	R/WO	0000h	Subsystem Identification: This value is used to identify a particular subsystem. This field should be programmed by BIOS during boot-up. Once written, this register becomes Read-Only. This register can only be cleared by a Reset.

7.2.16 ROMADR — Video BIOS ROM Base Address

B/D/F/Type: 0/2/0/PCI
Address Offset: 30-33h
Default Value: 00000000h
Access: RO;
Size: 32 bits

The IGD does not use a separate BIOS ROM, therefore this register is hardwired to 0s.

Bit	Access	Default Value	Description
31:18	RO	0000h	ROM Base Address: Hardwired to 0s.
17:11	RO	00h	Address Mask: Hardwired to 0s to indicate 256 KB address range.
10:1	RO	000h	Reserved: Hardwired to 0s.
0	RO	0b	ROM BIOS Enable: 0 = ROM not accessible.

7.2.17 CAPPOINT — Capabilities Pointer

B/D/F/Type: 0/2/0/PCI
Address Offset: 34h
Default Value: 90h
Access: RO;
Size: 8 bits

Bit	Access	Default Value	Description
7:0	RO	90h	Capabilities Pointer Value: This field contains an offset into the function's PCI Configuration Space for the first item in the New Capabilities Linked List which is the MSI Capabilities ID register at address 90h or the Power Management Capabilities ID registers at address D0h. The value is determined by CAPL[0].



7.2.18 INTRLINE — Interrupt Line

B/D/F/Type: 0/2/0/PCI
Address Offset: 3Ch
Default Value: 00h
Access: R/W;
Size: 8 bits

Bit	Access	Default Value	Description
7:0	R/W	00h	Interrupt Connection: Used to communicate interrupt line routing information. POST software writes the routing information into this register as it initializes and configures the system. The value in this register indicates which input of the system interrupt controller that the device's interrupt pin is connected to.

7.2.19 INTRPIN — Interrupt Pin

B/D/F/Type: 0/2/0/PCI
Address Offset: 3Dh
Default Value: 01h
Access: RO;
Size: 8 bits

Bit	Access	Default Value	Description
7:0	RO	01h	Interrupt Pin: As a single function device, the IGD specifies INTA# as its interrupt pin. 01h: INTA#.

7.2.20 MINGNT — Minimum Grant

B/D/F/Type: 0/2/0/PCI
Address Offset: 3Eh
Default Value: 00h
Access: RO;
Size: 8 bits

Bit	Access	Default Value	Description
7:0	RO	00h	Minimum Grant Value: The IGD does not burst as a PCI compliant master.



7.2.21 MAXLAT — Maximum Latency

B/D/F/Type: 0/2/0/PCI
Address Offset: 3Fh
Default Value: 00h
Access: RO;
Size: 8 bits

Bit	Access	Default Value	Description
7:0	RO	00h	Maximum Latency Value: The IGD has no specific requirements for how often it needs to access the PCI bus.

7.2.22 MCAPPTR — Capabilities Pointer (to Mirror of Dev0 CAPID)

B/D/F/Type: 0/2/0/PCI
Address Offset: 44h
Default Value: 48h
Access: RO;
Size: 8 bits

Bit	Access	Default Value	Description
7:0	RO	48h	Capabilities Pointer Value: In this case the first capability is the product-specific Capability Identifier (CAPID0).

7.2.23 MCAPID — Mirror of Dev 0 Capability Identification.

B/D/F/Type: 0/2/0/PCI
Address Offset: 48-51h
Default Value: [Device Specific]
Access: RO;
Size: 80 bits

This is an INTEL RESERVED register and should NOT be disclosed to customers. It is for test and debug purposes only and will not be included in external documentation. Control of bits in this register are only required for customer visible SKU differentiation.

Bit	Access	Default Value	Description
79:0	RO	--	Device Specific Bit Definitions



7.2.24 MGGC — Mirror of Dev0 GMCH Graphics Control

B/D/F/Type: 0/2/0/PCI
 Address Offset: 52-53h
 Default Value: 0030h
 Access: RO;
 Size: 16 bits

Bit	Access	Default Value	Description
15:7	RO	000000000b	Reserved
6:4	RO	011b	Graphics Mode Select (GMS): This field is used to select the amount of Main Memory that is pre-allocated to support the Internal Graphics device in VGA (non-linear) and Native (linear) modes. The BIOS ensures that memory is pre-allocated only when Internal graphics is enabled. Stolen Memory Base is located between (TOLUD - SMSize) to TOUD. 000 = No memory pre-allocated. Device 2 (IGD) does not claim VGA cycles (Mem and IO), and the Sub-Class Code field within Device 2 function 0. Class Code register is 80. 001 = DVMT (UMA) mode, 1 MB memory pre-allocated for frame buffer. 010 = DVMT (UMA) mode, 4 MB memory pre-allocated for frame buffer. 011 = DVMT (UMA) mode, 8 MB memory pre-allocated for frame buffer. 100 = DVMT (UMA) mode, 16 MB memory pre-allocated for frame buffer. 101 = DVMT (UMA) mode, 32 MB memory pre-allocated for frame buffer. 110 = DVMT (UMA) mode, 48 MB memory pre-allocated for frame buffer. 111 = DVMT (UMA) mode, 64 MB memory pre-allocated for frame buffer. Note: This register is locked and becomes Read Only when the D_LCK bit in the SMRAM register is set. Hardware does not clear or set any of these bits automatically based on IGD being disabled/enabled.
3:2	RO	00b	Reserved
1	RO	0b	IGD VGA Disable (IVD): 1:Disable. Device 2 (IGD) does not claim VGA cycles (Mem and IO), and the Sub-Class Code field within Device 2 function 0 Class Code register is 80. 0: Enable (Default). Device 2 (IGD) claims VGA memory and IO cycles, the Sub-Class Code within Device 2 Class Code register is 00.
0	RO	0b	Reserved



7.2.25 MDEVENdev0F0 — Mirror of Dev0 DEVEN

B/D/F/Type: 0/2/0/PCI
 Address Offset: 54-57h
 Default Value: [Device Specific]
 Access: RO;
 Size: 32 bits

Allows for enabling/disabling of PCI devices and functions that are within the MCH.

Bit	Access	Default Value	Description
31:0	RO	--	Device Specific Bit Definitions. See Device 0 documentation in the EDS.

7.2.26 SSRW — Software Scratch Read Write

B/D/F/Type: 0/2/0/PCI
 Address Offset: 58-5Bh
 Default Value: 00000000h
 Access: R/W;
 Size: 32 bits

Bit	Access	Default Value	Description
31:0	R/W	00000000h	Reserved

7.2.27 BSM — Base of Stolen Memory

B/D/F/Type: 0/2/0/PCI
 Address Offset: 5C-5Fh
 Default Value: [Device Specific]
 Access: RO;
 Size: 32 bits

Graphics Stolen Memory and TSEG are within DRAM space defined under TOLUD.
 From the top of low used DRAM, GMCH claims 1 to 64MBs of DRAM for internal graphics if enabled.

Bit	Access	Default Value	Description
31:20	RO	--	Base of Stolen Memory (BSM): This register contains bits 31 to 20 of the base address of stolen DRAM memory. The host interface determines the base of graphics stolen memory by subtracting the graphics stolen memory size from TOLUD. See Device 0 TOLUD in the EDS for more explanation.
19:0	RO	00000h	Reserved



7.2.28 HSRW — Hardware Scratch Read Write

B/D/F/Type: 0/2/0/PCI
Address Offset: 60-61h
Default Value: 0000h
Access: R/W;
Size: 16 bits

Bit	Access	Default Value	Description
15:0	R/W	0000h	Reserved R / W

7.2.29 MSAC — Multi Size Aperture Control

B/D/F/Type: 0/2/0/PCI
Address Offset: 62h
Default Value: 02h
Access: RO; R/W; R/W/L;
Size: 8 bits

This register determines the size of the graphics memory aperture in function 0. By default the aperture size is 256 MB. Only the system BIOS will write this register based on pre-boot address allocation efforts, but the graphics may read this register to determine the correct aperture size. System BIOS needs to save this value on boot so that it can reset it correctly during S3 resume.

Bit	Access	Default Value	Description
7:4	R/W	0h	Scratch Bits Only: Have no physical effect on hardware.
3	RO	0b	Reserved
2:1	R/W/L	01b	Aperture Size (LHSAS): 11: bits [28:27] of GMADR register are made Read only and forced to zero, allowing only 512MB of GMADR 01: bit [28] of GMADR is made R/W and bit [27] of GMADR is forced to zero allowing 256MB of GMADR 00: bits [28:27] of GMADR register are made R/W allowing 128MB of GMADR 10: Illegal programming.
0	RO	0b	Reserved



7.2.30 SCWBFC — Secondary CWB Flush Control ([DevBW] Only)

B/D/F/Type: 0/2/0/PCI
Address Offset: 68-6Fh
Default Value: 0000000000000000h
Access: W
Size: 64 bits

This register is for hardware debug purposes only. This is not relevant for software.
All the data stored in the secondary CWB is flushed to memory before a write to this page is completed on the Front side bus. The write data is discarded. All transactions from the CPU that follow are not processed by the chipset till the "flush write" completes creating a fence beyond which coherency is guaranteed.

A read to this page does not flush the primary CWB/DWB and returns Zeros.

Bit	Access	Default Value	Description
63:0	W	000000000 0000000h	Secondary CWB Flush Control (SCWBFC): A CPU Dword/Qword write to this space flushes the Secondary CWB/DWB of all writes. The data is discarded..

7.2.31 CAPL — Capabilities List Control

B/D/F/Type: 0/2/0/PCI
Address Offset: 7Fh
Default Value: 00h
Access: RO; R/W;
Size: 8 bits

Allows BIOS to hide capabilities that are part of the Device 2 PCI Capabilities Linked List. By setting the appropriate bits, certain capabilities will be "skipped" during a later phase of system initialization. This is an INTEL RESERVED register and should NOT be disclosed to customers. It is for test and debug purposes only and will not be included in external documentation.

Bit	Access	Default Value	Description
7:1	RO	00h	Reserved.:
0	R/W	0b	MSI Capability Hidden (MSICH): 0: MSI Capability at 90h is included in list. 1: MSI Capability is NOT included in list. Power Management Capability ID's (D0h) pointer is the next capability.



7.2.32 MSI_CAPID — Message Signaled Interrupts Capability ID

B/D/F/Type: 0/2/0/PCI
Address Offset: 90-91h
Default Value: D005h
Access: RO;
Size: 16 bits

When a device supports MSI it can generate an interrupt request to the processor by writing a predefined data item (a message) to a predefined memory address. The reporting of the existence of this capability can be disabled by setting MSICH (CAPL[0] @ 7Fh). In that case walking this linked list will skip this capability and instead go directly to the PCI PM capability.

Bit	Access	Default Value	Description
15:8	RO	D0h	Pointer to Next Capability: This contains a pointer to the next item in the capabilities list which is the Power Management Capability.
7:0	RO	05h	Capability ID: Value of 05h identifies this linked list item (capability structure) as being for MSI registers.



7.2.33 MC — Message Control

B/D/F/Type: 0/2/0/PCI
 Address Offset: 92-93h
 Default Value: 0000h
 Access: RO; R/W;
 Size: 16 bits

System software can modify bits in this register, but the device is prohibited from doing so.

If the device writes the same message multiple times, only one of those messages is guaranteed to be serviced. If all of them must be serviced, the device must not generate the same message again until the driver services the earlier one.

Bit	Access	Default Value	Description
15:8	RO	00h	Reserved
7	RO	0b	64-bit Address Capable: Hardwired to 0 to indicate that the function does not implement the upper 32 bits of the Message Address register and is incapable of generating a 64-bit memory address. This may need to change in future implementations when addressable system memory exceeds the 32bit/4GB limit.
6:4	R/W	000b	Multiple Message Enable (MME): System software programs this field to indicate the actual number of messages allocated to this device. This number will be equal to or less than the number actually requested. The encoding is the same as for the MMC field below.
3:1	RO	000b	Multiple Message Capable (MMC): System software reads this field to determine the number of messages being requested by this device. Value : Number of Messages Requested 000: 1 All of the following are reserved in this implementation: 001: 2 010: 4 011: 8 100: 16 101: 32 110: Reserved. 111: Reserved.
0	R/W	0b	MSI Enable (MSIEN): Controls the ability of this device to generate MSIs.



7.2.34 MA — Message Address

B/D/F/Type: 0/2/0/PCI
Address Offset: 94-97h
Default Value: 00000000h
Access: R/W; RO;
Size: 32 bits

A read from this register produces undefined results.

Bit	Access	Default Value	Description
31:2	R/W	00000000h	Message Address: Used by system software to assign an MSI address to the device. The device handles an MSI by writing the padded contents of the MD register to this address.
1:0	RO	00b	Force Dword Align: Hardwired to 0 so that addresses assigned by system software are always aligned on a dword address boundary.

7.2.35 MD — Message Data

B/D/F/Type: 0/2/0/PCI
Address Offset: 98-99h
Default Value: 0000h
Access: R/W;
Size: 16 bits

Bit	Access	Default Value	Description
15:0	R/W	0000h	Message Data: Base message data pattern assigned by system software and used to handle an MSI from the device. When the device must generate an interrupt request, it writes a 32-bit value to the memory address specified in the MA register. The upper 16 bits are always set to 0. The lower 16 bits are supplied by this register.



7.2.36 GDRST — Graphics Device Reset

B/D/F/Type: 0/2/0/PCI
 Address Offset: C0h
 Default Value: 00h
 Access: RO; RW/L;
 Size: 8 bits

Bit	Access	Default Value	Description
7:5	RO	0h	Reserved ():
4:2	RW/L	00b	Graphics Reset Domain (GRDOM): Graphics Reset Domain (GRDOM): 000 – Full Graphics Reset will be performed (Render and Media engines and Display clock domain resets asserted) 001 – Render Engine only will be reset 011 – Media Engine only will be reset 010 – Reserved (Illegal Programming) 1XX – Reserved (Illegal Programming)
1	RO	0h	Reserved ():
0	RW/L	0b	Graphics Reset Enable (GR): Setting this bit asserts graphics-only reset. The clock domains to be reset are determined by GRDOM. Hardware resets this bit when the reset is complete. Setting this bit without waiting for it to clear, is undefined behavior. Once this bit is set to a "1" all MMIO registers associated with the selected engine(s) are returned to power on default state. Ring buffer pointers are reset, command stream fetches are dropped and ongoing render pipeline processing is halted, state machines and State Variables returned to power on default state. If the Display is reset, all display engines are halted (garbage on screen). VGA memory is not available, Store DWORDs and interrupts associated with the reset engine(s) are not guaranteed to be completed. Device #2 IO registers are not available. Device #2 Config registers continue to be available while Graphics reset is asserted.



7.2.37 GMBUSFREQ — GMBUS frequency binary encoding

B/D/F/Type: 0/2/0/PCI
Address Offset: CC-CDh
Default Value: 0000h
Access: R/W; RO;
Size: 16 bits

Bit	Access	Default Value	Description
15:10	RO	000000b	Reserved (RSVD)
9:0	R/W	0000000 000b	CMBUS CDCLK frequency (cdfreq)

7.2.38 PMCAPID — Power Management Capabilities ID

B/D/F/Type: 0/2/0/PCI
Address Offset: D0-D1h
Default Value: 0001h
Access: RO;
Size: 16 bits

Bit	Access	Default Value	Description
15:8	RO	00h	NEXT_PTR: This contains a pointer to the next item in the capabilities list.
7:0	RO	01h	CAP_ID: SIG defines this ID is 01h for power management.



7.2.39 PMCAP — Power Management Capabilities

B/D/F/Type: 0/2/0/PCI
 Address Offset: D2-D3h
 Default Value: 0022h
 Access: RO;
 Size: 16 bits

Bit	Access	Default Value	Description
15:11	RO	00h	PME Support: This field indicates the power states in which the IGD may assert PME#. Hardwired to 0 to indicate that the IGD does not assert the PME# signal.
10	RO	0b	D2: The D2 power management state is not supported. This bit is hardwired to 0.
9	RO	0b	D1: Hardwired to 0 to indicate that the D1 power management state is not supported.
8:6	RO	000b	Reserved.
5	RO	1b	Device Specific Initialization (DSI): Hardwired to 1 to indicate that special initialization of the IGD is required before generic class device driver is to use it.
4	RO	0b	Auxiliary Power Source: Hardwired to 0.
3	RO	0b	PME Clock: Hardwired to 0 to indicate IGD does not support PME# generation.
2:0	RO	01-b	Version: [DevBW] Hardwired to 010b to indicate that there are 4 bytes of power management registers implemented and that this device complies with revision 1.1 of the PCI Power Management Interface Specification. [DevCL] 010b indicates compliant with revision 1.1 of the PCI Power Management Specification. 011b indicates compliance with revision 1.2 of the PCI Power Management Specification. The value in this register is determined by the value of MCHBAR offset C08[15].



7.2.40 PMCS — Power Management Control/Status

B/D/F/Type: 0/2/0/PCI
 Address Offset: D4-D5h
 Default Value: 0000h
 Access: RO; R/W;
 Size: 16 bits

Bit	Access	Default Value	Description
15	RO	0b	PME_Status: This bit is 0 to indicate that IGD does not support PME# generation from D3 (cold).
14:13	RO	00b	Data_Scale (Reserved): The IGD does not support data register. This bit always returns 0 when read, write operations have no effect.
12:9	RO	0h	Data_Select (Reserved): The IGD does not support data register. This bit always returns 0 when read, write operations have no effect.
8	RO	0b	PME_En: This bit is 0 to indicate that PME# assertion from D3 (cold) is disabled.
7:4	RO	00h	Reserved Always returns 0 when read, write operations have no effect.
3	RO	-	[DevBW] Only: Reserved, hardwired to 0. No_Soft_Reset. Will be set according to the state of MCHBAR C08[14]. When transitioning from D3hot to D0, a 0 indicates the device performs an internal reset, a 1 indicates that the device does not perform an internal reset, and Configuration Context is preserved. Note that the state of this bit has no hardware effect – it is programmable since there is some ambiguity as to which definition of the bit our hardware behavior better matches.
2	RO	0b	Reserved Always returns 0 when read, write operations have no effect.
1:0	R/W	00b	PowerState: This field indicates the current power state of the IGD and can be used to set the IGD into a new power state. If software attempts to write an unsupported state to this field, write operation must complete normally on the bus, but the data is discarded and no state change occurs. On a transition from D3 to D0 the graphics controller is optionally reset to initial values. Behavior of the graphics controller in supported states is detailed in the power management section of the PRM. Bits[1:0] Power state 00 D0Default 01 D1Not Supported 10 D2Not Supported 11 D3



7.2.41 SWSMI — Software SMI

B/D/F/Type: 0/2/0/PCI
 Address Offset: E0-E1h
 Default Value: 0000h
 Access: R/W; R/WC;
 Size: 16 bits

As long as there is the potential that DVO port legacy drivers exist which expect this register at this address, Dev#2F0address E0h-E1h must be reserved for this register.

Bit	Access	Default Value	Description
15:8	R/W	00h	SW scratch bits:
7:1	R/W	00h	Software Flag: Used to indicate caller and SMI function desired, as well as return result.
0	R/W	0b	GMCH Software SMI Event: When set this bit will trigger an SMI. Software must clear this bit to remove the SMI condition.

7.2.42 ASLE — System Display Event Register

B/D/F/Type: 0/2/0/PCI
 Address Offset: E4-E7h
 Default Value: 00000000h
 Access: R/W;
 Size: 32 bits

The exact use of these bytes including whether they are addressed as bytes, words, or as a dword, is not pre-determined but subject to change by driver and System BIOS teams (acting in unison).

Bit	Access	Default Value	Description
31:24	R/W	00h	ASLE Scratch Trigger3: When written, this scratch byte triggers an interrupt when IEF bit 0 is enabled and IMR bit 0 is unmasked. If written as part of a 16-bit or 32-bit write, only one interrupt is generated in common.
23:16	R/W	00h	ASLE Scratch Trigger2: When written, this scratch byte triggers an interrupt when IEF bit 0 is enabled and IMR bit 0 is unmasked. If written as part of a 16-bit or 32-bit write, only one interrupt is generated in common.
15:8	R/W	00h	ASLE Scratch Trigger 1: When written, this scratch byte triggers an interrupt when IEF bit 0 is enabled and IMR bit 0 is unmasked. If written as part of a 16-bit or 32-bit write, only one interrupt is generated in common.
7:0	R/W	00h	ASLE Scratch Trigger 0: When written, this scratch byte triggers an interrupt when IEF bit 0 is enabled and IMR bit 0 is unmasked. If written as part of a 16-bit or 32-bit write, only one interrupt is generated in common.



7.2.43 SWSCI — Software SCI

B/D/F/Type: 0/2/0/PCI
Address Offset: E8-E9h
Default Value: 0000h
Access: RWO; RW;
Size: 16 bits

This register serves 2 purposes:

- 1) Support selection of SMI or SCI event source (SMISCISEL - bit15)
- 2) SCI Event trigger (GSSCIE – bit 0).

To generate a SW SCI event, software (System BIOS/Graphics driver) should program bit 15 (SMISCISEL) to 1. This is typically programmed once (assuming SMIs are never triggered). On a "0" to "1" subsequent transition in bit 0 of this register (caused by a software write operation), GMCH sends a single SCI message (as currently defined in Grantsdale GMCH EDS) down the DMI link to ICH. ICH will set the DMISCI bit in its TCO1_STS register and TCOSCI_STS bit in its GPE0 register upon receiving this message from DMI. The corresponding SCI event handler in BIOS is to be defined as a _Lxx method, indicating level trigger to the operating system.

Once written as 1, software must write a "0" to this bit to clear it, and all other write transitions (1->0, 0->0, 1->1) or if bit 15 is "0" will not cause GMCH to send SCI message to DMI link.

To generate a SW SMI event, software should program bit 15 to 0 and trigger SMI via writes to SWSMI register (See SWSMI register for programming details).

Bit	Access	Default Value	Description
15	RWO	0b	SMI or SCI event select (SMISCISEL): SMI or SCI event select (SMISCISEL)- 0 = SMI (default) 1 = SCI If selected event source is SMI, SMI trigger and associated scratch bits accesses are performed via SWSMI register at offset E0h. If SCI event source is selected, the rest of the bits in this register provide SCI trigger capability and associated SW scratch pad area.
14:1	RW	00000000 000000b	Software scratch bits (SCISB): SW scratch bits (read/write bits not used by hardware) (SCISB)
0	RW	0b	GMCH Software SCI Event (GSSCIE): If SCI event is selected (SMISCISEL = 1), on a "0" to "1" transition of GSSCIE bit, GMCH will send a SCI message via DMI link to ICH to cause the TCOSCI_STS bit in its GPE0 register to be set to 1. Software must write a "0" to clear this bit.



7.2.44 LBB — Legacy Backlight Brightness ([DevCL] Only)

B/D/F/Type: 0/2/0/PCI
Address Offset: F4-F7h
Default Value: 00000000h
Access: R/W;
Size: 32 bits

This register can be accessed by either Byte, Word, or Dword PCI config cycles. A write to this register will cause the Backlight Event (Display B Interrupt) if enabled.

Bit	Access	Default Value	Description
31:24	R/W	00h	LBPC Scratch Trigger3: When written, this scratch byte triggers an interrupt when LBEE is enabled in the Pipe B Status register and the Display B Event is enabled in IER and unmasked in IMR etc. If written as part of a 16-bit or 32-bit write, only one interrupt is generated in common.
23:16	R/W	00h	LBPC Scratch Trigger2: When written, this scratch byte triggers an interrupt when LBEE is enabled in the Pipe B Status register and the Display B Event is enabled in IER and unmasked in IMR etc. If written as part of a 16-bit or 32-bit write, only one interrupt is generated in common.
15:8	R/W	00h	LBPC Scratch Trigger1: When written, this scratch byte triggers an interrupt when LBEE is enabled in the Pipe B Status register and the Display B Event is enabled in IER and unmasked in IMR etc. If written as part of a 16-bit or 32-bit write, only one interrupt is generated in common.
7:0	R/W	00h	Legacy Backlight Brightness (LBES): The value of zero is the lowest brightness setting and 255 is the brightest. A write to this register will cause a flag to be set (LBES) in the PIPEBSTATUS register and cause an interrupt if Backlight event in the PIPEBSTATUS register and cause an Interrupt if Backlight Event (LBEE) and Display B Event is enabled by software.



7.2.45 MID2 — Manufacturing ID

B/D/F/Type: 0/2/0/PCI
Address Offset: F8-FBh
Default Value: [Device Specific]
Access: RO;
Size: 32 bits

This is an INTEL RESERVED register and should NOT be disclosed to customers. It is for test and debug purposes only and will not be included in external documentation.

Bit	Access	Default Value	Description
31:24	RO	00h	Reserved
23:16	RO	--	Manufacturing Stepping ID (MSTEP)
15:8	RO	0Fh	Foundry Code (FOUND): 0Fh: Foundry code for Intel others: Reserved These bits identify the Foundry; code of 0000 1111b = foundry code for Intel.
7:3	RO	--	Process ID (PROC)
2:0	RO	--	Dot Process (DOT)

7.2.46 ASLS — ASL Storage

B/D/F/Type: 0/2/0/PCI
Address Offset: FC-FFh
Default Value: 00000000h
Access: R/W;
Size: 32 bits

This SW scratch register only needs to be read/write accessible. The exact bit register usage must be worked out in common between System BIOS and driver software, but storage for switching/indicating up to 6 devices is possible with this amount. For each device, the ASL control method will require two bits for _DOD (BIOS detectable yes or no, VGA/NonVGA), one bit for _DGS (enable/disable requested), and two bits for _DCS (enabled now/disabled now, connected or not).

Bit	Access	Default Value	Description
31:0	R/W	00000000h	RW according to a software controlled usage to support device switching.



7.3 Device 2, Function 1

Register Name	Register Symbol	Register Start	Register End	Default Value	Access
Vendor Identification	VID2	0	1	8086h	RO;
Device Identification	DID2	2	3	[Device Specific]	RO;
PCI Command	PCICMD2	4	5	0000h	RO; R/W;
PCI Status	PCISTS2	6	7	0090h	RO;
Revision Identification	RID2	8	8	00h	RO;
Class Code	CC	9	B	038000h	RO;
Cache Line Size	CLS	C	C	00h	RO;
Master Latency Timer	MLT2	D	D	00h	RO;
Header Type	HDR2	E	E	80h	RO;
Built In Self Test	BIST	F	F	00h	RO;
Memory Mapped Range Address	MMADR	10	17	00000000 0000004h	RO; R/W;
Subsystem Vendor Identification	SVID2	2C	2D	0000h	RO;
Subsystem Identification	SID2	2E	2F	0000h	RO;
Video BIOS ROM Base Address	ROMADR	30	33	00000000h	RO;
Capabilities Pointer	CAPPOINT	34	34	D0h	RO;
Minimum Grant	MINGNT	3E	3E	00h	RO;
Maximum Latency	MAXLAT	3F	3F	00h	RO;
Capabilities Pointer (to Mirror of Dev0 CAPID)	MCAPPTR	44	44	48h	RO;
Mirror of Dev 0 Capability Identification	MCAPID	48	51	[Device Specific]	RO;
Mirror of Dev0 GMCH Graphics Control	MGGC	52	53	0030h	RO;
Mirror of Dev0 DEVEN	MDEVENdev0 F0	54	57	[Device Specific]	RO;
Software Scratch Read Write	SSRW	58	5B	00000000h	RO;
Base of Stolen Memory	BSM	5C	5F	[Device Specific]	RO;
Hardware Scratch Read Write	HSRW	60	61	0000h	RO;
Multi Size Aperture Control	MSAC	62	62	02h	RO;



7.3.1 VID2 — Vendor Identification

B/D/F/Type: 0/2/0/PCI
Address Offset: 0-1h
Default Value: 8086h
Access: RO;
Size: 16 bits

This register combined with the Device Identification register uniquely identifies any PCI device.

Bit	Access	Default Value	Description
15:0	RO	8086h	Vendor Identification Number (VID): PCI standard identification for Intel.

7.3.2 DID2 — Device Identification

B/D/F/Type: 0/2/0/PCI
Address Offset: 2-3h
Default Value: [Device Specific]
Access: RO;
Size: 16 bits

This register combined with the Vendor Identification register uniquely identifies any PCI device.

Bit	Access	Default Value	Description
15:0	RO	--	Device Identification Number (DID): Identifier assigned to the GMCH core/primary PCI device. Intel Reserved Text: Some bits of this field are actually determined by fuses, which allows unique Device IDs to be used for different product SKUs. See the device EDS for details.



7.3.3 PCICMD2 — PCI Command

B/D/F/Type: 0/2/0/PCI
 Address Offset: 4-5h
 Default Value: 0000h
 Access: RO; R/W;
 Size: 16 bits

This 16-bit register provides basic control over the IGDs ability to respond to PCI cycles. The PCICMD Register in the IGD disables the IGD PCI compliant master accesses to main memory.

Bit	Access	Default Value	Description
15:11	RO	00h	Reserved
10	R/W	0b	Interrupt Disable: This bit disables the device from asserting INTx#. 0: Enable the assertion of this device's INTx# signal. 1: Disable the assertion of this device's INTx# signal. DO_INTx messages will not be sent to DMI.
9	RO	0b	Fast Back-to-Back (FB2B): Not Implemented. Hardwired to 0.
8	RO	0b	SERR Enable (SERRE): Not Implemented. Hardwired to 0.
7	RO	0b	Address/Data Stepping Enable (ADSTEP): Not Implemented. Hardwired to 0.
6	RO	0b	Parity Error Enable (PERRE): Not Implemented. Hardwired to 0. Since the IGD belongs to the category of devices that does not corrupt programs or data in system memory or hard drives, the IGD ignores any parity error that it detects and continues with normal operation.
5	RO	0b	Video Palette Snooping (VPS): This bit is hardwired to 0 to disable snooping.
4	RO	0b	Memory Write and Invalidate Enable (MWIE): Hardwired to 0. The IGD does not support memory write and invalidate commands.
3	RO	0b	Special Cycle Enable (SCE): This bit is hardwired to 0. The IGD ignores Special cycles.
2	R/W	0b	Bus Master Enable (BME): 0: Disable IGD bus mastering. 1: Enable the IGD to function as a PCI compliant master.
1	R/W	0b	Memory Access Enable (MAE): This bit controls the IGDs response to memory space accesses. 0: Disable. 1: Enable.
0	R/W	0b	I/O Access Enable (IOAE): This bit controls the IGDs response to I/O space accesses. 0: Disable. 1: Enable.



7.3.4 PCISTS2 — PCI Status

B/D/F/Type: 0/2/0/PCI
 Address Offset: 6-7h
 Default Value: 0090h
 Access: RO;
 Size: 16 bits

PCISTS is a 16-bit status register that reports the occurrence of a PCI compliant master abort and PCI compliant target abort. PCISTS also indicates the DEVSEL# timing that has been set by the IGD.

Bit	Access	Default Value	Description
15	RO	0b	Detected Parity Error (DPE): Since the IGD does not detect parity, this bit is always hardwired to 0.
14	RO	0b	Signaled System Error (SSE): The IGD never asserts SERR#, therefore this bit is hardwired to 0.
13	RO	0b	Received Master Abort Status (RMAS): The IGD never gets a Master Abort, therefore this bit is hardwired to 0.
12	RO	0b	Received Target Abort Status (RTAS): The IGD never gets a Target Abort, therefore this bit is hardwired to 0.
11	RO	0b	Signaled Target Abort Status (STAS): Hardwired to 0. The IGD does not use target abort semantics.
10:9	RO	00b	DEVSEL Timing (DEVT): N/A. These bits are hardwired to "00".
8	RO	0b	Master Data Parity Error Detected (DPD): Since Parity Error Response is hardwired to disabled (and the IGD does not do any parity detection), this bit is hardwired to 0.
7	RO	1b	Fast Back-to-Back (FB2B): Hardwired to 1. The IGD accepts fast back-to-back when the transactions are not to the same agent.
6	RO	0b	User Defined Format (UDF): Hardwired to 0.
5	RO	0b	66 MHz PCI Capable (66C): N/A - Hardwired to 0.
4	RO	1b	Capability List (CLIST): This bit is set to 1 to indicate that the register at 34h provides an offset into the function's PCI Configuration Space containing a pointer to the location of the first item in the list.
3	RO	0b	Interrupt Status: This bit reflects the state of the interrupt in the device. Only when the Interrupt Disable bit in the command register is a 0 and this Interrupt Status bit is a 1, will the device's INTx# signal be asserted. Setting the Interrupt Disable bit to a 1 has no effect on the state of this bit.
2:0	RO	000b	Reserved.



7.3.5 RID2 — Revision Identification

B/D/F/Type: 0/2/0/PCI
Address Offset: 8h
Default Value: 00h
Access: RO;
Size: 8 bits

Compatible Revision ID (CRID):

An 8 bit hardwired value assigned by the ID Council. Normally, the value assigned as the CRID will be identical to the SRID value of a previous stepping of the product with which the new product is deemed "compatible". Note that CRID is not an addressable PCI register. The CRID value is simply reflected through the RID register when appropriately selected. Lower 4 bits of the CRID are driven by Fuses. The CRID fuses are programmed based on the SKU.

Stepping Revision ID (SRID):

An 8 bit hardwired value assigned by the ID Council. The values assigned as the SRID of a product's steppings will be selectively incremented based on the degree of change to that stepping. It is suggested that the first stepping of any given product have an SRID value = 01h simply to avoid the "reserved register" value of 00h. Note that SRID is not an addressable PCI register. The SRID value is simply reflected through the RID register when appropriately selected.

RID Select Key Value:

This is hardwired value (69h). If the latched value written to the RID register address matches this RID Select Key Value, the CRID value be presented for reading from the RID register.

RID Definition:

This register contains the revision number of the GMCH Device #0. Following PCI Reset the SRID value is selected to be read. When a write occurs to this register the write data is compared to the hardwired RID Select Key Value which is 69h. If the data matches this key a flag is set that enables the CRID value to be read through this register.

Note that the flag is a "write once". Therefore once the CRID is selected to be read, the only way to again select the SRID is to PCI Reset the component. Also if any value other than the key (69h) is written to the RID register, the flag is locked such that the SRID is selected until the component is PCI Reset. Note that the RID register itself is not directly write-able.

This register contains the revision number for Device #2 Functions 0 and 1.

Bit	Access	Default Value	Description
7:0	RO	00h	Revision Identification Number (RID): This is an 8-bit value that indicates the revision identification number for the GMCH.



7.3.6 CC — Class Code

B/D/F/Type: 0/2/0/PCI
Address Offset: 9-Bh
Default Value: 038000h
Access: RO;
Size: 24 bits

This register contains the device programming interface information related to the Sub-Class Code and Base Class Code definition for the IGD. This register also contains the Base Class Code and the function sub-class in relation to the Base Class Code.

Bit	Access	Default Value	Description
23:16	RO	03h	Base Class Code (BCC): This is an 8-bit value that indicates the base class code for the GMCH. This code has the value 03h, indicating a Display Controller.
15:8	RO	80h	Sub-Class Code (SUBCC): 80h: Non VGA
7:0	RO	00h	Programming Interface (PI): 00h: Hardwired as a Display controller.

7.3.7 CLS — Cache Line Size

B/D/F/Type: 0/2/0/PCI
Address Offset: Ch
Default Value: 00h
Access: RO;
Size: 8 bits

The IGD does not support this register as a PCI slave.

Bit	Access	Default Value	Description
7:0	RO	00h	Cache Line Size (CLS): This field is hardwired to 0s. The IGD as a PCI compliant master does not use the Memory Write and Invalidate command and, in general, does not perform operations based on cache line size.



7.3.8 MLT2 — Master Latency Timer

B/D/F/Type: 0/2/0/PCI
 Address Offset: Dh
 Default Value: 00h
 Access: RO;
 Size: 8 bits

The IGD does not support the programmability of the master latency timer because it does not perform bursts.

Bit	Access	Default Value	Description
7:0	RO	00h	Master Latency Timer Count Value: Hardwired to 0s.

7.3.9 HDR2 — Header Type

B/D/F/Type: 0/2/0/PCI
 Address Offset: Eh
 Default Value: 80h
 Access: RO;
 Size: 8 bits

This register contains the Header Type of the IGD.

Bit	Access	Default Value	Description
7	RO	1b	Multi Function Status (MFunc): Indicates if the device is a Multi-Function Device. The Value of this register is determined by Device #0, offset 54h, DEVEN[4]. If Device #0 DEVEN[4] is set, the Mfunc bit is also set.
6:0	RO	00h	Header Code (H): This is a 7-bit value that indicates the Header Code for the IGD. This code has the value 00h, indicating a type 0 configuration space format.

7.3.10 BIST — Built In Self Test

B/D/F/Type: 0/2/0/PCI
 Address Offset: Fh
 Default Value: 00h
 Access: RO;
 Size: 8 bits

This register is used for control and status of Built In Self Test (BIST).

Bit	Access	Default Value	Description
7	RO	0b	BIST Supported: BIST is not supported. This bit is hardwired to 0.
6:0	RO	00h	Reserved



7.3.11 MMADR — Memory Mapped Range Address

B/D/F/Type: 0/2/1/PCI
Address Offset: 10-17h
Default Value: 0000000000000004h
Access: RO; R/W;
Size: 64 bits

This register requests allocation for the IGD registers and instruction ports. The allocation is for 512 KB and the base address is defined by bits [35:20].

Bit	Access	Default Value	Description
63:36	RO	0000000h	Reserved ():
35:20	R/W	0000h	Memory Base Address (): Set by the OS, these bits correspond to address signals [35:20].
19:4	RO	0000h	Address Mask (): Hardwired to 0's to indicate 512 KB address range (aligned to 1M boundary).
3	RO	0b	Prefetchable Memory (): Hardwired to 0 to prevent prefetching.
2	RO	1b	Memory Type (): 0 : To indicate 32 bit base address 1 : To indicate 64 bit base address
1	RO	0b	Reserved ():
0	RO	0b	Memory / IO Space (): Hardwired to 0 to indicate memory space.

7.3.12 SVID2 — Subsystem Vendor Identification

B/D/F/Type: 0/2/0/PCI
Address Offset: 2C-2Dh
Default Value: 0000h
Access: RO;
Size: 16 bits

Bit	Access	Default Value	Description
15:0	RO	0000h	Subsystem Vendor ID: This value is used to identify the vendor of the subsystem. This register should be programmed by BIOS during boot-up. Once written, this register becomes Read-Only. This register can only be cleared by a Reset. NOTE: This is a RO copy of the Dev2Fxn0 value.



7.3.13 SID2 — Subsystem Identification

B/D/F/Type: 0/2/0/PCI
Address Offset: 2E-2Fh
Default Value: 0000h
Access: RO;
Size: 16 bits

Bit	Access	Default Value	Description
15:0	RO	0000h	Subsystem Identification: This value is used to identify a particular subsystem. This field should be programmed by BIOS during boot-up. Once written, this register becomes Read-Only. This register can only be cleared by a Reset. NOTE: This is a RO copy of the Dev2Fxn0 value.

7.3.14 ROMADR — Video BIOS ROM Base Address

B/D/F/Type: 0/2/0/PCI
Address Offset: 30-33h
Default Value: 00000000h
Access: RO;
Size: 32 bits

The IGD does not use a separate BIOS ROM, therefore this register is hardwired to 0s.

Bit	Access	Default Value	Description
31:18	RO	0000h	ROM Base Address: Hardwired to 0s.
17:11	RO	00h	Address Mask: Hardwired to 0s to indicate 256 KB address range.
10:1	RO	000h	Reserved: Hardwired to 0s.
0	RO	0b	ROM BIOS Enable: 0 = ROM not accessible.

7.3.15 CAPPOINT — Capabilities Pointer

B/D/F/Type: 0/2/0/PCI
Address Offset: 34h
Default Value: D0h
Access: RO;
Size: 8 bits

Bit	Access	Default Value	Description
7:0	RO	D0h	Capabilities Pointer Value: This field contains an offset into the function's PCI Configuration Space for the first item in the New Capabilities Linked List which the Power Management Capabilities ID registers at address D0h.



7.3.16 MINGNT — Minimum Grant

B/D/F/Type: 0/2/0/PCI
Address Offset: 3Eh
Default Value: 00h
Access: RO;
Size: 8 bits

Bit	Access	Default Value	Description
7:0	RO	00h	Minimum Grant Value: The IGD does not burst as a PCI compliant master.

7.3.17 MAXLAT — Maximum Latency

B/D/F/Type: 0/2/0/PCI
Address Offset: 3Fh
Default Value: 00h
Access: RO;
Size: 8 bits

Bit	Access	Default Value	Description
7:0	RO	00h	Maximum Latency Value: The IGD has no specific requirements for how often it needs to access the PCI bus.

7.3.18 MCAPPTR — Capabilities Pointer (to Mirror of Dev0 CAPID)

B/D/F/Type: 0/2/0/PCI
Address Offset: 44h
Default Value: 48h
Access: RO;
Size: 8 bits

Bit	Access	Default Value	Description
7:0	RO	48h	Capabilities Pointer Value: In this case the first capability is the product-specific Capability Identifier (CAPID0).



7.3.19 MCAPID — Mirror of Dev 0 Capability Identification.

B/D/F/Type: 0/2/0/PCI
Address Offset: 48-51h
Default Value: [Device Specific]
Access: RO;
Size: 80 bits

This is an INTEL RESERVED register and should NOT be disclosed to customers. It is for test and debug purposes only and will not be included in external documentation. Control of bits in this register are only required for customer visible SKU differentiation.

Bit	Access	Default Value	Description
79:0	RO	--	Device Specific Bit Definitions – see the device EDS for details.



7.3.20 MGGC — Mirror of Dev0 GMCH Graphics Control

B/D/F/Type: 0/2/0/PCI
 Address Offset: 52-53h
 Default Value: 0030h
 Access: RO;
 Size: 16 bits

Bit	Access	Default Value	Description
15:7	RO	000000000b	Reserved
6:4	RO	011b	Graphics Mode Select (GMS): This field is used to select the amount of Main Memory that is pre-allocated to support the Internal Graphics device in VGA (non-linear) and Native (linear) modes. The BIOS ensures that memory is pre-allocated only when Internal graphics is enabled. Stolen Memory Base is located between (TOLUD - SMSize) to TOUTD. 000 = No memory pre-allocated. Device 2 (IGD) does not claim VGA cycles (Mem and IO), and the Sub-Class Code field within Device 2 function 0. Class Code register is 80. 001 = DVMT (UMA) mode, 1 MB memory pre-allocated for frame buffer. 010 = DVMT (UMA) mode, 4 MB memory pre-allocated for frame buffer. 011 = DVMT (UMA) mode, 8 MB memory pre-allocated for frame buffer. 100 = DVMT (UMA) mode, 16 MB memory pre-allocated for frame buffer. 101 = DVMT (UMA) mode, 32 MB memory pre-allocated for frame buffer. 110 = DVMT (UMA) mode, 48 MB memory pre-allocated for frame buffer. 111 = DVMT (UMA) mode, 64 MB memory pre-allocated for frame buffer. Note: This register is locked and becomes Read Only when the D_LCK bit in the SMRAM register is set. Hardware does not clear or set any of these bits automatically based on IGD being disabled/enabled.
3:2	RO	00b	Reserved
1	RO	0b	IGD VGA Disable (IVD): 1 = Disable. Device 2 (IGD) does not claim VGA cycles (Mem and IO), and the Sub-Class Code field within Device 2 function 0 Class Code register is 80. 0 = Enable (Default). Device 2 (IGD) claims VGA memory and IO cycles, the Sub-Class Code within Device 2 Class Code register is 00.
0	RO	0b	Reserved

7.3.21 MDEVNdev0F0 — Mirror of Dev0 DEVEN

B/D/F/Type: 0/2/0/PCI



Address Offset: 54-57h
Default Value: [Device Specific]
Access: RO;
Size: 32 bits

Allows for enabling/disabling of PCI devices and functions that are within the MCH.

Bit	Access	Default Value	Description
31:0	RO	--	Device Specific Bit Definitions. See Device 0 documentation in the EDS.

7.3.22 SSRW — Software Scratch Read Write

B/D/F/Type: 0/2/0/PCI
Address Offset: 58-5Bh
Default Value: 00000000h
Access: RO;
Size: 32 bits

Bit	Access	Default Value	Description
31:0	RO	00000000h	Reserved

7.3.23 BSM — Base of Stolen Memory

B/D/F/Type: 0/2/0/PCI
Address Offset: 5C-5Fh
Default Value: [Device Specific]
Access: RO;
Size: 32 bits

Graphics Stolen Memory and TSEG are within DRAM space defined under TOLUD. From the top of low used DRAM, GMCH claims 1 to 64MBs of DRAM for internal graphics if enabled.

Bit	Access	Default Value	Description
31:20	RO	--	Base of Stolen Memory (BSM): This register contains bits 31 to 20 of the base address of stolen DRAM memory. The host interface determines the base of graphics stolen memory by subtracting the graphics stolen memory size from TOLUD. See Device 0 TOLUD in the EDS for more explanation.
19:0	RO	00000h	Reserved

7.3.24 HSRW — Hardware Scratch Read Write

B/D/F/Type: 0/2/0/PCI
Address Offset: 60-61h
Default Value: 0000h



Access: RO;
Size: 16 bits

Bit	Access	Default Value	Description
15:0	RO	0000h	Reserved

7.3.25 MSAC — Multi Size Aperture Control

B/D/F/Type: 0/2/0/PCI
Address Offset: 62h
Default Value: 02h
Access: RO;
Size: 8 bits

This register determines the size of the graphics memory aperture in function. By default the aperture size is 256 MB. Only the system BIOS will write this register based on pre-boot address allocation efforts, but the graphics may read this register to determine the correct aperture size. System BIOS needs to save this value on boot so that it can reset it correctly during S3 resume.

Bit	Access	Default Value	Description
7:4	RO	0h	Scratch Bits Only: Have no physical effect on hardware.
3	RO	0b	Reserved
2:1	RO	01b	Aperture Size (LHSAS): 11: bits [28:27] of GMADR register are made Read only and forced to zero, allowing only 512MB of GMADR 01: bit [28] of GMADR is made R/W and bit [27] of GMADR is forced to zero allowing 256MB of GMADR 00: bits [28:27] of GMADR register are made R/W allowing 128MB of GMADR 10: Illegal programming.
0	RO	0b	Reserved



8 *Memory Interface Registers*

8.1 Introduction

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

The registers detailed in this chapter are used across the Gen4 family of products. However, slight changes may be present in some registers (i.e., for features added or removed), or some registers may be removed entirely. These changes are clearly marked within this chapter.

8.2 Virtual Memory Control

Gen4 products differ somewhat in the types of virtual memory they support and how they support it. The following table describes the structures to support Global virtual memory (shared between all GFX processes) and per-process virtual memory.

Virtual Memory Structure	All
Global (GGTT)	Anywhere
Per-Process (PPGTT)	Single-level, anywhere

8.2.1 Global Virtual Memory

Global Virtual Memory is the default target memory if a PPGTT is not enabled (or for products that don't support PPGTT). If a PPGTT is also present, the method to choose which is targeted by memory and rendering operations varies by product. See the sections on Per-Process Virtual Memory for more information. High priority graphics clients such as Display and Cursor always access global virtual memory.



8.2.1.1 PGTBL_CTL—Page Table Control Register

PGTBL_CTL—Page Table Control Register	
Register Type: MMIO Address Offset: 2020h Project: All Default Value: 0000 0000h Access: R/W Size (in bits): 32	
The PGTBL_CTL register is used to enable or disable the mapping of graphics memory using the Global Graphics Translation Table (GGTT), set the size, and to set the base (physical) address of the GGTT. <i>Software must use the following steps to modify the Global GTT directly or update the Global GTT base:</i> 1. Flush the Gfx Pipeline 2. Flush the Chipset write buffers using the flush (GFX_FLSH_CTL) register 3. Update Global GTT using physical address/Update the Global GTT base register 4. Flush Chipset write buffers using the flush (GFX_FLSH_CTL) register The GGTT must be 4KByte aligned. The GGTT must reside in un-snooped Main Memory and must be contiguous. The GGTT must be completely contained within physical memory. A memory access that requires fetching a GGTT entry that is not in physical memory will have UNDEFINED results. [All Devices]: Software can use the GTTADR space to update entries in the GGTT. This allows the device to “snoop” writes to GTTADR and invalidate internal Translation Look-aside Buffers (TLBs) as required. This register is <i>not</i> reset by a <u>graphics</u> reset. It will maintain its value unless a full chipset reset is performed.	
Bit	Description
31:12	Page Table Base Address Project: All Default Value: 0h Address: GraphicsAddress[31:12] Surface Type: PageTableEntry This field specifies Bits 31:12 of the starting address of the global GTT. This address is a physical offset into system memory. This address must be in physical memory, i.e., it must be below the top of memory. Furthermore, the GGTT must be entirely contained within physical memory, i.e., the GTT Size added to the Page Table Base Address must be below top of memory. This field is only valid when the Page Table Enable field is specified as ENABLED. Programming Notes The base address for the GTT is expected to be size aligned in memory. Eg for 512KB size of the GTT bits 18:12 of the address need to be zero Project DevCL
11:8	Reserved Project: All Format: MBZ



PGTBL_CTL—Page Table Control Register																																				
7:4	Physical Start Address Extension Project: All Default Value: 0h Address: GraphicsAddress[35:32] This field specifies Bits 35:32 of the starting address of the GGTT.																																			
3:1	Size of the Global GTT Project: All Default Value: 0h Format: U3 <table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>000</td><td>512KB</td><td>512KB</td><td>All</td></tr><tr><td>001</td><td>256KB</td><td>256KB</td><td>All</td></tr><tr><td>010</td><td>128KB</td><td>128KB</td><td>All</td></tr><tr><td>011</td><td>1MB</td><td>1MB</td><td>Reserved</td></tr><tr><td>100</td><td>2MB</td><td>2MB</td><td>Reserved</td></tr><tr><td>101</td><td>1.5MB</td><td>1.5MB</td><td>Reserved</td></tr><tr><td>11X</td><td>Reserved</td><td>Reserved</td><td>All</td></tr></table>				Value	Name	Description	Project	000	512KB	512KB	All	001	256KB	256KB	All	010	128KB	128KB	All	011	1MB	1MB	Reserved	100	2MB	2MB	Reserved	101	1.5MB	1.5MB	Reserved	11X	Reserved	Reserved	All
Value	Name	Description	Project																																	
000	512KB	512KB	All																																	
001	256KB	256KB	All																																	
010	128KB	128KB	All																																	
011	1MB	1MB	Reserved																																	
100	2MB	2MB	Reserved																																	
101	1.5MB	1.5MB	Reserved																																	
11X	Reserved	Reserved	All																																	
0	Page Table Enable Project: All Security: None Default Value: 0h Format: Enable This field determines whether GM mappings are enabled. If disabled, GM mapping does not occur except for requests from the CPU and VGA streams. Any accesses to GM (other than CPU read, and VGA streams) while this bit is clear generates a Page Table HW Error (see Page Table Error in <i>Programming Interface</i>). Note: The source of the Page Table HW Error is available only via the <i>Debug</i> PGTBL_ER register. <table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td>Disable</td><td>GM mapping does not occur except for requests from the CPU and VGA streams.</td><td>All</td></tr><tr><td>1h</td><td>Enable</td><td>ENABLED</td><td>All</td></tr></table>				Value	Name	Description	Project	0h	Disable	GM mapping does not occur except for requests from the CPU and VGA streams.	All	1h	Enable	ENABLED	All																				
Value	Name	Description	Project																																	
0h	Disable	GM mapping does not occur except for requests from the CPU and VGA streams.	All																																	
1h	Enable	ENABLED	All																																	



8.2.1.2 PGTBL_ER—Page Table Error Register (*Debug*)

PGTBL_ER—Page Table Error Register				
Register Type: MMIO				
Address Offset: 2024h				
Project: All				
Default Value: 0000 0000h				
Access: RO				
Size (in bits): 32				
This register applies when the Per-Process Virtual Address Space and Run List Enable is clear else see below The PGTBL_ER <i>Debug</i> register stores information indicating the source of an error associated with GM mapping via the GTT. Note that this is a READ-ONLY register and cannot be modified by software. Error Types: • XX_INVALID_GTT_PTE: Translated Page Table Entry (PTE) is marked as not valid. Implemented by all streams. Detected at translation time for either Global or Per-Process GTT. • XX_INVALID_PTE_DATA: The PTE was marked valid, though the memory space or page mapped is not considered legal (i.e., Address points to PAM, SMM, over TOM and other restricted spaces in Main Memory). Implemented by Host Only. • CS_INVALID_GTT: Set if a ring buffer is active and the Page table is not enabled. This register identifies the TLB that detected the error. After an error, Normal priority data streams Commands, Render Cache and Mapping Cache stop execution. GTT errors on Host reads are not recorded. If there is an error on a read access a read request is forwarded to a memory address and data obtained from memory is returned to the CPU. Errors on Host writes are recorded and the write is completed with byte enables off. Each Source records the first error and ignores subsequent errors.				
Bit	Description			
31:27	Reserved	Project:	All	Format: MBZ
26	MT_INVALID_GTT_PTE	Project:	All	Format: Flag
Invalid Sampler Cache GTT entry				
25	Reserved	Project:	All	Format: MBZ
24	LC_INVALID_GTT_PTE	Project:	All	Format: Flag
Default Value: 0h				
Invalid Render Cache GTT entry				
Errata		Description		Project
GEN4016		This bit will never be set.		All
23	ISC_INVALID_GTT_PTE	Project:	All	Format: Flag
Invalid Instruction/State Cache GTT entry				



PGTBL_ER—Page Table Error Register											
22	ROC_INVALID_GTT_PTE Reserved since there is no ROC		Project:	All	Format: Flag						
21	CS_VertexData_INVALID_GTT_PTE Invalid GTT Entry during Vertex Fetch		Project:	All	Format: Flag						
20	CS_Command_INVALID_GTT_PTE Invalid GTT Entry during Command Fetch		Project:	All	Format: Flag						
19	CS_INVALID_GTT		Project:	All	Format: Flag						
18	CRSR_INVALID_GTT_PTE Invalid GTT Entry during Cursor Fetch		Project:	All	Format: Flag						
17	Reserved	Project: All	Format: MBZ								
16	OVRL_INVALID_GTT_PTE Project: All Default Value: 0h Format: Flag Invalid GTT Entry during Overlay Fetch <table><tr><th>Errata</th><th>Description</th><th>Project</th></tr><tr><td>BWT010</td><td>Invalid GTT Entry during Overlay Fetch is ignored.</td><td>DevBW-A, DevBW-B</td></tr></table>					Errata	Description	Project	BWT010	Invalid GTT Entry during Overlay Fetch is ignored.	DevBW-A, DevBW-B
Errata	Description	Project									
BWT010	Invalid GTT Entry during Overlay Fetch is ignored.	DevBW-A, DevBW-B									
15:13	Reserved	Project: All	Format: MBZ								
12	DISPC_INVALID_GTT_PTE Invalid GTT Entry during Display C Fetch		Project:	All	Format: Flag						
11:9	Reserved	Project: All	Format: MBZ								
8	DISPB_INVALID_GTT_PTE Invalid GTT Entry during Display B Fetch		Project:	All	Format: Flag						
7:5	Reserved	Project: All	Format: MBZ								
4	DISPA_INVALID_GTT_PTE Invalid GTT Entry during Display A Fetch		Project:	All	Format: Flag						
3:2	Reserved	Project: All	Format: MBZ								
1	HOST_INVALID_PTE_DATA Valid PTE references illegal memory, such as PAM, SMM or TOM		Project:	All	Format: Flag						



PGTBL_ER—Page Table Error Register								
0	HOST_INVALID_GTT_PTE Project: All Default Value: 0h Format: Flag Invalid GTT Entry during Fetch on behalf of the Host <table><thead><tr><th>Errata</th><th>Description</th><th>Project</th></tr></thead><tbody><tr><td>BWT015</td><td>This bit will never be set.</td><td>DevBW</td></tr></tbody></table>		Errata	Description	Project	BWT015	This bit will never be set.	DevBW
Errata	Description	Project						
BWT015	This bit will never be set.	DevBW						

8.2.1.3 Graphics Translation Table (GTT) Range (GTTADR)

Address Offset: GTTADR in CPU Physical Space

Access: Aligned DWord Read/Write

The GTTADR memory BAR defined in graphics device config space is an alias for the Global GTT.

Programming Notes: It is recommended that the driver map all graphics memory pages in the GGTT to some physical page, if only a dummy page.



8.2.1.4 GTT Page Table Entries (PTEs)

Page Table Entry: 1 DWord per 4KB Graphics Memory page.

31	12	11:8	7:4	3	2	1	0
Physical Page Address 31:12		Reserved:MBZ	Physical Page Address 35:32	Reserved	Mapping Type		Valid

Bits	Description
31: 12	Physical Page Address 31:12: If the Valid bit is set, This field provides the page number of the physical memory page backing the corresponding Graphics Memory page.
11:8	Reserved: MBZ
7:4	Physical Start Address Extension: This field specified Bits 35:32 of the page table entry. This field must be zero for 32 bit addresses.
3	Reserved: MBZ
2:1	Mapping Type: If the Valid bit is set, this field specifies the type of physical memory backing this Graphics Memory page, as defined below: 0: Physical address targets Main Memory (not snooped). Physical address is a main memory page number (including pages in stolen memory). 1-2: Reserved 3: Physical address targets cacheable Main Memory (aka System Memory) (causes snoop on processor bus). Must not be targeted by the processor through graphics memory range. Accesses via the Instruction stream are permitted (no error generated), yet treated as unsnooped Main Memory. This removes restrictions regarding Instruction stream overfetches into dissimilar graphics memory regions.
0	Valid PTE: This field indicates whether the mapping of the corresponding Graphics Memory page is valid. 1: Valid 0: Invalid. An access (other than a CPU Read) through an invalid PTE will result in Page Table Error (Invalid PTE).



8.2.2 Single-Level (Flat) Per-Process Virtual Memory

8.2.2.1 PGTBL_CTL2— Per Process Page Table Control Register

PGTBL_CTL2— Per Process Page Table Control Register	
Register Type: MMIO Address Offset: 20C4h Project: All Default Value: 0000 0000h Access: R/W Size (in bits): 32	
The PGTBL_CTL2 register is used to enable the secondary mapping of graphics memory addresses by defining the starting point of the per-process Graphics Translation Table (PPGTT). Software must assure that a pipeline flush occurs subsequent to updating any PPGTT entries or changing the value of the Page Table Base Address and prior to any new access in the PPGTT aperture. Once a PPGTT is established, software can update entries of the PPGTT using physical writes. The PPGTT does not have an access window corresponding to GTTADR that will trigger snoops and/or flushes when possibly pre-fetched entries are modified. The PPGTT can be up to 1MB in size as programmed below. Each 4B entry in the PPGTT corresponds to a 4KB page of memory mapped through the PPGTT aperture. The PPGTT must be 4KByte-aligned. The PPGTT must reside in unsnooped Main Memory and must be contiguously size aligned. This register is saved and restored per context. If the valid bit for this register is not set, the hardware uses the Global GTT.	
Bit	Description
31:12	Page Table Base Address Project: All Default Value: 0h Address: GraphicsAddress[31:12] Surface Type: PageTableEntry This field specifies Bits 31:12 of the starting address of the GTT. Bit 1 of the address is MBZ. This address is a physical offset into system memory. This field is only valid when the Page Table Enable field is specified as ENABLED. Format = "Effective Local Memory Address" Bits 31:2
11:8	Reserved Project: All Format: MBZ
7:4	Physical Start Address Extension Project: All Default Value: 0h Address: GraphicsAddress[35:2] This field specified Bits 35:32 of the page table entry. This field must be zero for 32 bit addresses.



PGTBL_CTL2— Per Process Page Table Control Register				
3:1	Size of the PPGTT			
	Project:		All	
	Default Value:		0h	
	Format:		U3	
	Value	Name	Description	Project
	000	64KB	64KB	All
	001	128KB	128KB	All
	010	256KB	256KB	All
	011	512KB	512KB	All
	100	1MB	1MB	All
	101-111	Reserved	Reserved	All
0	Page Table Enable			
	Project:		All	
	Default Value:		0h	
	Format:		Enable	
	This field determines whether GM mappings are enabled. If enabled, the Page Table Base Address specifies the starting address of the PGTT. If disabled, GM mapping will proceed using the global GTT.			
	Value	Name	Description	Project
	0h	Disable	GM mapping will proceed using the global GTT.	All
1h	Enable	The Page Table Base Address specifies the starting address of the PGTT	All	



8.2.2.2 PGTBL_STR2—Page Table Steer Register (Per Process)

PGTBL_STR2—Page Table Steer Register (Per Process)			
Register Type: MMIO			
Address Offset: 20C8h			
Project: All			
Default Value: 0000 0000h			
Access: R/W			
Size (in bits): 32			
The PGTBL_STR2 register is used to map the graphics functions to either the per-process GTT or the global GTT. This register is saved and restored with context. If the valid bit for the per-process GTT is not set, the hardware uses the Global GTT for all functions and ignores the contents of this register.			
Bit	Description		
31:22	Reserved	Project: All	Format: MBZ
21:16	Write enable bits	Project: All	Format: Mask[5:0]
	This bit needs to be set in order to change the value for the corresponding location of register bits 5:0		
15:6	Reserved	Project: All	Format: MBZ
5	Location of the render batch buffer		
	Project:	All	
	Default Value:	0h	
	Format:	U1	
	Location of the render batch buffer		
	Value	Name	Description
	0h		Batch buffer accesses are translated through the global GTT
	1h		Batch buffer accesses are translated through the per-process GTT (PGTT).
4	Location of indirect state buffers includes states and instructions		
	Project:	All	
	Default Value:	0h	
	Format:	U1	
	Location of indirect state buffers includes states and instructions		
	Value	Name	Description
	0h		Indirect state buffer accesses are translated through the global GTT
	1h		Indirect state buffer accesses are translated through the per-process GTT (PGTT).



PGTBL_STR2—Page Table Steer Register (Per Process)				
3	Location of Vertex buffer			
	Project:		All	
	Default Value:		0h	
	Format:		U1	
	Location of Vertex buffer			
	Value	Name	Description	Project
	0h		Vertex buffer accesses are translated through the global GTT	All
	1h		Vertex buffer accesses are translated through the per-process GTT (PGTT).	All
2	Reserved	Project:	All	Format: MBZ
1	Location of functions using the Sampler cache			
	Project:		All	
	Default Value:		0h	
	Format:		U1	
	Location of functions using the Sampler cache			
	Value	Name	Description	Project
	0h		Sampler surface accesses are translated through the global GTT	All
	1h		Sampler surface accesses are translated through the per-process GTT (PGTT).	All
0	Location of functions using the render cache			
	Project:		All	
	Default Value:		0h	
	Format:		U1	
	Includes Render targets, constants, Scratch Space access and direct reads/writes from EUs to memory.			
	Value	Name	Description	Project
	0h		Render surface accesses are translated through the global GTT	All
	1h		Render surface accesses are translated through the per-process GTT (PGTT).	All



8.2.3 TLB Read Interface

It may be necessary for one or more pages belonging to a context to be unmapped from its PPGTT in order to map other pages when resolving a page fault. Pages that get unmapped cannot be one of the set that the HW is currently using. SW should read all of the TLB entry virtual addresses in order to report these virtual page addresses to the OS/Scheduler such that it can avoid swapping these pages out in order to bring in a page to resolve a fault.

8.2.3.1 TLB_RD_EXT — TLB Read Extent

TLB_RD_EXT -- TLB Read Extent	
Register Type: MMIO Address Offset: 251Ch Project: All Default Value: 0000 0780h Access: RO Size (in bits): 32	
This RO register can be read by software to determine how many TLB Read entries follow. SW must read the entire set to make sure all in-use pages are reported during the servicing of a page fault.	
Bit	Description
31:2	TLB Read Extent Project: All Default Value: 01E0h ??? Format: U30 This RO register is hardwired to a count of the total number of TLB read registers SW can read this register to determine the total range of potentially valid DWs in the TLB read range.
1:0	Reserved Project: All Format: MBZ

Unused registers in the range below, from B000h to B000h + TLB Read Extent, should be treated as reserved and read as 0. This allows SW to read the entire range contiguously and maintain proper behavior when reading unused, reserved registers.



8.2.3.2 Instruction/State Cache (ISC)

Instruction/State Cache (ISC)		
Register Type: MMIO Address Offset: B000h Project: All Default Value: TBD Access: RO Size (in bits): 16x32		
DWord	Bit	Description
0..15	31:12	TLB Page Address Project: All Address: GraphicsVirtualAddress[31:12] If the Valid bit is set, this field contains the page address of the TLB entry.
	11:2	Reserved Project: All Format: MBZ
	1	Global GTT Address Project: All Format: MBZ Hardwired to 0.
	0	Valid Project: All Format: Enable If this bit is set, this entry contains a valid TLB entry. If clear, this TLB entry is effectively "empty" and may be disregarded.



8.2.3.3 Vertex Fetch (VF)

Vertex Fetch (VF)		
Register Type: MMIO Address Offset: B100h Project: All Default Value: TBD Access: RO Size (in bits): 19x32		
DWord	Bit	Description
0..18	31:12	TLB Page Address Project: All Address: GraphicsVirtualAddress[31:12] If the Valid bit is set, this field contains the page address of the TLB entry.
	11:2	Reserved Project: All Format: MBZ
	1	Global GTT Address Project: All Format: MBZ Hardwired to 0.
	0	Valid Project: All Format: Enable If this bit is set, this entry contains a valid TLB entry. If clear, this TLB entry is effectively "empty" and may be disregarded.



8.2.3.4 Command Streamer (CS)

Command Streamer (CS)		
Register Type: MMIO Address Offset: B200h Project: All Default Value: TBD Access: RO Size (in bits): 6x32		
DWord	Bit	Description
0..5	31:12	TLB Page Address Project: All Address: GraphicsVirtualAddress[31:12] If the Valid bit is set, this field contains the page address of the TLB entry.
	11:2	Reserved Project: All Format: MBZ
	1	Global GTT Address Project: All Format: Flag If set, this virtual address is a global GTT address, and is guaranteed to remain mapped. Only TLB entries with this bit clear need to be communicated as being part of a minimum set that must remain mapped during the servicing of a page fault. Only the Command Streamer (CS) may contain global GTT entries in its TLB; all the other clients will hardwire this bit to 0 in all of their TLB read registers.
	0	Valid Project: All Format: Enable If this bit is set, this entry contains a valid TLB entry. If clear, this TLB entry is effectively “empty” and may be disregarded.



8.2.3.5 Texture Cache (MT)

Texture Cache (MT)		
Register Type: MMIO Address Offset: B300h Project: All Default Value: ??? Access: RO Size (in bits): 32x32		
DWord	Bit	Description
0..31	31:12	TLB Page Address Project: All Address: GraphicsVirtualAddress[31:12] If the Valid bit is set, this field contains the page address of the TLB entry.
	11:2	Reserved Project: All Format: MBZ
	1	Global GTT Address Project: All Format: MBZ Hardwired to 0.
	0	Valid Project: All Format: Enable If this bit is set, this entry contains a valid TLB entry. If clear, this TLB entry is effectively "empty" and may be disregarded.



8.2.3.6 Render Cache (RC)

Render Cache (RC)		
Register Type: MMIO Address Offset: B400h Project: All Default Value: ??? Access: RO Size (in bits): 224x32		
224 DWords		
DWord	Bit	Description
0.223	31:12	TLB Page Address Project: All Address: GraphicsVirtualAddress[31:12] If the Valid bit is set, this field contains the page address of the TLB entry.
	11:2	Reserved Project: All Format: MBZ
	1	Global GTT Address Project: All Format: MBZ Hardwired to 0.
	0	Valid Project: All Format: Enable If this bit is set, this entry contains a valid TLB entry. If clear, this TLB entry is effectively "empty" and may be disregarded.



8.3 GFX_MODE – Graphics Mode Register

GFX_MODE – Graphics Mode Register				
Register Type: MMIO Address Offset: 2520h Project: All Default Value: 00000000h Access: R/W Size (in bits): 32				
This register contains a control bit for the new run list and 2-level PPGTT functions. This register is not saved/restored with context. This register is not reset with single-engine GFX reset; it is only reset by a global graphics reset (all engines including display).				
Bit	Description			
31:16	Mask Bits Format: Mask[15:0] Must be set to modify corresponding bit in Bits 15:0. (All implemented bits)			
15	Reserved	Project:	All	Format: MBZ
14	Reserved	Project:	All	Format: MBZ
13	Reserved	Project:	All	Format: MBZ
12:0	Reserved	Project:	All	Format: MBZ



8.4 EXCC—Execute Condition Code Register

EXCC—Execute Condition Code Register			
Register Type: MMIO Address Offset: 2028h Project: All Default Value: 00000000h Access: R/W,RO Size (in bits): 32			
This register contains user defined and hardware generated conditions that are used by MI_WAIT_FOR_EVENT commands. An MI_WAIT_FOR_EVENT instruction excludes the executing ring from arbitration if the selected event evaluates to a "1", while instruction is discarded if the condition evaluates to a "0". Once excluded a ring is enabled into arbitration when the selected condition evaluates to a "0".			
Bit	Description		
31:22	Reserved	Project: All	Format: MBZ
21	Mask Bits Format: Mask[5] This bit serves as a write enable for bit 5. If this register is written with this bit clear the corresponding bit in the field 5 will not be modified. Reading these bits always returns 0s.		
20:16	Mask Bits Format: Mask[4:0] These bits serves as a write enable for bits 4:0. If this register is written with any of these bits clear the corresponding bit in the field 4:0 will not be modified. Reading these bits always returns 0s.		
15:12	Reserved	Project: All	Format: MBZ
11	Pending Indirect State Dirty Bit Project: All Format: U32 This field keeps track of whether or not an indirect state pointer command has been parsed in the current context. Clears either on a context save or explicitly through a flush command		
10:8	Pending Indirect State Counter This field keeps track of the maximum number of indirect state pointers pending in the system. When the register is saved/restored, it saves either a value of 1 or 0. This field is Read-Only		
7:6	Reserved	Project: All	Format: MBZ



EXCC—Execute Condition Code Register				
5	Indirect State Pointer Force Restore			
	Determines whether to use pending indirect state counter to restore data to memory, or restore indirect data			
	Value	Name	Description	Project
	0h	Use Pending	Use the pending indirect state counter to restore data to memory	All
4:0	1h	Don't Use Pending	Don't use pending indirect state counter to restore data to memory. Always restore indirect data	All
	User Defined Condition Codes			
4:0	The software may signal a Stream Semaphore by setting the Mask bit and Signal Bit together to match the bit field specified in a WAIT_FOR_EVENT (Semaphore).			



8.5 RINGBUF—Ring Buffer Registers

RING_BUFFER_TAIL	
Register Type: MMIO Address Offset: 2030h Project: All Default Value: 00000000h Access: R/W Size (in bits): 32	
These registers are used to define and operate the “ring buffer” mechanism which can be used to pass instructions to the command interface. The buffer itself is located in a physical memory region. The ring buffer is defined by a 4 Dword register set that includes starting address, length, head offset, tail offset, and control information. Refer to the <i>Programming Interface</i> chapter for a detailed description of the parameters specified in this ring buffer register set, restrictions on the placement of ring buffer memory, arbitration rules, and in how the ring buffer can be used to pass instructions. <u>Ring Buffer Tail Offsets must be properly programmed before ring is enabled. A Ring Buffer can be enabled when empty.</u>	
Bit	Description
31:21	Reserved Project: All Format: MBZ
20:3	Tail Offset Project: All Format: U18 QWord Offset This field is written by software to specify where the valid instructions placed in the ring buffer end. The value written points to the QWord <i>past</i> the last valid QWord of instructions. In other words, it can be defined as the <i>next</i> QWord that software will write instructions into. Software must write subsequent instructions to QWords following the Tail Offset, possibly wrapping around to the top of the buffer (i.e., software can't skip around within the buffer). Note that all DWords prior to the location indicated by the Tail Offset must contain valid instruction data – which may require instruction padding by software. See Head Offset for more information.
2:0	Reserved Project: All Format: MBZ



RING_BUFFER_HEAD	
Register Type: MMIO Address Offset: 2034h Project: All Default Value: 00000000h Access: R/W Size (in bits): 32	
These registers are used to define and operate the “ring buffer” mechanism which can be used to pass instructions to the command interface. The buffer itself is located in a physical memory region. The ring buffer is defined by a 4 Dword register set that includes starting address, length, head offset, tail offset, and control information. Refer to the <i>Programming Interface</i> chapter for a detailed description of the parameters specified in this ring buffer register set, restrictions on the placement of ring buffer memory, arbitration rules, and in how the ring buffer can be used to pass instructions. <u>Ring Buffer Head Offsets must be properly programmed before ring is enabled. A Ring Buffer can be enabled when empty.</u>	
Bit	Description
31:21	Wrap Count Project: All Default Value: 0h Format: U11 count of ring buffer wraps This field is incremented by 1 whenever the Head Offset wraps from the end of the buffer back to the start (i.e., whenever it wraps back to 0). Appending this field to the Head Offset field effectively creates a virtual 4GB Head “Pointer” which can be used as a tag associated with instructions placed in a ring buffer. The Wrap Count itself will wrap to 0 upon overflow. The Wrap Count will get cleared as a result of writes of the Starting Address field.
20:2	Head Offset Project: All Format: U19 DWord Offset This field is written by software to specify where the valid instructions placed in the ring buffer end. The value written points to the QWord <i>past</i> the last valid QWord of instructions. In other words, it can be defined as the <i>next</i> QWord that software will write instructions into. Software must write subsequent instructions to QWords following the Tail Offset, possibly wrapping around to the top of the buffer (i.e., software can’t skip around within the buffer). Note that all DWords prior to the location indicated by the Tail Offset must contain valid instruction data – which may require instruction padding by software. See Head Offset for more information. Programming Notes Project A RB can be enabled empty or containing some number of valid instructions. All Head Offset is cleared as a result of writes of the Starting Address field. All
1	Reserved Project: All Format: MBZ
0	Wait for Condition Indicator Project: All Format: Enabled This is a read only value used to indicate whether or not the command streamer is currently waiting for a conditional code to be cleared from 0x2028



RING_BUFFER_START	
Register Type:	MMIO
Address Offset:	2038h
Project:	All
Default Value:	00000000h
Access:	R/W
Size (in bits):	32
These registers are used to define and operate the “ring buffer” mechanism which can be used to pass instructions to the command interface. The buffer itself is located in a physical memory region. The ring buffer is defined by a 4 Dword register set that includes starting address, length, head offset, tail offset, and control information. Refer to the <i>Programming Interface</i> chapter for a detailed description of the parameters specified in this ring buffer register set, restrictions on the placement of ring buffer memory, arbitration rules, and in how the ring buffer can be used to pass instructions.	
Bit	Description
31:12	Starting Address Project: All Address: GraphicsAddress[31:12] Surface Type: RingBuffer This field specifies Bits 31:12 of the 4KB-aligned starting Graphics Address of the ring buffer. Address bits 31 down to 29 must be zero. Writing this register also causes the Head Offset to be reset to zero, and the Wrap Count to be reset to zero. All ring buffer pages must map to Main Memory (uncached) pages. Ring Buffer addresses are always translated through the global GTT. Per-process address space can only be used via a batch buffer with the appropriate Memory Space Select.
11:0	Reserved Project: All Format: MBZ



RING_BUFFER_CONTROL																											
Register Type: MMIO																											
Address Offset: 203Ch																											
Project: All																											
Default Value: 00000000h																											
Access: R/W																											
Size (in bits): 32																											
These registers are used to define and operate the “ring buffer” mechanism which can be used to pass instructions to the command interface. The buffer itself is located in a physical memory region. The ring buffer is defined by a 4 Dword register set that includes starting address, length, head offset, tail offset, and control information. Refer to the <i>Programming Interface</i> chapter for a detailed description of the parameters specified in this ring buffer register set, restrictions on the placement of ring buffer memory, arbitration rules, and in how the ring buffer can be used to pass instructions.																											
Bit	Description																										
31:0	Buffer Length Project: All Format: U9-1 Count of 4 KB pages Range 0..1FFh This field is written by SW to specify the length of the ring buffer in 4 KB Pages. Range = [0 = 1 page = 4 KB, 1FFh = 512 pages = 2 MB]																										
11	RB Wait Project: All Format: Boolean Indicates that this ring has executed a WAIT_FOR_EVENT instruction and is currently waiting. Software can write a “1” to clear this bit, write of “0” has no effect. When the RB is waiting for an event and this bit is cleared, the wait will be terminated and the RB will be returned to arbitration.																										
10:3	Reserved Project: All Format: MBZ																										
2:1	Automatic Report Head Pointer Project: All This field is written by software to control the automatic “reporting” (write) of this ring buffer’s “Head Pointer” register (register DWord 1) to the corresponding location within the Hardware Status Page. Automatic reporting can either be disabled or enabled at 4KB, 64KB or 128KB boundaries within the ring buffer. <table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td>MI_AUTOREPORT_OFF</td><td>Automatic reporting disabled</td><td>All</td></tr><tr><td>1h</td><td>MI_AUTOREPORT_64KB</td><td>Report every 16 pages (64KB)</td><td>All</td></tr><tr><td></td><td>MI_AUTOREPORT_4KB</td><td>When the Per-Process Virtual Address Space and Run List Enable bit is set, the ring buffer reports every 4KB</td><td></td></tr><tr><td>2h</td><td>Reserved</td><td>Reserved</td><td>All</td></tr><tr><td>3h</td><td>MI_AUTOREPORT_128KB</td><td>Report every 32 pages (128KB)</td><td>All</td></tr></table>			Value	Name	Description	Project	0h	MI_AUTOREPORT_OFF	Automatic reporting disabled	All	1h	MI_AUTOREPORT_64KB	Report every 16 pages (64KB)	All		MI_AUTOREPORT_4KB	When the Per-Process Virtual Address Space and Run List Enable bit is set, the ring buffer reports every 4KB		2h	Reserved	Reserved	All	3h	MI_AUTOREPORT_128KB	Report every 32 pages (128KB)	All
Value	Name	Description	Project																								
0h	MI_AUTOREPORT_OFF	Automatic reporting disabled	All																								
1h	MI_AUTOREPORT_64KB	Report every 16 pages (64KB)	All																								
	MI_AUTOREPORT_4KB	When the Per-Process Virtual Address Space and Run List Enable bit is set, the ring buffer reports every 4KB																									
2h	Reserved	Reserved	All																								
3h	MI_AUTOREPORT_128KB	Report every 32 pages (128KB)	All																								



RING_BUFFER_CONTROL	
0	Ring Buffer Enable Project: All Format: Enable This field is used to enable or disable this ring buffer. It can be enabled or disabled regardless of whether there are valid instructions pending.

8.5.1 UHPTR — Pending Head Pointer Register

UHPTR — Pending Head Pointer Register				
Register Type: MMIO				
Address Offset: 2134h				
Project: All				
Default Value: 0000 0000h				
Access: R/W				
Size (in bits): 32				
Bit	Description			
31:3	Head Pointer Address Project: All Default Value: 0h Address: GraphicsAddress[31:3] This register represents the GFX address offset where execution should continue in the ring buffer following execution of an MI_ARB_CHECK command.			
2:1	Reserved	Project: All	Format: MBZ	
0	Head Pointer Valid Project: All Default Value: 0h Format: U1 This bit is set by the software to request a pre-emption. It is reset by hardware after the head pointer in this register is read. The hardware uses the head pointer programmed in this register at the time the reset is generated.			
	Value	Name	Description	Project
	0h		No valid updated head pointer register, resume execution at the current location in the ring buffer	All
	1h		Indicates that there is an updated head pointer programmed in this register	All



8.6 Debug Registers Control

8.6.1 HW_MEMRD—Memory Read Sync Register (Debug)

HW_MEMRD—Memory Read Sync Register (Debug)	
Register Type: MMIO Address Offset: 2060h Project: All Default Value: 0000 0000h Access: RO Size (in bits): 32	
This register is used to flush the data from the Graphics dedicated chipset buffers into memory. A read to the register is generated post-flush completion of the graphics pipeline by the software. Read to this register is expected to be used in debug mode. The hardware will always return 0 for this register.	
Bit	Description
31:0	Reserved Project: All Format: MBZ



8.6.2 IPEIR—Instruction Parser Error Identification Register (Debug)

IPEIR—Instruction Parser Error Identification Register (Debug)			
Register Type: MMIO Address Offset: 2064h Project: All Default Value: 0000 0000h Access: RO Size (in bits): 32			
The IPEIR register identifies the general location of instructions that generate a Invalid Instruction Errors for the Renderer IP. (Note: The header (DWord 0) of the offending instruction will be stored in the IPEHR register).			
Bit	Description		
31:4	Reserved	Project: All	Format: MBZ
3	Batch Buffer Error	Project: All	Format: Flag
	If this bit is set the faulting instruction was executed from a batch buffer. If this bit is clear the faulting instruction was executed directly from a ring buffer.		
2:0	Ring ID	Project: All Default Value: 0h Format: U3 This field indicates which ring buffer is associated with the faulting	
	Value	Name	Description
	0	Ring Buffer 0	Ring Buffer 0
	1-7	Reserved	Reserved
			Project
			All
			All



8.6.3 IPEHR—Instruction Parser Error Header Register (Debug)

IPEHR—Instruction Parser Error Header Register (Debug)	
Register Type: MMIO Address Offset: 2068h Project: All Default Value: 0000 0000h Access: RO Size (in bits): 32	
The IPEHR register is used to identify the instructions that generate Invalid Instruction Errors. This register is loaded with the header (DWORD 0) of each instruction that is executed. It will therefore hold the header of an instruction that generates an Invalid Instruction Error.	
Bit	Description
31:0	Header Project: All Format: U32 This field will contain the header (DWORD 0) of a Renderer IP instruction that generates an Invalid Instruction Error.

8.6.4 INSTDONE—Instruction Stream Interface Done Register (Debug)

INSTDONE—Instruction Stream Interface Done Register (Debug)	
Register Type: MMIO Address Offset: 206Ch Project: All Default Value: FFE7 FFEh Access: RO Size (in bits): 32	
This read-only register reports “Done” signals associated with the various internal engines and instruction transport mechanisms. In general, when the rendering engines of the device are idle, all bits will be set. If, for some reason, the device hangs, this register can be used to determine which functions are stalled with pending operations.	
Bit	Description
31	Row 0, EU 0 Done
30	Row 0, EU 1 Done
29	Row 0, EU 2 Done
28	Row 0, EU 3 Done
27	Row 1, EU 0 Done
26	Row 1, EU 1 Done



INSTDONE—Instruction Stream Interface Done Register (Debug)	
25	Row 1, EU 2 Done
24	Row 1, EU 3 Done
23	Strips and Fans (SF) Done
22	Setup (SE) Done
21	Windower (WM) Done
20	Reserved. Read as “0”
19	Reserved. Read as “0”
18	Dispatcher (DIP) Done
17	Projection and LOD (PL) Done
16	Dependent Address Generator (DG) Done
15	Quad Cache Controller (QC) Done
14	Texture Fetch (FT) Done
13	Texture Decompressor (DM) Done
12	Sampler Cache (SC) Done
11	Filter (FL) Done
10	Bypass FIFO (BY) Done
9	Pixel Shader (PS) Done
8	Color Calculator (CC) Done
7	Map Filter Done: FL_done
6	Map L2 Cache Idle.
5	Message Arbiter Row 0 (EU output and EU input for Row 0) Done
4	Message Arbiter Row 1 (EU output and EU input for Row 1) Done
3	Instruction Cache Row 0 Done
2	Instruction Cache Row 1 Done
1	Command Parser (CP) Done
0	Ring 0 Enable



8.6.5 INSTPS—Instruction Parser State Register (Debug)

INSTPS—Instruction Parser State Register (Debug)	
Register Type:	MMIO
Address Offset:	2070h
Project:	All
Default Value:	UUUU UUUUh
Access:	RO
Size (in bits):	32
This register contains the state code of the Instruction Parser in the CSI. Decoding the contents of this register will indicate what the Instruction Parser is currently doing.	
Bit	Description
31:0	Instruction Parser State Project: All Format: <i>Implementation Specific</i> Fields in this register identify the active Ring Buffer or Batch Buffer, and Batch buffer type.

8.6.6 ACTHD — Active Head Pointer Register (Debug)

ACTHD — Active Head Pointer Register (Debug)	
Register Type:	MMIO
Address Offset:	2074h
Project:	All
Default Value:	0000 0000h
Access:	RO
Size (in bits):	32
This register contains the Head "Pointer" (DWORD Graphics Memory Address) of the currently-active ring buffer.	
Bit	Description
31:2	Head Pointer Project: All Default Value: 0h Address: GraphicsAddress[31:2] DWord Graphics Address corresponding to the Head Pointer of the currently-active ring or batch buffer.
1:0	Reserved Project: All Format: MBZ



8.6.7 DMA_FADD_P — Primary DMA Engine Fetch Address (Debug)

DMA_FADD_P — Primary DMA Engine Fetch Address (Debug)	
Register Type: MMIO Address Offset: 2078h Project: All Default Value: 0000 0000h Access: RO Size (in bits): 32	
This register contains the QWord offset from the start address of the instruction being fetched by the Primary DMA engine.	
Bit	Description
31:3	Current DMA QWord Offset Project: All Format: U30 This field contains the offset of the QWord (from the start of the ring buffer or batch buffer) that the “Primary” instruction parser DMA engine is currently accessing (fetching). Note that this offset will typically lead the Head offset (as instructions must be fetched before execution).
2:0	Reserved Project: All Format: MBZ

8.6.8 INSTDONE_1 — Additional Instruction Stream Interface Done (Debug)

INSTDONE_1 — Additional Instruction Stream Interface Done (Debug)	
Register Type: MMIO Address Offset: 207Ch Project: All Default Value: 0000 0000h Access: RO Size (in bits): 32	
Bit	Description
31:20	Reserved
19	gw_cs_done_cr
18	svsm_cs_done_cr
17	svdw_cs_done_cr
16	svdr_cs_done_cr



INSTDONE_1 — Additional Instruction Stream Interface Done (Debug)	
15	svrw_cs_done_cr
14	svrr_cs_done_cr
13	svts_cs_done_cr
12	masm_cs_done_cr
11	masf_cs_done_cr
10	mawb_cs_done_cr
9	em1_cs_done_cr
8	em0_cs_done_cr
7	uc1_cs_done
6	uc0_cs_done
5	urb_cs_done
4	isc_cs_done
3	cl_cs_done
2	gs_cs_done
1	vs0_cs_done
0	vf_cs_done



8.6.9 GFX_FLSH_CNTL — Graphics Flush Control

GFX_FLSH_CNTL — Graphics Flush Control	
Register Type: MMIO Address Offset: 2170h Project: All Default Value: 0000 0000 0000 0000h Access: Write Only Size (in bits): 64	
The flush initiated by this register is required whenever the GTT base address is changed or GTT entries are updated directly in memory by the host. See the description of the PGTBL_CTL_0 register for the sequence of operations required to update the GTT base or directly update GTT entries without using GTTADR.	
Bit	Description
63:0	Project: All Format: U64 A CPU Dword/Qword write to this space flushes the GWB of all writes. The data associated with the write to this register is discarded. A command stream write to this space has no effect and the write data is discarded; the cycle is completed. It is UNDEFINED to read from this register.



8.7 NOPID — NOP Identification Register

NOPID — NOP Identification Register

Register Type:

MMIO

Address Offset:

2094h

Project:

All

Default Value:

00000000h

Access:

RO

Size (in bits):

32

The NOPID register contains the Noop Identification value specified by the last MI_NOOP instruction that enabled this register to be updated.

Bit	Description			
31:22	Reserved	Project:	All	Format: MBZ
21:0	Identification Number Project: All Security: None Default Value: 0h DefaultVaueDesc This field contains the 22-bit Noop Identification value specified by the last MI_NOOP instruction that enabled this field to be updated Programming Notes This register is expected to be used for debug purposes to keep track of the execution of the command buffer			
				Project: All



8.8 Interrupt Control Registers

The Interrupt Control Registers described below all share the same bit definition. The bit definition is as follows:

Table 8-1. Bit Definition for Interrupt Control Registers

Bit	Description
31:26	Reserved. These bits may be assigned to interrupts on future products/steppings.
25	Reserved.
24	Reserved.
23	Reserved.
22	Reserved.
21	Reserved.
20	Reserved.
19	Reserved.
18	PIPE_CONTROL Notify Interrupt: The Pipe Control packet (Fences) specified in <i>3D pipeline</i> document may optionally generate an Interrupt. The Store QW associated with a fence is completed ahead of the MSI. This ordering is not guaranteed if PCI Line Intr# mechanism is used.
17	Display Port Interrupt: This status bit is set when a port hotplug/unplug event has been detected. The specific trigger of this interrupt can be read in the port hotplug status register.
16	Reserved. MBZ
15	Render Command Parser Master Error: When this status bit is set, it indicates that the hardware has detected an error. It is set by the device upon an error condition and cleared by a CPU write of a one to the appropriate bit contained in the Error ID register followed by a write of a one to this bit in the IIR. Further information on the source of the error comes from the “Error Status Register” which along with the “Error Mask Register” determine which error conditions will cause the error status bit to be set and the interrupt to occur. Page Table Error: Indicates a page table error. Instruction Parser Error: The Renderer Instruction Parser encounters an error while parsing an instruction.
14	GMCH Thermal Sensor Event: This bit is set on “thermal events” detected by the Thermal Sensor logic.
13	Reserved. MBZ
12	Sync Status: This bit is toggled when the Instruction Parser completes a flush with the sync enable bit active in the INSTPM register. The toggle event will happen after all the graphics engines are flushed. The HW Status DWord write resulting from this toggle will cause the CPU's view of graphics memory to be coherent as well (flush and invalidate the render cache).
11	Display Plane A Flip Pending: This status bit is set on a Display Plane A pending flip (i.e., resulting from the execution of an MI_DISPLAY_BUFFER_INFO instruction). This is only used when the MI_DISPLAY_BUFFER_INFO instruction is being used. See that instruction for additional information.
10	Display Plane B Flip Pending: Flip Pending status for Display B. See Display Plane A Flip Pending



Bit	Description
9	Overlay Plane Flip Pending: This status bit is set to reflect a pending overlay plane flip (i.e., resulting from the execution of an MI_OVERLAY_FLIP instruction). This is only affected by the use of MI_OVERLAY_FLIP instructions and not through the manual method.
8	Display Plane C Flip Pending: Flip Pending status for Display Plane C. See Display Plane A Flip Pending
7	Display Pipe A VBLANK: This status bit is set at leading edge of Display Pipe A VBLANK, though delayed to allow all internal hardware VBLANK events to occur before the interrupt is generated (to avoid race conditions). These events include the update of the display and overlay status bits and loading of the overlay registers. [DevCL] If trunk clock gating is enabled, this interrupt should never be used.
6	Display Pipe A Event: This status bit is set by the device on the active-going edge of the OR of unmasked Display Pipe A event bits. The specific cause of the event can be determined by reading the display status register. Note that the display line compare status can also be observed through the instruction interface.
5	Display Pipe B VBLANK: This status bit is set at leading edge of Display B VBLANK. This is actually delayed to allow all VBLANK events to occur before the interrupt is generated. These events include the update of the overlay registers. [DevCL] If trunk clock gating is enabled, this interrupt should never be used.
4	Display Pipe B Event: This status bit is set by the device on the active-going edge of the OR of unmasked Display Pipe A event bits. The specific cause of the event can be determined by reading the display status register. Note that the display line compare status can also be observed through the instruction interface.
3	Reserved. MBZ
2	Debug Interrupt: When this bit is set, the EU is indicating that it has encountered an interrupt in the kernel program. Refer to the Gen4 Debug PRM for more details
1	Render Command Parser User Interrupt: This status bit is set when an MI_USER_INTERRUPT instruction is executed on the Render Command Parser. Note that instruction execution is not halted and proceeds normally. A mechanism such as an MI_STORE_DATA instruction is required to associate a particular meaning to a user interrupt.
0	ASLE Interrupt: This status bit is set when ASLE (PCI Configuration register Device 2, Function 0, E4) is written by the System BIOS (any byte or all). The meaning of the interrupt is determined by the contents written.



The following table specifies the settings of interrupt bits stored upon a “Hardware Status Write” due to ISR changes:

Bit	Interrupt Bit	ISR bit Reporting via Hardware Status Write (when unmasked via HWSTAM)
25	Reserved. MBZ	
24	Reserved. MBZ	
23	Reserved. MBZ	
22	Reserved. MBZ	
21	Reserved. MBZ	
20	Reserved. MBZ	
19	Reserved. MBZ	
18	PIPE_CONTROL packet - Notify Enable	0
17	Display Port Interrupt	Set when event occurs, cleared when event cleared
16	Reserved. MBZ	0
15	Master Error	Set when error occurs, cleared when error cleared
14	GMCH Thermal Sensor Event	Should always be disabled for Hardware Status Write reporting.
13	Reserved. MBZ	0
12	Sync Status	Toggled every SyncFlush Event
11	Display Plane A Flip Pending	Set when flip is pending
10	Display Plane B Flip Pending	Set when flip is pending
9	Overlay Flip Pending	Set when flip is pending
8	Display Plane C Flip Pending	Set when Flip requested, cleared when flip occurs.
7	Display Pipe A VBlank	0
6	Display Pipe A Event	Set when event occurs, cleared when event cleared
5	Display Pipe B VBlank	0
4	Display Pipe B Event	Set when event occurs, cleared when event cleared
3	Reserved. MBZ	0
2	Debug Interrupt	Set when debug interrupt occurs.
1	User Interrupt	0
0	ASLE Interrupt	0



8.8.1 HWS_PGA — Hardware Status Page Address Register

HWS_PGA — Hardware Status Page Address Register	
Register Type: MMIO Address Offset: 2080h Project: All Default Value: 1FFFF000h Access: R/W Size (in bits): 32	
This register is used to program the 4 KB-aligned System Memory address of the Hardware Status Page used to report hardware status into (typically cacheable) System Memory.	
Bit	Description
31:12	Address Project: All Security: None Address: PhysicalAddress[31:12] Surface Type: U32 Range: $0..2^{32}-1$ This field is used by SW to specify Bits 31:12 of the 4 KB-aligned System Memory address of the 4 KB page known as the “Hardware Status Page”. The system address space is expected to be cacheable in memory.
12:8	Reserved Project: All Format: MBZ
7:4	Physical Start Address Extension Project: All Security: None Address: PhysicalAddress[35:32] This field specifies Bits 35:32 of the starting physical address.
3:0	Reserved Project: All Format: MBZ



The following table defines the layout of the Hardware Status Page:

DWord Offset	Description
0	Interrupt Status Register Storage: The content of the ISR register is written to this location whenever an “unmasked” bit of the ISR (as determined by the HWSTAM register) changes state.
3:1	Reserved. Must not be used.
4	Ring Head Pointer Storage: The contents of the Ring Buffer Head Pointer register (register DWord 1) are written to this location either as result of an MI_REPORT_HEAD instruction or as the result of an “automatic report” (see RINGBUF registers).
Fh:5h	Reserved. Must not be used.
10h-1Bh	Reserved. Must not be used.
1Ch-1Fh	Reserved. Must not be used.
1Fh	Reserved. Must not be used.
20h-3FFh	These locations can be used for general purpose via the MI_STORE_DATA_INDEX or MI_STORE_DATA_IMM instructions.

8.8.2 PWRCTXA — Power Context Register Address ([DevCL] Only)

PWRCTXA — Power Context Register Address	
Register Type:	MMIO
Address Offset:	2088h
Project:	DevCL
Default Value:	0000 0000h
Access:	R/W
Size (in bits):	32
The PWRCTXA register has the address of the Global GTT translated memory location which stores the hardware context if the voltage is removed from the render clock well. The format of the hardware “power” context is specified in the Memory Data Formats.	
Bit	Description
31:12	Power Context Address Project: DevCL Default Value: 0h Address: GraphicsAddress[31:12] This field is used by SW to specify Bits 31:12 of the 4 KB-aligned Graphics Memory address. The graphics memory address is translated using the Global GTT.
11:5	Reserved Project: DevCL Format: MBZ



PWRCTXA — Power Context Register Address																
4:1	Power Context Size Project: DevCL Default Value: 0h Format: U4 Field specifies the size of the power context allocated by the software. The size is in terms of 4K pages This field is ReadOnly <table><thead><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr></thead><tbody><tr><td>001-111</td><td>Reserved</td><td>Reserved</td><td>All</td></tr><tr><td>000</td><td>4KB</td><td></td><td>DevCL</td></tr></tbody></table>				Value	Name	Description	Project	001-111	Reserved	Reserved	All	000	4KB		DevCL
Value	Name	Description	Project													
001-111	Reserved	Reserved	All													
000	4KB		DevCL													
0	Power Context Enable Project: DevCL Default Value: 0h Format: Enable This field determines whether the power context is enabled. If enabled, the Power Context Address specifies the starting address of the hardware context in memory. If the power context is not enabled, the hardware will disable reducing the render voltage. <table><thead><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr></thead><tbody><tr><td>0h</td><td>Disable</td><td>DISABLED</td><td>DevCL</td></tr><tr><td>1h</td><td>Enable</td><td>ENABLED</td><td>DevCL</td></tr></tbody></table>				Value	Name	Description	Project	0h	Disable	DISABLED	DevCL	1h	Enable	ENABLED	DevCL
Value	Name	Description	Project													
0h	Disable	DISABLED	DevCL													
1h	Enable	ENABLED	DevCL													

8.8.3 HWSTAM — Hardware Status Mask Register

Hardware Status Mask Register				
Register Type: MMIO Address Offset: 2098h Project: All Default Value: FFFE DFFFh Access: R/W Size (in bits): 32				
The HWSTAM register has the same format as the Interrupt Control Registers. The bits in this register are “mask” bits that prevent the corresponding bits in the Interrupt Status Register from generating a “Hardware Status Write” (PCI write cycle). Any unmasked interrupt bit (HWSTAM bit set to 0) will allow the Interrupt Status Register to be written to the ISR location (within the memory page specified by the Hardware Status Page Address Register) when that Interrupt Status Register bit changes state.				
Bit	Description			
31:13	Reserved	Project: All	Format: MB1	



Hardware Status Mask Register	
12	Sync Status Project: All Security: None Default Value: 1h Masked by default When this mask bit is clear, a change in the corresponding ISR bit will trigger a DWord write of the ISR contents to the “ISR location” in the Hardware Status Page. When a bit in this mask is set, a write will not be triggered by a change in the corresponding ISR bit. Programming Notes Project This bit is toggled when the Instruction Parser completes a flush with the sync enable bit active in the INSTPM register. The toggle event will happen after all the graphics engines are flushed. The HW Status DWord write resulting from this toggle will cause the CPU's view of graphics memory to be coherent as well (flush and invalidate the render cache). All
11	Display Plane A Flip Pending Project: All Security: None Default Value: 1h Masked by default When this mask bit is clear, a change in the corresponding ISR bit will trigger a DWord write of the ISR contents to the “ISR location” in the Hardware Status Page. When a bit in this mask is set, a write will not be triggered by a change in the corresponding ISR bit. Programming Notes Project This status bit is set on a Display Plane A pending flip (i.e., resulting from the execution of an MI_DISPLAY_BUFFER_INFO instruction). This is only used when the MI_DISPLAY_BUFFER_INFO instruction is being used. See that instruction for additional information. All
10	Display Plane B Flip Pending Project: All Security: None Default Value: 1h Masked by default When this mask bit is clear, a change in the corresponding ISR bit will trigger a DWord write of the ISR contents to the “ISR location” in the Hardware Status Page. When a bit in this mask is set, a write will not be triggered by a change in the corresponding ISR bit. Programming Notes Project This status bit is set on a Display Plane B pending flip (i.e., resulting from the execution of an MI_DISPLAY_BUFFER_INFO instruction). This is only used when the MI_DISPLAY_BUFFER_INFO instruction is being used. See that instruction for additional information. All



Hardware Status Mask Register				
9	Overlay Plane Flip Pending Project: All Security: None Default Value: 1h Masked by default When this mask bit is clear, a change in the corresponding ISR bit will trigger a DWord write of the ISR contents to the "ISR location" in the Hardware Status Page. When a bit in this mask is set, a write will not be triggered by a change in the corresponding ISR bit. Programming Notes This status bit is set to reflect a pending overlay plane flip (i.e., resulting from the execution of an MI_OVERLAY_FLIP instruction). This is only affected by the use of MI_OVERLAY_FLIP instructions and not through the manual method. Project All			
8	Display Plane C Flip Pending Project: All Security: None Default Value: 1h Masked by default When this mask bit is clear, a change in the corresponding ISR bit will trigger a DWord write of the ISR contents to the "ISR location" in the Hardware Status Page. When a bit in this mask is set, a write will not be triggered by a change in the corresponding ISR bit. Programming Notes Flip Pending status for Display Plane C. See Display Plane A Flip Pending Project All			
7:0	Reserved	Project: All	Format: MB1	



8.8.4 IER — Interrupt Enable Register

IER — Interrupt Enable Register			
Register Type: MMIO Address Offset: 20A0h Project: All Default Value: 0000 0000h Access: R/W Size (in bits): 32			
The IER register contains an interrupt enable bit for each interrupt bit in the IIR register. A disabled interrupt will still appear in the Interrupt Identity Register to allow polling of interrupt sources.			
Bit	Description		
31:0	Interrupt Enable Bits Project: All Default Value: 0h Format: Array of Enables refer to Table 8-1 in Interrupt Control Register section for bit definitions The bits in this register enable a CPU interrupt to be generated whenever the corresponding bit in the Interrupt Identity Register becomes set.		
	Value	Name	Description
	0h	Disable	DISABLED
	1h	Enable	ENABLED
			Project
			All
			All



8.8.5 IIR — Interrupt Identity Register

IIR — Interrupt Identity Register											
Register Type:	MMIO										
Address Offset:	20A4h										
Project:	All										
Default Value:	0000 0000h										
Access:	R/WC										
Size (in bits):	32										
The IIR register contains the interrupt bits that are “unmasked” by the IMR and thus can generate CPU bits (if enabled via the IER). When a CPU interrupt is generated, this should be the first register to be interrogated to determine the source of the interrupt. Writing a ‘1’ into the appropriate bit position within this register clears interrupts. Programming Note: Prior to clearing a Display Pipe-sourced interrupt (e.g., Display Pipe A VBLANK) in the IIR the corresponding interrupt (source) status in the PIPEASTAT register (e.g., Pipe A VBLANK Interrupt Status bit of PIPEASTAT) must first be cleared. Note that clearing these status bits requires writing a ‘1’ to the appropriate bit position.											
Bit	Description										
31:0	Interrupt Identity Bits Project: All Default Value: 0h Format: Array of unmasked Persistent interrupt bits (refer to Table 9-1 in Interrupt Control Register section for bit definitions) This field holds the persistent values of the interrupt bits from the ISR which are “unmasked” by the IMR. If enabled by the IER, bits set in this register will generate a CPU interrupt. Bits set in this register will remain set (persist) until the interrupt condition is “cleared” via software by writing a ‘1’ to the appropriate bit(s). <table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>1h</td><td>Interrupt Condition Detected</td><td>Interrupt Condition Detected (may or may not have actually generated a CPU interrupt)</td><td>All</td></tr></table> Programming Notes Bit 12 of the Interrupt Identity register is used for the sync status flush. The hardware toggles the bit at the completion of the flush. It is not expected that this bit will be used to generate interrupt. In case an interrupt is desired, software needs to toggle the bit back to 0 (by programming another sync flush) before clearing the IIR.			Value	Name	Description	Project	1h	Interrupt Condition Detected	Interrupt Condition Detected (may or may not have actually generated a CPU interrupt)	All
Value	Name	Description	Project								
1h	Interrupt Condition Detected	Interrupt Condition Detected (may or may not have actually generated a CPU interrupt)	All								



8.8.6 IMR—Interrupt Mask Register

IMR—Interrupt Mask Register			
Register Type: MMIO Address Offset: 20A8h Project: All Default Value: FFFE DFFFh Access: R/W Size (in bits): 32			
The IMR register is used by software to control which Interrupt Status Register bits are “masked” or “unmasked”. “Unmasked” bits will be reported in the IIR, possibly triggering a CPU interrupt, and will persist in the IIR until cleared by software. “Masked” bits will not be reported in the IIR and therefore cannot generate CPU interrupts.			
Bit	Description		
31:0	Interrupt Mask Bits Project: All Default Value: FFFE DFFFh Format: Array of interrupt mask bits Refer to Table 9-1 in Interrupt Control Register section for bit definitions This field contains a bit mask which selects which interrupt bits (from the ISR) are reported in the IIR.		
	Value	Name	Description
	0h	Not Masked	Will be reported in the IIR
	1h	Masked	Will not be reported in the IIR
			Project
			All
			All



8.8.7 ISR — Interrupt Status Register

ISR — Interrupt Status Register			
Register Type: MMIO Address Offset: 20ACh Project: All Default Value: 0000 0000h Access: RO Size (in bits): 32			
The ISR register contains the non-persistent value of all interrupt status bits. The IMR register selects which of these interrupt conditions are reported in the persistent IIR (i.e., set bits must be cleared by software). Bits in the IER are used to selectively enable IIR bits to cause CPU interrupts. Programming Note: The User Interrupt bit in this register is a short pulse therefore software should not expect to use this register to sample these conditions.			
Bit	Description		
31:0	Interrupt Status Bits		
	Project:	All	
	Default Value:	0h	
	Format:	Array of interrupt status bits	Refer to Table 9-1 in Interrupt Control Register section for bit definitions
	This field contains the non-persistent values of all interrupt status bits.		
	Value	Name	Description
			Project
	1h	Interrupt Condition Exists	Interrupt Condition currently exists
			All



8.9 Hardware-Detected Error Bit Definitions (for EIR, EMR, ESR)

This section defines the Hardware-Detected Error bit definitions and ordering that is common to the EIR, EMR and ESR registers. The EMR selects which error conditions (bits) in the ESR are reported in the EIR. Any bit set in the EIR will cause the Master Error bit in the ISR to be set. EIR bits will remain set until the appropriate bit(s) in the EIR is cleared by writing the appropriate EIR bits with '1'.

The following table describes the Hardware-Detected Error bits:

Table 8-2. Hardware-Detected Error Bits

Bit	Description
15:10	Reserved: MBZ
9	Reserved.
8	Reserved.
7:6	Reserved: MBZ
5	Reserved.
4	Page Table Error: This bit is set when a Graphics Memory Mapping Error is detected. The cause of the error is indicated (to some extent) in the PGTBL_ER register. Note: This error indications cannot be cleared except by reset (i.e., it is a fatal error). 1 = Page table error
3	Reserved.
2	Reserved.
1	Main Memory Refresh Timer Error: This bit is set when the device detects a timeout related to refreshing Main Memory. [DevBW]: Reserved.
0	Instruction Error: This bit is set when the Renderer Instruction Parser detects an error while parsing an instruction. Instruction errors include: 1) Client ID value (Bits 31:29 of the Header) is not supported (only MI, 2D and 3D are supported). 2) Defeatured MI Instruction Opcodes: The (<i>debug</i>) INSTPS register may provide more information as to the cause of the error. The (<i>debug</i>) IPEHR register contains the header (DWord 0) of the faulting instruction. The (<i>debug</i>) IPEIR, BBP_PTR, ABB_PTR, ABB_END and DMA_FADD registers provide an indication of where the faulting instruction is located and which instruction stream mechanism caused the instruction to be executed. 1: Instruction Error detected Programming Note: The bit for the error mask of this register is reserved. The mask should be set to a value of 1.



8.9.1 EIR — Error Identity Register

EIR — Error Identity Register													
Register Type: MMIO Address Offset: 20B0h Project: All Default Value: 0000 0000h Access: R/WC Size (in bits): 32													
The EIR register contains the persistent values of Hardware-Detected Error Condition bits. Any bit set in this register will cause the Master Error bit in the ISR to be set. The EIR register is also used by software to clear detected errors (by writing a '1' to the appropriate bit(s)).													
Bit	Description												
31:16	Reserved	Project: All	Format: MBZ										
15:0	Error Identity Bits Project: All Default Value: 0h Format: Array of Error condition bits See Table 9-5. Hardware-Detected Error Bits This register contains the persistent values of ESR error status bits that are unmasked via the EMR register. (See Table 8-2. Hardware-Detected Error Bits). The logical OR of all (defined) bits in this register is reported in the Master Error bit of the Interrupt Status Register. In order to clear an error condition, software must first clear the error by writing a '1' to the appropriate bit(s) in this field. If required, software should then proceed to clear the Master Error bit of the IIR. <table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>1h</td><td>Error occurred</td><td>Error occurred</td><td>All</td></tr></table> Programming Notes Writing a '1' to a set bit will cause that error condition to be cleared. However, the Page Table Error bit (Bit 4) cannot be cleared except by reset (i.e., it is a fatal error). <table><tr><th>Project</th></tr><tr><td>All</td></tr></table>			Value	Name	Description	Project	1h	Error occurred	Error occurred	All	Project	All
Value	Name	Description	Project										
1h	Error occurred	Error occurred	All										
Project													
All													



8.9.2 EMR—Error Mask Register

EMR—Error Mask Register			
Register Type: MMIO			
Address Offset: 20B4h			
Project: All			
Default Value: FFFF FFDFh			
Access: R/W			
Size (in bits): 32			
The EMR register is used by software to control which Error Status Register bits are “masked” or “unmasked”. “Unmasked” bits will be reported in the EIR, thus setting the Master Error ISR bit and possibly triggering a CPU interrupt, and will persist in the EIR until cleared by software. “Masked” bits will not be reported in the EIR and therefore cannot generate Master Error conditions or CPU interrupts.			
Bit	Description		
31:16	Reserved	Project: All	Format: MBZ
15:0	Error Mask Bits		
	Project: All		
	Default Value: FFFF FFDFh		
	Format: Array of error condition mask bits		
	See Table 9-5. Hardware-Detected Error Bits		
	This register contains a bit mask that selects which error condition bits (from the ESR) are reported in the EIR.		
	Value	Name	Description
0h	Not Masked	Will be reported in the EIR	All
1h	Masked	Will not be reported in the EIR	All



8.9.3 ESR—Error Status Register

ESR—Error Status Register											
Register Type: MMIO Address Offset: 20B8h Project: All Default Value: 0000 0000h Access: RO Size (in bits): 32											
The ESR register contains the current values of all Hardware-Detected Error condition bits (these are all by definition “persistent”). The EMR register selects which of these error conditions are reported in the persistent EIR (i.e., set bits must be cleared by software) and thereby causing a Master Error interrupt condition to be reported in the ISR.											
Bit	Description										
31:16	Reserved Project: All Format: MBZ										
15:0	Error Status Bits Project: All Default Value: 0h Format: Array of error condition bits See Table 9-5. Hardware-Detected Error Bits This register contains the non-persistent values of all hardware-detected error condition bits. <table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>1h</td><td>Error Condition Detected</td><td>Error Condition detected</td><td>All</td></tr></table>			Value	Name	Description	Project	1h	Error Condition Detected	Error Condition detected	All
Value	Name	Description	Project								
1h	Error Condition Detected	Error Condition detected	All								



8.10 Register Definitions for Context Save

8.10.1 INSTPM—Instruction Parser Mode Register

INSTPM—Instruction Parser Mode Register	
Register Type: MMIO Address Offset: 20C0h Project: All Default Value: 00000000h Access: R/W Size (in bits): 32	
The INSTPM register is used to control the operation of the Instruction Parser. Certain classes of instructions can be disabled (ignored) – often useful for detecting performance bottlenecks. Also, “Synchronizing Flush” operations can be initiated – useful for ensuring the completion (vs. only parsing) of rendering instructions. Programming Notes: • If an instruction type is disabled, the parser will read those instructions but not process them. • Error checking will be performed even if the instruction is ignored. • All Reserved bits are implemented. • This Register is saved and restored as part of Context.	
Bit	Description
31:16	Mask Bits Format: Mask[15:0] Masks: These bits serve as write enables for bits 15:0. If this register is written with any of these bits clear the corresponding bit in the field 15:0 will not be modified. Reading these bits always returns 0s.
15:11	Reserved Project: All Format: MBZ
10	Reserved. MBZ
9:8	Reserved Project: All Format: MBZ
7	CONSTANT_BUFFER Surface Address Offset Enable Project: All Format: U1 When this bit is set, the CONSTANT_BUFFER Buffer Starting Address is used as a SurfaceStateOffset. I.e., it serves as an offset from the Surface State Base Address. Accesses will be subject to Surface State bounds checking. When this bit is not set, the CONSTANT_BUFFER Buffer Starting Address is based on bit 6 of the address. No bounds checking will be performed during access. Format = Enable



INSTPM—Instruction Parser Mode Register			
6	CONSTANT_BUFFER Address Offset Disable Project: All Format: U1 When this bit is clear, the CONSTANT_BUFFER Buffer Starting Address is used as a GeneralStateOffset. I.e., it serves as an offset from the General State Base Address. Accesses will be subject to General State bounds checking. When this bit is set, the CONSTANT_BUFFER Buffer Starting Address is used as a true GraphicsAddress (not an offset). No bounds checking will be performed during access. Format = Disable		
5	Sync Flush Enable Project: All Format: U1 This field is used to request a Sync Flush operation. The device will automatically clear this bit before completing the operation. See Sync Flush (<i>Programming Environment</i>). Programming Note: - The command parser must be stopped prior to issuing this command by setting the Stop Rings bit in register MI_MODE. Only after observing Rings Idle set in MI_MODE can a Sync Flush be issued by setting this bit. Once this bit becomes clear again, indicating flush complete, the command parser is re-enabled by clearing Stop Rings. Errata: Sync Flush cannot be used while a media scoreboard kernel is running. Format = Enable (cleared by HW)		
4	Global Debug Enable Project: All Format: U1 This field is used to enable the debug capability. Setting this bit allows the hardware to start incrementing the registers corresponding to the debug feature. Format = Enable		
3	BLT Instruction Disable Project: All Format: U1 This bit instructs the Renderer instruction parser to parse and error-check BLT instructions, but not execute them. Format = Disable		
2	3D Rendering Instruction Disable Project: All Format: U1 This bit instructs the Renderer instruction parser to parse and error-check 3D Rendering instructions, but not execute them. This bit must always be set by software if 3D State Instruction Disable is set. Setting this bit <i>without</i> setting 3D State Instruction Disable is allowed. Format = Disable		
1	3D State Instruction Disable Project: All Format: U1 This bit instructs the Renderer instruction parser to parse and error-check 3D State instructions, but not execute them. This bit should <i>not</i> be set unless 3D Rendering Instruction Disable (bit 2) is also set. Format = Disable		
0	Texture Palette Load Instruction Disable Project: All Format: U1 This bit instructs the Renderer instruction parser to parse and error-check Texture Palette Load instructions, but not execute them. Format = Disable		



8.10.2 Cache_Mode_0— Cache Mode Register 0

Cache_Mode_0— Cache Mode Register 0																						
Register Type: MMIO																						
Address Offset: 2120h																						
Project: All																						
Default Value: 0000 6820h																						
Access: R/W																						
Size (in bits): 32																						
This register is used to control the operation of the Render and Sampler L2 Caches. All reserved bits are implemented as read/write.																						
This Register is saved and restored as part of Context.																						
Bit	Description																					
31:16	Masks Format: Mask[15:0] A “1” in a bit in this field allows the modification of the corresponding bit in Bits 15:0.																					
15	Sampler L2 Disable Project: All Default Value: 0h Format: Disable <table><thead><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr></thead><tbody><tr><td>0h</td><td></td><td>Sampler L2 Cache Enabled.</td><td>All</td></tr><tr><td>1h</td><td></td><td>Sampler L2 Cache Disabled all accesses are treated as misses.</td><td>All</td></tr></tbody></table> <table><thead><tr><th>Errata</th><th>Description</th><th>Project</th></tr></thead><tbody><tr><td>BWT012</td><td>Setting this bit is UNDEFINED.</td><td>DevBW-A,B</td></tr></tbody></table>				Value	Name	Description	Project	0h		Sampler L2 Cache Enabled.	All	1h		Sampler L2 Cache Disabled all accesses are treated as misses.	All	Errata	Description	Project	BWT012	Setting this bit is UNDEFINED.	DevBW-A,B
Value	Name	Description	Project																			
0h		Sampler L2 Cache Enabled.	All																			
1h		Sampler L2 Cache Disabled all accesses are treated as misses.	All																			
Errata	Description	Project																				
BWT012	Setting this bit is UNDEFINED.	DevBW-A,B																				



Cache_Mode_0— Cache Mode Register 0				
14:13	Sampler L2 Page Gathering Fifo Modes			
	Project:		All	
	Default Value:		3h	
	Format:		U3	
	Value	Name	Description	Project
	00		No Page Gathering, No Interleaving.	All
	01		On Page Gathering based on Page Size described in Low Priority Grace Period Page Size. No Interleaving.	All
	10		Interleaved based on Tile Type and address bits A6, A9 and A10.	All
	11		Interleaved on page gathering as combination of modes 1 and 2.	All
	12:10	Page Gather Limit		
Project:		All		
Default Value:		2h		
Format:		U3		
Used when bits 14:13 are set to 1 or 3. Determines the maximum number of on page requests gathered.				
Value		Name	Description	Project
000			4 Requests.	All
001			6 Requests.	All
010			8 Requests.	All
011			10 Requests.	All
100			12 Requests.	All
101			14 Requests.	All
110			16 Requests.	All
111			As much as the FIFO allows.	All
9		Sampler L2 TLB Prefetch Enable		
	Project:		All	
	Default Value:		0h	
	Format:		Enable	
	Value	Name	Description	Project
	0h		TLB Prefetch Disabled	All
	1h		TLB Prefetch Enabled	All



Cache_Mode_0— Cache Mode Register 0				
8	Reserved Project: All Format: MBZ			
7:6	Sampler L2 Request Arbitration			
	Project: All			
	Default Value: 0h			
	Format: U2			
	Value	Name	Description	Project
	00		Round Robin	All
	01		Fetch are Highest Priority	All
	10		Constants are Highest Priority	All
11		Reserved	All	
5	Reserved. This bit must be 0. Note that it defaults to 1.			
4:3	Reserved Project: All Format: MBZ			
2	Reserved Project: All Format: MBZ			
1	Reserved Project: All Format: MBZ			
0	Render Cache Operational Flush Enable			
	Project: All			
	Default Value: 0h			
	Format: Enable			
	Value	Name	Description	Project
	0h	Disable	Operational Flush Disabled (recommended for performance when not rendering to the front buffer)	All
	1h	Enable	Operational Flush Enabled (required when rendering to the front buffer)	All
Errata Description Project				
BWT006 This bit must be clear; Operational Flush cannot be enabled. DevBW-A,B				



8.10.3 Cache_Mode_1— Cache Mode Register 1

Cache_Mode_1— Cache Mode Register 1															
Register Type: MMIO Address Offset: 2124h Project: All Default Value: 0000 0180h Access: Read/32 bit Write Size (in bits): 32															
This Register is saved and restored as part of Context.															
Bit	Description														
31:16	Mask Bits for 15:0 Format: Mask[15:0] Must be set to modify corresponding data bit. Reads to this field returns zero.														
15:13	Reserved Project: All Format: MBZ														
12	Enable the indirect load of Data through the Vertex Fetch Project: All Default Value: 0h Format: U1 <table> <tr> <th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr> <tr> <td>0h</td><td></td><td>Media Object Data transferred through the command streamer</td><td>All</td></tr> <tr> <td>1</td><td></td><td>Media Object Data transferred through the Vertex Fetch</td><td>All</td></tr> </table>			Value	Name	Description	Project	0h		Media Object Data transferred through the command streamer	All	1		Media Object Data transferred through the Vertex Fetch	All
Value	Name	Description	Project												
0h		Media Object Data transferred through the command streamer	All												
1		Media Object Data transferred through the Vertex Fetch	All												



Cache_Mode_1— Cache Mode Register 1

Cache_Mode_1— Cache Mode Register 1				
11	Instruction and State Cache Invalidate			
	Project:		All	
	Default Value:		0h	
	Format:		U1	
	When this field is set, all instruction and state caches (level 1 and level 2) are invalidated. It is intended for debug use. For example, it may be used in conjunction with EU breakpoint control to provide single stepping kernel debugging capability and dynamic breakpoint capability. Before setting this field, host (debug) software must make sure that the graphics render engine has reached idle state – there is no activity to/from the instruction and state caches. For example, during kernel debug, upon a breakpoint exception, host debug software may delay for a sufficiently long period and then check the EU done signals to make sure that all EUs other than the one(s) causing the breakpoint exception are set. It can then set this field to invalidate the instruction and state caches. This field generates a level control signal. Host software must clear this field, before letting execution to continue (e.g. by clearing the host notification MMIO registers to let the kernel under debug to proceed).			
	Value	Name	Description	Project
	0h		Normal Cache operation.	All
	1		Invalidate will be sent to Level 1 and Level 2 caches. (DEBUG ONLY)	All
10	Instruction Level 1 Cache and In-Flight Queue Disable			
	Project:		All	
	Default Value:		0h	
	Format:		Disable	
	Value	Name	Description	Project
	0h		Cache is enabled.	All
	1h		Cache is disabled and all accesses to this cache are treated as misses and sent to L2 cache. Setting this bit overrides the setting of bit 0. (DEBUG ONLY)	All
9	Instruction and State Level 2 Cache Fill Buffers Disable			
	Project:		All	
	Default Value:		0h	
	Format:		Disable	
	Value	Name	Description	Project
	0h		Fill Buffers are enabled.	All
	1h		Fill Buffers are disabled. (DEBUG ONLY)	All



Cache_Mode_1— Cache Mode Register 1				
8:7	Sampler Cache Set XOR selection Project: All Default Value: 3h Format: U2 These bits have an impact only when the Sampler cache is configured in 16 way set associative mode. If the cache is being used for immediate data or for blitter data these bits have no effect.			
	Value	Name	Description	Project
	00	Default value	Default behavior to calculate set address, no XOR.	All
	01	Scheme 1	$\text{New\_set\_mask}[3:0] = \text{Tiled\_address}[16:13]$ $\text{New\_set}[3:0] \leq \text{New\_set\_mask}[3:0] \wedge \text{Old\_set}[3:0]$ Rationale: These bits can distinguish among 16 different equivalent classes of virtual pages. These bits also represent the lsb for tile rows ranging from a pitch of 1 tile to 16 tiles.	All
	10	Scheme 2	$\text{New\_set\_mask}[3] = \text{Tiled\_address}[17] \wedge \text{Tiled\_address}[16]$ $\text{New\_set\_mask}[2] = \text{Tiled\_address}[16] \wedge \text{Tiled\_address}[15]$ $\text{New\_set\_mask}[1] = \text{Tiled\_address}[15] \wedge \text{Tiled\_address}[14]$ $\text{New\_set\_mask}[0] = \text{Tiled\_address}[14] \wedge \text{Tiled\_address}[13]$ $\text{New\_set}[3:0] \leq \text{New\_set\_mask}[3:0] \wedge \text{Old\_set}[3:0]$ Rationale: More bits on each XOR can give better statistical uniformity on sets and since two lsbs are taken for each tile row size, it reduces the chance of aliasing on sets.	All
	11	Scheme 3	$\text{New\_set\_mask}[3] = \text{Tiled\_address}[22] \wedge \text{Tiled\_address}[21] \wedge \text{Tiled\_address}[20] \wedge \text{Tiled\_address}[19]$ $\text{New\_set\_mask}[2] = \text{Tiled\_address}[18] \wedge \text{Tiled\_address}[17] \wedge \text{Tiled\_address}[16]$ $\text{New\_set\_mask}[1] = \text{Tiled\_address}[15] \wedge \text{Tiled\_address}[14]$ $\text{New\_set\_mask}[0] = \text{Tiled\_address}[13]$ $\text{New\_set}[3:0] \leq \text{New\_set\_mask}[3:0] \wedge \text{Old\_set}[3:0]$ Rationale: More bits on each XOR can give better statistical uniformity on sets and since each XOR has different bits, it reduces the chance of aliasing on sets even more.	All
6:5	Reserved	Project:	All	Format: MBZ
4	Reserved	Project:	All	Format: MBZ



Cache_Mode_1— Cache Mode Register 1																
3	Reserved	Project: DevCL	Format: MBZ													
3	A-step bug fix bit for rcc allocation Project: DevBW-A, DevBW-B Default Value: 0h Format: U1 This bit <u>should always be set</u> for proper operation on BW-A,B <table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td></td><td>This bug fix is disabled.</td><td>DevBW-A, DevBW-B</td></tr><tr><td>1h</td><td></td><td>Bug fix is active and will solve random pixel corruption issues due to this bug. It slows down allocation to one allocation every 4 clock. In the A and B-steps, 3d performance will not be bottlenecked by this bug fix. Media performance impact will be minor.</td><td>DevBW-A, DevBW-B</td></tr></table>				Value	Name	Description	Project	0h		This bug fix is disabled.	DevBW-A, DevBW-B	1h		Bug fix is active and will solve random pixel corruption issues due to this bug. It slows down allocation to one allocation every 4 clock. In the A and B-steps, 3d performance will not be bottlenecked by this bug fix. Media performance impact will be minor.	DevBW-A, DevBW-B
Value	Name	Description	Project													
0h		This bug fix is disabled.	DevBW-A, DevBW-B													
1h		Bug fix is active and will solve random pixel corruption issues due to this bug. It slows down allocation to one allocation every 4 clock. In the A and B-steps, 3d performance will not be bottlenecked by this bug fix. Media performance impact will be minor.	DevBW-A, DevBW-B													
2	Reserved	Project: All	Format: MBZ													
1	Instruction and State Level 2 Cache Disable Project: All Default Value: 0h Format: Disable <table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td></td><td>Cache is enabled.</td><td>All</td></tr><tr><td>1h</td><td></td><td>Cache is disabled and all accesses to this cache are treated as misses. (DEBUG ONLY)</td><td>All</td></tr></table>				Value	Name	Description	Project	0h		Cache is enabled.	All	1h		Cache is disabled and all accesses to this cache are treated as misses. (DEBUG ONLY)	All
Value	Name	Description	Project													
0h		Cache is enabled.	All													
1h		Cache is disabled and all accesses to this cache are treated as misses. (DEBUG ONLY)	All													
0	Instruction Level 1 Cache Disable Project: All Default Value: 0h Format: Disable <table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td></td><td>Cache is enabled.</td><td>All</td></tr><tr><td>1h</td><td></td><td>Cache is disabled and all accesses to this cache are treated as misses, but only requests with unique addresses are sent to the L2. (DEBUG ONLY)</td><td>All</td></tr></table>				Value	Name	Description	Project	0h		Cache is enabled.	All	1h		Cache is disabled and all accesses to this cache are treated as misses, but only requests with unique addresses are sent to the L2. (DEBUG ONLY)	All
Value	Name	Description	Project													
0h		Cache is enabled.	All													
1h		Cache is disabled and all accesses to this cache are treated as misses, but only requests with unique addresses are sent to the L2. (DEBUG ONLY)	All													



8.10.4 BB_ADDR—Batch Buffer Head Pointer Register

BB_ADDR—Batch Buffer Head Pointer Register				
Register Type: MMIO Address Offset: 2140h Project: All Default Value: 0000 0000 0000 0000h Access: RO Size (in bits): 64				
This register contains the current DWord Graphics Memory Address of the last-initiated batch buffer.				
Bit	Description			
63:32	Reserved	Project:	All	Format: MBZ
31:2	Batch Buffer Head Pointer	Project:	All	Format: GraphicsAddress[31:2]
	This field specifies the DWord-aligned Graphics Memory Address where the last initiated Batch Buffer is currently fetching commands. If no batch buffer is currently active, the Valid bit will be 0 and this field will be meaningless.			
1	Reserved	Project:	All	Format: MBZ
0	Valid	Project:	All	Format: U1
	Value	Name	Description	Project
	0h	Invalid	Batch buffer Invalid	All
	1h	Valid	Batch buffer Valid	All



8.10.5 BB_STATE – Batch Buffer State Register

BB_STATE – Batch Buffer State Register			
Register Type: MMIO Address Offset: 2110h Project: All Default Value: 0000 0000h Access: R/W Size (in bits): 32			
This register contains the attributes of the last batch buffer initiated from the Ring Buffer. These include the memory space select and security indicator. This register should <i>not</i> be written by software. These fields should only get written by a context restore. Software should always set these fields via the MI_BATCH_BUFFER_START command when initiating a batch buffer. This register is saved and restored with context.			
Bit	Description		
31:6	Reserved	Project: All	Format: MBZ
5	Buffer Security Indicator Project: All Default Value: 0h Format: MI_BufferSecurityType If set, this batch buffer is non-secure and cannot execute privileged commands nor access privileged (GGTT) memory. It will be accessed via the PPGTT. If clear, this batch buffer is secure and will be accessed via the GGTT. Note: This field reflects the effective security level and may not be the same as the Buffer Security Indicator written using MI_BATCH_BUFFER_START.		
	Value	Name	Description
	0h	MIBUFFER_SECURE	Located in GGTT memory
	1h	MIBUFFER_NONSECURE	Located in PPGTT memory
4:0	Reserved	Project: All	Format: MBZ



8.10.6 CTXT_SR_CTL – Context Save/Restore Control Register

CTXT_SR_CTL – Context Save/Restore Control Register															
Register Type: MMIO Address Offset: 2714h Project: All Default Value: 0000 0000h Access: R/W (Debug Only) Size (in bits): 32															
This register is saved and restored with context.															
Bit	Description														
31:2	Reserved	Project: All	Format: MBZ												
1	Extended Context Enable Project: All Default Value: 0h Format: Enable If this bit is set, the extended portion of the render context will be saved and restored with the current context. If clear, extended context will not be a part of this context. Note that since this register is part of ring context, each context can have its own setting for this bit. Extended context can thus be selected on a per-context basis. Note that extended context is part of render context, so that if Render Context Restore Inhibit is set in the context image, extended context will not be restored (the first time) even if this bit is set. <table> <tr> <th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr> <tr> <td>0h</td><td>Disable</td><td>The current context does not include extended context</td><td>All</td></tr> <tr> <td>1h</td><td>Enable</td><td>The current context does include extended context.</td><td>All</td></tr> </table>			Value	Name	Description	Project	0h	Disable	The current context does not include extended context	All	1h	Enable	The current context does include extended context.	All
Value	Name	Description	Project												
0h	Disable	The current context does not include extended context	All												
1h	Enable	The current context does include extended context.	All												
0	Render Context Restore Inhibit Project: All Format: U1 This is not a true register bit. This bit should be set in the context image of a ring context that is being submitted for the first time. Setting this bit will inhibit the restoring of render context (including extended context if applicable) so that restoring of an uninitialized render context can be prevented. This bit will always be set on a context save (since the render context cannot be uninitialized on context save – it will always contain at least default values.)														



8.11 Logical Context Support

8.11.1 CCID—Current Context ID Register

CCID—Current Context ID Register	
Register Type: MMIO Address Offset: 2180h Project: All Default Value: 0000 0000h Access: R/W Size (in bits): 32	
This register contains the current “logical rendering context address” associated with the ring buffer. Programming Note: The CCID register must not be written directly (via MMIO) unless the Command Streamer is completely idle (i.e., the Ring Buffer is empty and the pipeline is idle). Note that, under normal conditions, the CCID register should only be updated from the command stream using the MI_SET_CONTEXT command.	
Bit	Description
31:11	Logical Render Context Address (LRCA) Project: All Default Value: 0h Address: GraphicsAddress[31:11] This field contains the 4 KB-aligned Graphics Memory Address of the current Logical Rendering Context. Bit 11 MBZ. It will point to a Logical Pipeline Context (a subset of a Logical Rendering Context) if loaded using MI_SET_CONTEXT. If this register was set using MI_SET_CONTEXT with the Memory Space Select set to Physical Main Memory, this field contains the 2 KB-aligned “Effective Local Memory” <i>physical</i> Main Memory address of the current Logical Pipeline Context.
10:8	Reserved Project: All Format: MBZ
7:4	Physical Start Address Extension Project: All Default Value: 0h Address: GraphicsAddress[35:32] This field specifies Bits 35:32 of the starting <i>physical</i> address <i>if</i> Memory Space Select of the last MI_SET_CONTEXT command was set to Physical Main Memory.
3	Extended State Save Enable Project: All Format: Enable If set, the extended state identified in the Logical Context Data section of the Memory Data Formats chapter, is saved as part of switching <u>away from</u> this logical context.



CCID—Current Context ID Register																
2	Extended State Restore Enable Project: All Format: Enable If set, the extended state identified in the Logical Context Data section of the Memory Data Formats chapter, was loaded (or restored) as part of switching <u>to</u> this logical context.															
1	Reserved Project: All Format: MBZ															
0	Valid Project: All Default Value: 0h Format: U1 <table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td>Invalid</td><td>The other fields of this register are invalid. A switch away from the context will not invoke a context save operation.</td><td>All</td></tr><tr><td>1h</td><td>Valid</td><td>The other fields of this register are valid, and a switch from the context will invoke the normal context save/restore operations.</td><td>All</td></tr></table>				Value	Name	Description	Project	0h	Invalid	The other fields of this register are invalid. A switch away from the context will not invoke a context save operation.	All	1h	Valid	The other fields of this register are valid, and a switch from the context will invoke the normal context save/restore operations.	All
Value	Name	Description	Project													
0h	Invalid	The other fields of this register are invalid. A switch away from the context will not invoke a context save operation.	All													
1h	Valid	The other fields of this register are valid, and a switch from the context will invoke the normal context save/restore operations.	All													



8.11.2 CXT_SIZE—Context Size with Extended State

CXT_SIZE—Context Size with Extended State	
Register Type: MMIO Address Offset: 21A0h Project: All Default Value: 0000 0013h Access: Read/32 bit Write Size (in bits): 32	
Bit	Description
31:5	Reserved Project: All Format: MBZ
4:0	Size Project: All Default Value: 13h Format: U5-1 Size of pipeline logical rendering context <u>including</u> the extended state in <u>64B quantities minus one</u> .

8.11.3 CXT_SIZE_NOEXT—Context Size without the Extended State

CXT_SIZE_NOEXT—Context Size without the Extended State	
Register Type: MMIO Address Offset: 21A4h Project: All Default Value: 0000 000Fh Access: Read/32 bit Write Size (in bits): 32	
Bit	Description
31:5	Reserved Project: All Format: MBZ
4:0	Size Project: All Default Value: Fh Format: U5-1 Size of pipeline logical rendering context <u>excluding</u> the extended state in <u>64B quantities minus one</u> .



8.12 Arbitration Control, and Scratch Bits

8.12.1 MI_DISPLAY_POWER_DOWN—Display Power Down ([DevCL] Only)

MI_DISPLAY_POWER_DOWN—Display Power Down	
Register Type: MMIO Address Offset: 20E0h Project: DevCL Default Value: 0000 0000h Access: R/W Size (in bits): 32	
The MI_DISPLAY_POWER_DOWN register contains the Display Power Down Enable bit which is used to enable display power down prior to entering C3SR state. This Register is NOT saved and restored as part of Context.	
Bit	Description
31:16	Reserved Project: DevCL Format: MBZ
15	Display Power Down Enable Project: DevCL Format: Enable The bit enables the chipset to put the DIMMs in self refresh when the display conditions are right (No VGA or Overlay, only 1 display pipe enabled) and the CPU is in the C3+ state. Note that setting this bit is not required for DIMMs to enter self-refresh for any device state higher than D0.
14:0	Reserved Project: DevCL Format: MBZ



8.12.2 MI_ARB_STATE—Memory Interface Arbitration State Register

MI_ARB_STATE—Memory Interface Arbitration State Register																
Register Type: MMIO																
Address Offset: 20E4h																
Project: All																
Default Value: 0000 0040h																
Access: R/W																
Size (in bits): 32																
The MI_ARB_STATE register contains state information that controls arbitration aspects of the Memory Interface function.																
Bit	Description															
31:16	Mask Bits Format: Mask[15:0] Must be set to modify corresponding bit in Bits 15:0. (All bits implemented)															
15	Render / Sampler TLB Request Priority Project: All Default Value: 0h Format: U1 <table><thead><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr></thead><tbody><tr><td>0h</td><td></td><td>TLBs are above the corresponding data requests in priority. That is Render TLB fetch is above Render reads and writes, Sampler TLB fetches are above Sampler Fetches. This is the default setting and used for normal operation.</td><td>All</td></tr><tr><td>1h</td><td></td><td>TLBs are at the lowest priority (above FBC) with Sampler TLB fetches higher than render.</td><td>All</td></tr></tbody></table>				Value	Name	Description	Project	0h		TLBs are above the corresponding data requests in priority. That is Render TLB fetch is above Render reads and writes, Sampler TLB fetches are above Sampler Fetches. This is the default setting and used for normal operation.	All	1h		TLBs are at the lowest priority (above FBC) with Sampler TLB fetches higher than render.	All
Value	Name	Description	Project													
0h		TLBs are above the corresponding data requests in priority. That is Render TLB fetch is above Render reads and writes, Sampler TLB fetches are above Sampler Fetches. This is the default setting and used for normal operation.	All													
1h		TLBs are at the lowest priority (above FBC) with Sampler TLB fetches higher than render.	All													
14:9	Reserved Project: All Format: MBZ Read/Write (SW must maintain setting)															



MI_ARB_STATE—Memory Interface Arbitration State Register			
8	Suppress Cacheable indicator from Render Command Stream write requests Project: All Default Value: 0h Format: U1		
	Value	Name	Description
	0h	Snooped	Cacheable write cycles from Render Command Stream are snooped on the FSB.
	1h	Non-Snooped	Cacheable write cycles from Render Command Stream are not snooped on the FSB. These writes are processed as non-snoop.
	Errata	Description	Project
	BWT010	Setting this bit may cause UNDEFINED behavior (extra cycles issued to different addresses in addition to the specified address.)	DevBW-A
7:5	Time Slice Project: All Default Value: 2h Format: U3 Applicable to Render Cache, Sampler Cache, Pixel Shader, Frame Buffer, Command stream and Host Requests. Time Slice is fixed at 1 for TLB and snoop requests, and not applicable to Isochronous Streams. The (value programmed –1) determines the number of Page Hits before arbitration switch for a low priority stream interrupted by a higher priority stream as long as the lower priority stream is active. If set to '000' the arbiter does apply a page hit grace period. In 64B Requests		
	Value	Name	Description
	000	1 Request	1 Requests (This setting implies that the grace period is disabled)
	001	2 Requests	2 Requests
	010	4 Requests	4 Requests
	011	6 Requests	6 Requests
	100	8 Requests	8 Requests
	101	10 Requests	10 Requests
	110	14 Requests	14 Requests
	111	16 Requests	16 Requests



MI_ARB_STATE—Memory Interface Arbitration State Register																
4	Low Priority Grace Period Page Size Project: All Default Value: 0h Format: U1 <table><thead><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr></thead><tbody><tr><td>0h</td><td></td><td>Grace period on-page indicator uses 4KB pages in the command streams and caches. (Default)</td><td>All</td></tr><tr><td>1h</td><td></td><td>Grace period on-page indicator uses 8KB pages in the command streams and caches.</td><td>All</td></tr></tbody></table>				Value	Name	Description	Project	0h		Grace period on-page indicator uses 4KB pages in the command streams and caches. (Default)	All	1h		Grace period on-page indicator uses 8KB pages in the command streams and caches.	All
Value	Name	Description	Project													
0h		Grace period on-page indicator uses 4KB pages in the command streams and caches. (Default)	All													
1h		Grace period on-page indicator uses 8KB pages in the command streams and caches.	All													
3	Reserved Project: All Format: MBZ Read/Write (SW must maintain setting)															
2	Display A/B Trickle Feed Disable Project: All Default Value: 0h Format: Disable <table><thead><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr></thead><tbody><tr><td>0h</td><td></td><td>Enable</td><td>All</td></tr><tr><td>1h</td><td></td><td>Disable (Turn off trickle feed Display request)</td><td>All</td></tr></tbody></table> Programming Notes For mobile devices ([DevCL]), this bit should always be set to disable trickle feed. [DevBW] must always set to disable trickle feed Project DevCL DevBW				Value	Name	Description	Project	0h		Enable	All	1h		Disable (Turn off trickle feed Display request)	All
Value	Name	Description	Project													
0h		Enable	All													
1h		Disable (Turn off trickle feed Display request)	All													
1	Reserved Project: All Format: MBZ Read/Write (SW must maintain setting)															
0	Display A/B Priority Select Project: All Default Value: 0h Format: U1 This bit determines the arbitration priority of accesses among the high priority streams. <table><thead><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr></thead><tbody><tr><td>0h</td><td>DA/DB/Others</td><td>Set this when Display Plane A is the Primary</td><td>All</td></tr><tr><td>1h</td><td>DB/DA/Others</td><td>Set this when Display Plane B is the Primary</td><td>All</td></tr></tbody></table>				Value	Name	Description	Project	0h	DA/DB/Others	Set this when Display Plane A is the Primary	All	1h	DB/DA/Others	Set this when Display Plane B is the Primary	All
Value	Name	Description	Project													
0h	DA/DB/Others	Set this when Display Plane A is the Primary	All													
1h	DB/DA/Others	Set this when Display Plane B is the Primary	All													



8.12.3 MI_RDRET_STATE—Memory Interface Read Return State Register

MI_RDRET_STATE—Memory Interface Read Return State Register																
Register Type: MMIO																
Address Offset: 20FCh [DevCL] 20E0h [DevBW]																
Project: All																
Default Value: 0000 0000h																
Access: R/W																
Size (in bits): 32																
The MI_RDRET_STATE register contains state information that controls data return aspects of the Memory Interface function. This register is used strictly for HVM testing. Any functional usage of this register is undefined. None of the TLB read returns from memory are impacted by this register.																
This Register is NOT saved and restored as part of Context.																
Bit	Description															
31:16	Mask Bits Format: Mask[15:0] Must be set to a 1 to allow modification of corresponding bit in Bits 15:0. (All implemented bits)															
15	HVM Enable Bit Project: All Format: Enable This bit must be set to '1' to enable HVM loopback mode and enable random internal data returns from CI. This bit must be programmed after the other client specific bits are programmed to desired values.															
14:9	Reserved Project: All Format: MBZ															
8	Vertex Fetch Cache select Project: All Default Value: 0h Format: U1 This bit determines the read return for Vertex Fetch Reads <table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td></td><td>Return Data from memory</td><td>All</td></tr><tr><td>1h</td><td></td><td>Return data from a random data generator on-chip</td><td>All</td></tr></table>				Value	Name	Description	Project	0h		Return Data from memory	All	1h		Return data from a random data generator on-chip	All
Value	Name	Description	Project													
0h		Return Data from memory	All													
1h		Return data from a random data generator on-chip	All													
7	Reserved Project: All Format: MBZ Was Read Only Cache select															



MI_RDRET_STATE—Memory Interface Read Return State Register																
6	Overlay return Select Project: DevCL Default Value: 0h Format: U1 This bit determines the read return for Overlay streamer Reads <table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td></td><td>Return Data from memory</td><td>DevCL</td></tr><tr><td>1h</td><td></td><td>Return data from a random data generator on-chip</td><td>DevCL</td></tr></table>				Value	Name	Description	Project	0h		Return Data from memory	DevCL	1h		Return data from a random data generator on-chip	DevCL
Value	Name	Description	Project													
0h		Return Data from memory	DevCL													
1h		Return data from a random data generator on-chip	DevCL													
6	Reserved	Project: DevBW	Format: MBZ													
5	Color / Z return Select Project: All Default Value: 0h Format: U1 This bit determines the read return for Low Priority Reads <table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td></td><td>Return Data from memory</td><td>All</td></tr><tr><td>1h</td><td></td><td>Return data from a random data generator on-chip</td><td>All</td></tr></table>				Value	Name	Description	Project	0h		Return Data from memory	All	1h		Return data from a random data generator on-chip	All
Value	Name	Description	Project													
0h		Return Data from memory	All													
1h		Return data from a random data generator on-chip	All													
4	Sampler Read return Select Project: All Default Value: 0h Format: U1 This bit determines the read return for Low Priority Reads <table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td></td><td>Return Data from memory</td><td>All</td></tr><tr><td>1h</td><td></td><td>Return data from a random data generator on-chip</td><td>All</td></tr></table>				Value	Name	Description	Project	0h		Return Data from memory	All	1h		Return data from a random data generator on-chip	All
Value	Name	Description	Project													
0h		Return Data from memory	All													
1h		Return data from a random data generator on-chip	All													
3	Cursor (A and B) Read return Select Project: All Default Value: 0h Format: U1 This bit determines the read return for Display Reads <table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td></td><td>Return Data from memory</td><td>All</td></tr><tr><td>1h</td><td></td><td>Return data from a random data generator on-chip</td><td>All</td></tr></table>				Value	Name	Description	Project	0h		Return Data from memory	All	1h		Return data from a random data generator on-chip	All
Value	Name	Description	Project													
0h		Return Data from memory	All													
1h		Return data from a random data generator on-chip	All													



MI_RDRET_STATE—Memory Interface Read Return State Register															
2	Display C Read return Select Project: All Default Value: 0h Format: U1 This bit determines the read return for Display C Reads <table><thead><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr></thead><tbody><tr><td>0h</td><td></td><td>Return Data from memory</td><td>All</td></tr><tr><td>1h</td><td></td><td>Return data from a random data generator on-chip</td><td>All</td></tr></tbody></table>			Value	Name	Description	Project	0h		Return Data from memory	All	1h		Return data from a random data generator on-chip	All
Value	Name	Description	Project												
0h		Return Data from memory	All												
1h		Return data from a random data generator on-chip	All												
1	Display B Read return Select Project: All Default Value: 0h Format: U1 This bit determines the read return for Display Reads <table><thead><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr></thead><tbody><tr><td>0h</td><td></td><td>Return Data from memory</td><td>All</td></tr><tr><td>1h</td><td></td><td>Return data from a random data generator on-chip</td><td>All</td></tr></tbody></table>			Value	Name	Description	Project	0h		Return Data from memory	All	1h		Return data from a random data generator on-chip	All
Value	Name	Description	Project												
0h		Return Data from memory	All												
1h		Return data from a random data generator on-chip	All												
0	Display A Read return Select Project: All Default Value: 0h Format: U1 This bit determines the read return for Display Reads <table><thead><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr></thead><tbody><tr><td>0h</td><td></td><td>Return Data from memory</td><td>All</td></tr><tr><td>1h</td><td></td><td>Return data from a random data generator on-chip</td><td>All</td></tr></tbody></table>			Value	Name	Description	Project	0h		Return Data from memory	All	1h		Return data from a random data generator on-chip	All
Value	Name	Description	Project												
0h		Return Data from memory	All												
1h		Return data from a random data generator on-chip	All												



8.12.4 MI_MODE — Mode Register for Software Interface

MI_MODE — Mode Register for Software Interface																
Register Type: MMIO Address Offset: 209Ch Project: All Default Value: 00000000h Access: R/W Size (in bits): 32																
The MI_MODE register contains information that controls software interface aspects of the Memory Interface function.																
Bit	Description															
31:16	Masks Format: Mask[15:0] A “1” in a bit in this field allows the modification of the corresponding bit in Bits 15:0															
15	Reserved Read/Write	Project:	All	Format: MBZ												
14	Reserved Read/Write	Project:	All	Format: MBZ												
13	Flush Performance mode Project: DevCL Default Value: 0h Format: U1 <table><thead><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr></thead><tbody><tr><td>0h</td><td>run fast restore</td><td>No NonPipelined SV flush.</td><td>DevCL</td></tr><tr><td>1h</td><td>run slow legacy restore</td><td>With NonPipelined SV flush.</td><td>DevCL</td></tr></tbody></table>				Value	Name	Description	Project	0h	run fast restore	No NonPipelined SV flush.	DevCL	1h	run slow legacy restore	With NonPipelined SV flush.	DevCL
Value	Name	Description	Project													
0h	run fast restore	No NonPipelined SV flush.	DevCL													
1h	run slow legacy restore	With NonPipelined SV flush.	DevCL													
13	Reserved Read/Write	Project:	DevBW	Format: MBZ												
12	Reserved	Project:	All	Format: MBZ												
11	Invalidate UHPTR enable If bit set H/W clears the valid bit of UHPTR (2134h, bit 0) when current active head pointer is equal to UHPTR.															
10	Power of 2 Fences Enable This field is used to indicate to the hardware that the fences in use currently are for Power of 2 tile pitch. This bit is used by the chipset for performance enhancement.															



MI_MODE — Mode Register for Software Interface				
9	Rings Idle			
	Project:		All	
	Default Value:		0h	
	Format:		U1	
	Read Only Status bit			
	Value	Name	Description	Project
	0h	Not Idle	Parser not Idle or Ring Arbiter not Idle.	All
	1h	Idle	Parser Idle and Ring Arbiter Idle.	All
	Programming Notes			Project
	Writes to this bit are not allowed.			All
8	Stop Rings			
	Project:		All	
	Default Value:		0h	
	Format:		U1	
	Value	Name	Description	Project
	0h		Normal Operation.	All
	1h		Parser is turned off and Ring arbitration is turned off.	All
	Programming Notes			Project
	Software must set this bit to force the Rings and Command Parser to Idle. Software must read a “1” in Ring Idle bit after setting this bit to ensure that the hardware is idle.			All
Software must clear this bit for Rings to resume normal operation.			All	
7	Vertex Shader Cache Mode			
	Project:		All	
	Default Value:		0h	
	Format:		U1	
	Value	Name	Description	Project
	0h	Non-LRA	Non-LRA mode of allocation. Vertex shader cache is allocated on the basis of the reference count of individual vertices	All
1h	LRA	LRA mode of allocation. Used for validation purposes.	All	



MI_MODE — Mode Register for Software Interface				
6	Vertex Shader Timer Dispatch Enable Project: All Default Value: 0h Format: Enable			
	Value	Name	Description	Project
	0h	Disable	Disable the timer for dispatch of single vertices from the vertex shader. Vertex shader will try to collect 2 vertices before a dispatch	All
	1h	Enable	Enable the timer for dispatch of single vertices. Dispatch a single vertex shader thread after the timer expires.	All
	Programming Notes To avoid deadlock conditions in hardware this bit needs to be set for normal operation.			Project All
5	FBC2 Modification Enable Project: All Default Value: 0h Format: Enable			
	Value	Name	Description	Project
	0h	Disable	FBC logic does not look at the modifications to the frame buffer.	All
	1h	Enable	FBC logic looks at the modifications into the frame buffer.	All
4	Reserved: MBZ			
3	Physical Batch Buffer 4K size limit disable (test mode) Project: All Default Value: 0h Format: Disable			
	Value	Name	Description	Project
	0h	Disable	Physical batch buffers more than 4K in size are not permitted.	All
	1h	Enable	Physical batch buffers more than 4K in size are permitted.	All
2	Reserved Read/Write	Project:	All	Format: MBZ



MI_MODE — Mode Register for Software Interface				
1	Dummy Read Disable	Project:	All	Format: Disable
Nominally a command stream flush is completed with a dummy read to memory to push all pending writes. Setting this bit to a “1” disables the dummy read.				
0	Mask IIR disable	Project:	All	Format: Disable
Mask IIR disable. Nominally the Interrupt controller masks interrupts in the IIR register if an interrupt acknowledge from the 3gio interface is pending. Setting this bit to a “1” allows interrupts to be visible to the interrupt controller while an interrupt acknowledge is pending.				



8.12.5 ECOSKPD—ECO Scratch Pad (DEBUG)

ECOSKPD—ECO Scratch Pad (DEBUG)																
Register Type: MMIO																
Address Offset: 21D0h																
Project: All																
Default Value: 00000307h																
Access: R/W																
Size (in bits): 32																
Bit	Description															
31:16	Mask Bits Format: Mask[15:0] Must be set to modify corresponding bit in Bits 15:0. (All implemented bits)															
15	Reserved	Project: All	Format: MBZ													
14	Vertex Shader Dual dispatch disable Project: DevBW-E Security: None Default Value: 0h Enable the dual dispatch Mask: MMIO(0x21D0)#30 <table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td>Enable</td><td>HW implements the fix for the enhanced dual dispatch. Dual dispatch is triggered only when the top entry in the tracking FIFO is a</td><td>DevBW-E</td></tr><tr><td>1h</td><td>Disable</td><td>Disable the HW fix for the enhanced dual dispatch. Vertex shader will be dispatched as a single vertex everytime the tracking FIF becomes full.</td><td>DevBW-E</td></tr></table>				Value	Name	Description	Project	0h	Enable	HW implements the fix for the enhanced dual dispatch. Dual dispatch is triggered only when the top entry in the tracking FIFO is a	DevBW-E	1h	Disable	Disable the HW fix for the enhanced dual dispatch. Vertex shader will be dispatched as a single vertex everytime the tracking FIF becomes full.	DevBW-E
Value	Name	Description	Project													
0h	Enable	HW implements the fix for the enhanced dual dispatch. Dual dispatch is triggered only when the top entry in the tracking FIFO is a	DevBW-E													
1h	Disable	Disable the HW fix for the enhanced dual dispatch. Vertex shader will be dispatched as a single vertex everytime the tracking FIF becomes full.	DevBW-E													
13	Clipper Performance Fix Disable Project: DevBW-E Security: None Default Value: 0h DefaultVaueDesc Mask: MMIO(0x21D0)#29 <table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td>Disable</td><td>Desc</td><td>All</td></tr><tr><td>1h</td><td>Enable</td><td>Desc</td><td>All</td></tr></table>				Value	Name	Description	Project	0h	Disable	Desc	All	1h	Enable	Desc	All
Value	Name	Description	Project													
0h	Disable	Desc	All													
1h	Enable	Desc	All													



ECOSKPD—ECO Scratch Pad (DEBUG)				
12	Clipper software workaround for DX10 Enable			
	Project:		DevBW-E	
	Security:		None	
	Default Value:		0h	DefaultVaueDesc
	Mask:		MMIO(0x21D0) #28	
	Value	Name	Description	Project
	0h	Disable	Desc	DevBW-E
	1h	Enable	Desc	DevBW-E
	Programming Notes			Project
	Notes			DevBW-E
	This bit is expected to be used with bit 9 in this register:			
	bit9	bit12		
0	0	PerformanceECO. No software workaround in vs0.		
1	0	Not valid.		
0	1	Software workaround for Dx10		
1	1	No ECO. Will need Software workaround for Dx9.		
11	PL Unit bug fix	Project:	All	Format: U1
Unspecified ECO disable in the PL unit				
10	RCC Unit Bug fix	Project:	All	Format: U1
Unspecified ECO disable in the RCC unit				
9	Clipper fix for definition of Bad vertex			
	Project:		DevCL, DevBW-E	
	Security:		None	
	Default Value:		0h	DefaultVaueDesc
	Mask:		MMIO(0x21D0) #25	
	Value	Name	Description	Project
0h	Disable	BAD vertex is dealt as a trivial reject	DevCL, DevBW-E	
1h	Enable	BAD vertex is dealt as a must clip instead of trivial reject	DevCL, DevBW-E	
8	Clock gating for the RCC (Disable one clock gate cell)			
	Project:		DevCL	
	Default Value:		0h	DefaultVaueDesc
	Format:		U1	FormatDesc
	0 = Disable Clock gating			
	1 = Enable clock gating			



ECOSKPD—ECO Scratch Pad (DEBUG)				
7	Clock gating for the MAWB Project: DevCL Default Value: 0h DefaultVaueDesc Format: U1 FormatDesc 0 = Disable Clock gating 1 = Enable clock gating			
6	Reserved	Project: All	Format: MBZ	
5	Reserved	Project: All	Format: MBZ	
4	Constant Buffer Save / Restore Disable Project: DevBW-C1+ Default Value: 0h DefaultVaueDesc Format: U1 FormatDesc "0" : constant buffer should part of context save/restore "1": constant buffer should not be part of context save/restore			
3	WIZunit Scratch Space ECO Project: DevBW-C+ Default Value: 0h DefaultVaueDesc Format: U1 FormatDesc Enable ECO: Max scratch space (indicated by Per Thread Scratch Space set to 11) is 256KB. 256KB scratch base must be 8M aligned. Disable ECO: Max scratch space is 12KB.			
2:0	Reserved	Project: All	Format: MBZ	



8.13 Debug Registers

These registers are used to reflect internal hardware state. The intention is to be used for silicon debug

8.13.1 CSFLFSM — Flush FSM (Debug)

CSFLFSM — Flush FSM (Debug)			
Register Type: MMIO Address Offset: 2200h Project: All Default Value: 0000 0000h Access: R/W Size (in bits): 32			
Bit	Description		
31:16	Reserved: 0x0	Project: All	Format: MBZ
15:13	Project: All Format: U3 "000" * (CSFLSHFIFOIDLE_s == '1') + "001" * (CSFLSHFIFOVIRXPHY_s == '1') + "010" * (CSFLSHFIFOWT4ACK_s == '1') + "011" * (CSFLSHFIFOLDSTDW_s == '1') + "100" * (CSFLSHFIFOISCFLUSH_s == '1') + "101" * (CSFLSHFIFOMSI_s == '1') + "110" * (CSFLSHFIFODMYRD_s == '1') + "111"		
12:10	Project: All Format: U3 "000" * (CS3DCNTRLIDLE_s == '1') + "001" * (CS3DCNTRLDW1_s == '1') + "010" * (CS3DCNTRLDW2_s == '1') + "011" * (CS3DCNTRLDFIFO_s == '1') + "100" * (CS3DCNTRLWT4DONE_s == '1') + "101" * (CS3DCNTRLNULL_s == '1') + "111"		
9:8	Project: All Format: U2 "00" * (URBIDLE_s == '1') + "01" * (URBPIPESEL_s == '1') + "10" * (URBCURBECLEAR_s == '1') + "11" * (URBDEALLOC_s == '1')		



CSFLFSM — Flush FSM (Debug)			
7:4	Project:	All	Format: U4
	"0000" * (URBNIDLE_s == '1') + "0001" * (URBNCLR_s == '1') + "0010" * (URBNCLRS_s == '1') + "0011" * (URBNSET_s == '1') + "0100" * (URBNRPLC_s == '1') + "0101" * (URBNRPLC_W_s == '1') + "0110" * (URBCLRWT_s == '1') + "0111" * (URBNPRIM_s == '1') + "1000" * (URBNRPLC_WVS0_s == '1') + "1111"		
3:0	Project:	All	Format: U4
	"0000" * (IDLE_S == '1')+ "0001" * (NF3DADDR_S == '1')+ "0010" * (NF3DADDR_URB_S == '1')+ "0011" * (NFNPRIM_URBCLR_S == '1')+ "0100" * (NFMDADDR_S == '1')+ "0101" * (NF3DNPRIM_S == '1')+ "0110" * (NFMDNPRIM_S == '1')+ "0111" * (NFURBNPRIM_S == '1')+ "1000" * (NFURBWALLOC_S == '1')+ "1111"		



8.13.2 CSFLFLAG — Flush FLAG (Debug)

CSFLFLAG — Flush FLAG (Debug)	
Register Type: MMIO Address Offset: 2204h Project: All Default Value: 0000 0000h Access: R/W Size (in bits): 32	
Bit	Description
31:17	Reserved: 0x0 Project: All Format: MBZ
16:9	Project: All Format: U8 csprsrallflsh& csctxlclflsh& csynclclflush &fi_write & fi_depth & fi_timestamp & fi_isclflush & fi_globalcnt_rst
8	cs_media_select Project: All Format: U1
7:0	Project: All Format: U8 fi_MURB_chng & fi_MSP_flag & fi_URB_chng & fi_PSP_flag & fi_BTP_flag & fi_curbe_opcodes & fi_only_one_curbe_avail & cs_curbe_set



8.13.3 CSFLTRK — Flush Track (Debug)

CSFLTRK — Flush Track (Debug)	
Register Type: MMIO Address Offset: 2208h Project: All Default Value: 0000 0000h Access: R/W Size (in bits): 32	
Bit	Description
31:13	Reserved: 0x0 Project: All Format: MBZ
12:8	Project: All Format: U5 fi_3dcntrl_ldfifo & fi_3dcntrlfifo_full & fi_3dcntrl_ram_wren & fi_3dcntrl_ram_wraddr[1:0]
7:0	Project: All Format: U8 fi_3dcntrl_rdp[1:0] & fi_fiford & fi_3dcntrl_ramwrp[1:0] & fi_3dcntrl_completeptr_crb2clk[2:0]

8.13.4 CSCMDOP — Instruction DWORD (Debug)

CSCMDOP — Instruction DWORD (Debug)	
Register Type: MMIO Address Offset: 220Ch Project: All Default Value: 0000 0000h Access: R/W Size (in bits): 32	
Bit	Description
31:0	Command Buffer Data Project: All Format: U32 This field represents the data being parsed by the command streamer currently



8.13.5 CSCMDVLD — Instruction DWORD Valid (Debug)

CSCMDVLD — Instruction DWORD Valid (Debug)	
Register Type: MMIO Address Offset: 2210h Project: All Default Value: 0000 0000h Access: R/W Size (in bits): 32	
Bit	Description
31:1	Reserved Project: All Format: MBZ
0	Command Buffer Valid Project: All Format: U1 Command buffer currently has valid data

8.13.6 CLKCMP — Compare count clock stop (Debug)

CLKCMP — Compare count clock stop (Debug)	
Register Type: MMIO Address Offset: 2360h Project: All Default Value: 0000 0000 0000 0000h Access: R/W This register is <i>not</i> set by the context restore. Size (in bits): 64	
This register stores the value of the count of clock ticks that should cause the clock to stop. An internal hardware counter keeps track of the clock ticks. The internal hardware counter is reset when this register is written. The reference clock used by this counter is the core render clock (crclk). Crclk is chosen here specifically because it is the operating frequency for a majority of the logic in the 3D pipeline. See the EDS for details for the frequency of the crclk. See section 1.21.	
Bit	Description
63:0	Clock Stop Value Project: All Format: U64 This register reflects the total number of crclk ticks that need to pass before the crclk is stopped. A write to this register causes the internal clock counter to reset.



8.13.7 VFDC—Set Value of Draw Count (DEBUG)

VFDC—Set Value of Draw Count (DEBUG)	
Register Type:	MMIO
Address Offset:	2450h
Project:	All
Default Value:	UUUU UUUUh
Access:	R/W
Size (in bits):	32
The VFDC register is to set the initial DRAW count starting point. This is needed to be able to reset and start at different draw counts.	
Bit	Description
31:24	Reserved Project: All Format: MBZ
23:0	Set Value of Draw Count Project: All Format: U24 This value must be set before enabling the Skip Initial Primitive or Max Primitives Limit Enable . If not then the start of the Draw Count is undefined.

8.13.8 VFSKPD—VF Scratch Pad (DEBUG)

VFSKPD—VF Scratch Pad (DEBUG)	
Register Type:	MMIO
Address Offset:	2470h
Project:	All
Default Value:	0000 0000h
Access:	R/W
Size (in bits):	32
Bit	Description
31:16	Mask Bits Format: Mask[15:0] Must be set to modify corresponding bit in Bits 15:0. (All bits implemented)
15	SnapShot Continue Project: All Format: U1 Write a '1' to this field with the mask will allow VF to continue once a SnapShot occurs. Writing a '0' has no effect.
14:3	Reserved Project: All Format: MBZ



VFSKPD—VF Scratch Pad (DEBUG)			
2	Vertex Cache Implicit Disable Inhibit		
	Project:	All	
	Default Value:	0h	
	Format:	U1	
	Value	Name	Description
	0h		Allow VF to disable VS0 when Sequential index or Prim ID is a valid Element.
	1h		VF never implicitly disables the vertex cache. Software must disable the VS0 Cache when required.
1	Disable Over Fetch Cache		
	Project:	All	
	Default Value:	0h	
	Format:	Disable	
	Value	Name	Description
	0h		Cache will check for data in cache before making a request to memory
	1h		Always re-fetch new data from memory.
0	Disable Pending FIFO		
	Project:	All	
	Default Value:	0h	
	Format:	Disable	
	Value	Name	Description
	0h		Allow VFunit to request TLB data without waiting for pending TLB data to return.
	1h		Only allow one pending TLB request at a time



8.14 Software Visible Counter Registers

These registers keep continuous count of time and pixels passing the depth test. They are saved and restored with context but should not be changed by software except to reset them to 0 at context creation time. These registers may be read at any time; however, to obtain a meaningful result, a pipeline flush just prior to reading the registers is necessary in order to synchronize the counts with the primitive stream.

8.14.1 PS_DEPTH_COUNT — Reported Pixels Passing Depth Test Counter

PS_DEPTH_COUNT	
Register Type: MMIO Address Offset: 2350h Project: All Default Value: 00000000h; 00000000h; Access: R/W Size (in bits): 64	
This register stores the value of the count of pixels that have passed the depth test. This register is part of the context save and restore. Note that the value of this register can be obtained in a pipeline-synchronous fashion without a pipeline flush by using the 3DCONTROL command. See 3D Overview in the 3D volume.	
Bit	Description
63:0	Depth Count This register reflects the total number of pixels that have passed the depth test (i.e., will be visible). All pixels are counted when Statistics Enable is set in the Windower State. See the Windower chapter of the 3D volume for details. Pixels that pass the depth test but fail the stencil test will <i>not</i> be counted.



8.14.2 TIMESTAMP — Reported Timestamp Count

TIMESTAMP — Reported Timestamp Count	
Register Type:	MMIO
Address Offset:	2358h
Project:	All
Default Value:	0000 0000 0000 0000h
Access:	R/W. This register is <i>not</i> set by the context restore.
Size (in bits):	64
This register stores the value of the count of clock ticks that have passed since it was last reset. Note that the value of this register can be obtained in a 3D pipeline-synchronous fashion without a pipeline flush by using the 3DCONTROL command. See 3D Pipeline in the <i>3D and Media</i> volume. The reference clock used by this counter is the GMCH core and Processor-Side Bus (PSB) clock referred to as “hclk”. Hclk is not used elsewhere in the graphics device and is chosen here specifically because it is not subject to throttling as the graphics device clock is. The hclk used is not gated, throttled or selectively powered down so that the TIMESTAMP can remain accurate even during power management activity (as long as the GMCH does not have all of its clocks stopped, as when it is fully powered down.) The frequency of hclk is determined externally to the GMCH and can be discovered through the “Clocking Configuration” (“CLKCFG”) MCHBAR register. See the EDS for details. Note that the MCHBAR registers can be accessed through the MCHBAR aperture in MMIO space. See section 8.20. TIMESTAMP is <i>not</i> reset by a <u>graphics</u> reset. It will maintain its value unless a full chipset reset is performed.	
Bit	Description
63:0	TIMESTAMP Project: All Format: U64 This register reflects the total number of ticks that have passed since reset or the last time 0000 0000 0000 0000h was written to this register. SW should not write a non-zero value to this register. The value in this register increments once every 16 hclks. A full GMCH reset is required to reset this register; since this register is in the hclk domain it is not reset by a graphics reset alone.



8.15 MTCH_CID_RST – Matched Context ID Reset Register

MTCH_CID_RST – Matched Context ID Reset Register				
Register Type: MMIO				
Address Offset: 2524h				
Project: All				
Default Value: 0000 0000h				
Access: R/W				
Size (in bits): 32				
This register is used to generate a Context ID specific reset (Render Only). To initiate a reset, the register is written with the pending bit set. Hardware compares the current context ID with the register and on match generates a Render Only reset. After reset is complete, HW clears the pending bit and can be programmed to generate an interrupt. The match bit is set. If the current context ID does not match this register, the pending bit is reset and an interrupt is generated. The match bit is reset. The match indicates the result of the last comparison, and its valid only when pending bit is zero. Please see MCIDRST interrupt bit assignment in the Interrupt Control Registers.				
Bit	Description			
31:12	Reserved	Project:	All	Format: MBZ
11:2	Reserved	Project:	All	Format: MBZ
1	Reserved	Project:	All	Format: MBZ
0	Reserved	Project:	All	Format: MBZ



8.16 Interrupt Control Registers

The Interrupt Control Registers described below all share the same bit definition. The bit definition is as follows:

Table 8-3. Bit Definition for Interrupt Control Registers

Bit	Description
31:4	Reserved. MBZ: These bits may be assigned to interrupts on future products/steppings.
7	Timeout Counter Expired: Set when the VCS timeout counter has reached the timeout threshold value.
6:4	Reserved: MBZ
3	Reserved: MBZ
2	Render Command Parser Master Error: When this status bit is set, it indicates that the hardware has detected an error. It is set by the device upon an error condition and cleared by a CPU write of a one to the appropriate bit contained in the Error ID register followed by a write of a one to this bit in the IIR. Further information on the source of the error comes from the “Error Status Register” which along with the “Error Mask Register” determines which error conditions will cause the error status bit to be set and the interrupt to occur. Page Table Error: Indicates a page table error. Instruction Parser Error: The Renderer Instruction Parser encounters an error while parsing an instruction.
1	Sync Status: This bit is toggled when the Instruction Parser completes a flush with the sync enable bit active in the INSTPM register. The toggle event will happen after all the graphics engines are flushed. The HW Status DWord write resulting from this toggle will cause the CPU’s view of graphics memory to be coherent as well (flush and invalidate the render cache).
0	Render Command Parser User Interrupt: This status bit is set when an MI_USER_INTERRUPT instruction is executed on the Render Command Parser. Note that instruction execution is not halted and proceeds normally. A mechanism such as an MI_STORE_DATA instruction is required to associate a particular meaning to a user interrupt.



8.16.1.1 BCS_IPEIR—Instruction Parser Error Identification Register (Debug)

Address Offset: 04064h–04067h
Default Value: 0000 0000h
Access: Read Only
Size: 32 bits

The IPEIR register identifies the general location of instructions that generated an Invalid Instruction Errors for the Renderer IP. (Note: The header (DWord 0) of the offending instruction will be stored in the IPEHR register).

Bit	Description
31:4	Reserved. Read as zero
3	Batch Buffer Error: If this bit is set the faulting instruction was executed from a batch buffer. If this bit is clear the faulting instruction was executed directly from a ring buffer.
2:0	Reserved. Read as zero

8.16.1.2 BCS_IPEHR—Instruction Parser Error Header Register (Debug)

Address Offset: 04068h–0406Bh
Default Value: 0000 0000h
Access: Read Only
Size: 32 bits

The IPEHR register is used to identify the instructions that generate Invalid Instruction Errors. This register is loaded with the header (DWord 0) of each instruction that is executed. It will therefore hold the header of an instruction that generates an Invalid Instruction Error.

Bit	Description
31:0	Header: This field will contain the header (DWord 0) of a Media Decode IP instruction that generates an Invalid Instruction Error.

8.16.1.3 BCS_ACTHD — Active Head Pointer Register (Debug)

Address Offset: 04074h–04077h
Default Value: 0000 0000h
Access: Read Only
Size: 32 bits

This register contains the Head “Pointer” (DWord Graphics Memory Address) of the ring buffer.

Bit	Description
31:2	Head Pointer: DWord Graphics Address corresponding to the Head Pointer of the ring or batch buffer.
1:0	Reserved: MBZ



8.16.1.4 BCS_DMA_FADD —DMA Engine Fetch Address (Debug)

Address Offset: 04078h – 0407Bh
 Default Value: 0000 0000h
 Access: Read Only
 Size: 32 bits

This register contains the QWord offset from the start address of the instruction being fetched by the DMA engine.

Bit	Description
31:3	Current DMA QWord Offset: This field contains the offset of the QWord (from the start of the ring buffer or batch buffer) that the “Video Decode” instruction parser DMA engine is currently accessing (fetching). Note that this offset will typically lead the Head offset (as instructions must be fetched before execution).
2:0	Reserved: MBZ

8.16.1.5 BCS_HWS_PGA — Hardware Status Page Address Register

Address Offset: 04080h–04083h
 Default Value: 1FFF F000h
 Access: Read/Write
 Size: 32 bits

This register is used to program the 4 KB-aligned System Memory address of the Hardware Status Page used to report hardware status into (typically cacheable) System Memory.

Bit	Description
31:12	Address: This field is used by SW to specify Bits 31:12 of the 4 KB-aligned System Memory address of the 4 KB page known as the “Hardware Status Page”. Bits 11:0 of the address MBZ. Format = Bits 31:12 of Graphics Memory Address
11:0	Reserved: MBZ

The following table defines the layout of the Hardware Status Page:

DWord Offset	Description
3:0	Reserved. Must not be used.
4	Head Pointer Storage: The contents of the Ring Buffer Head Pointer register (register DWord 1) are written to this location either as result of an MI_REPORT_HEAD instruction or as the result of an “automatic report” (see RINGBUF registers).
0Fh:05h	Reserved. Must not be used.
(3FFh – 010h)	These locations can be used for general purpose via the MI_STORE_DATA_INDEX or MI_STORE_DATA_IMM instructions.



8.16.1.6 BCS_NOPID — NOP Identification Register

Address Offset: 04094h–04097h
 Default Value: 0000 0000h
 Access: Read Only
 Size: 32 bits

The BCS_NOPID register contains the Noop Identification value specified by the last MI_NOOP instruction that enabled this register to be updated.

Bit	Description
31:22	Reserved: MBZ
21:0	Identification Number: This field contains the 22-bit Noop Identification value specified by the last MI_NOOP instruction that enabled this field to be updated.

8.16.1.7 BCS_MI_MODE — Mode Register for Software Interface

Address Offset: 0409Ch–0409Fh
 Default Value: 0000 0000h
 Access: Read/Write
 Size: 32 bits

The MI_MODE register contains information that controls software interface aspects of the command parser.

Bit	Description
31:16	Masks: A “1” in a bit in this field allows the modification of the corresponding bit in Bits 15:0
15:12	Reserved Read/Write
11	Invalidate UHPTR enable: If bit set H/W clears the valid bit of BCS_UHPTR (4134h, bit 0) when current active head pointer is equal to UHPTR.
10	Reserved Read/Write
9	Ring Idle (Read Only Status bit) 0 = Parser not Idle 1 = Parser Idle <i>Writes to this bit are not allowed.</i>
8	Stop Ring 0 = Normal Operation. 1 = Parser is turned off. Software must set this bit to force the Ring and Command Parser to Idle. Software must read a “1” in Ring Idle bit after setting this bit to ensure that the hardware is idle. <i>Software must clear this bit for Ring to resume normal operation.</i>
7:2	Reserved Read/Write
1	Dummy Read Disable. Nominally a command stream flush is completed with a dummy read to memory to push all pending writes. Setting this bit to a “1” disables the dummy read.
0	Reserved Read/Write



8.16.1.8 BCS_INSTPM—Instruction Parser Mode Register

Address Offset: 040C0h–040C3h
Default Value: 0000 0000h
Access: Read/Write
Size: 32 bits

The BCS_INSTPM register is used to control the operation of the BCS Instruction Parser. Certain classes of instructions can be disabled (ignored) – often useful for detecting performance bottlenecks. Also, “Synchronizing Flush” operations can be initiated – useful for ensuring the completion (vs. only parsing) of rendering instructions.

Programming Notes:

- All Reserved bits are implemented.

Bit	Description
31:16	Masks: These bits serve as write enables for bits 15:0. If this register is written with any of these bits clear the corresponding bit in the field 15:0 will not be modified. Reading these bits always returns 0s.
15:6	Reserved: MBZ
5	Sync Flush Enable: This field is used to request a Sync Flush operation. The device will automatically clear this bit before completing the operation. See Sync Flush (<i>Programming Environment</i>). Programming Note: • The command parser must be stopped prior to issuing this command by setting the Stop Ring bit in register BCS_MI_MODE. Only after observing Ring Idle set in BCS_MI_MODE can a Sync Flush be issued by setting this bit. Once this bit becomes clear again, indicating flush complete, the command parser is re-enabled by clearing Stop Ring. Format = Enable (cleared by HW)
4:0	Reserved: MBZ



8.16.1.9 BCS_UHPTR — Pending Head Pointer Register

Address Offset: 04134h–04137h
 Default Value: 0000 0000h
 Access: Read/Write
 Size: 32 bits

Bit	Description
31:3	Head Pointer Address: This register represents the GFX address offset where execution should continue in the ring buffer following execution of an MI_ARB_CHECK command. Format = MI_Graphics_Offset
2:1	Reserved: MBZ
0	Head Pointer Valid: 1 = Indicates that there is an updated head pointer programmed in this register 0 = No valid updated head pointer register, resume execution at the current location in the ring buffer This bit is set by the software to request a pre-emption. It is reset by hardware after the head pointer in this register is read. The hardware uses the head pointer programmed in this register at the time the reset is generated.

8.16.1.10 BCS_CNTR—Counter for the Bit Stream Decode Engine

Address Offset: 04178h–0417Bh
 Default Value: FFFF FFFFh
 Access: Read/Write
 Size: 32 bits

Bit	Description
31:0	Count Value: Writing a Zero value to this register starts the counting. Writing a Value of FFFF FFFF to this counter stops the counter

8.16.1.11 BCS_THRSH—Threshold for the Counter of Bit Stream Decode Engine

Address Offset: 0417Ch–0417Fh
 Default Value: 00014500h
 Access: Read/Write
 Size: 32 bits

Bit	Description
31:0	Threshold Value: The value in this register reflects the number of clocks the bit stream decode engine is expected to run. If the value is exceeded the counter is reset and an interrupt may be enabled in the device.



8.16.1.12 BCS_BB_ADDR—Batch Buffer Head Pointer Register

Address Offset: 04140h–04147h
Default Value: 0000 0000 0000 0000h
Access: Read-Only
Size: 64 bits

This register contains the current QWord Graphics Memory Address of the last-initiated batch buffer.

Bit	Description
63:32	Reserved: MBZ
31:3	Batch Buffer Head Pointer: This field specifies the QWord-aligned Graphics Memory Address where the last initiated Batch Buffer is currently fetching commands. If no batch buffer is currently active, the Valid bit will be 0 and this field will be meaningless. .
2:1	Reserved: MBZ
0	Valid: 1 = Batch buffer Valid 0 = Batch buffer Invalid

8.16.1.13 BCS_RCCID—Ring Buffer Current Context ID Register

Address Offset: 04190h–04193h
Default Value: 00 00 00 00h
Access: Read/Write
Size: 32 bits

This register contains the current “ring context ID” associated with the ring buffer.

Programming Notes:

- The current context registers must not be written directly (via MMIO). The RCCID register should only be updated indirectly from RNCID.

Bit	Description
31:12	Logical Ring Context Address (LRCA): This field contains the 4 KB-aligned Memory address of the current Ring Context Descriptor associated with this ring buffer. See the RNCID register for the Descriptor format. Format = GlobalGraphicsVirtualAddress[31:12]
11:1	Reserved: MBZ
0	Valid: 1 = The other field of this register is valid. A ring context is executing and the LRCA field contains the address of its context descriptor. 0 = The other field of this register is invalid. No ring context is executing. This streamer is idle or it is being used in Basic Scheduler mode where the ring buffer registers are manipulated directly and no ring context is used.



8.16.1.14 BCS_RNCID—Ring Buffer Next Context ID Register

Address Offset: 04194h–04197h
Default Value: 00 00 00 00h
Access: Read/Write
Size: 32 bits

This register contains the *next* “ring context ID” associated with the ring buffer.

Programming Notes:

- The current context (RCCID) register can be updated indirectly from this register on a context switch event. Note that this can only be triggered by executing an MI_ARB_CHECK command in the current context or if the current context runs dry (head pointer becomes equal to tail pointer).

Bit	Description
31:12	Logical Ring Context Address (LRCA): This field contains the 4 KB-aligned Memory address of the next Ring Context Descriptor associated with this ring buffer. Format = GlobalGraphicsVirtualAddress[31:12]
11:1	Reserved: MBZ
0	Valid: 1 = The other field of this register is valid. A valid ring context is pointed at by the LRCA field of this register. 0 = The other field of this register is invalid. No next context is available to run should the current one execute MI_ARB_CHECK or run out of instructions. This bit is reset by HW when the current context ends and the “next” context becomes the current one. Once that happens, SW may submit a new “next” context.

8.17 Software Control Bit Definitions

Registers in the range 22XX are not protected from the load register immediate instruction if the command is executed in the non-secure batch buffer.



8.18 Frame Buffer Compression Control ([DevCL] Only)

This section describes the registers associated with the Frame Buffer Compression function. The primary motivation of FBC is power savings and thus it is only applicable to the Mobile Product.

Programming Notes:

- Frame buffer compression has to be disabled (via FBC_CONTROL[31] = 0), and software has to wait until compression not in progress (FBC_STATUS[31] == 0) before changing any of the following fields:
 - FBC_CFB_BASE
 - FBC_LL_BASE
 - FBC_CONTROL[Mode Select]
 - FBC_CONTROL[Compressed Frame Buffer Stride]
 - FBC_CONTROL[Fence Number]

8.18.1 FBC_CFB_BASE — Compressed Frame Buffer Base Address

FBC_CFB_BASE — Compressed Frame Buffer Base Address	
Register Type: MMIO Address Offset: 3200h Project: DevCL Default Value: 0000 0000h Access: R/W Size (in bits): 32	
This register specifies the physical memory address at which the Compressed Frame Buffer is located. Note that the Compressed Frame Buffers must be in Non Cacheable memory and not relocated while FBC is active.	
Bit	Description
31:12	Compressed Frame Buffer Address Project: DevCL Default Value: 0h Address: PhysicalAddress[31:12] This register specifies Bits 31:12 of the physical address of the Compressed Frame Buffer. Programming Notes Software must guarantee that the Compressed Frame Buffer is stored in contiguous physical memory. The buffer must be 4K byte aligned. This field should not be changed unless FBC is inactive (the first VBlank start after Enable Frame Buffer Compression has been cleared.)
11:0	Reserved Project: DevCL Format: MBZ



8.18.2 FBC_LL_BASE — Compressed Frame Line Length Buffer Address

FBC_LL_BASE — Compressed Frame Line Length Buffer Address	
Register Type: MMIO Address Offset: 3204h Project: DevCL Default Value: 0000 0000h Access: R/W Size (in bits): 32	
This register specifies the physical memory address at which the Compressed Frame Line Length Buffer is located. Note that the Compressed Frame Buffers must be in Non Cacheable memory and not relocated while FBC is active.	
Bit	Description
31:12	Compressed Frame Line Length Buffer Address Project: DevCL Default Value: 0h Address: PhysicalAddress[31:12] This register specifies Bits 31:12 of the physical address of the Compressed Frame Line Length Buffer. Programming Notes Software must guarantee that the Compressed Frame Line Length Buffer is stored in contiguous physical memory. The buffer must be 4K byte aligned. This field should not be changed unless FBC is inactive (the first VBlank start after Enable Frame Buffer Compression has been cleared.)
11:0	Reserved Project: DevCL Format: MBZ



8.18.3 FBC_CONTROL — Frame Buffer Compression Control Register

FBC_CONTROL — Frame Buffer Compression Control Register			
Register Type: MMIO Address Offset: 3208h Project: DevCL Default Value: 0000 0000h Access: R/W Size (in bits): 32			
This register is used to control the operation of RLE-FBC.			
Bit	Description		
31	Enable Frame Buffer Compression Project: DevCL Default Value: 0h Format: Enable This bit is used to globally enable or disable the RLE-FBC function (compression and decompression) at the next VBlank start.		
	Value	Name	Project
	0h	Disable	DevCL
	1h	Enable	DevCL
30	Mode Select Project: DevCL Default Value: 0h Format: U1		
	Value	Name	Project
	0h	Single Pass	DevCL
	1h	Periodic Pass	DevCL
29:16	Interval Project: DevCL Default Value: 0h Format: U14 Range [1,16383] This is interval for which the compressor waits between passes. In Periodic Mode this field determines the interval length, in terms of frames (VBlanks). Zero is an illegal value.		



FBC_CONTROL — Frame Buffer Compression Control Register				
15	Stop Compressing on Modification (DEBUG ONLY)	Project:	DevCL	Format: Enable
	If set to '1' the compressor will abort a subsequent compressing pass when any modification to the source frame buffer is detected.			
14	Uncompressible Enable	Project:	DevCL	Format: Enable
	If set to a '1' the compressor marks as "Uncompressible 10" (see the FBC_TAG register) if any scanline in a pair cannot be compressed. In Default mode Uncompressible mode is turned off.			
13	Reserved	Project:	DevCL	Format: MBZ
12:5	Compressed Frame Buffer Stride	Project:	DevCL	Format: (Stride in 64Byte units) – 1
	This is the stride for the compressed frame buffer. This value is used to determine the line-to-line increment for the compressed frame buffer. Lines that cannot be compressed to a stride size or less are not compressed at all.			
	This field must be set to a value less than or equal to the stride of the source (uncompressed) frame buffer.			
	00h = 64B stride			
4	Reserved	Project:	DevCL	Format: MBZ
3:0	Fence Number	Project:	DevCL	Format: U3
	This field specifies the FENCE number corresponding to the placement of the uncompressed frame buffer. (Note that only tiled frame buffers can be compressed). This field is double buffered in hardware. Only the host accesses the uncompressed frame buffer using a fence.			

8.18.4 FBC_COMMAND — Frame Buffer Compression Command Register

FBC_COMMAND — Frame Buffer Compression Command Register				
Register Type: MMIO				
Address Offset: 320Ch				
Project: DevCL				
Default Value: 0000 0000h				
Access: R/W				
Size (in bits): 32				
This register is used to request a frame buffer compression pass while in Single Pass mode.				
Bit	Description			
31:1	Reserved	Project:	DevCL	Format: MBZ
0	Compress Enable	Project:	DevCL	Format: Enable
	Software can set this bit to trigger compression in Single Pass mode. The compressor clears this bit after the compression pass is completed. This bit is ignored in Periodic Mode (i.e., it will not cause a compression pass and will always read as '0').			



8.18.5 FBC_STATUS — Frame Buffer Compression Status Register

FBC_STATUS — Frame Buffer Compression Status Register	
Register Type: MMIO Address Offset: 3210h Project: DevCL Default Value: 2000 0000h Access: RO / R/W Size (in bits): 32	
This register contains status information associated with the RLE-FBC function. The information is read-only in normal operation, though some fields can be programmed as a TEST MODE.	
Bit	Description
31	Compressing Project: DevCL Security: RO Default Value: 0h Format: Flag This status bit indicates that the device is currently within a compression pass.
30	Compressed Project: DevCL Security: RO normally, R/W TEST MODE Default Value: 0h Format: Flag This bit indicates that a compressed frame buffer is available at the address contained in the FB_CFB_BASE register. In normal operation the compressor sets this bit when it has completed the compression pass. During compression this bit is not set. As a test mode this bit can be set if there is a software-created compressed buffer available at the address in the FB_CFB_BASE register. <u>Test-Mode software must check that compression is not in progress before setting this bit.</u> If RLE-FBC is enabled, the compressor will clear this bit when it starts the next recompression pass.



FBC_STATUS — Frame Buffer Compression Status Register	
29	Any Modified Project: DevCL Security: RO normally, R/W TEST MODE Default Value: 1h Format: Flag 1 = (default) Indicates that the frame buffer has been modified since the last compression pass. The compressor sets this bit on the first write to the frame buffer from the application/driver or upon an allocation within the render cache (e.g., as a result of Blt, 3D or MPEG activity). The fence number and frame buffer base address are used to determine if a write modified the frame buffer. The bit is cleared by the compressor at the start of the next compression pass. In normal operation this bit is read only (software must not write this bit) and defaults to a “1”. As a test mode this bit can be set if there is a software-created compressed buffer with modified lines available at the address contained the FB_CFB_BASE register. <u>SW must check that compression is not in progress before setting this bit.</u> If enabled, the compressor will clear this bit when it initiates the next compression pass. This test mode is used for continuous-mode compression testing.
28:11	Reserved Project: DevCL Format: MBZ
10:0	Current Line Compressing Project: DevCL Security: RO Default Value: 0h Format: U11 This read only field indicates the line number that the compressor is currently processing. If this field is 0 and the Compressing bit (Bit 31) is set, the compressor is currently on display frame line 1.



8.18.6 FBC_CONTROL2— Frame Buffer Compression 2nd Control Register

FBC_CONTROL2— Frame Buffer Compression 2 nd Control Register																							
Register Type: MMIO Address Offset: 3214h Project: DevCL Default Value: 0000 0000h Access: R/W Size (in bits): 32																							
This register is used to control the operation of RLE-FBC.																							
Bit	Description																						
31:3	Reserved Project: DevCL Format: MBZ																						
4	Double Buffer FBC Fence and Fence_DisplayY Offset Register Fields Project: DevCL Default Value: 0h Format: Disable <table> <thead> <tr> <th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr> </thead> <tbody> <tr> <td>0h</td><td></td><td>Double buffer</td><td>DevCL</td></tr> <tr> <td>1h</td><td></td><td>Don't double buffer</td><td>DevCL</td></tr> </tbody> </table>			Value	Name	Description	Project	0h		Double buffer	DevCL	1h		Don't double buffer	DevCL								
Value	Name	Description	Project																				
0h		Double buffer	DevCL																				
1h		Don't double buffer	DevCL																				
3:2	FBC C3 Mode Project: DevCL Default Value: 0h Format: U2 <table> <thead> <tr> <th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr> </thead> <tbody> <tr> <td>00</td><td></td><td>FBC IDLENESS is not looked at in order to enter Self Refresh</td><td>DevCL</td></tr> <tr> <td>01</td><td></td><td>FBC IDLENESS is looked at in order to enter Self Refresh</td><td>DevCL</td></tr> <tr> <td>10</td><td></td><td>FBC IDLENESS is looked at in order to enter Self Refresh. But FBC enters IDLE as it finishes compressing the current scanline pair and enters IDLE as soon as csunit asserts the inc3 signal.</td><td>DevCL</td></tr> <tr> <td>11</td><td>Reserved</td><td>Reserved</td><td>DevCL</td></tr> </tbody> </table>			Value	Name	Description	Project	00		FBC IDLENESS is not looked at in order to enter Self Refresh	DevCL	01		FBC IDLENESS is looked at in order to enter Self Refresh	DevCL	10		FBC IDLENESS is looked at in order to enter Self Refresh. But FBC enters IDLE as it finishes compressing the current scanline pair and enters IDLE as soon as csunit asserts the inc3 signal.	DevCL	11	Reserved	Reserved	DevCL
Value	Name	Description	Project																				
00		FBC IDLENESS is not looked at in order to enter Self Refresh	DevCL																				
01		FBC IDLENESS is looked at in order to enter Self Refresh	DevCL																				
10		FBC IDLENESS is looked at in order to enter Self Refresh. But FBC enters IDLE as it finishes compressing the current scanline pair and enters IDLE as soon as csunit asserts the inc3 signal.	DevCL																				
11	Reserved	Reserved	DevCL																				



FBC_CONTROL2— Frame Buffer Compression 2 nd Control Register				
1	CPU Fence enable			
	Project:	DevCL		
	Default Value:	0h		
	Format:	Enable		
	Value	Name	Description	Project
	0h		Display Buffer is not in a CPU fence. No modifications are expected from CPU to the Display Buffer.	DevCL
	1h		Display Buffer exists in a CPU fence.	DevCL
0	Frame Buffer Compression Display Plane Select A/B			
	Project:	DevCL		
	Default Value:	0h		
	Format:	Flag		
	Value	Name	Description	Project
	0h		Enable frame buffer compression on Plane A.	All
	1h		Enable frame buffer compression on Plane B.	All
	Programming Notes			Project
	Before changing this bit s/w needs to make sure that FBC is disabled and the "COMPRESSING" bit in the FBC_CONTROL register comes to a "0".			DevCL

8.18.7 FBC_DISPYOFF — FBC Fence Display Buffer Y Offset

FBC_DISPYOFF — FBC Fence Display Buffer Y Offset				
Register Type: MMIO				
Address Offset: 321Bh				
Project: DevCL				
Default Value: 0000 0000h				
Access: R/W				
Size (in bits): 32				
Bit	Description			
31:12	Reserved	Project:	DevCL	Format: MBZ
11:0	Fence_YDisp	Project:	DevCL	Format: U12
	Y offset from the fence to the Display Buffer base			



8.18.8 FBC_MOD_NUM— FBC Number of Modifications for Recompression

FBC_MOD_NUM— FBC Number of Modifications for Recompression	
Register Type: MMIO Address Offset: 3220h Project: DevCL Default Value: 0000 0000h Access: R/W Size (in bits): 32	
The purpose of this register is to avoid SR exit unless the programmed number of modifications have been made to the Display buffer.	
Bit	Description
31:1	FBC_Mod_Num Project: DevCL Format: U12 Number of modifications to the display buffer required before recompression is attempted. If the number of modifications to the Frame Buffer is not equal to the programmed count value at the end of the interval, re-compression is not attempted.
0	FBC_Mod_Num_Valid Project: DevCL Format: Flag Only if this bit is set will the above count value be looked at.



8.18.9 FBC_TAG — Frame Buffer Compression TAG Interface (DEBUG)

FBC_TAG — Frame Buffer Compression TAG Interface (DEBUG)		
Register Type: MMIO Address Offset: 3300h Project: All Default Value: 00000000h; Access: R/W Size (in bits): 49x32		
The device implements 49 DWords of Tag data for RLE-FBC compression. Each DWord contains storage for a 2-bit Tag value associated with a frame buffer line pair. 49 DWords are required to support the required 1536 display lines (= 48 x 32), as an extra DWord may be required due to the alignment of the source (uncompressed) frame buffer. I.e., if the source frame buffer starts on an odd tile line, line 0 corresponds to bit 1 of 3300 (bit 0 is unused) and the 49th DWord may be required. If the source frame buffer starts on an even tile line, line 0 corresponds to bit 0 of 3300.		
DWord	Bit	Description
0..48	31:30	Tag for lines 30&31 Project: All Format: FBC Tag For lines: (DWord) + 30 and (DWord) + 31
	29:28	Tag for lines 29&28 Project: All Format: FBC Tag For lines: (DWord) + 30 and (DWord) + 31
	27:26	Tag for lines 27&26 Project: All Format: FBC Tag For lines: (DWord) + 30 and (DWord) + 31
	25:24	Tag for lines 25&24 Project: All Format: FBC Tag For lines: (DWord) + 30 and (DWord) + 31
	23:22	Tag for lines 23&22 Project: All Format: FBC Tag For lines: (DWord) + 30 and (DWord) + 31
	21:20	Tag for lines 21&20 Project: All Format: FBC Tag For lines: (DWord) + 30 and (DWord) + 31
	19:18	Tag for lines 19&18 Project: All Format: FBC Tag For lines: (DWord) + 30 and (DWord) + 31
	17:16	Tag for lines 17&16 Project: All Format: FBC Tag For lines: (DWord) + 30 and (DWord) + 31



FBC_TAG — Frame Buffer Compression TAG Interface (DEBUG)																							
15:14	Tag for lines 15&14 Project: All Format: FBC Tag For lines: (DWord) + 30 and (DWord) + 31																						
13:12	Tag for lines 13&12 Project: All Format: FBC Tag For lines: (DWord) + 30 and (DWord) + 31																						
11:10	Tag for lines 11&10 Project: All Format: FBC Tag For lines: (DWord) + 30 and (DWord) + 31																						
9:8	Tag for lines 9&8 Project: All Format: FBC Tag For lines: (DWord) + 30 and (DWord) + 31																						
7:6	Tag for lines 7&6 Project: All Format: FBC Tag For lines: (DWord) + 30 and (DWord) + 31																						
5:4	Tag for lines 5&4 Project: All Format: FBC Tag For lines: (DWord) + 30 and (DWord) + 31																						
3:2	Tag for lines 3&2 Project: All Format: FBC Tag For lines: (DWord) + 30 and (DWord) + 31																						
1:0	Tag for lines 1&0 Project: All Format: FBC Tag For lines: (DWord) + 30 and (DWord) + 31																						
31:0	Tag for lines DW# + 1&0 Project: All Format: FBC Tag See below For lines: (DWord) + 30 and (DWord) + 31 <table> <thead> <tr> <th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr> </thead> <tbody> <tr> <td>00</td><td>Modified</td><td>At least one of the associated lines was modified since the last compression pass started.</td><td>All</td></tr> <tr> <td>01</td><td>Uncompressed</td><td>The associated lines are uncompressed and are candidate for compression in the next pass</td><td>All</td></tr> <tr> <td>10</td><td>Uncompressible</td><td>The associated lines are uncompressible and are not candidate for compression in the next pass.</td><td>All</td></tr> <tr> <td>11</td><td>Compressed</td><td>The associated lines are compressed</td><td>All</td></tr> </tbody> </table>	Value	Name	Description	Project	00	Modified	At least one of the associated lines was modified since the last compression pass started.	All	01	Uncompressed	The associated lines are uncompressed and are candidate for compression in the next pass	All	10	Uncompressible	The associated lines are uncompressible and are not candidate for compression in the next pass.	All	11	Compressed	The associated lines are compressed	All		
Value	Name	Description	Project																				
00	Modified	At least one of the associated lines was modified since the last compression pass started.	All																				
01	Uncompressed	The associated lines are uncompressed and are candidate for compression in the next pass	All																				
10	Uncompressible	The associated lines are uncompressible and are not candidate for compression in the next pass.	All																				
11	Compressed	The associated lines are compressed	All																				



8.19 Fence Registers

8.19.1 FENCE — Graphics Memory Fence Table Registers

FENCE — Graphics Memory Fence Table Registers	
Register Type:	MMIO
Address Offset:	3000h
Project:	All
Default Value:	00000000h;
Access:	R/W
Size (in bits):	16x64
Address Offset:	03000h – 03007h: FENCE_0
:	
:	
	0307Ch – 0307Fh: FENCE_15
The graphics device performs address translation from linear space to tiled space for a CPU access to graphics memory (See <i>Memory Interface Functions</i> chapter for information on these memory layouts) using the fence registers. Note that the fence registers are used only for CPU accesses to gfx memory. Graphics rendering/display pipelines use Per Surface Tiling (PST) parameters (found in SURFACE_STATE – see the <i>Sampling Engine</i> chapter) to access tiled gfx memory. The intent of tiling is to locate graphics data that are close (in X and Y surface axes) in one physical memory page while still locating some amount of line oriented data sequentially in memory for display efficiency. All 3D rendering is done such that the QWords of any one span are all located in the same memory page, improving rendering performance. Applications view surfaces as linear, hence when the cpu access a surface that is tiled, the gfx hardware must perform linear to tiled address conversion and access the correct physical memory location(s) to get the data. Tiled memory is supported for rendering and display surfaces located in graphics memory. A tiled memory surface is a surface that has a width and height that are subsets of the tiled region's pitch and height. The device maintains the constants required by the memory interface to perform the address translations. Each tiled region can have a different pitch and size. The CPU-memory interface needs the surface pitch and tile height to perform the address translation. It uses the GMAddr (PCI-BAR) offset address to compare with the fence start and end address, to determine if the rendering surface is tiled. The tiled address is generated based on the tile orientation determined from the matching fence register. Fence ranges are at least 4 KB aligned. Note that the fence registers are used <u>only for CPU accesses</u> to graphics memory. A Tile represents 4 KB of memory. Tile height is 8 rows for X major tiles and 32 rows for Y major tiles. Tile Pitch is 512Bs for X major tiles and 128Bs for Y major tiles. The surface pitch is programmed in 128B units such that the pitch is an integer multiple of "tile pitch". Engine restrictions on tile surface usage are detailed in Surface Placement Restrictions (Memory Interface Functions). Note that X major tiles can be used for Sampler, Color, Depth, motion compensation references and motion compensation destination, Display, Overlay, GDI Blt source and destination surfaces. Y major tiles can be used for Sampler, depth, color and motion compensation assuming they do not need to be displayed. GDI Blit operations, overlay and display cannot used Tiled Y orientations. A "PST" graphics surface that will also be accessed via fence needs its base address to be tile row aligned. Hardware handles the flushing of any pending cycles when software changes the fence upper/lower bounds. Fence Table Registers occupy the address range specified above. Each Fence Table Register has the following format. FENCE registers are <i>not</i> reset by a <u>graphics</u> reset. They will maintain their values unless a full chipset reset is performed.	



FENCE — Graphics Memory Fence Table Registers																
DWord	Bit	Description														
0..15	63:44	Fence Upper Bound Project: All Address: GraphicsAddress[31:12] Bits 31:12 of the ending Graphics Address of the fence region. Fence regions must be aligned to a 4KB page. This address represents the last 4KB page of the fence region (Upper Bound is included in the fence region). Graphics Address is the offset within GMADR space.														
	45:32	Reserved	Project: All	Format: MBZ												
	31:12	Fence Lower Bound Project: All Address: GraphicsAddress[31:12] Bits 31:12 of the starting Graphics Address of the fence region. Fence regions must be aligned to 4KB. This address represents the first 4KB page of the fence region (Lowe Bound is included in the fence region). Graphics Address is the offset within GMADR space.														
	11:2	Fence Pitch Project: All Default Value: 0h DefaultVaueDesc Format: U10-1 Width in 128 bytes This field specifies the width (pitch) of the fence region in multiple of “tile width”. For Tile X this field must be programmed to a multiple of 512B (“003” is the minimum value) and for Tile Y this field must be programmed to a multiple of 128B (“000” is the minimum value). 000h = 128B 001h = 256B ... 3FFh = 128KB														
	1	Tile Walk Project: All Format: MI_TileWalk This field specifies the spatial ordering of QWords within tiles.														
<table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td>MI_TILE_XMAJOR</td><td>Consecutive SWords (32 Bytes) sequenced in the X direction</td><td>All</td></tr><tr><td>1h</td><td>MI_TILE_YMAJOR</td><td>Consecutive OWords (16 Bytes) sequenced in the Y direction</td><td>All</td></tr></table>					Value	Name	Description	Project	0h	MI_TILE_XMAJOR	Consecutive SWords (32 Bytes) sequenced in the X direction	All	1h	MI_TILE_YMAJOR	Consecutive OWords (16 Bytes) sequenced in the Y direction	All
Value	Name	Description	Project													
0h	MI_TILE_XMAJOR	Consecutive SWords (32 Bytes) sequenced in the X direction	All													
1h	MI_TILE_YMAJOR	Consecutive OWords (16 Bytes) sequenced in the Y direction	All													



FENCE — Graphics Memory Fence Table Registers				
	0	Fence Valid		
		Project:	All	
		Format:	MI_ FenceValid	
		This field specifies whether or not this fence register defines a fence region.		
	Value	Name	Description	Project
	0h	MI_FENCE_INVALID		All
1h	MI_FENCE_VALID		All	

8.20 GFX MMIO – MCHBAR Aperture

Address Offset: 10000h – 13FFFh
 Default Value: Same as MCHBAR
 Access: Aligned Word, Dword or Qword Read/Write

This range defined in the graphics MMIO range is an alias with which graphics driver can read and write registers defined in the MCHBAR MMIO space claimed thru Device #0. Attributes for registers defined within the MCHBAR space are preserved when the same registers are accessed via this space. Registers that the graphics driver requires access to are Rank Throttling, GMCH Throttling, Thermal Sensor etc. Product specific EDS has the details of MCHBAR register set.

The Alias functions works for MMIO access from the CPU only. A command stream load register immediate will drop the data and store register immediate will return all Zeros.

Graphics MMIO registers can be accessed thru MMIO BARs in function #0 and function #1 in Device #2. The aliasing mechanism is turned off if memory access to the corresponding function is turned off via software or in certain power states.

§§





9 Memory Interface Commands for Rendering Engine

9.1 Introduction

This chapter describes the formats of the “Memory Interface” commands, including brief descriptions of their use. The functions performed by these commands are discussed fully in the *Memory Interface Functions* Device Programming Environment chapter.

This chapter describes MI Commands for the original graphics processing engine. The term “for Rendering Engine” in the title has been added to differentiate this chapter from a similar one describing the MI commands for the Media Decode Engine.

The commands detailed in this chapter are used across products within the Gen4 family. However, slight changes may be present in some commands (i.e., for features added or removed), or some commands may be removed entirely. Refer to the *Preface* chapter for product specific summary.

9.2 MI_ARB_CHECK

MI_ARB_CHECK		
Project:	All	Length Bias: 1
The MI_ARB_CHECK instruction is used to check the ring buffer double buffered head pointer (register UHPTR). This instruction can be used to pre-empt the current execution of the ring buffer. Note that the valid bit in the updated head pointer register needs to be set for the command streamer to be pre-empted. Programming Note: • The current head pointer is loaded with the updated head pointer register independent of the location of the updated head • If the current head pointer and the updated head pointer register are equal, hardware will automatically reset the valid bit corresponding to the UHPTR • This instruction can be placed only in a ring buffer, never in a batch buffer. • For pre-emption, the wrap count in the ring buffer head register is no longer maintained by hardware. The hardware updates the wrap count to the value in the UHPTR register.		
DWord	Bit	Description
0	31:29	Command Type Default Value: 0h MI_COMMAND Format: OpCode
	28:23	MI Command Opcode Default Value: 05h MI_ARB_CHECK Format: OpCode
	22:0	Reserved Project: All Format: MBZ



9.3 MI_BATCH_BUFFER_END

MI_BATCH_BUFFER_END			
Project: All		Length Bias:	1
The MI_BATCH_BUFFER_END command is used to terminate the execution of commands stored in a <i>batch buffer</i> initiated using a MI_BATCH_BUFFER_START command.			
DWord	Bit	Description	
0	31:29	Command Type Default Value: 0h MI_COMMAND Format: OpCode	
	28:23	MI Command Opcode Default Value: 0Ah MI_BATCH_BUFFER_END Format: OpCode	
	22:0	Reserved Project: All Format: MBZ	

9.4 MI_BATCH_BUFFER_START

MI_BATCH_BUFFER_START			
Project: All		Length Bias:	2
The MI_BATCH_BUFFER_START command is used to initiate the execution of commands stored in a <i>batch buffer</i> . For restrictions on the location of batch buffers, see Batch Buffers in the Device Programming Interface chapter of <i>MI Functions</i> . The batch buffer can be specified as secure or non-secure, determining the operations considered valid when initiated from within the buffer and any attached (chained) batch buffers. See Batch Buffer Protection in the Device Programming Interface chapter of <i>MI Functions</i> .			
Programming Notes: - Batch buffers referenced with physical addresses must not extend beyond the end of the starting physical page (can't span physical pages). However, a batch buffer initiated using a physical address can chain to another buffer in another physical page. - A batch buffer initiated with this command must end either with a MI_BATCH_BUFFER_END command or by chaining to another batch buffer with an MI_BATCH_BUFFER_START command. - For virtual batch buffers, it is essential that the address location beyond the current page be populated inside the GTT. HW performs over-fetch of the command addresses and any over-fetch requires a valid TLB entry. A single extra page beyond the batch buffer is sufficient.			
DWord	Bit	Description	
0	31:29	Command Type Default Value: 0h MI_COMMAND Format: OpCode	
	28:23	MI Command Opcode Default Value: 31h MI_BATCH_BUFFER_START Format: OpCode	
	22:12	Reserved Project: All Format: MBZ	
	11	Reserved Project: All Format: MBZ	



MI_BATCH_BUFFER_START			
10:9	Command Arbitration Control This field controls where command arbitration can occur during the batch buffer.		
	Value	Name	Description
	0h	Arbitrate only at chain points	Legacy Mode. Overridden by MI_ARB_ON_OFF = Off
	1h	Arbitrate between commands	Arbitration can occur between any pair of commands, or during execution of a primitive command. Overridden by MI_ARB_ON_OFF = Off
2h	Reserved		All
	3h	No Arbitration	The Batch Buffer execution cannot be pre-empted until control returns to the initiating ring. I.e., command arbitration does not occur during or between batch buffer chains. This avoids software from having to place MI_ARB_ON_OFF packets around batch buffers to prevent interruption.
8	Reserved Project: All Format: MBZ Although Buffer Security Indicator is implemented, there is no usage model for it and it need not be validated.		
7	Memory Space Select Project: All Specifies memory space associated with the Buffer Start Address .		
	Value	Name	Description
	0h	Physical Memory	Physical Main (unsnooped) Memory. The 4 bits of the Batch Buffer Start Address Extension are prefixed to bits 31:6 of Buffer Start Address to specify an address within physical main memory. In this mode the hardware must not fetch data beyond a 4KB boundary.
	1h	Graphics Memory	(GTT-mapped) Bits 31:2 of a graphics memory address. The GGTT is used to translate this address.
6	Programming Notes Batch buffers referenced with physical addresses must not extend beyond the end of the starting physical page (can't span physical pages). However, a batch buffer initiated using a physical address can chain to another buffer in another physical page. Batch buffers can chain between (but cannot span) memory spaces.		Project
			All
6	Reserved Project: All Format: MBZ		



MI_BATCH_BUFFER_START		
	5:0	DWord Length Default Value: 0h Excludes DWord (0,1) Format: =n Total - Bias
1	31:6	Batch Buffer Start Address Project: All Address: SelectableAddress(Memory Space Select)[31:6] Surface Type: BatchBuffer This field specifies Bits 31:6 of the starting address of the 64B aligned batch buffer. The address space used depends on <i>Memory Space Select</i> (see above).
	5:4	Reserved Project: All Format: MBZ
	3:0	Batch Buffer Start Address Extension Project: All Address: PhysicalAddressExtension[35:32] This field specifies bits 35:32 of the starting address of the 64B-aligned physical batch buffer. This field must be zero for non-physical Batch Buffers.



9.5 MI_DISPLAY_FLIP

MI_DISPLAY_FLIP		
Project:	All	Length Bias: 2
The MI_DISPLAY_FLIP command is used to request a specific display plane to switch (flip) to display a new buffer. The buffer is specified with a starting address and pitch. The tiled attribute of the buffer start address is programmed as part of the packet. The operation this command performs is also known as a “display flip request” operation – in that the flip operation itself will occur at some point in the future. This command specifies when the flip operation is to occur: either synchronously with vertical retrace to avoid tearing artifacts (possibly on a future frame), or asynchronously (as soon as possible) to minimize rendering stalls at the cost of tearing artifacts. Programming Notes: 1. Prior to a display flip operation being requested, software must ensure that the new display buffer is coherent in memory. This will typically require MI_DISPLAY_FLIP to be included in a PIPE_CONTROL command to flush pending rendering operations and any pending write buffers/caches, although the use of an MI_FLUSH command will also suffice albeit with greater performance penalty. (Note that completion of the MI_FLUSH command does not guarantee that previous outstanding flip operations have completed). 2. This command simply requests a display flip operation -- command execution then continues normally. There is no guarantee that the flip (even if asynchronous) will occur prior to subsequent commands being executed. (Note that completion of the MI_FLUSH command does not guarantee that outstanding flip operations have completed). The MI_WAIT_FOR_EVENT command can be used to provide this synchronization – by pausing command execution until a pending flip has actually completed. This synchronization can also be performed by use of the Display Flip Pending hardware status. See Display Flip Synchronization in the Device Programming Interface chapter of <i>MI Functions</i>. 3. After a display flip operation is requested, software is responsible for initiating any required synchronization with subsequent buffer clear or rendering operations. For multi-buffering (e.g., double buffering) operations, this will typically require updating SURFACE_STATE or the binding table to change the rendering (back) buffer. In addition, prior to any subsequent clear or rendering operations, software must typically ensure that the new rendering buffer is not actively being displayed. Again, the MI_WAIT_FOR_EVENT command or Display Flip Pending hardware status can be used to provide this synchronization. See Display Flip Synchronization in the Device Programming Interface chapter of <i>MI Functions</i>. 4. The display buffer command uses the X and Y offset for the tiled buffers from the Display Interface registers. Software is allowed to change the offset via the MMIO interface irrespective of the flip commands enqueued in the command stream. For tiled buffers, the display subsystem uses the X and Y offset in generation of the final request to memory. The offset is always updated on the next vblank for both Synchronous and Asynch Flips. It is not necessary to have a flip enqueued to update the X and Y offset 5. The display buffer command uses the linear dword offset for the linear buffers from the Display Interface registers. Software is allowed to change the offset via the MMIO interface irrespective of the flip commands enqueued in the command stream. For linear buffers, the display subsystem uses the dword offset in generation of the final request to memory. • For synchronous flips the offset is updated on the next vblank. It is not necessary to have a sync flip enqueued to update the dword offset. • Linear memory does not support asynchronous flips		



MI_DISPLAY_FLIP																		
6. DWord 3 (panel fitter flip) must not be sent with asynchronous flips. It is only allowed to be sent with synchronous flips.																		
DWord	Bit	Description																
0	31:29	Command Type Default Value: 0h MI_COMMAND Format: OpCode																
	28:23	MI Command Opcode Default Value: 14h MI_DISPLAY_FLIP Format: OpCode																
	22	Asynchronous Flip Project: All Format: Boolean This field specifies whether the flip operation should be performed asynchronously to vertical retrace. If FALSE, the flip will occur during the vertical blanking interval – thus avoiding any tearing artifacts. If TRUE, the flip will occur “as soon as possible” – and may exhibit tearing artifacts <table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td>Asynchronous Flip</td><td></td><td>All</td></tr><tr><td>1h</td><td>Synchronous Flip</td><td></td><td>All</td></tr></table> Programming Notes <table><tr><th></th><th>Project</th></tr><tr><td> - This command must not be used to perform an Asynchronous Flip to the <u>same address</u> as specified in the previously executed Asynchronous Flip, or the device operation is UNDEFINED. - The Display Buffer Pitch and Tile parameter fields are ignored for asynchronous flips (i.e., the new buffer must have the same pitch/tile format as the previous buffer). Supported on X-Tiled Frame buffers only. - For Asynch Flips the Buffers used must be 32KB aligned. - The display stride must be >=8KB when doing Asynch Flips together with 180 display rotation. - The display stride must be power of 2 when doing Asynch Flips. - Supported on Display Planes A and B only - Not supported via the flip queue (if this bit is set, Flip Queue Select must be 0) </td><td>All</td></tr></table>		Value	Name	Description	Project	0h	Asynchronous Flip		All	1h	Synchronous Flip		All		Project	- This command must not be used to perform an Asynchronous Flip to the <u>same address</u> as specified in the previously executed Asynchronous Flip, or the device operation is UNDEFINED. - The Display Buffer Pitch and Tile parameter fields are ignored for asynchronous flips (i.e., the new buffer must have the same pitch/tile format as the previous buffer). Supported on X-Tiled Frame buffers only. - For Asynch Flips the Buffers used must be 32KB aligned. - The display stride must be >=8KB when doing Asynch Flips together with 180 display rotation. - The display stride must be power of 2 when doing Asynch Flips. - Supported on Display Planes A and B only - Not supported via the flip queue (if this bit is set, Flip Queue Select must be 0)
Value	Name	Description	Project															
0h	Asynchronous Flip		All															
1h	Synchronous Flip		All															
	Project																	
- This command must not be used to perform an Asynchronous Flip to the <u>same address</u> as specified in the previously executed Asynchronous Flip, or the device operation is UNDEFINED. - The Display Buffer Pitch and Tile parameter fields are ignored for asynchronous flips (i.e., the new buffer must have the same pitch/tile format as the previous buffer). Supported on X-Tiled Frame buffers only. - For Asynch Flips the Buffers used must be 32KB aligned. - The display stride must be >=8KB when doing Asynch Flips together with 180 display rotation. - The display stride must be power of 2 when doing Asynch Flips. - Supported on Display Planes A and B only - Not supported via the flip queue (if this bit is set, Flip Queue Select must be 0)	All																	



MI_DISPLAY_FLIP																																
	21:20	Display (Plane) Select Project: All Format: U2 This field selects which display plane is to perform the flip operation. <table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td>Display Plane A</td><td></td><td>All</td></tr><tr><td>1h</td><td>Display Plane A</td><td></td><td>All</td></tr><tr><td>2h</td><td>Display Plane C</td><td></td><td>All</td></tr><tr><td>3h</td><td>Display Sprite A</td><td></td><td>Reserved</td></tr><tr><td>3h</td><td>Reserved</td><td></td><td>All</td></tr><tr><td>3h</td><td>Display Sprite B</td><td></td><td>Reserved</td></tr></table>			Value	Name	Description	Project	0h	Display Plane A		All	1h	Display Plane A		All	2h	Display Plane C		All	3h	Display Sprite A		Reserved	3h	Reserved		All	3h	Display Sprite B		Reserved
	Value	Name	Description	Project																												
	0h	Display Plane A		All																												
	1h	Display Plane A		All																												
2h	Display Plane C		All																													
3h	Display Sprite A		Reserved																													
3h	Reserved		All																													
3h	Display Sprite B		Reserved																													
19:6	Reserved	Project: All	Format: MBZ																													
5:0	DWord Length Default Value: 0h Excludes DWord (0,1) Format: =n Total Length - 2																															
1																																
	31:30	Reserved	Project: All	Format: MBZ																												
	29	Flip Queue Select Project: All This field selects whether this flip is placed in the flip queue or is a standard (legacy) flip request. <table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td>Standard Flip</td><td>Use standard (legacy) synchronous or asynchronous flipping</td><td>All</td></tr><tr><td>1h</td><td>Enqueue Flip</td><td>Enqueue Flip (see <i>Display Functions</i> for a description of the Flip Queue)</td><td>All</td></tr></table> Programming Notes Performing a legacy synchronous or asynchronous flip will drop any outstanding flips in the flip queue as well as any previous synchronous flip that has not yet completed.			Value	Name	Description	Project	0h	Standard Flip	Use standard (legacy) synchronous or asynchronous flipping	All	1h	Enqueue Flip	Enqueue Flip (see <i>Display Functions</i> for a description of the Flip Queue)	All	Project All															
	Value	Name	Description	Project																												
0h	Standard Flip	Use standard (legacy) synchronous or asynchronous flipping	All																													
1h	Enqueue Flip	Enqueue Flip (see <i>Display Functions</i> for a description of the Flip Queue)	All																													
28:15	Reserved	Project: All	Format: MBZ																													



MI_DISPLAY_FLIP															
	14:3	Display Buffer Pitch Project: All Default Value: 0h DefaultVaueDesc Format: U12 Quad Words For Synchronous or Queued Flips only, this field specifies the QWord pitch of the new display buffer. For Asynchronous Flips, this parameter is ignored. All the flips in a flip chain should maintain the same pitch as programmed with the last synchronous flip or direct thru mmio.													
	2:0	Reserved Project: All Format: MBZ													
2	31:12	Display Buffer Base Address Project: All Address: GraphicsAddress[31:12] This field specifies Bits 31:12 of the Graphics Address of the new display buffer. The display buffer must be pixel aligned within the Graphics Address space. (Refer to the Display Address Start Address Register description in the <i>Display Registers</i> chapter). Programming Notes - The Display buffer must reside completely in Main Memory - This address is always translated via the <i>global</i> (rather than per-process) GTT													
	11:1	Reserved Project: All Format: MBZ													
	0	Tile Parameter Project: All Default Value: 0h DefaultVaueDesc Address: GraphicsAddress[31:0] For Asynchronous Flips, this parameter is ignored. All the flips in a flip chain should maintain the same tile parameter as programmed with the last synchronous flip or direct thru mmio. For Synchronous Flips, tile parameter can change for different flips in the flip chain <table><thead><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr></thead><tbody><tr><td>0h</td><td>Linear</td><td>For Synchronous Flips Only</td><td>All</td></tr><tr><td>1h</td><td>Tiled X</td><td></td><td>All</td></tr></tbody></table>			Value	Name	Description	Project	0h	Linear	For Synchronous Flips Only	All	1h	Tiled X	
Value	Name	Description	Project												
0h	Linear	For Synchronous Flips Only	All												
1h	Tiled X		All												
3	31	Enable Panel Fitter Project: All Format: Enable Enables the panel fitter on the pipe attached to the plane selected for this flip.													



MI_DISPLAY_FLIP				
	30	Panel Fitter Select Project: All		
		Value	Name	Description
				Project
	0h	7x5	Select 7x5 capable panel fitter	All
	1h	3x3	Select 3x3 capable panel fitter	All
29:28	Reserved	Project: All	Format: MBZ	
27:16	Pipe Horizontal Source Image Size Project All Format: U32 :			
	This 12-bit field specifies Horizontal source image size up to 4096. This determines the size of the image created by the display planes sent to the blender. The value programmed should be the source image size minus one.			
	This field obeys all the rules of the Horizontal Source Image Size registers.			
	The pipe affected will be the pipe attached to the plane selected for this flip.			
15:12	Reserved	Project: All	Format: MBZ	
11:0	Pipe Vertical Source Image ReSize Project All Format: U32 :			
	This 12-bit field specifies the new vertical source image size up to 4096 lines. This determines the size of the image created by the display planes sent to the blender. The value programmed should be the source image size minus one.			
	This field obeys all the rules of the Vertical Source Image Size registers.			
	The pipe affected will be the pipe attached to the plane selected for this flip.			



9.6 MI_FLUSH

MI_FLUSH															
Project:	All	Length Bias:	1												
The MI_FLUSH command is used to perform an internal “flush” operation. The parser pauses on an internal flush until all drawing engines have completed any pending operations and the read caches are invalidated including the texture cache accessed via the Sampler or the data port. In addition, this command can also be used to: 1. Flush any dirty data in the Render Cache to memory. This is done by default, however this can be inhibited. 2. Invalidate the state and command cache. Usage note: After this command is completed and followed by a Store DWord-type command, CPU access to graphics memory will be coherent (assuming the Render Cache flush is not inhibited).															
DWord	Bit	Description													
0	31:29	Command Type Default Value: 0h MI_COMMAND Format: OpCode													
	28:23	MI Command Opcode Default Value: 04h MI_FLUSH Format: OpCode													
	22:6	Reserved Project: All Format: MBZ													
	5:4	Reserved Project: All Format: MBZ													
	3	Global Snapshot Count Reset Project: All Format: Boolean If set, the snapshot registers defined for the Gen4 debug capability are reset after the flush completes. The Statistics Counters are also reset; SW should never set this bit during normal operation since the Statistics Counters are intended to be free running.													
		Programming Notes Project PS_DEPTH_COUNT and TIMESTAMP are <i>not</i> reset by MI_FLUSH with this bit set. TIMESTAMP and PS_DEPTH_COUNT can be reset by writing 0 to them All													
		<table> <tr> <th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr> <tr> <td>0h</td><td>Don't Reset</td><td>Do not reset the snapshot counts or Statistics Counters.</td><td>All</td></tr> <tr> <td>1h</td><td>Reset</td><td>Reset the snapshot count in Gen4 for all the units and reset the Statistics Counters except as noted above.</td><td>All</td></tr> </table>	Value	Name	Description	Project	0h	Don't Reset	Do not reset the snapshot counts or Statistics Counters.	All	1h	Reset	Reset the snapshot count in Gen4 for all the units and reset the Statistics Counters except as noted above.	All	
Value	Name	Description	Project												
0h	Don't Reset	Do not reset the snapshot counts or Statistics Counters.	All												
1h	Reset	Reset the snapshot count in Gen4 for all the units and reset the Statistics Counters except as noted above.	All												
	2	Render Cache Flush Inhibit Project All Format: Boolean : If set, the Render Cache is not flushed as part of the processing of this command.													
		<table> <tr> <th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr> <tr> <td>0h</td><td>Flush</td><td>Flush the Render Cache</td><td>All</td></tr> <tr> <td>1h</td><td>Don't Flush</td><td>Do not flush the Render Cache</td><td>All</td></tr> </table>	Value	Name	Description	Project	0h	Flush	Flush the Render Cache	All	1h	Don't Flush	Do not flush the Render Cache	All	
Value	Name	Description	Project												
0h	Flush	Flush the Render Cache	All												
1h	Don't Flush	Do not flush the Render Cache	All												



MI_FLUSH				
	1	State / Instruction Cache Invalidate Project All Format: Boolean		
		If set, Invalidates the State and Instruction Cache		
		Value	Name	Description
		0h	Don't Invalidate	Leave State/Instruction Cache unaffected
		1h	Invalidate	Invalidate State/Instruction Cache
	0	Reserved	Project: All	Format: MBZ

9.7 MI_LOAD_REGISTER_IMM

MI_LOAD_REGISTER_IMM				
Project: All		Length Bias: 2		
The MI_LOAD_REGISTER_IMM command requests a write of up to a DWord constant supplied in the command to the specified Register Offset (i.e., offset into Memory-Mapped Register Range). The register is loaded before the next command is executed. Programming Notes: The behavior of this command is controlled by Dword 3, Bit 8 (Disable Register Access) of the RINGBUF register. If this command is disallowed then the command stream converts it to a NOOP. If this command is executed from a BB then the behavior of this command is controlled by Dword 0, Bit 8 (Security Indicator) of the BATCH_BUFFER_START Command. If the batch buffer is insecure then the command stream converts this command to a NOOP. Note that the corresponding ring buffer must allow a register update for this command to execute.				
DWord	Bit	Description		
0	31:29	Command Type Default Value: 0h MI_COMMAND Format: OpCode		
	28:23	MI Command Opcode Default Value: 22h MI_ Format: OpCode		
	22:12	Reserved Project: All Format: MBZ		
	11:8	Byte Write Disables Format: Enable[4] Bit 8 corresponds to Data DWord [7:0] Range Must specify a valid register write operation This field specifies which bytes of the Data DWord are not to be written to the DWord offset specified in <i>Register Offset</i> .		
	7:6	Reserved Project: All Format: MBZ		



MI_LOAD_REGISTER_IMM		
	5:0	DWord Length Default Value: 1h Excludes DWord (0,1) Format: =n Total Length - 2
1	31:2	Register Offset Format: U30 Address: MmioAddress[31:2] This field specifies bits [31:2] of the offset into the Memory Mapped Register Range (i.e., this field specifies a DWord offset).
	1:0	Reserved Project: All Format: MBZ
2	31:0	Data DWord Mask: Bytes Write Disables Format: U32 This field specifies the DWord value to be written to the targeted location.

9.8 MI_LOAD_SCAN_LINES_EXCL

MI_LOAD_SCAN_LINES_EXCL		
Project:	All	Length Bias: 2
The MI_LOAD_SCAN_LINES_EXCL command is used to initialize the Scan Line Window registers for a specific Display Pipe. If the display refresh is outside this window the Display Engine asserts a signal that is used by the command parser to process the WAIT_FOR_EVENT command (i.e., the parser will wait while outside). This command overrides the Scan Line Window defined by any previous MI_LOAD_SCAN_LINES_INCL or MI_LOAD_SCAN_LINES_EXCL commands targeting the specific display pipe. Note: The two scan-line numbers are inclusive. If programmed to the same values, that single line defines the region in question. Always place an even number of MI_LOAD_SCAN_LINES_EXCL/INCL at a time into the ring buffer. If only a single MI_LOAD_SCAN_LINES_EXCL/INCL is desired, just add a second identical MI_LOAD_SCAN_LINES_EXCL/INCL command.		
DWord	Bit	Description
0	31:29	Command Type Default Value: 0h MI_COMMAND Format: OpCode
	28:23	MI Command Opcode Default Value: 13h MI_LOAD_SCAN_LINES_EXCL Format: OpCode
	22	Reserved Project: All Format: MBZ



MI_LOAD_SCAN_LINES_EXCL																
	21:20	Display Pipe Select Project: All Format: U2 This field selects which Display Engine (pipe) this command is targeting. <table><thead><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr></thead><tbody><tr><td>0h</td><td>Display Pipe A</td><td></td><td>All</td></tr><tr><td>1h</td><td>Display Pipe B</td><td></td><td>All</td></tr></tbody></table>			Value	Name	Description	Project	0h	Display Pipe A		All	1h	Display Pipe B		All
	Value	Name	Description	Project												
	0h	Display Pipe A		All												
	1h	Display Pipe B		All												
19:6	Reserved	Project: All	Format: MBZ													
	5:0	DWord Length Default Value: 0h Excludes DWord (0,1) Format: =n Total Length - 2														
1	31:16	Start Scan Line Number Project: All Format: U16 In scan lines, where scan line 0 is the first line of the display frame. Range [0,Display Buffer height in lines-1] This field specifies the starting scan line number of the Scan Line Window.														
	31:16	End Scan Line Number Project: All Format: U16 In scan lines, where scan line 0 is the first line of the display frame. Range [0,Display Buffer height in lines-1] This field specifies the ending scan line number of the Scan Line Window.														



9.9 MI_LOAD_SCAN_LINES_INCL

MI_LOAD_SCAN_LINES_INCL															
Project: All		Length Bias:	2												
The MI_LOAD_SCAN_LINES_INCL command is used to initialize the Scan Line Window registers for a specific Display Engine. If the display refresh is <i>within</i> this window the Display Engine asserts a signal that is used by the command parser to process the WAIT_FOR_EVENT command (i.e., the parser will wait while inside of the window). This command overrides the Scan Line Window defined by any previous MI_LOAD_SCAN_LINES_INCL or MI_LOAD_SCAN_LINES_EXCL commands targeting the specific display. Always place an even number of MI_LOAD_SCAN_LINES_EXCL/INCL at a time into the ring buffer. If only a single MI_LOAD_SCAN_LINES_EXCL/INCL is desired, just add a second identical															
DWord	Bit	Description													
0	31:29	Command Type Default Value: 0h MI_COMMAND Format: OpCode													
	28:23	MI Command Opcode Default Value: 12h MI_LOAD_SCAN_LINES_INCL Format: OpCode													
	22	Reserved Project: All Format: MBZ													
	21:20	Display Pipe Select Project: All Format: U2 This field selects which Display Engine (pipe) this command is targeting. <table><thead><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr></thead><tbody><tr><td>0h</td><td>Display Pipe A</td><td></td><td>All</td></tr><tr><td>1h</td><td>Display Pipe B</td><td></td><td>All</td></tr></tbody></table>		Value	Name	Description	Project	0h	Display Pipe A		All	1h	Display Pipe B		All
	Value	Name	Description	Project											
	0h	Display Pipe A		All											
1h	Display Pipe B		All												
19:6	Reserved Project: All Format: MBZ														
5:0	DWord Length Default Value: 0h Excludes DWord (0,1) Format: =n Total Length - 2														
1	31:16	Start Scan Line Number Project: All Format: U16 In scan lines, where scan line 0 is the first line of the display frame. Range [0,Display Buffer height in lines-1] This field specifies the starting scan line number of the Scan Line Window.													



MI_LOAD_SCAN_LINES_INCL		
	31:16	End Scan Line Number Project: All Format: U16 In scan lines, where scan line 0 is the first line of the display frame. Range [0,Display Buffer height in lines-1] This field specifies the ending scan line number of the Scan Line Window.

9.10 MI_NOOP

MI_NOOP																
Project:		All		Length Bias:	1											
The MI_NOOP command basically performs a “no operation” in the command stream and is typically used to pad the command stream (e.g., in order to pad out a batch buffer to a QWord boundary). However, there is one minor (optional) function this command can perform – a 22-bit value can be loaded into the MI NOPID register. This provides a general-purpose command stream tagging (“breadcrumb”) mechanism (e.g., to provide sequencing information for a subsequent breakpoint interrupt). Performance Note: The process time to execute a NOP command is min of 6 clock cycles. One example usage of the improved NOP throughput is for some multi-pass media application whereas some unwanted media object commands are replaced by MI_NOOP without repacking the commands in a batch buffer.																
DWord	Bit	Description														
0	31:29	Command Type Default Value: 0h MI_COMMAND Format: OpCode														
	28:23	MI Command Opcode Default Value: 0h MI_NOOP Format: OpCode														
	22	Identification Number Register Write Enable Project: All Format: Enable This field enables the value in the Identification Number field to be written into the MI NOPID register. If disabled, that register is unmodified – making this command an effective “no operation” function. <table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td>Disable</td><td>Do not write the NOP_ID register.</td><td>All</td></tr><tr><td>1h</td><td>Enable</td><td>Write the NOP_ID register.</td><td>All</td></tr></table>			Value	Name	Description	Project	0h	Disable	Do not write the NOP_ID register.	All	1h	Enable	Write the NOP_ID register.	All
	Value	Name	Description	Project												
0h	Disable	Do not write the NOP_ID register.	All													
1h	Enable	Write the NOP_ID register.	All													
31:0	Identification Number Project All Format: U22 : This field contains a 22-bit number which can be written to the MI NOPID register.															



9.11 MI_OVERLAY_FLIP

MI_OVERLAY_FLIP			
Project:	All	Length Bias:	2
The MI_OVERLAY_FLIP command is used to specify memory buffers that will (optionally) be used during the next Vertical Blank period to update the specified Overlay control register set and Overlay filter coefficients (respectively). The update of the Overlay registers is referred to as an "Overlay Flip", making this command an "Overlay Flip Request". Programming Notes: 1. Prior to an overlay flip operation being requested, software must ensure that the memory buffer used to update the overlay registers is coherent (i.e., there are no outstanding buffered writes to that memory buffer). 2. Prior to an overlay flip operation being requested, software must ensure that the new overlay buffer is coherent in memory. This will typically require the use of an MI_FLUSH command to flush pending rendering operations and any pending write buffers/caches. 3. This command simply requests an overlay flip operation -- command execution then continues normally. There is no mechanism to prevent a new flip request from overriding any outstanding flip request. (Note that completion of the MI_FLUSH command does not guarantee that outstanding flip operations have completed). The MI_WAIT_FOR_EVENT command can be used to provide this synchronization -- by pausing command execution until a pending overlay flip has actually completed or that the display refresh has proceeded past a specific scan line window. This synchronization can also be performed by use of the Overlay Flip Pending hardware status. See Overlay Flip Synchronization in the Device Programming Interface chapter of <i>MI Functions</i>. 4. After an overlay flip operation is requested, software is responsible for initiating any required synchronization with subsequent buffer clear or rendering operations targeting the previous ("flipped-from") overlay buffer. 5. Registers and Coefficients are located in Main memory.			
DWord	Bit	Description	
0	31:29	Command Type Default Value: 0h MI_COMMAND	Format: OpCode
	28:23	MI Command Opcode Default Value: 11h MI_OVERLAY_FLIP	Format: OpCode



MI_OVERLAY_FLIP				
	22:21	Mode Flags		
		Project: All		
		Format: U2		
		Value	Name	Description
		Project		
00b	Flip Continue	Do not flush or change the state of the Render Cache or Overlay.	All	
01b	Flip On	Flush Render Cache, drawing pipeline and then set render cache in overlay Mode before executing the Flip. The Flip turns on the overlay engine. This Render Cache flush is not applicable in a Mobile Gfx controller which has an independent overlay data buffer.	All	
10b	Flip Off	Flush Render Cache, drawing pipeline and then clear Overlay Mode and turn off the overlay engine. Do not update registers and coefficients from memory. This Render Cache flush is required because overlay shares the render cache in desktop graphics controllers. This bit is generally not applicable in a Mobile graphics controller which has an independent overlay data buffer.	All	
11b	Reserved		All	
20:6	Reserved	Project: All	Format: MBZ	
	5:0	DWord Length		
		Default Value: 0h Excludes DWord (0,1)		
		Format: =n Total Length - 2		
1	31:12	Register and Coefficient Update Address		
		Project: All		
		Address: GlobalGraphicsAddress[31:12]		
		Surface Type: U32		
		This field specifies the memory buffer used to update the overlay registers and Coefficients. The Overlay Update Address Register specifies a Global GTT address used by the Overlay at the next VBLANK event to start requesting overlay control register and Coefficient data from memory. Software should ensure that the Global GTT address is page-aligned , so that the entire overlay control registers and coefficients are within one 4K page.		
11:1	Reserved	Project: All	Format: MBZ	



MI_OVERLAY_FLIP															
	0	Overlay Filter Coefficient Register Update Flag (OFC_UPDATE) Project: All This field indicates if hardware should load overlay filter coefficients from memory. Turning overlay off without loading the Overlay Filter Coefficient registers via MI_OVERLAY_FLIP can lead to a hang. <table><thead><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr></thead><tbody><tr><td>0h</td><td>Don't Update</td><td>Do not update overlay filter coefficients.</td><td>All</td></tr><tr><td>1h</td><td>Update</td><td>Hardware loads the overlay filter coefficients from memory to on-chip registers.</td><td>All</td></tr></tbody></table>		Value	Name	Description	Project	0h	Don't Update	Do not update overlay filter coefficients.	All	1h	Update	Hardware loads the overlay filter coefficients from memory to on-chip registers.	All
Value	Name	Description	Project												
0h	Don't Update	Do not update overlay filter coefficients.	All												
1h	Update	Hardware loads the overlay filter coefficients from memory to on-chip registers.	All												

9.11.1 Turning the Overlay Off

The Overlay Engine is turned off by issuing an MI_OVERLAY_FLIP with the **Mode Flags** set to '10'b (aka "Flip Off"), thereby flushing and reconfiguring the internal caches and putting the Overlay Engine into a low-power state. Software must ensure that the subsequent Overlay Flip has occurred at the next associated VBlank, typically by use of the **Overlay Flip Pending Wait Enable** bit of the MI_WAIT_FOR_EVENT command. In addition, the Display Pipe to which the overlay is attached must continue running until the sequence completes, or device operation is UNDEFINED.

In order to completely shutdown the Overlay Engine, an additional step is required before the use of the "Flip Off" sequence (as described above). The Overlay Enable (OV_ENBL) bit of the Overlay Command (OCOMD) Register must be cleared via a normal Overlay Register load accomplished via issuance of an MI_OVERLAY_FLIP with Mode Flags = '00'b (aka Flip Continue). This operation will effectively turn off the display of the overlay. Note that a wait-for-overlay-VBlank must be used to ensure this Flip Continue has completed. The subsequent Flip-Off sequence (above) will reconfigure the cache for non-overlay operation and gracefully power down the Overlay Engine.

9.11.2 Valid Overlay Flip Sequences

The only architecturally valid Overlay Flip sequence is shown below:

- FlipOn
- some number of FlipContinues
- FlipOff

For example, multiple FlipOn commands (without intervening FlipOff commands) are invalid; multiple FlipOff commands (without intervening FlipOn commands) are invalid; FlipContinue without a preceding FlipOn is invalid.



9.12 MI_REPORT_HEAD

MI_REPORT_HEAD			
Project:	All	Length Bias:	1
The MI_REPORT_HEAD command causes the Head Pointer value of the active ring buffer to be written to a cacheable (snooped) system memory location. The location written is relative to the address programmed in the Hardware Status Page Address Register. Programming Notes: This command must not be executed from a Batch Buffer (Refer to the description of the HSW_PGA register).			
DWord	Bit	Description	
0	31:29	Command Type Default Value: 0h MI_COMMAND Format: OpCode	
	28:23	MI Command Opcode Default Value: 07h MI_REPORT_HEAD Format: OpCode	
	22:0	Reserved Project: All Format: MBZ	

9.13 MI_SET_CONTEXT

MI_SET_CONTEXT			
Project:	All	Length Bias:	2
The MI_SET_CONTEXT command is used to specify the <i>logical</i> context associated with the hardware context. A logical context is an area in memory used to store hardware context information, and the context is referenced via a 2KB-aligned pointer. If the (new) logical context is different (i.e., at a different memory address), the device will proceed to save the current HW context values to the current logical context address, and then restore (load) the new logical context by reading the context from the new address and loading it into the hardware context state. If the logical context address specified in this command matches the current logical context address, this command is effectively treated as a NOP. This command also includes some controls over the context save/restore process. • The Force Restore bit can be used to refresh the on-chip device state from the same memory address if the indirect state buffers have been modified. • The Restore Inhibit bit can be used to prevent the new context from being loaded at all. This must be used to prevent an uninitialized context from being loaded. Once software has initialized a context (by setting all state variables to initial values via commands), the context can then be stored and restored normally. • This command needs to be always followed by a single MI_NOOP instruction to workaround a Gen4 silicon issue.			



MI_SET_CONTEXT														
DWord	Bit	Description												
0	31:29	Command Type Default Value: 0h MI_COMMAND Format: OpCode												
	28:23	MI Command Opcode Default Value: 18h MI_SET_CONTEXT Format: OpCode												
	22:6	Reserved Project: All Format: MBZ												
	5:0	DWord Length Default Value: 0h Excludes DWord (0,1) Format: =n Total Length - 2												
1	31:11	Logical Context Address Project: All Address: PhysicalAddress[31:11] Surface Type: Logical Context This field contains the 2KB-aligned physical address of the Logical Context that is <u>to be loaded</u> into the hardware context. If this address is equal to the CCID register associated with the current ring, no load will occur. Prior to loading this new context, the device will save the existing context as required. After the context switch operation completes, this address will be loaded into the associated CCID register.												
	10	Reserved Project: All Format: MBZ												
	9	HD DVD Context Project: All <table> <tr> <th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr> <tr> <td>0h</td><td>Regular Context</td><td></td><td>All</td></tr> <tr> <td>1h</td><td>HD DVD Context</td><td>Special considerations for TDP allow for higher voltage and frequency.</td><td>All</td></tr> </table>		Value	Name	Description	Project	0h	Regular Context		All	1h	HD DVD Context	Special considerations for TDP allow for higher voltage and frequency.
Value	Name	Description	Project											
0h	Regular Context		All											
1h	HD DVD Context	Special considerations for TDP allow for higher voltage and frequency.	All											



MI_SET_CONTEXT																
	8	Memory Space Select Project: All BitFieldDesc <table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td>Physical Memory</td><td>Physical Main (unsnooped) Memory. The 4 bits of Physical Start Address Extension are prefixed to bits 31:11 to specify a 2KB aligned address within physical main memory. In this mode the hardware must not fetch data beyond a 4KB boundary.</td><td>All</td></tr><tr><td>1h</td><td>Global Graphics Memory</td><td>Global Graphics (GTT-mapped) Memory. Bits 31:11 of a graphics memory address. The GTT whose address is contained in the PGTBL_CTL register is used to translate this address.</td><td>All</td></tr></table>			Value	Name	Description	Project	0h	Physical Memory	Physical Main (unsnooped) Memory. The 4 bits of Physical Start Address Extension are prefixed to bits 31:11 to specify a 2KB aligned address within physical main memory. In this mode the hardware must not fetch data beyond a 4KB boundary.	All	1h	Global Graphics Memory	Global Graphics (GTT-mapped) Memory. Bits 31:11 of a graphics memory address. The GTT whose address is contained in the PGTBL_CTL register is used to translate this address.	All
	Value	Name	Description	Project												
	0h	Physical Memory	Physical Main (unsnooped) Memory. The 4 bits of Physical Start Address Extension are prefixed to bits 31:11 to specify a 2KB aligned address within physical main memory. In this mode the hardware must not fetch data beyond a 4KB boundary.	All												
	1h	Global Graphics Memory	Global Graphics (GTT-mapped) Memory. Bits 31:11 of a graphics memory address. The GTT whose address is contained in the PGTBL_CTL register is used to translate this address.	All												
	7:4	Logical Context Address Extension Project: All Address: PhysicalAddressExtension[35:32] Surface Type: Logical Context This field specified Bits 35:32 of the starting address of the 2KB-aligned physical logical context address. This field must be zero for global gtt context address.														
3	Extended State Save Enable Project: All Format: U32 If set, the extended state identified in the Logical Context Data section of the Memory Data Formats chapter is saved as part of switching <u>away from</u> this logical context. This bit will be stored in the associated CCID register to control the context save operation when switching <u>away from</u> this context (as part of a subsequent MI_SET_CONTEXT command).															
2	Extended State Restore Enable Project: All Format: U32 : If set, the extended state identified in the Logical Context Data section of the Memory Data Formats chapter is loaded (or restored) as part of switching <u>to</u> this logical context. This method can be used to restore things such as filter coefficients using the indirect state restore followed by a restore of the extended logical context data. This bit affects the switch (if required) to the context specified in Logical Context Address . This bit will also be stored in the associated CCID register to control a subsequent context save operation when switching <u>to</u> this context (as part of a subsequent ring buffer switch).															
1	Force Restore Project: All Format: U32 When switching <u>to</u> this logical context a comparison between Logical Context Address and the contents of the CCID register is performed. Normally, matching addresses prevent a context restore from occurring; however, when this bit is set a context restore is forced to occur. This bit cannot be set with Restore Inhibit. Note: This bit is not saved in the associated CCID register. It only affects the processing of this command.															



MI_SET_CONTEXT		
	0	Restore Inhibit Project: All Format: U32 If set, the restore of the HW context from the logical context specified by Logical Context Address is inhibited (i.e., the existing HW context values are maintained). This bit must be used to prevent the loading of an uninitialized logical context. If clear, the context switch proceeds normally. This bit cannot be set with Force Restore. Note: This bit is not saved in the associated CCID register. It only affects the processing of this command.

9.14 MI_STORE_DATA_IMM

MI_STORE_DATA_IMM		
Project:	All	Length Bias: 2
The MI_STORE_DATA_IMM command requests a write of the QWord constant supplied in the packet to the specified Memory Address. As the write targets a System Memory Address, the write operation is coherent with the CPU cache (i.e., the processor cache is snooped). Programming Notes: • This command should not be used within a “non-secure” batch buffer to access per-process virtual space. Doing so will cause the command parser to perform the write with byte enables turned off. This command can be used within ring buffers and/or “secure” batch buffers. • This command can be used for general software synchronization through variables in cacheable memory (i.e., where software does not need to poll un-cached memory or device registers). • This command simply initiates the write operation with command execution proceeding normally. Although the write operation is guaranteed to complete “eventually”, there is no mechanism to synchronize command execution with the completion (or even initiation) of these operations.		
DWord	Bit	Description
0	31:29	Command Type Default Value: 0h MI_COMMAND Format: OpCode
	28:23	MI Command Opcode Default Value: 20h MI_STORE_DATA_IMM Format: OpCode



MI_STORE_DATA_IMM				
	22	Memory Address Type Project: All		
		Value	Name	Description
				Project
	0h	Physical Address		All
	1h	Graphics Address	Hardware will translate this address using the operating GTT. The GTT (global or per-process) used for the translation will be the same GTT used to access the buffer executing this command.	All
	21	BitFieldName Project: All This bit will be ignored and treated as if clear when executing from a non-privileged batch buffer. It is allowed for this bit to be clear when executing this command from a privileged (secure) batch buffer.		
		Value	Name	Description
				Project
	0h	Per Process Graphics Address		All
	1h	Global Graphics Address	This command will use the global GTT to translate the Address and this command must be executing from a privileged (secure) batch buffer.	All
		Programming Notes Notes		
	20:6	Reserved	Project: All	Format: MBZ
	5:0	DWord Length Default Value: 2h Excludes DWord (0,1) = 2 for DWord, 3 for QWord Format: =n Total Length - 2		
	1	31:4	Reserved	Project: All
		3:0	Physical Start Address Extension Project: All Address: PhysicalAddressExtension[35:32] Surface Type: U64 This field specifies bits 35:32 of the physical address where the data will be stored. This field must be zero for a virtual address.	



MI_STORE_DATA_IMM		
2	31:2	Address Project: All Address: SelectableAddress(Memory Address Type) [31:2] Surface Type: U32(2) This field specifies Bits 31:2 of the Address where the DWord will be stored. As the store address must be DWord-aligned, Bits 1:0 of that address MBZ. This address must be 8B aligned for a store "QW" command. Format = U30, Range = valid System Memory Address (not mapped by GTT) if Physical Format = Bits[31:2] of a Graphics Memory Address If Virtual
	1:0	Reserved Project: All Format: MBZ
3	31:0	Data DWord 0 Project: All Format: U32 This field specifies the DWord value to be written to the targeted location. For a QWord write this DWord is the lower DWord of the QWord to be reported (DW 0).
4	31:0	Data DWord 1 Project: All Format: U32 This field specifies the upper DWord value to be written to the targeted QWord location (DW 1).

9.15 MI_STORE_DATA_INDEX

MI_STORE_DATA_INDEX			
Project:	All	Length Bias:	2
- The MI_STORE_DATA_INDEX command requests a write of the data constant supplied in the packet to the specified offset from the System Address defined by the Hardware Status Page Address Register. As the write targets a System Address, the write operation is coherent with the CPU cache (i.e., the processor cache is snooped). - Programming Notes: - Use of this command with an invalid or uninitialized value in the Hardware Status Page Address Register is UNDEFINED. - This command can be used for general software synchronization through variables in cacheable memory (i.e., where software does not need to poll uncached memory or device registers). - This command simply initiates the write operation with command execution proceeding normally. Although the write operation is guaranteed to complete "eventually", there is no mechanism to synchronize command execution with the completion (or even initiation) of these operations.			



MI_STORE_DATA_INDEX		
DWord	Bit	Description
0	31:29	Command Type Default Value: 0h MI_COMMAND Format: OpCode
	28:23	MI Command Opcode Default Value: 21h MI_STORE_DATA_INDEX Format: OpCode
	22:21	Reserved Project: All Format: MBZ
	20:6	Reserved Project: All Format: MBZ
	5:0	DWord Length Default Value: 1h Excludes DWord (0,1) = 1 for DWord, 2 for QWord Format: =n Total Length - 2
1	31:12	Reserved Project: All Format: MBZ
	11:2	Offset Project: All Format: U10 zero-based DWord offset into the HW status page. Address: HardwareStatusPageOffset[11:2] Surface Type: U32 Range [16, 1023] This field specifies the offset (into the hardware status page) to which the data will be written. Note that the first few DWords of this status page are reserved for special-purpose data storage – targeting these reserved locations via this command is UNDEFINED.
	1:0	Reserved Project: All Format: MBZ
2	31:0	Data DWord 0 Project: All Format: U32 This field specifies the DWord value to be written to the targeted location. For a QWord write this DWord is the lower DWord of the QWord to be reported (DW 0).
3	31:0	Data DWord 1 Project: All Format: U32 This field specifies the upper DWord value to be written to the targeted QWord location (DW 1).



9.16 MI_STORE_REGISTER_MEM

MI_STORE_REGISTER_MEM					
Project:		All		Length Bias:	2
The MI_STORE_REGISTER_MEM command requests a register read from a specified memory mapped register location in the device and store of that DWord to memory. The register address is specified along with the command to perform the read.					
Programming Notes:					
• The command temporarily halts command execution. • The memory address for the write is snooped on the host bus. • This command should not be used within a “non-secure” batch buffer to access per-process virtual space. Doing so will cause the command parser to perform the write with byte enables turned off. This command can be used within ring buffers and/or “secure” batch buffers. • This command will cause undefined data to be written to memory if given register addresses for the PGTBL_CTL_0 or FENCE registers					
DWord	Bit	Description			
0	31:29	Command Type Default Value: 0h MI_COMMAND Format: OpCode			
	28:23	MI Command Opcode Default Value: 24h MI_STORE_REGISTER_MEM Format: OpCode			
	22	Reserved Project: DevBW-A,B Format: MBZ			
	22	Memory Address Type Project: All, except DevBW-A,B			
		Value	Name	Description Project	
		0h	Physical Address	All	
	1h	Graphics Address	Hardware will translate this address using the operating GTT. The GTT (global or per-process) used for the translation will be the same GTT used to access the buffer executing this command.	All	
21:6	Reserved Project: All Format: MBZ				
5:0	DWord Length Default Value: 1h Excludes DWord (0,1) Format: =n Total Length - 2				



MI_STORE_REGISTER_MEM		
1	31:28	Physical Start Address Extension Project: All Address: PhysicalAddressExtension[35:32] Surface Type: MMIO Register This field specifies bits 35:32 of the starting address of the physical address.
	27:19	Reserved Project: All Format: MBZ
	18:1	Register Address Project: All Address: MMIO Address[18:2] Surface Type: MMIO Register This field specifies Bits 18:2 of the Register offset the DWord will be read from. As the register address must be DWord-aligned, Bits 1:0 of that address MBZ. Programming Notes Storing a VGA register is not permitted and will store an UNDEFINED value. Project All The values of PGTBL_CTL0 or any of the FENCE registers cannot be stored to memory; UNDEFINED values will be written to memory if the addresses of these registers are specified. Project All
	1	Reserved Project: All Format: MBZ
	0	Reserved Project: All Format: MBZ
2	31:2	Memory Address Project: All Address: SelectableAddress(Memory Address Type)[31:2] Surface Type: MMIO Register This field specifies the address of the memory location where the register value specified in the DWord above will be written. The address specifies the DWord location of the data. If Memory Address Type = 0, Range = Physical_Address [31:2] If Memory Address Type = 1, Range = GraphicsMemoryAddress[31:2]
	1:0	Reserved Project: All Format: MBZ



9.17 MI_USER_INTERRUPT

MI_USER_INTERRUPT			
Project: All		Length Bias:	1
The MI_USER_INTERRUPT command is used to generate a User Interrupt condition. The parser will continue parsing after processing this command. See User Interrupt.			
DWord	Bit	Description	
0	31:29	Command Type Default Value: 0h MI_COMMAND Format: OpCode	
	28:23	MI Command Opcode Default Value: 02h MI_USER_INTERRUPT Format: OpCode	
	22:0	Reserved Project: All Format: MBZ	

9.18 MI_WAIT_FOR_EVENT

MI_WAIT_FOR_EVENT			
Project: All		Length Bias:	1
The MI_WAIT_FOR_EVENT command is used to pause command stream processing until a specific event occurs or while a specific condition exists. See Wait Events/Conditions, Device Programming Interface in <i>MI Functions</i>. Only one event/condition can be specified -- specifying multiple events is UNDEFINED. The effect of the wait operation depends on the source of the command. If executed from a batch buffer, the parser will halt (and suspend command arbitration) until the event/condition occurs. If executed from a ring buffer, further processing of that ring will be suspended, although command arbitration (from other rings) will continue. Note that if a specified condition does not exist (the condition code is inactive) at the time the parser executes this command, the parser proceeds, treating this command as a no-operation. If execution of this command from a primary ring buffer causes a wait to occur, the active ring buffer will <i>effectively</i> give up the remainder of its time slice (required in order to enable arbitration from other primary ring buffers).			
DWord	Bit	Description	
0	31:29	Command Type Default Value: 0h MI_COMMAND Format: OpCode	
	28:23	MI Command Opcode Default Value: 03h MI_WAIT_FOR_EVENT Format: OpCode	
	22:19	Reserved Project: All Format: MBZ	



MI_WAIT_FOR_EVENT																
	18	Display Pipe B Start of V Blank Wait Enable This field enables a wait until the start of next Display Pipe B “Vertical Blank” event occurs. This event is defined as the start of the next Display B Vertical blank period. Note that this can cause a wait for up to a frame. See Start of Vertical Blank Event in the Device Programming Interface chapter of <i>MI Functions</i> . <table><tr><th>Errata</th><th>Description</th><th>Project</th></tr><tr><td>BWT013</td><td>MBZ</td><td>DevBW</td></tr></table>	Errata	Description	Project	BWT013	MBZ	DevBW	Project: All	Format: Enable						
Errata	Description	Project														
BWT013	MBZ	DevBW														
	17	Display Pipe A Start of V Blank Wait Enable This field enables a wait until the start of next Display Pipe A “Vertical Blank” event occurs. This event is defined as the start of the next Display A Vertical blank period. Note that this can cause a wait for up to a frame. See Start of Vertical Blank Event in the Device Programming Interface chapter of <i>MI Functions</i> . <table><tr><th colspan="2">Programming Notes</th><th>Project</th></tr><tr><td colspan="2">Notes</td><td>All</td></tr><tr><th>Errata</th><th>Description</th><th>Project</th></tr><tr><td>BWT013</td><td>MBZ</td><td>DevBW</td></tr></table>	Programming Notes		Project	Notes		All	Errata	Description	Project	BWT013	MBZ	DevBW	Project: All	Format: Enable
Programming Notes		Project														
Notes		All														
Errata	Description	Project														
BWT013	MBZ	DevBW														
	16	Overlay Flip Pending Wait Enable This field enables a wait for the duration of an Overlay “Flip Pending” condition. If a flip request is pending, the parser will wait until the flip operation has completed (i.e., the new overlay address has been loaded into the corresponding overlay registers). See Overlay Flip Pending Condition in the Device Programming Interface chapter of <i>MI Functions</i> .	Project: All	Format: Enable												
	15	Reserved	Project: All	Format: MBZ												
	14	Display Pipe B H Blank Wait Enable This field enables a wait until the start of next Display Pipe B “Horizontal Blank” event occurs. This event is defined as the start of the next Display B Horizontal blank period. Note that this can cause a wait for up to a line. See Horizontal Blank Event in the Device Programming Interface chapter of <i>MI Functions</i> .	Project: All	Format: Enable												
	13	Display Pipe A H Blank Wait Enable This field enables a wait until the start of next Display Pipe A “Horizontal Blank” event occurs. This event is defined as the start of the next Display A Horizontal blank period. Note that this can cause a wait for up to a line. See Horizontal Blank Event in the Device Programming Interface chapter of <i>MI Functions</i> .	Project: All	Format: Enable												



MI_WAIT_FOR_EVENT				
12:9	Condition Code Wait Select			
	Project:		All	
	This field enables a wait for the duration that the corresponding condition code is active. These enable select one of 15 condition codes in the EXCC register, that cause the parser to wait until that condition-code in the EXCC is cleared.			
	Value	Name	Description	Project
	0h	Not Enabled	Condition Code Wait not enabled	All
1h-5h	Enabled	Condition Code select enabled; selects one of 5 codes, 0 – 4	All	
	6h-15h	Reserved	All	
	Programming Notes			
Note that not all condition codes are implemented. The parser operation is UNDEFINED if an unimplemented condition code is selected by this field. The description of the EXCC register (<i>Memory Interface Registers</i>) lists the codes that are implemented.			All	
8	Display Plane C Flip Pending Wait Enable	Project:	All	Format: Enable
This field enables a wait for the duration of a Display Plane C “Flip Pending” condition. If a flip request is pending, the parser will wait until the flip operation has completed (i.e., the new front buffer address has now been loaded into the active front buffer registers). See Display Flip Pending Condition in the Device Programming Interface chapter of <i>MI Functions</i> .				
7	Display Pipe B Vertical Blank Wait Enable	Project:	All	Format: Enable
This field enables a wait until the next Display Pipe B “Vertical Blank” event occurs. This event is defined as the start of the next Display Pipe B vertical blank period. Note that this can cause a wait for up to an entire refresh period. See Vertical Blank Event (See <i>Programming Interface</i>).				
Programming Notes				Project
Prior to using the MI_WAIT_FOR_EVENT command to wait on Display Pipe A/B VBlank events, the corresponding Vertical Blank Interrupt Enable (bit 17) of the corresponding PIPEASTAT (70024h) or PIPEBSTAT (71024h) register must be set. Note that this does not require an actual VBlank interrupt to be enabled.				All
6	Display Plane B Flip Pending Wait Enable	Project:	All	Format: Enable
This field enables a wait for the duration of a Display Plane B “Flip Pending” condition. If a flip request is pending, the parser will wait until the flip operation has completed (i.e., the new front buffer address has now been loaded into the active front buffer registers). See Display Flip Pending Condition (in the Device Programming Interface chapter of <i>MI Functions</i>).				



MI_WAIT_FOR_EVENT				
	5	Display Pipe B Scan Line Window Wait Enable	Project: All	Format: Enable
		This field enables a wait while a Display B “In Scan Line Window” condition exists. This condition is defined as the period of time the Display B refresh is inside the scan line window as specified by a previous MI_LOAD_SCAN_LINES_INCL or MI_LOAD_SCAN_LINES_EXCL command. If the Display B refresh is outside this window, or a window has not been specified, the parser proceeds, treating this command as a no-op. If the Display B refresh is currently inside this window, the parser will wait until the refresh exits the window. See Scan Line Window Condition in the Device Programming Interface chapter of <i>MI Functions</i>.		
	4	Frame Buffer Compression Idle Wait Enable	Project: All	Format: Enable
		This field enables a wait while the Frame Buffer compressor is busy. The ring that this command got executed from is removed from arbitration for the wait period and is inserted into arbitration as soon as the frame buffer compressor is idle.		
	3	Display Pipe A Vertical Blank Wait Enable	Project: All	Format: Enable
		This field enables a wait until the next Display Pipe A “Vertical Blank” event occurs. This event is defined as the start of the next Display A vertical blank period. Note that this can cause a wait for up to an entire refresh period. See Vertical Blank Event in the Device Programming Interface chapter of <i>MI Functions</i>. Programming Notes Project Prior to using the MI_WAIT_FOR_EVENT command to wait on Display Pipe A/B VBlank events, the corresponding Vertical Blank Interrupt Enable (bit 17) of the corresponding PIPEASTAT (70024h) or PIPEBSTAT (71024h) register must be set. Note that this does not require an actual VBlank interrupt to be enabled. All		
	2	Display Plane A Flip Pending Wait Enable	Project: All	Format: Enable
		This field enables a wait for the duration of a Display Plane A “Flip Pending” condition. If a flip request is pending, the parser will wait until the flip operation has completed (i.e., the new front buffer address has now been loaded into the active front buffer registers). See Display Flip Pending Condition in the Device Programming Interface chapter of <i>MI Functions</i>.		
	1	Display Pipe A Scan Line Window Wait Enable	Project: All	Format: Enable
		This field enables a wait while a Display Pipe A “In Scan Line Window” condition exists. This condition is defined as the period of time the Display A refresh is inside the scan line window as specified by a previous MI_INCLUSIVE_SCAN_WINDOW or MI_EXCLUSIVE_SCAN_WINDOW command. If the Display A refresh is outside this window, or a window has not been specified, the parser proceeds, treating this command as a no-op. If the Display A refresh is currently inside this window, the parser will wait until the refresh exits the window. See Scan Line Window Condition in the Device Programming Interface chapter of <i>MI Functions</i>.		
	0	Reserved	Project: All	Format: MBZ



§§



10 *Memory Interface Commands for Blitter Engine*

10.1 Introduction

This chapter describes the formats of the “Memory Interface” commands, including brief descriptions of their use. The functions performed by these commands are discussed fully in the *Memory Interface Functions* Device Programming Environment chapter.

This chapter describes MI Commands for the blitter graphics processing engine. The term “for Blitter Engine” in the title has been added to differentiate this chapter from a similar one describing the MI commands for the Media Decode Engine and the Rendering Engine.

The commands detailed in this chapter are used across products within the Gen4 family. However, slight changes may be present in some commands (i.e., for features added or removed), or some commands may be removed entirely. Refer to the *Preface* chapter for product specific summary.



10.2 MI_LOAD_REGISTER_IMM

MI_LOAD_REGISTER_IMM			
Project:	All	Length Bias:	2
The MI_LOAD_REGISTER_IMM command requests a write of up to a DWord constant supplied in the command to the specified Register Offset (i.e., offset into Memory-Mapped Register Range). The register is loaded before the next command is executed. Programming Notes: The behavior of this command is controlled by Dword 3, Bit 8 (Disable Register Access) of the RINGBUF register. If this command is disallowed then the command stream converts it to a NOOP. If this command is executed from a BB then the behavior of this command is controlled by Dword 0, Bit 8 (Security Indicator) of the BATCH_BUFFER_START Command. If the batch buffer is insecure then the command stream converts this command to a NOOP. Note that the corresponding ring buffer must allow a register update for this command to execute.			
DWord	Bit	Description	
0	31:29	Command Type Default Value: 0h MI_COMMAND Format: OpCode	
	28:23	MI Command Opcode Default Value: h MI_ Format: OpCode	
	22:12	Reserved Project: All Format: MBZ	
	11:8	Byte Write Disables Format: Enable[4] Bit 8 corresponds to Data DWord [7:0] Range Must specify a valid register write operation This field specifies which bytes of the Data DWord are not to be written to the DWord offset specified in <i>Register Offset</i> .	
	7:6	Reserved Project: All Format: MBZ	
	5:0	DWord Length Default Value: 1h Excludes DWord (0,1) Format: =n Total Length - 2	
1	31:2	Register Offset Format: U30 Address: MmioAddress[31:2] This field specifies bits [31:2] of the offset into the Memory Mapped Register Range (i.e., this field specifies a DWord offset).	
	1:0	Reserved Project: All Format: MBZ	
2	31:0	Data DWord Mask: Bytes Write Disables Format: U32 This field specifies the DWord value to be written to the targeted location.	



10.3 MI_NOOP

MI_NOOP			
Project: All		Length Bias:	1
The MI_NOOP command basically performs a “no operation” in the command stream and is typically used to pad the command stream (e.g., in order to pad out a batch buffer to a QWord boundary). However, there is one minor (optional) function this command can perform – a 22-bit value can be loaded into the MI NOPID register. This provides a general-purpose command stream tagging (“breadcrumb”) mechanism (e.g., to provide sequencing information for a subsequent breakpoint interrupt).			
DWord	Bit	Description	
0	31:29	Command Type Default Value: 0h MI_COMMAND Format: OpCode	
	28:23	MI Command Opcode Default Value: 0h MI_NOOP Format: OpCode	
	22	Identification Number Register Write Enable Project: All Format: Enable This field enables the value in the Identification Number field to be written into the MI NOPID register. If disabled, that register is unmodified – making this command an effective “no operation” function.	
	31:0	Identification Number Project All Format: U22 : This field contains a 22-bit number which can be written to the MI NOPID register.	



10.4 MI_STORE_DATA_IMM

MI_STORE_DATA_IMM														
Project:		All		Length Bias:	2									
The MI_STORE_DATA_IMM command requests a write of the QWord constant supplied in the packet to the specified Memory Address. As the write targets a System Memory Address, the write operation is coherent with the CPU cache (i.e., the processor cache is snooped). Programming Notes: - This command can be used for general software synchronization through variables in cacheable memory (i.e., where software does not need to poll un-cached memory or device registers). However, the cacheable nature of the transaction is determined by the setting of the “mapping type” in the GTT entry. - This command simply initiates the write operation with command execution proceeding normally. Although the write operation is guaranteed to complete “eventually”, there is no mechanism to synchronize command execution with the completion (or even initiation) of these operations. All writes to memory generated using this command are expected to finish in order.														
DWord	Bit	Description												
0	31:29	Command Type Default Value: 0h MI_COMMAND Format: OpCode												
	28:23	MI Command Opcode Default Value: 20h MI_STORE_DATA_IMM Format: OpCode												
	22	Memory Address Type Project: All												
		<table><thead><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr></thead><tbody><tr><td>0h</td><td>Reserved</td><td>Physical address</td><td>All</td></tr><tr><td>1h</td><td>Reserved</td><td>Virtual address. Hardware will translate this address using the GTT. The GTT (global or per-process) used for the translation will be the same GTT used to access the buffer executing this instruction.</td><td>All</td></tr></tbody></table>	Value	Name	Description	Project	0h	Reserved	Physical address	All	1h	Reserved	Virtual address. Hardware will translate this address using the GTT. The GTT (global or per-process) used for the translation will be the same GTT used to access the buffer executing this instruction.	All
	Value	Name	Description	Project										
0h	Reserved	Physical address	All											
1h	Reserved	Virtual address. Hardware will translate this address using the GTT. The GTT (global or per-process) used for the translation will be the same GTT used to access the buffer executing this instruction.	All											
21:6	Reserved	Project: All	Format: MBZ											
	5:0	DWord Length Default Value: 2h Excludes DWord (0,1) = 2 for DWord, 3 for QWord Format: =n Total Length - 2												
1	31:0	Reserved	Project: All	Format: MBZ										
2	31:0	Reserved	Project: All	Format: MBZ										



MI_STORE_DATA_IMM		
3	31:0	Data DWord 0 Project: All Format: U32 This field specifies the DWord value to be written to the targeted location. For a QWord write this DWord is the lower DWord of the QWord to be reported (DW 0).
4	31:0	Data DWord 1 Project: All Format: U32 This field specifies the upper DWord value to be written to the targeted QWord location (DW 1).

10.5 MI_STORE_DATA_INDEX

MI_STORE_DATA_INDEX		
Project:	All	Length Bias: 2
The MI_STORE_DATA_INDEX command requests a write of the data constant supplied in the packet to the specified offset from the System Address defined by the Hardware Status Page Address Register. As the write targets a System Address, the write operation is coherent with the CPU cache (i.e., the processor cache is snooped).		
Programming Notes: - Use of this command with an invalid or uninitialized value in the Hardware Status Page Address Register is UNDEFINED. - This command can be used for general software synchronization through variables in cacheable memory (i.e., where software does not need to poll uncached memory or device registers). - This command simply initiates the write operation with command execution proceeding normally. Although the write operation is guaranteed to complete "eventually", there is no mechanism to synchronize command execution with the completion (or even initiation) of these operations.		
DWord	Bit	Description
0	31:29	Command Type Default Value: 0h MI_COMMAND Format: OpCode
	28:23	MI Command Opcode Default Value: 21h MI_STORE_DATA_INDEX Format: OpCode
	22	Reserved Project: All Format: Setting this bit will cause this command to offset in the Surface Probe List instead of the hardware status page. This is intended to be used internally only (it is UNDEFINED to set this bit in a command in a ring or batch buffer.)
	21:6 5:0	Reserved Project: All Format: MBZ DWord Length Default Value: 1h Excludes DWord (0,1) = 1 for DWord, 2 for QWord Format: =n Total Length - 2



MI_STORE_DATA_INDEX		
1	31:12	Reserved Project: All Format: MBZ
	11:2	Offset Project: All Format: U10 zero-based DWord offset into the HW status page. Address: HardwareStatusPageOffset[11:2] Surface Type: U32 Range [16, 1023] This field specifies the offset (into the hardware status page) to which the data will be written. Note that the first few DWords of this status page are reserved for special-purpose data storage – targeting these reserved locations via this command is UNDEFINED.
	1:0	Reserved Project: All Format: MBZ
2	31:0	Data DWord 0 Project: All Format: U32 This field specifies the DWord value to be written to the targeted location. For a QWord write this DWord is the lower DWord of the QWord to be reported (DW 0).
3	31:0	Data DWord 1 Project: All Format: U32 This field specifies the upper DWord value to be written to the targeted QWord location (DW 1).



10.6 MI_USER_INTERRUPT

MI_USER_INTERRUPT			
Project: All		Length Bias:	1
The MI_USER_INTERRUPT command is used to generate a User Interrupt condition. The parser will continue parsing after processing this command. See User Interrupt.			
DWord	Bit	Description	
0	31:29	Command Type Default Value: 0h MI_COMMAND Format: OpCode	
	28:23	MI Command Opcode Default Value: 02h MI_USER_INTERRUPT Format: OpCode	
	22:0	Reserved Project: All Format: MBZ	

10.7 MI_WAIT_FOR_EVENT

MI_WAIT_FOR_EVENT			
Project: All		Length Bias:	1
The MI_WAIT_FOR_EVENT command is used to pause command stream processing until a specific event occurs or while a specific condition exists. See Wait Events/Conditions, Device Programming Interface in <i>MI Functions</i>. Only one event/condition can be specified -- specifying multiple events is UNDEFINED. The effect of the wait operation depends on the source of the command. If executed from a batch buffer, the parser will halt (and suspend command arbitration) until the event/condition occurs. If executed from a ring buffer, further processing of that ring will be suspended, although command arbitration (from other rings) will continue. Note that if a specified condition does not exist (the condition code is inactive) at the time the parser executes this command, the parser proceeds, treating this command as a no-operation. If execution of this command from a primary ring buffer causes a wait to occur, the active ring buffer will <i>effectively</i> give up the remainder of its time slice (required in order to enable arbitration from other primary ring buffers).			
DWord	Bit	Description	
0	31:29	Command Type Default Value: 0h MI_COMMAND Format: OpCode	
	28:23	MI Command Opcode Default Value: 03h MI_WAIT_FOR_EVENT Format: OpCode	
	22:0	Reserved Project: All Format: MBZ	



§§



11 Graphics Memory Interface Functions

11.1 Introduction

The major role of an integrated graphics device's Memory Interface (MI) function is to provide various client functions access to “graphics” memory used to store commands, surfaces, and other information used by the graphics device. This chapter describes the basic mechanisms and paths by which graphics memory is accessed.

Information not presented in this chapter includes:

- Microarchitectural and implementation-dependent features (e.g., internal buffering, caching and arbitration policies).
- MI functions and paths specific to the operation of external (discrete) devices attached via external connections.
- MI functions essentially unrelated to the operation of the internal graphics devices, e.g., traditional “chipset functions” (refer to the device's C-Spec for this information).

11.2 Graphics Memory Clients

The MI function provides memory access functionality to a number of external and internal graphics memory *clients*, as described in Table 11-1.

Table 11-1. Graphics Memory Clients

MI Client	Access Modes
Host Processor	Read/Write of Graphics Operands located in Main Memory. Graphics Memory is accessed using Device 2 Graphics Memory Range Addresses
External PEG Graphics Device	Write-Only of Graphics Operands located in Main Memory via the Graphics Aperture. (This client is not described in this chapter).
Peer PCI Device	Write-Only of Graphics Operands located in Main Memory. Graphics Memory is accessed using Device 2 Graphics Memory Range Addresses (i.e., mapped by GTT). <i>Note that DMI access to Graphics registers is not supported.</i>
Snooped Read/Write (internal)	Internally-generated snooped reads/writes.
Command Stream (internal)	DMA Read of graphics commands and related graphics data.



MI Client	Access Modes
Vertex Stream (internal)	DMA Read of indexed vertex data from Vertex Buffers by the 3D Vertex Fetch (VF) Fixed Function.
Instruction/State Cache (internal)	Read of pipelined 3D rendering state used by the 3D/Media Functions and instructions executed by the EUs.
Render Cache (internal)	Read/Write of graphics data operated upon by the graphics rendering engines (Blit, 3D, MPEG, etc.) Read of render surface state.
Sampler Cache (internal)	Read of texture (and other sampled surface) data stored in graphics memory.
Display/Overlay Engines (internal)	Read of display, overlay, cursor and VGA data.

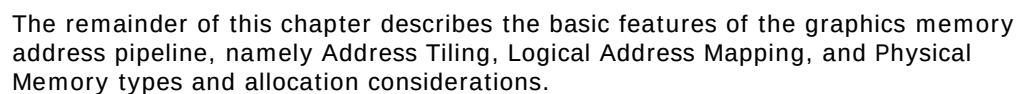
11.3 Graphics Memory Addressing Overview

The Memory Interface function provides access to graphics memory (GM) clients. It accepts memory addresses of various types, performs a number of optional operations along *address paths*, and eventually performs reads and writes of graphics memory data using the resultant addresses. The remainder of this subsection will provide an overview of the graphics memory clients and address operations.

11.3.1 Graphics Address Path

Figure 11-1 shows the internal graphics memory address path, connection points, and optional operations performed on addresses. Externally-supplied addresses are normalized to zero-based *Graphics Memory (GM) addresses* (GM_Address). If the GM address is determined to be a tiled address (based on inclusion in a fenced region or via explicit surface parameters), *address tiling* is performed. At this point the address is considered a *Logical Memory address*, and is translated into a *Physical Memory address* via the GTT and associated TLBs. The physical memory location is then accessed.

CPU accesses to graphics memory are not snooped on the front side bus post GTT translation. Hence pages that are mapped cacheable in the GTT will not be coherent with the CPU cache if accessed through graphics memory aperture. Also, such accesses may have side effects in the hardware.





11.4 Graphics Memory Address Spaces

Table 11-2 lists the five supported Graphics Memory Address Spaces. Note that the Graphics Memory Range Removal function is automatically performed to transform system addresses to internal, zero-based Graphics Addresses.

Table 11-2. Graphics Memory Address Types

Address Type	Description	Range
Dev2_GM_Address	Address range allocated via the Device 2 (integrated graphics device) GMADR register. The processor and other peer (DMI) devices utilize this address space to read/write graphics data that resides in Main Memory. This address is internally converted to a GM_Address.	Some 64MB, 128MB, 256MB or 512MB address range normally above TOM
GM_Address	Zero-based logical Graphics Address, utilized by internal device functions to access GTT-mapped graphics operands. GM_Addresses are typically passed in commands and contained in state to specify operand location.	[0, 64MB-1], [0, 128MB-1], [0, 256MB-1] or [0, 512MB-1]
PGM_Address	Zero-based logical Per-Process Graphics Address, utilized by internal device functions to access render GTT (PPGTT) mapped graphics operands. Memory in this space is not accessible by the processor and other peer (DMI) devices unless aliased to a GM_Address.	[0, 64MB-1], [0,128MB-1], [0,256MB-1], [0,512MB-1] or [0, 1GB – 1]

11.5 Address Tiling Function

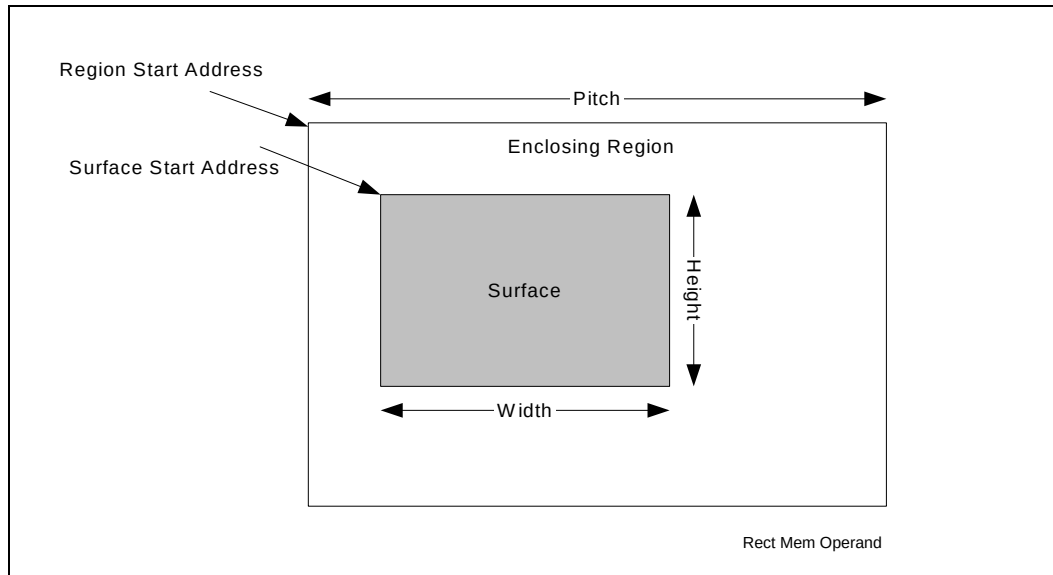
When dealing with memory operands (e.g., graphics surfaces) that are inherently rectangular in nature, certain functions within the graphics device support the storage/access of the operands using alternative (tiled) memory formats in order to increase performance. This section describes these memory storage formats, why/when they should be used, and the behavioral mechanisms within the device to support them.

11.5.1 Linear vs. Tiled Storage

Regardless of the memory storage format, “rectangular” memory operands have a specific *width* and *height*, and are considered as residing within an enclosing rectangular region whose width is considered the *pitch* of the region and surfaces contained within. Surfaces stored within an enclosing region must have widths less than or equal to the region pitch (indeed the enclosing region may coincide exactly with the surface). Figure 11-2 shows these parameters.

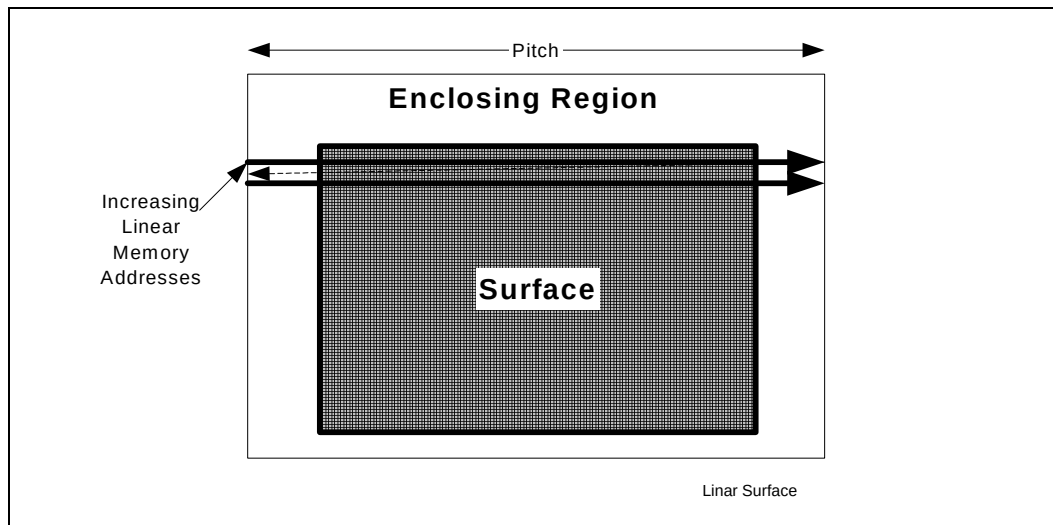


Figure 11-2. Rectangular Memory Operand Parameters



The simplest storage format is the *linear* format (see Figure 11-3), where each row of the operand is stored in sequentially increasing memory locations. If the surface width is less than the enclosing region's pitch, there will be additional memory storage between rows to accommodate the region's pitch. The pitch of the enclosing region determines the distance (in the memory address space) between vertically-adjacent operand elements (e.g., pixels, texels).

Figure 11-3. Linear Surface Layout



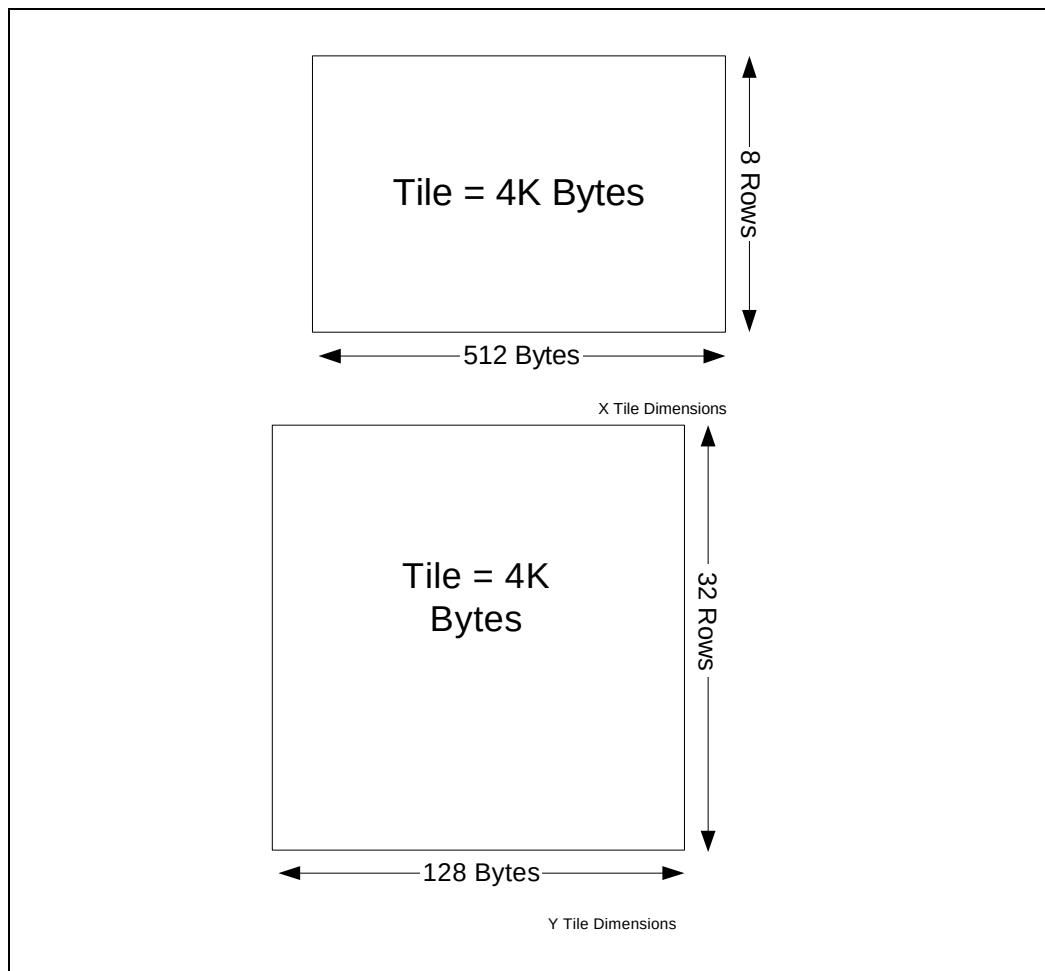
The linear format is best suited for 1-dimensional row-sequential access patterns (e.g., a display surface where each scanline is read sequentially). Here the fact that one object element may reside in a different memory page than its vertically-adjacent neighbors is not significant; all that matters is that horizontally-adjacent elements are stored contiguously. However, when a device function needs to access a 2D subregion

within an operand (e.g., a read or write of a 4x4 pixel span by the 3D renderer, a read of a 2x2 texel block for bilinear filtering), having vertically-adjacent elements fall within different memory pages is to be avoided, as the page crossings required to complete the access typically incur increased memory latencies (and therefore lower performance).

One solution to this problem is to divide the enclosing region into an array of smaller rectangular regions, called memory *tiles*. Surface elements falling within a given tile will all be stored in the same physical memory page, thus eliminating page-crossing penalties for 2D subregion accesses within a tile and thereby increasing performance.

Tiles have a fixed 4KB size and are aligned to physical DRAM page boundaries. They are either 8 rows high by 512 bytes wide or 32 rows high by 128 bytes wide (see Figure 11-4). Note that the dimensions of tiles are irrespective of the data contained within – e.g., a tile can hold twice as many 16-bit pixels (256 pixels/row x 8 rows = 2K pixels) than 32-bit pixels (128 pixels/row x 8 rows = 1K pixels).

Figure 11-4. Memory Tile Dimensions

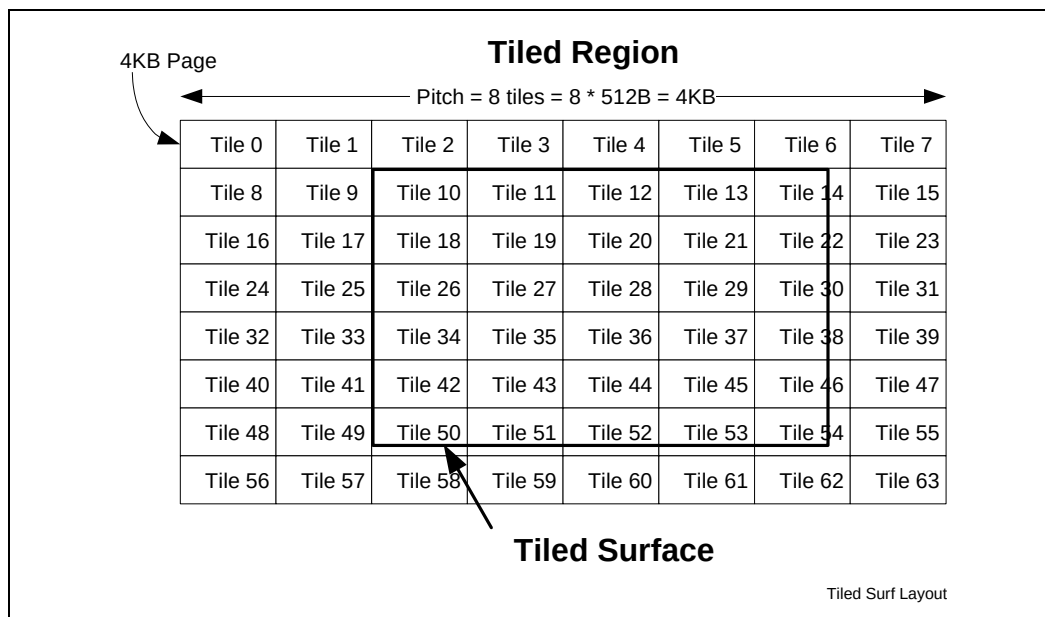




The pitch of a tiled enclosing region must be an integral number of tile widths. The 4KB tiles within a tiled region are stored sequentially in memory in row-major order.

Figure 11-5 shows an example of a tiled surface located within a tiled region with a pitch of 8 tile widths (512 bytes * 8 = 4KB). Note that it is the enclosing region that is divided into tiles – the surface is not necessarily aligned or dimensioned to tile boundaries.

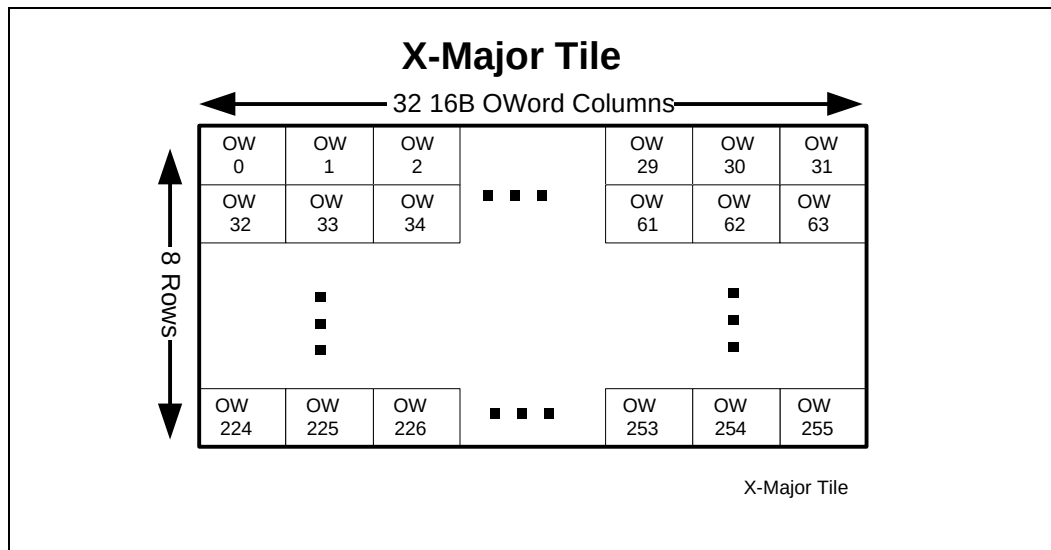
Figure 11-5. Tiled Surface Layout



11.5.2 Tile Formats

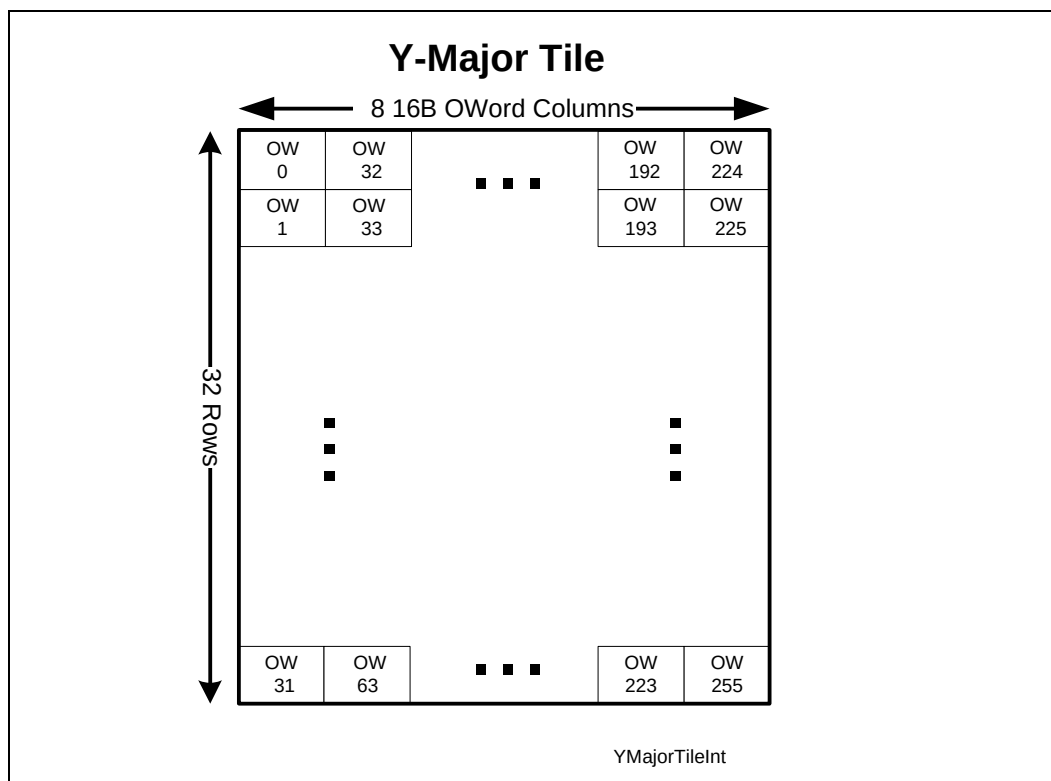
The device supports both *X-Major* (row-major) and *Y-Major* (column major) storage of tile data units, as shown in the following figures. A 4KB tile is subdivided into an 8-high by 32-wide array of 16-byte OWords for X-Major Tiles (X Tiles for short), and 32-high by 8-wide array of OWords for Y-Major Tiles (Y Tiles). The selection of tile direction only impacts the internal organization of tile data, and does not affect how surfaces map onto tiles. Note that the diagrams are not to scale – the first format defines the contents of an 8-high by 512-byte wide tile, and the 2nd a 32-high by 128-byte wide tile. The storage of tile data units in X-Major or Y-Major fashion is sometimes refer to as the *walk* of the tiling.

Table 11-3. X-Major Tile Layout



Note that an X-major tiled region with a tile pitch of 1 tile is actually stored in a linear fashion.

Figure 11-6. Y-Major Tile Layout





11.5.3 Tiling Algorithm

The following pseudocode describes the algorithm for translating a tiled memory surface in graphics memory to an address in logical space.

```
Inputs: LinearAddress(offset into aperture in terms of bytes),
        Pitch(in terms of tiles),
        WalkY (1 for Y and 0 for X)

Static Parameters: TileH (Height of tile, 8 for X and 32 for Y),
                  TileW (Width of Tile in bytes, 512 for X and
                        128 for Y)

TileSize = TileH * TileW;
RowSize = Pitch * TileSize;

If (Fenced) {
    LinearAddress = LinearAddress - FenceBaseAddress
    LinearAddrInTileW = LinearAddress div TileW;
    Xoffset_inTile = LinearAddress mod TileW;
    Y = LinearAddrInTileW div Pitch;
    X = LinearAddrInTileW mod Pitch + Xoffset_inTile;
}

// Internal graphics clients that access tiled memory already have
// the X, Y
// coordinates and can start here
YOff_Within_Tile = Y mod TileH;
XOff_Within_Tile = X mod TileW;

TileNumber_InY = Y div TileH;
TileNumber_InX = X div TileW;

TiledOffsetY = RowSize * TileNumber_InY + TileSize *
               TileNumber_InX + TileH * XOff_Within_Tile +
               YOff_Within_Tile * 16 + (XOff_Within_Tile mod 16);

TiledOffsetX = RowSize * TileNumber_InY + TileSize *
               TileNumber_InX + TileW * YOff_Within_Tile +
               XOff_Within_Tile;

TiledOffset = WalkY? TiledOffsetY : TiledOffsetX;

TiledAddress = Tiled? (BaseAddress + TiledOffset): (BaseAddress +
               Y*LinearPitch + X);
}
```

The Y-Major tile formats have the characteristic that a surface element in an even row is located in the same aligned 64-byte cacheline as the surface element immediately below it (in the odd row). This spatial locality can be exploited to increase performance when reading 2x2 texel squares for bilinear texture filtering, or reading and writing aligned 4x4 pixel spans from the 3D Render pipeline.



On the other hand, the X-Major tile format has the characteristic that horizontally-adjacent elements are stored in sequential memory addresses. This spatial locality is advantageous when the surface is scanned in row-major order for operations like display refresh. For this reason, the Display and Overlay memory streams only support linear or X-Major tiled surfaces (Y-Major tiling is not supported by these functions). This has the side effect that 2D- or 3D-rendered surfaces must be stored in linear or X-Major tiled formats if they are to be displayed. Non-displayed surfaces, e.g., “rendered textures”, can also be stored in Y-Major order.

11.5.4 Tiling Support

The rearrangement of the surface elements in memory must be accounted for in device functions operating upon tiled surfaces. (Note that not all device functions that access memory support tiled formats). This requires either the modification of an element's linear memory address or an alternate formula to convert an element's X,Y coordinates into a tiled memory address.

However, before tiled-address generation can take place, some mechanism must be used to determine whether the surface elements accessed fall in a linear or tiled region of memory, and if tiled, what the tile region pitch is, and whether the tiled region uses X-Major or Y-Major format. There are two mechanisms by which this detection takes place: (a) an implicit method by detecting that the pre-tiled (linear) address falls within a “fenced” tiled region, or (b) by an explicit specification of tiling parameters for surface operands (i.e., parameters included in surface-defining instructions).

The following table identifies the tiling-detection mechanisms that are supported by the various memory streams.

Access Path	Tiling-Detection Mechanisms Supported
Processor access through the Graphics Memory Aperture	Fenced Regions
3D Render (Color/Depth Buffer access)	Explicit Surface Parameters
Sampled Surfaces	Explicit Surface Parameters
Blit operands	Explicit Surface Parameters
Display and Overlay Surfaces	Explicit Surface Parameters

11.5.4.1 Tiled (Fenced) Regions

The only mechanism to support the access of surfaces in tiled format by the host or external graphics client is to place them within “fenced” tiled regions within Graphics Memory. A fenced region is a block of Graphics Memory specified using one of the sixteen FENCE device registers. (See *Memory Interface Registers* for details). Surfaces contained within a fenced region are considered tiled from an external access point of view. Note that fences cannot be used to untile surfaces in the PGM_Address space since external devices cannot access PGM_Address space. Even if these surfaces (or any surfaces accessed by an internal graphics client) fall within a region covered by an enabled fence register, that enable will be effectively masked during the internal graphics client access. Only the explicit surface parameters described in the next section can be used to tile surfaces being accessed by the internal graphics clients.



Each FENCE register (if its Fence Valid bit is set) defines a Graphics Memory region ranging from 4KB to the aperture size. The region is considered rectangular, with a pitch in tile widths from 1 tile width (128B or 512B) to 256 tile X widths ($256 * 512B = 128KB$) and 1024 tile Y widths ($1024 * 128B = 128KB$). Note that fenced regions must not overlap, or operation is UNDEFINED.

Also included in the FENCE register is a Tile Walk field that specifies which tile format applies to the fenced region.

11.5.4.2 Tiled Surface Parameters

Internal device functions require explicit specification of surface tiling parameters via information passed in commands and state. This capability is provided to limit the reliance on the fixed number of fence regions.

The following table lists the surface tiling parameters that can be specified for 3D Render surfaces (Color Buffer, Depth Buffer, Textures, etc.) via SURFACE_STATE.

Surface Parameter	Description
Tiled Surface	If ENABLED, the surface is stored in a tiled format. If DISABLED, the surface is stored in a linear format.
Tile Walk	If Tiled Surface is ENABLED, this parameter specifies whether the tiled surface is stored in Y-Major or X-Major tile format.
Base Address	Additional restrictions apply to the base address of a Tiled Surface vs. that of a linear surface.
Pitch	Pitch of the surface. Note that, if the surface is tiled, this pitch must be a multiple of the tile width.

11.5.4.3 Tiled Surface Restrictions

Additional restrictions apply to the Base Address and Pitch of a surface that is tiled. In addition, restrictions for tiling via SURFACE_STATE are subtly different from those for tiling via fence regions. The most restricted surfaces are those that will be accessed both by the host (via fence) and by internal device functions. An example of such a surface is a tiled texture that is initialized by the CPU and then sampled by the device.

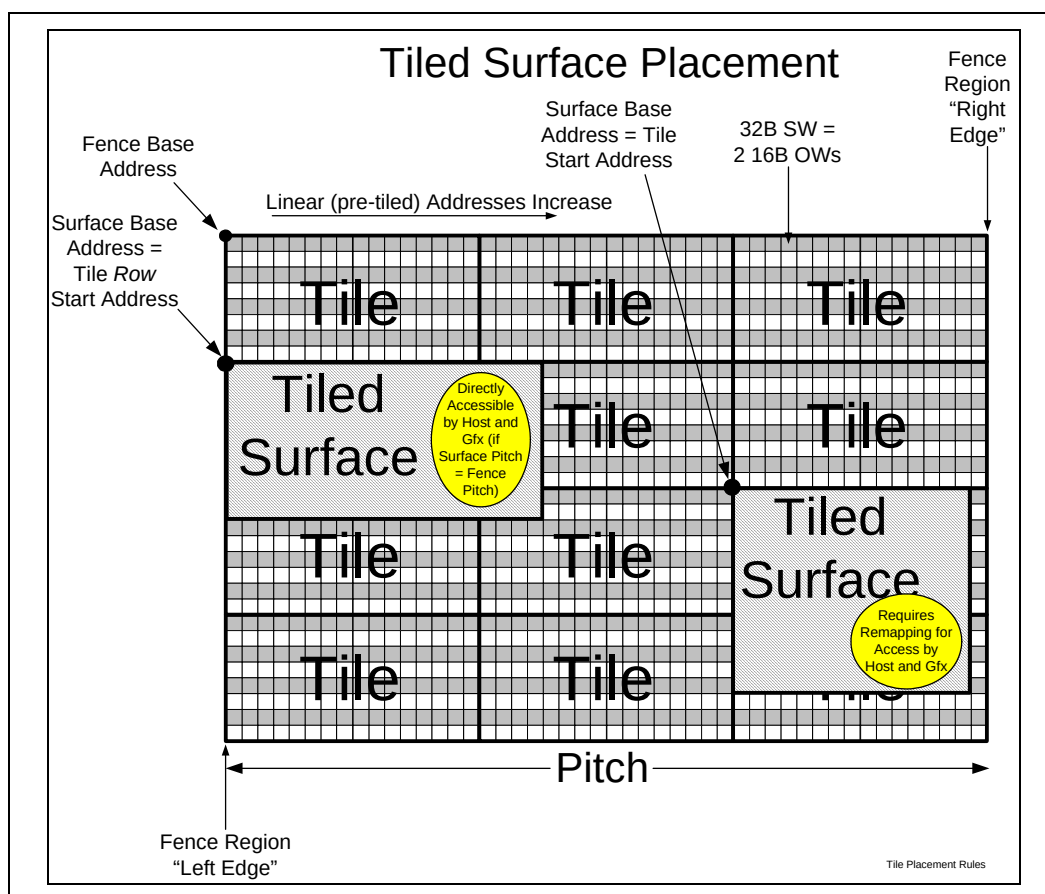
The tiling algorithm for internal device functions is different from that of fence regions. Internal device functions always specify tiling in terms of a surface. The surface must have a base address, and this base address is not subject to the tiling algorithm. Only *offsets* from the base address (as calculated by X, Y addressing within the surface) are transformed through tiling. The base address of the surface must therefore be 4KB-aligned. This forces the 4KB tiles of the tiling algorithm to exactly align with 4KB device pages once the tiling algorithm has been applied to the offset. The width of a surface must be less than or equal to the surface pitch. There are additional considerations for surfaces that are also accessed by the host (via a fence region).

Fence regions have no base address per se. Host linear addresses that fall in a fence region are translated in their entirety by the tiling algorithm. It is as if the surface being tiled by the fence region has a base address in graphics memory equal to the fence base address, and all accesses of the surfaces are (possibly quite large) offsets from the fence base address. Fence regions have a virtual “left edge” aligned with the

fence base address, and a “right edge” that results from adding the fence pitch to the “left edge”. Surfaces in the fence region must not straddle these boundaries.

Base addresses of surfaces that are to be accessed both by an internal graphics client and by the host have the tightest restrictions. In order for the surface to be accessed without GTT re-mapping, the surface base address (as set in SURFACE_STATE) must be a “Tile Row Start Address” (TRSA). The first address in each tile row of the fence region is a Tile Row Start Address. The first TRSA is the fence base address. Each TRSA can be generated by adding an integral multiple of the row size to the fence base address. The row size is simply the fence pitch in tiles multiplied by 4KB (the size of a tile.)

Figure 11-7. Tiled Surface Placement



The pitch in SURFACE_STATE must be set equal to the pitch of the fence that will be used by the host to access the surface if the same GTT mapping will be used for each access. If the pitches differ, a different GTT mapping must be used to eliminate the “extra” tiles (4KB memory pages) that exist in the excess rows at the right side of the larger pitch. Obviously no part of the surface that will be accessed can lie in pages that exist only in one mapping but not the other. The new GTT mapping can be done manually by SW between the time the host writes the surface and the device reads it, or it can be accomplished by arranging for the client to use a different GTT than the host (the PPGTT -- see Logical Memory Mapping below).



The width of the surface (as set in SURFACE_STATE) must be less than or equal to both the surface pitch and the fence pitch in any scenario where a surface will be accessed by both the host and an internal graphics client. Changing the GTT mapping will not help if this restriction is violated.

Surface Access	Base Address	Pitch	Width	Tile "Walk"
Host only	No restriction	Integral multiple of tile size $\leq$ 128KB	Must be $\leq$ Fence Pitch	No restriction
Client only	4KB-aligned	Integral multiple of tile size $\leq$ 256KB	Must be $\leq$ Surface Pitch	Restrictions imposed by the client (see Per-Stream Tile Format Support)
Host and Client, No GTT Remapping	Must be TRSA	Fence Pitch = Surface Pitch = integral multiple of tile size $\leq$ 256KB	Width $\leq$ Pitch	Surface Walk must meet client restriction, Fence Walk = Surface Walk
Host and Client, GTT Remapping	4KB-aligned for client (will be tile-aligned for host)	Both must be Integral multiple of tile size $\leq$ 128KB, but not necessarily the same	Width $\leq$ Min(Surface Pitch, Fence Pitch)	Surface Walk must meet client restriction, Fence Walk = Surface Walk



11.5.5 Per-Stream Tile Format Support

MI Client	Tile Formats Supported						
CPU Read/Write	All						
Display/Overlay	Y-Major not supported. X-Major required for Async Flips						
Blit	Linear and X-Major only No Y-Major support						
3D Sampler	All Combinations of TileY, TileX and Linear are supported. TileY is the fastest, Linear is the slowest.						
3D Color,Depth	<table> <tr> <th>Rendering Mode Color-vs-Depth bpp</th><th>Buffer Tiling Supported</th></tr> <tr> <td>Classical Same Bpp</td><td>Both Linear Both TileX Both TileY Linear & TileX Linear & TileY TileX & TileY</td></tr> <tr> <td>Classical Mixed Bpp</td><td>Both Linear Both TileX Both TileY Linear & TileX Linear & TileY TileX & TileY</td></tr> </table> NOTE: 128BPE Format Color buffer (render target) MUST be either TileX or Linear.	Rendering Mode Color-vs-Depth bpp	Buffer Tiling Supported	Classical Same Bpp	Both Linear Both TileX Both TileY Linear & TileX Linear & TileY TileX & TileY	Classical Mixed Bpp	Both Linear Both TileX Both TileY Linear & TileX Linear & TileY TileX & TileY
Rendering Mode Color-vs-Depth bpp	Buffer Tiling Supported						
Classical Same Bpp	Both Linear Both TileX Both TileY Linear & TileX Linear & TileY TileX & TileY						
Classical Mixed Bpp	Both Linear Both TileX Both TileY Linear & TileX Linear & TileY TileX & TileY						

11.6 Logical Memory Mapping

In order to provide a contiguous address space for graphics operands (surfaces, etc.) yet allow this address space to be mapped onto possibly discontinuous physical memory pages, the internal graphics device supports a Logical Memory Space (see Figure 11-10). A global *Graphics Translation Table* (GTT) is provided to map zero-based (and post-tiled) Logical Memory Addresses into a set of 4KB physical memory pages. (This mapping is also used for external PEG devices.)

There is another logical mapping function available local to each graphics process; this works identically to the global GTT with some additional restrictions. The base address for this per-process GTT (PPGTT) is determined by the PGTBL_CTL2 register. This register is saved and restored with ring context, thus providing each graphics context with its own local translation table and protected memory space (see Rendering Context Management later in this chapter).

The GTT and PPGTT are arrays of 4-byte *Page Table Entries* (PTEs) physically located in Main Memory. The GTT and PPGTT are comprised of a number of locked (non-



swappable) physically-contiguous 4KB memory pages, with a maximum size (each) of 128 4KB pages (128K DWords map $128K \times 4KB = 512MB$ max) for Global GTT, and up to 512 4KB pages for PPGTT, for total up to 2GB max. GTT and PPGTT base addresses must be 4KB-aligned.

Note that the PTEs within the global GTT must be written only through GTTADDR (see the Device #2 Config registers for a description of this range), as the MI function needs to snoop PTE updates in order to invalidate TLBs, which cache PTEs. The PGTBL_CTL register also contains a Page Table Enable bit used to enable/disable Logical Memory mapping. With the exception of processor Read, Cursor and VGA clients, access to graphics memory is not permitted when the Page Table Enable bit is clear (i.e., disabled). The PGTBL_ER debug register provides information pertaining to HW-detected errors in the Logical Memory Mapping function (e.g., invalid PTEs, invalid mappings, etc.).

The PPGTT base address is also 4KB aligned, but it is programmed directly in physical memory space rather than through an alias mechanism like GTTADDR. Note that not all clients may use the PPGTT; only the global GTT is available for processor accesses as well as graphics accesses from display engines (including overlay and cursor). Any per-process access that occurs while the PPGTT is disabled (via a bit in PGTBL_CTL2) will default to a translation via the global GTT.

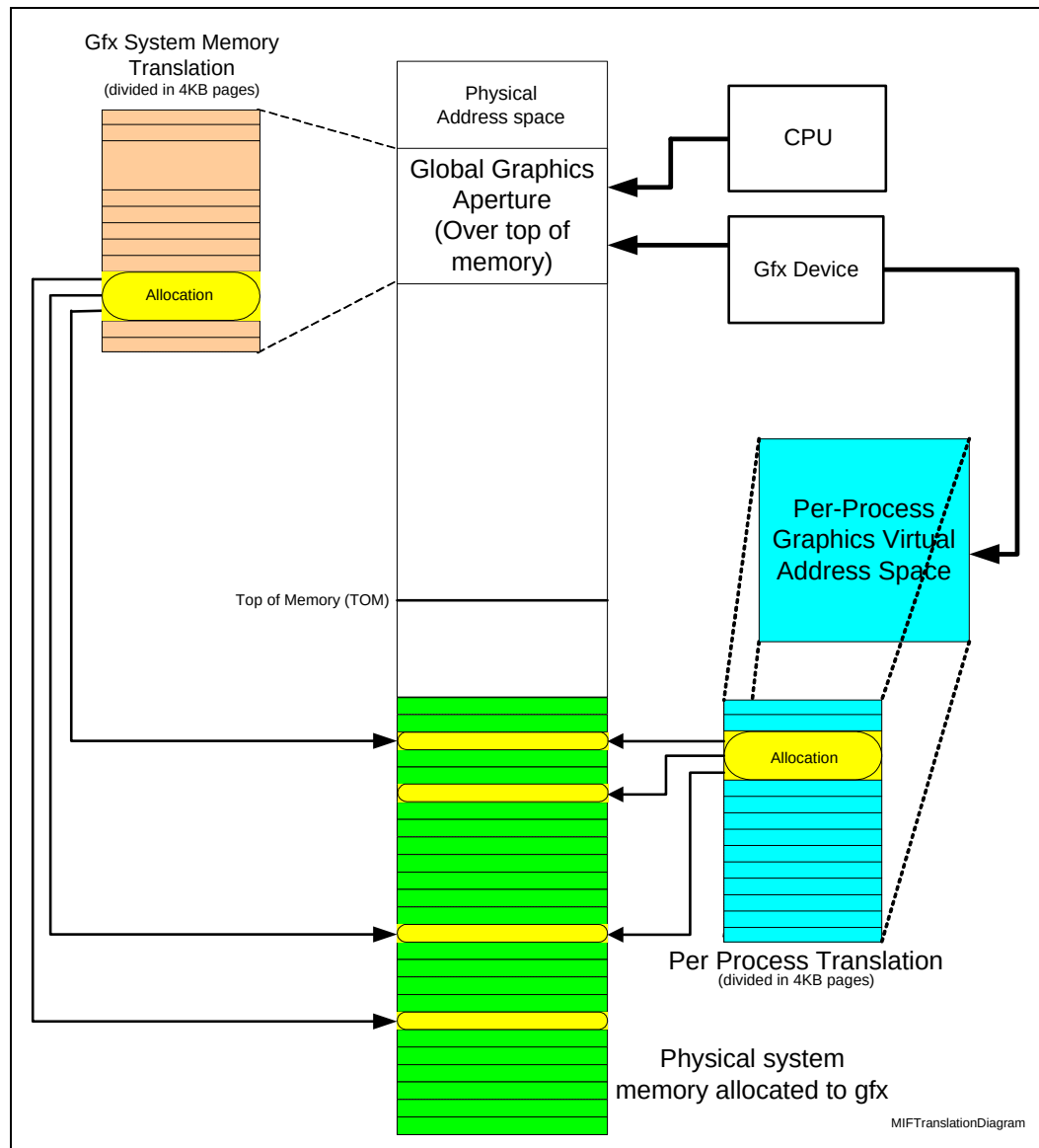
11.6.1 Logical Memory Space Mappings

Each valid PTE maps a 4KB page of Logical Memory to an independent 4KB page of:

- MM: Main Memory (unsnooped), or
- SM: System Memory (snooped, therefore coherent with the processor cache, must not be accessed through the Dev2_GM_Address range by the CPU)

PTEs marked as invalid have no backing physical memory, and therefore the corresponding Logical Memory Address pages must not be accessed in normal operation.

Figure 11-8. Global and Render GTT Mapping





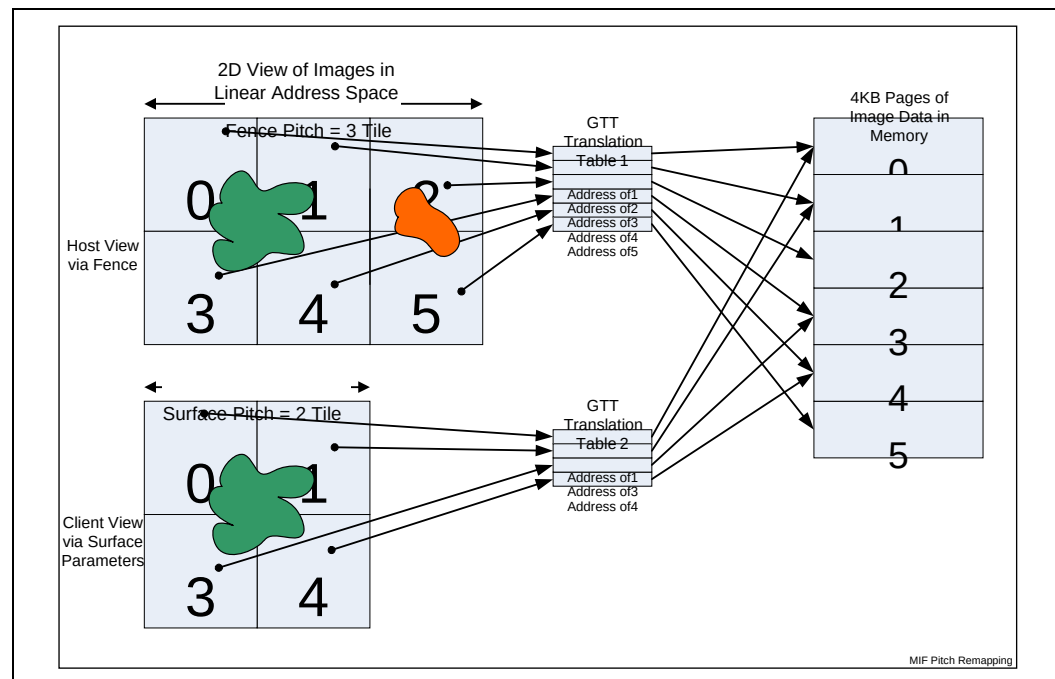
The following table lists the memory space mappings valid for each MI client:

MI Client	Logical Memory Space Mappings Supported	xGTT Usage
External Clients		
Host Processor	MM	GTT only
External PEG Device	None	n/a
Snooped Read/Write	None	n/a
Internal GPU Clients		
Render Command Ring Buffers	MM	GTT/PGTT, selected by PGTBL_STR2<2>
Render Command Batch Buffers	MM	GTT/PGTT, selected by PGTBL_STR2<5>
Indirect State Buffers	MM	GTT/PGTT, selected by PGTBL_STR2<4>
CURBE Constant Data	MM	Same xGTT used to fetch the CONSTANT_BUFFER command.
Media Object Indirect Data	MM	Same xGTT used to fetch the MEDIA_OBJECT command.
Vertex Fetch Data	MM, SM	GTT/PGTT, selected by PGTBL_STR2<3>
Sampler Cache (RO)	MM, SM	GTT/PGTT, selected by PGTBL_STR2<1>
DataPort Render Cache (R/W)	MM, SM	GTT/PGTT, selected by PGTBL_STR2<0>
Depth Buffer Cache (R/W)	MM	GTT/PGTT, selected by PGTBL_STR2<0>
Blit Engine	MM, SM	GTT/PGTT, selected by PGTBL_STR2<0>
MI_STORE_DATA_IMM Destination (if virtual addressed)	MM, SM	Same xGTT used to fetch the command.
PIPE_CONTROL Write Destination	MM, SM	GTT/PGTT, selected by the command
Display/Overlay Engines (internal)	MM	GTT only

Usage Note: Since the CPU cannot directly access memory pages mapped through a Graphics Process' local GTT (PPGTT), these pages must also be mapped through the global GTT (at least temporarily) in order for the CPU to initialize graphics data for a Graphics Process.

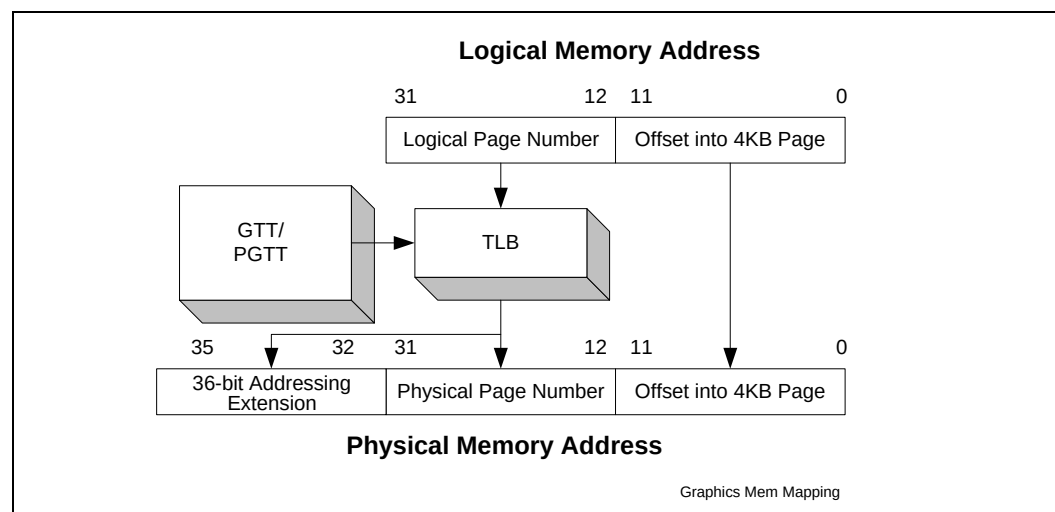
The PPGTT mechanism can be used by a client to access a surface with a pitch that is smaller than that of the fence region used by the host to initialize the surface, without having to physically move the data in memory.

Figure 11-9. GTT Re-mapping to Handle Differing Pitches



Refer to the "Graphics Translation Table (GTT) Range (GTTADR) & PTE Description" in *Memory Interface Registers* for details on PTE formats and programming information. Refer to the *Memory Data Formats* chapter for device-specific details/restrictions regarding the placement/storage of the various data objects used by the graphics device.

Figure 11-10. Logical-to-Physical Graphics Memory Mapping



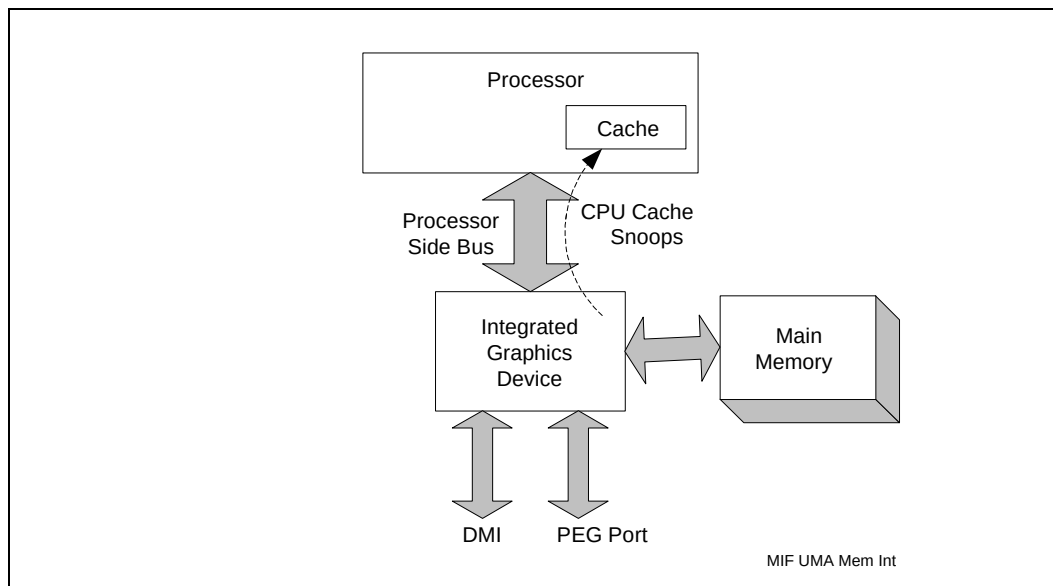


11.7 Physical Graphics Memory

The integrated graphics device satisfies all of its memory requirements using portions of main system memory. The integrated graphics device operates without any dedicated local memory, in a lower-cost configuration typically (though not necessarily officially) known as *Unified Graphics Memory (UMA)*.

Figure 11-11 shows how the Main Memory is interfaced to the device.

Figure 11-11. Memory Interfaces



11.7.1 Physical Graphics Address Types

Table 11-4 lists the various physical address types supported by the integrated graphics device. Physical Graphics Addresses are either generated by Logical Memory mappings or are directly specified by graphics device functions. These physical addresses are not subject to tiling or GTT re-mappings.

Table 11-4. Physical Memory Address Types

Address Type	Description	Range
MM_Address	Main Memory Address. Offset into physical, <u>unsnooped</u> Main Memory.	[0,TopOfMemory-1]
SM_Address	System Memory Address. Accesses are snooped in processor cache, allowing shared graphics/ processor access to (locked) cacheable memory data.	[0,4GB]



11.7.2 Main Memory

The integrated graphics device is capable of using 4KB pages of physical main (system) memory for graphics functions. Some of this main memory can be “stolen” from the top of system memory during initialization (e.g., for a VGA buffer). However, most graphics operands are dynamically allocated to satisfy application demands. To this end the graphics driver will frequently need to allocate locked-down (i.e., non-swappable) physical system memory pages – typically from a cacheable non-paged pool. The locked pages required to back large surfaces are typically non-contiguous. Therefore a means to support “logically-contiguous” surfaces backed by discontinuous physical pages is required. The Graphics Translation Table (GTT) that was described in previous sections provides the means.

11.7.2.1 Optimizing Main Memory Allocation

This section includes information for software developers on how to allocate SDRAM Main Memory (SM) for optimal performance in certain configurations. The general idea is that these memories are divided into some number of page types, and careful arrangement of page types both within and between surfaces (e.g., between color and depth surfaces) will result in fewer page crossings and therefore yield somewhat higher performance.

The algorithm for allocating physical SDRAM Main Memory pages to logical graphics surfaces is somewhat complicated by (1) permutations of memory device technologies (which determine page sizes and therefore the number of pages per device row), (2) memory device row population options, and (3) limitations on the allocation of physical memory (as imposed by the OS).

However, the theory to optimize allocation by limiting page crossing penalties is simple: (a) switching between open pages is optimal (again, the pages do not need to be sequential), (b) switching between memory device rows does not in itself incur a penalty, and (c) switching between pages within a particular bank of a row incurs a page miss and should therefore be avoided.

11.7.2.2 Application of the Theory (Page Coloring)

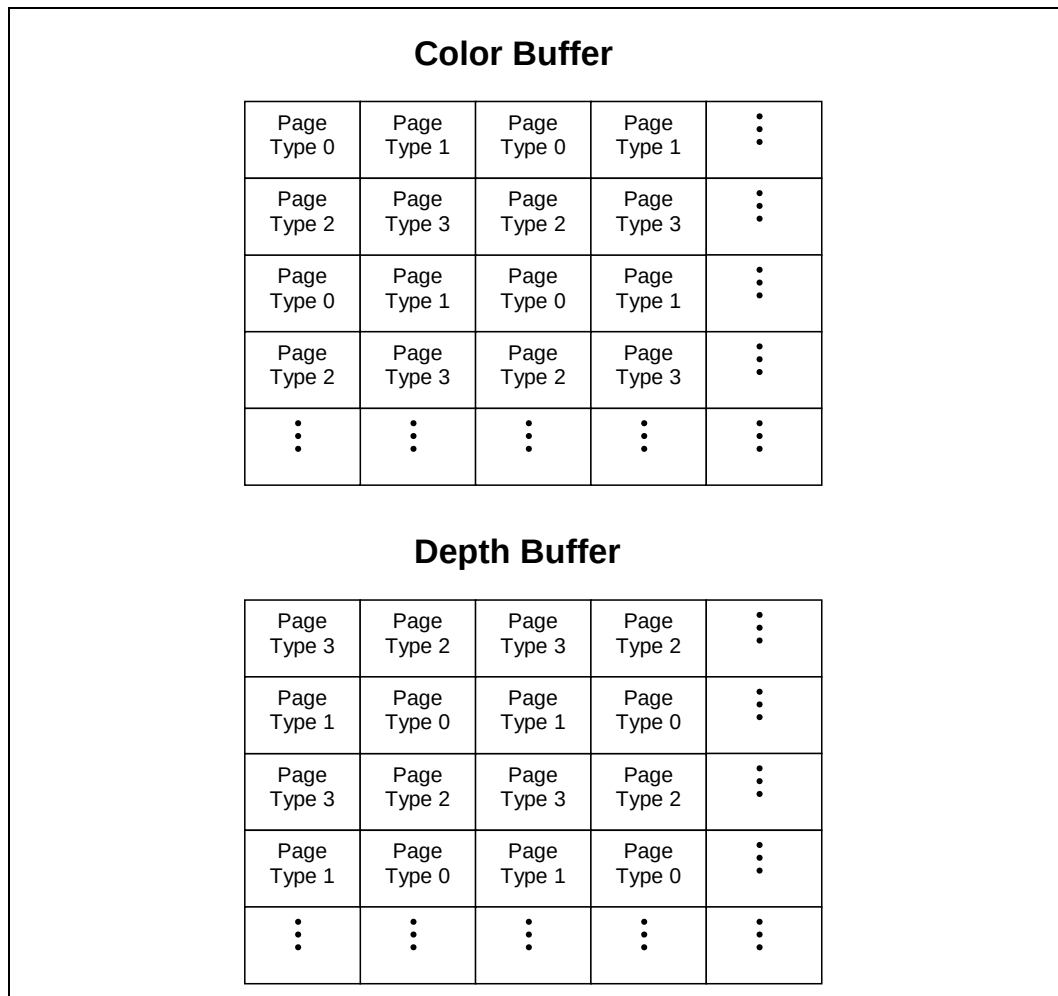
This section provides some scenarios of how Main Memory page allocation can be optimized.

11.7.2.2.1 3D Color and Depth Buffers

Here we want to minimize the impact of page crossings (a) between corresponding pages (1-4 tiles) in the Color and Depth buffers, and (b) when moving from a page to a neighboring page within a Color or Depth buffer. Therefore corresponding pages in the Color and Depth Buffers, and adjacent pages within a Color or Depth Buffer should be mapped to different page types (where a page’s “type” or “color” refers to the row and bank it’s in).



Figure 11-12. Memory Pages backing Color and Depth Buffers



For higher performance, the Color and Depth Buffers could be allocated from different memory device rows.

11.7.2.2.2 Media/Video

The Y surfaces can be allocated using 4 page types in a similar fashion to the Color Buffer diagram above. The U and V surfaces would split the same 4 page types as used in the Y surface.

§§





12 Device Programming Environment

The graphics device contains an extensive set of registers and commands (also referred to as “commands” or “packets”) for controlling 2D, 3D, video I/O, and other operations. This chapter describes the programming environment and software interface to these registers/commands. The registers and commands themselves are described elsewhere in this document.

12.1 Programming Model

The graphics device is programmed via the following three basic mechanisms:

POST-Time Programming of Configuration Registers

These registers are the graphics device registers residing in PCI space. A majority of these registers are programmed once during POST of the video device. Configuration registers are not covered in this section. For details on accessing the graphics device's configuration space see the EDS.

Direct (Physical I/O and/or Memory-Mapped I/O) Access of Graphics Registers

Various graphics functions can only be controlled via direct register access. In addition, direct register access is required to initiate the (asynchronous) execution of graphics command streams. This programming mechanism is “direct” and synchronous with software execution on the CPU.

Command Stream DMA (via the Command Ring Buffer and Batch Buffers)

This programming mechanism utilizes the indirect and asynchronous execution of graphics command streams to control certain graphics functions, e.g., all 2D, 3D drawing operations. Software writes commands into a command buffer (either a Ring Buffer or Batch Buffer) and informs the graphics device (using the Direct method above) that the commands are ready for execution. The graphics device's Command Parser (CP) will then, or at some point in the future, read the commands from the buffer via DMA and execute them.

12.2 Graphics Device Register Programming

The graphics device registers (except for the Configuration registers) are memory mapped. The base address of this 512 KB memory block is programmed in the MMADR Configuration register. For a detailed description of the register map and register categories, refer to the *Register Maps* chapter.

Programming Note:

Software must only access GR06, MSR0, MSR1, and Paging registers (see *Register Maps*) via Physical I/O, never via Memory Mapped I/O.



12.3 Graphics Device Command Streams

This section describes how command streams can be used to initiate and control graphics device operations.

12.3.1 Command Use

Memory-resident commands are used to control drawing engines and other graphics device functional units:

- **Memory Interface (MI) Commands.** The MI commands can be used to control and synchronize the command stream as well as perform various auxiliary functions (e.g., perform display/overlay flips, etc.)
- **2D Commands (BLT).** These commands are used to perform various 2D (Blt) operations.
- **3D Commands.** 3D commands are used to program the 3D pipeline state and perform 3D rendering operations. There are also a number of 3D commands that can be used to accelerate 2D and video operations, e.g., "StretchBlt" operations, 2D line drawing, etc.
- **Video (MPEG, WMV, etc.) Decode Commands.** A set of commands are supported to perform video decode acceleration including Motion Compensation operations via the Sampling Engine of the 3D pipeline.

12.3.2 Command Transport Overview

Commands are not written directly to the graphics device – instead they are placed in memory by software and later read via DMA by the graphics device's Command Parser (CP) within the Memory Interface function. The primary mechanism used to transport commands is through the use of a Ring Buffer.

An additional, indirect mechanism for command transport is through the use of Batch Buffers initiated from the Ring buffer.

The Command Parser uses a set of rules to determine the order in which commands are executed. Following sections in this chapter provide descriptions of the Ring Buffer, Batch Buffers, and Command Parser arbitration rules.



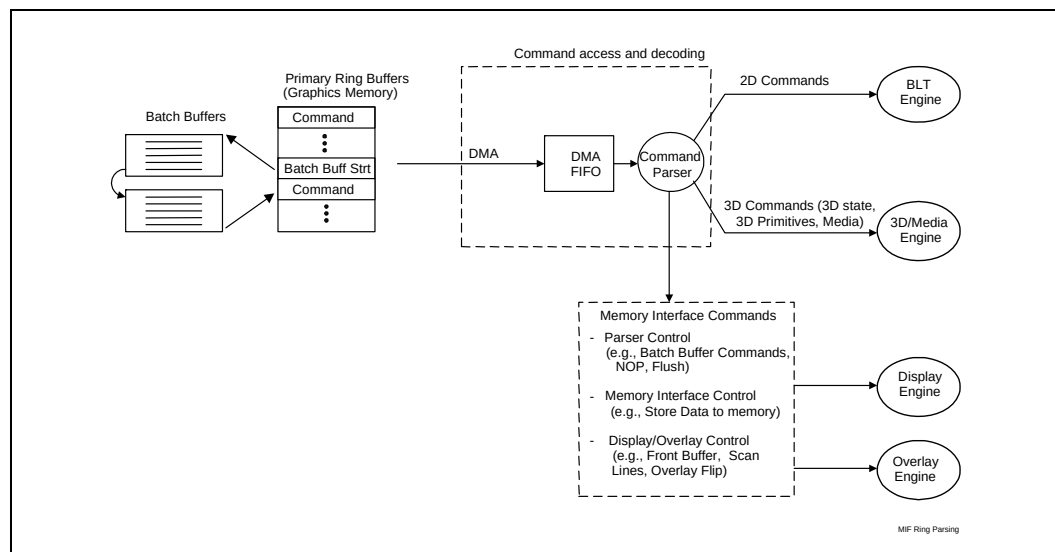
12.3.3 Command Parser

The graphics device's Command Parser (CP) is responsible for:

- Detecting the presence of commands (within the Ring Buffer).
- Reading commands from the Ring Buffer and Batch Buffers via DMA. This includes support of the automatic head report function.
- Parsing the common "Command Type" (destination) field of commands.
- Execution of Memory Interface commands that control CP functionality, provide synchronization functions, and provide display and overlay flips as well as other miscellaneous control functions.
- Redirection of 2D, 3D and Media commands to the appropriate destination (as qualified by the INSTPM register) while enforcing drawing engine concurrency and coherency rules.
- Performing the "Sync Flush" mechanism
- Enforcing the Batch Buffer protection mechanism

Figure 12-1 is a high-level diagram of the graphics device command interface.

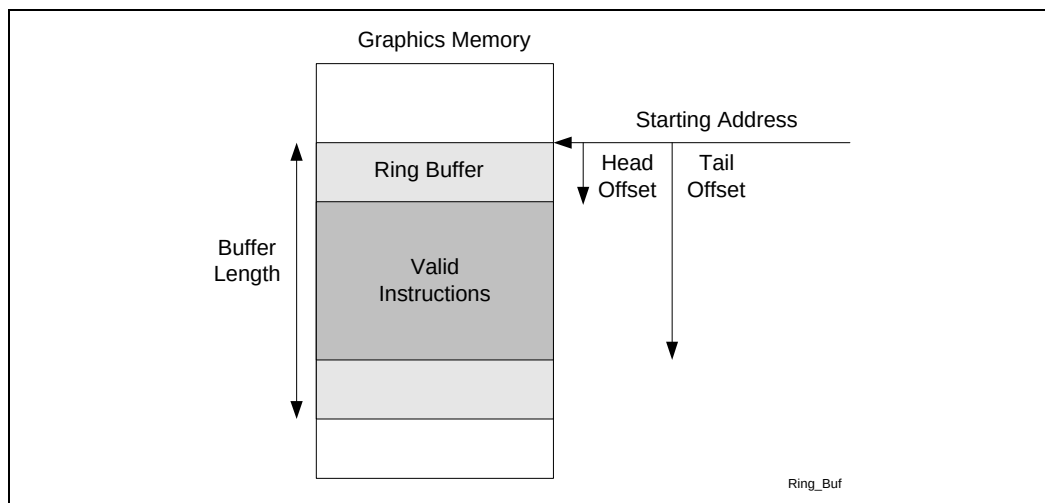
Figure 12-1. Graphics Controller Command Interface



12.3.4 The Ring Buffer

The ring buffer is defined by a set of Ring Buffer registers and a memory area that is used to hold the actual commands. The Ring Buffer registers (described in full below) define the start and length of the memory area, and include two "offsets" (head and tail) into the memory area. Software uses the Tail Offset to inform the CP of the presence of valid commands that must be executed. The Head Offset is incremented by the CP as those commands are parsed and executed. The list of commands can wrap from the bottom of the buffer back to the top. Also included in the Ring Buffer registers are control fields that enable the ring and allow the head pointer to be reported to cacheable memory for more efficient flow control algorithms.

Figure 12-2. Ring Buffer



12.3.4.1 The Ring Buffer (RB)

Ring Buffer support:

- Batch Buffer initiation
- Indirect Data (operand access)

12.3.4.2 Ring Buffer Registers

A Ring Buffer is defined by a set of 4 Ring Buffer registers. Before a Ring Buffer can be used for command transport, software needs to program these registers. The fields contained within these registers are as follows:

- **Ring Buffer Valid:** This bit controls whether the Ring Buffer is included in the command arbitration process. Software must program all other Ring Buffer parameters before enabling a Ring Buffer. Although a Ring Buffer can be enabled in the non-empty state, it must not be disabled unless it is empty. Attempting to disable a Ring Buffer in the non-empty state is UNDEFINED. Enabling or disabling a Ring Buffer does not of itself change any other Ring Buffer register fields.
- **Start Address:** This field points to a contiguous, 4KB-aligned, linear (i.e., must not be tiled), mapped graphics memory region which provides the actual command buffer area. Writing the Start Address has the side effect of clearing the Head Offset and Head Wrap Count fields.
- **Buffer Length:** The size of the buffer, in 4KB increments, up to 2MB.
- **Head Offset:** This is the DWord offset (from Start Address) of the next command that the CP will parse (i.e., it points one DWord past the last command parsed). The CP will update this field as commands are parsed – the CP typically continues parsing new commands before the previous command operations complete. (Note that, if commands are pending execution, the CP will likely have prefetched commands past the Head Offset). As the graphics device does not "reset" the Head Offset when a Ring Buffer is enabled, software must program the Head Offset field before enabling the Ring Buffer. Software can enable a Ring Buffer with any legal values for Head/Tail (i.e., can enable the Ring Buffer in a non-empty state). It is anticipated, but not required, that software enable The Ring



Buffer with Head and Tail Offsets of 0. Once the Head Offset reaches the QWord specified by the Tail Offset (i.e., the offsets are equal), the CP considers the Ring Buffer "empty".

- **Head Wrap Count:** This field is incremented by the CP every time the Head Offset wraps back to the start of the buffer. As it is included in the DWord written in the "report head" process, software can use this field to track CP progress as if the Ring Buffer had a "virtual" length of 2048 times the size of the actual physical buffer (up to 4GB).
- **Tail Offset:** This is the offset (from Start Address) of the next QWord of command data that software will request to be executed (i.e., it points one DWord past the last command DWord submitted for execution). The Tail Offset can only point to an command boundary – submitting partial commands is UNDEFINED. As the Tail Offset is a QWord offset, this requires software to submit commands in multiples of QWords (both DWords of the last QWord submitted must contain valid command data). Software may therefore need to insert a "pad" command to meet this restriction. After writing commands into the Ring Buffer, software updates the Tail Offset field in order to submit the commands for execution (by setting it to the QWord offset past the last command). The commands submitted can wrap from the end of the buffer back to the top, in which case the Tail Offset written will be less than the previous value. As the "empty" condition is defined as "Head Offset == Tail Offset", the largest amount of data that can be submitted at any one time is one QWord less than the Ring Buffer length.
- **IN USE Semaphore Bit:** This bit (included in the Tail Pointer register) is used to provide a HW semaphore that SW can use to manage access to the individual The Ring Buffer. See the Ring Buffer Semaphore section below.
- **Automatic Report Head Enable:** Software can request to have the hardware Head Pointer register contents written ("reported") to snooped system memory on a periodic basis. Auto-reports can be programmed to occur whenever the Head Offset crosses either a 64KB or 128KB boundary. (Note therefore that a Ring Buffer must be at least 64KB in length for the auto-report mechanism to be useful). The complete Head Pointer register will be stored at a Ring Buffer-specific DWord offset into the "hardware status page" (defined by the HWSTAM register). The auto-report mechanism is desirable as software needs to use the Head Offset to determine the amount of free space in the Ring Buffer -- and having the Head Pointer periodically reported to system memory provides a fairly up-to-date Head Offset value automatically (i.e., without having to explicitly store a Head Pointer value via the MI_REPORT_HEAD command).

Table 12-1. Ring Buffer Characteristics

Characteristic	Description
Alignment	4 KB page aligned.
Max Size	2 MB
Length	Programmable in numbers of 4 KB pages.
Start Pointer	Programmable 4KB page-aligned address of the buffer
Head pointer	Hardware maintained DWord Offset into the ring buffer. Commands can wrap. Programmable to initially set up ring.
Tail pointer	Programmable QWord Offset into the ring buffer – indicating the <i>next</i> QWord where software can insert new commands.



12.3.4.3 Ring Buffer Placement

Ring Buffer memory buffers are defined via a Graphics Address and must physically reside in (uncached) Main Memory. There is no support for The Ring Buffer in cacheable system memory.

12.3.4.4 Ring Buffer Initialization

Before initializing a Ring Buffer, software must first allocate the desired number of 4KB pages for use as buffer space. Then the Ring Buffer registers associated with the Ring Buffer can be programmed. Once the Ring Buffer Valid bit is set, the Ring Buffer will be considered for command arbitration, and the Head and Tail Offsets will either indicate an empty Ring Buffer (i.e., Head Offset == Tail Offset), or will define some amount of command data to be executed.

12.3.4.5 Ring Buffer Use

Software can write new commands into the "free space" of the Ring Buffer, starting at the Tail Offset QWord and up to the QWord prior to the QWord indicated by the Head Offset. Note that this "free space" may wrap from the end of the Ring Buffer back to the start (hence the "ring" in the name).

While the "free space" wrap may allow commands to be wrapped around the end of the Ring Buffer, the wrap should only occur between commands. Padding (with NOP) may be required to follow this restriction.

Software is required to use some mechanism to track command parsing progress in order to determine the "free space" in the Ring Buffer. This can be accomplished in one of the following ways:

1. A direct read (poll) of the Head Pointer register. This gives the most accurate indication but is expensive due to the uncached read.
2. The automatic reporting of the Head Pointer register in the Hardware Status Page. This has low impact as no uncached reads or command overhead is involved. However, given the 64KB/128KB granularity of auto-reports, this mechanism only works well on fairly large The Ring Buffer.
3. The explicit reporting of the Head Pointer register via the MI_REPORT_HEAD command. This allows for flexible and more accurate reporting but comes at the cost of command bandwidth and execution time, in addition to the software overhead to determine how often to report the head.
4. Some other "implicit" means by which software can determine how far the CP has progressed in retiring commands from a Ring Buffer. This could include the use of "Store DWORD" commands to write sequencing data to system memory. This has similar characteristics to using the MI_REPORT_HEAD mechanism.

Once the commands have been written and, if necessary, padded out to a QWord, software can write the Tail Pointer register to submit the new commands for execution. The uncached write of the Tail Pointer register will ensure that any pending command writes are flushed from the processor.

If the Ring Buffer Head Pointer and the Tail Pointer are on the same cacheline, the Head Pointer must not be greater than the Tail Pointer.



12.3.4.6 Ring Buffer Semaphore

When the **Ring Buffer Mutex Enable** (RBME) bit of the INSTPM MI register is clear, all Tail Pointer IN USE bits are disabled (read as zero, writes ignored). When RBME is enabled, the IN USE bit acts as a Ring Buffer semaphore. If the Tail Pointer is read, and IN USE is clear, it is immediately set after the read. Subsequent Tail Pointer reads will return a set IN USE bit, until IN USE is cleared by a Tail Pointer write.

This allows SW to maintain exclusive ring access through the following protocol: A SW agent needing exclusive ring access must read the Tail Pointer before accessing the Ring Buffer: if the IN USE bit is clear, the agent gains access to the Ring Buffer; if the IN USE bit is set, the agent has to wait for access to the Ring Buffer (as some other agent has control). The mechanism to inform pending agents upon release of the IN USE semaphore is unspecified (i.e., left up to software).

12.3.5 Batch Buffers

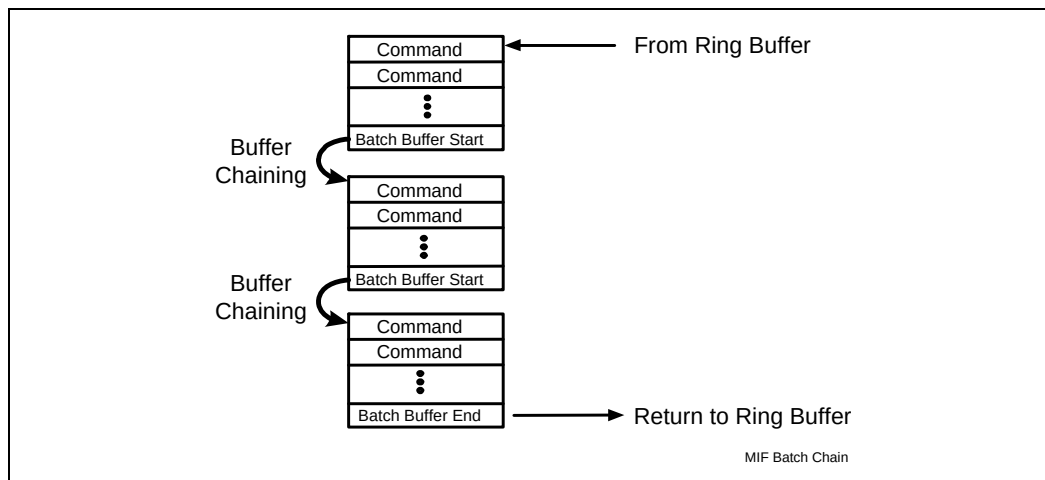
The graphics device provides for the execution of command sequences *external* to the Ring buffer. These sequences are called "Batch Buffers", and are initiated through the use of various Batch Buffer commands described below. When a Batch Buffer command is executed, a batch buffer sequence is initiated, where the graphics device fetches and executes the commands sequentially via DMA from the batch buffer memory.

12.3.5.1 Batch Buffer Chaining

What happens when the end of the Batch Buffer is reached depends on the final command in the buffer. Normally, when a Batch Buffer is initiated from a Ring Buffer, the completion of the Batch Buffer will cause control to pass back to the Ring Buffer at the command following the initiating Batch Buffer command.

However, the final command of a Batch Buffer can be another Batch Buffer-initiating command (MI_BATCH_BUFFER_START). In this case control will pass to the new Batch Buffer. This process, called *chaining*, can continue indefinitely, terminating with a Batch Buffer that does not chain to another Batch Buffer (ends with MI_BATCH_BUFFER_END) – at which point control will return to the Ring Buffer.

Figure 12-3. Batch Buffer Chaining





12.3.5.2 Ending Batch Buffers

The end of the Batch Buffer is determined as the buffer is being executed: either by (a) an MI_BATCH_BUFFER_END command, or (b) a “chaining” MI_BATCH_BUFFER_START command. There is no explicit limit on the size of a Batch Buffer that uses GTT-mapped memory. Batch buffers in physical space cannot exceed one physical page (4KB).

12.3.6 Indirect Data

In addition to Ring Buffer and Batch Buffers, the MI supports the access of *indirect* data for some specific command types. (Normal read/write access to surfaces isn't considered indirect access for this discussion).

12.3.6.1 Logical Contexts

Logical Contexts, indirectly referenced via the MI_SET_CONTEXT command, must reside in (unsnooped) Main Memory.

12.3.7 Command Arbitration

The command parser employs a set of rules to arbitrate among these command stream sources. This section describes these rules and discusses the reasoning behind the algorithm.

12.3.7.1 Arbitration Policies and Rationale

The Ring buffer (RB) is considered the primary mechanism by which drivers will pass commands to the graphics device.

The insertion of command sequences into the Ring Buffer must be a “synchronous” operation, i.e., software must guarantee mutually exclusive access to the Ring Buffer among contending sources (drivers). This ensures that one driver does not corrupt another driver's partially-completed command stream. There is currently no support for unsynchronized multi-threaded insertion of commands into ring buffer.

Another requirement for asynchronous command generation arises from competing (and asynchronous) drivers (e.g., “user-mode” driver libraries). In this case, the desire is to allow these entities to construct command sequences in an asynchronous fashion, via batch buffers. Synchronization is then only required to “dispatch” the batch buffers via insertion of Batch Buffer commands inserted into the Ring Buffer.

Software retains some control over this arbitration process. The MI_ARB_ON_OFF command disables all other sources of command arbitration until re-enabled by a subsequent MI_ARB_ON_OFF command from the same command stream. This can be used to define uninterruptible “critical sections” in an command stream (e.g., where some device operation needs to be protected from interruption). Disabling arbitration from a batch buffer without re-enabling before the batch is complete is UNDEFINED.

Batch Buffers can be (a) interruptible at command boundaries, (b) interruptible only at chain points, or (c) non-interruptible. See MI_BATCH_BUFFER_START in *Memory Interface Commands* for programming details.



12.3.7.2 Wait Commands

The MI_WAIT_EVENT command is provided to allow command streams to be held pending until an asynchronous event occurs or condition exists. An *event* is defined as occurring at a specific point in time (e.g., the leading edge of a signal, etc.) while a *condition* is defined as a finite period of time. A wait on an event will (for all intents and purposes) take some non-zero period of time before the subsequent command can be executed. A wait on a condition is effectively a noop if the condition exists when the MI_WAIT_EVENT command is executed.

A Wait in the Ring Buffer or batch buffer will cause the CP to treat the Ring Buffer as if it were empty until the specific event/condition occurs. This will temporarily stall the Ring Buffer.

While the Ring Buffer is waiting, the **RB Wait** bit of the corresponding RBN_CTL register will be set. Software can cancel the wait by clearing this bit (along with setting the **RB Wait Write Enable** bit). This will terminate the wait condition and the Ring Buffer will be re-enabled. This sequence can be included when software is required to flush all pending device operations and pending Ring Buffer waits cannot be tolerated.

12.3.7.3 Wait Events/Conditions

This section describes the wait events and conditions supported by the MI_WAIT_EVENT command. Only one event or condition can be specified in an MI_WAIT_EVENT, though different command streams can be simultaneously waiting on different events.

12.3.7.3.1 Display Pipe A,B Vertical Blank Event

The Vertical Blank event is defined as “shortly after” the *leading edge* of the next display VBLANK period of the corresponding display pipe. The delay from the leading edge is provided to allow for internal device operations to complete (including the update of display and overlay status bits, and the update of overlay registers).

12.3.7.3.2 Display Pipe A,B Horizontal Blank Event

The Horizontal Blank event is defined as “shortly after” the *leading edge* of the next display HBLANK period of the corresponding display pipe.

12.3.7.3.3 Display Plane A, B, C , Flip Pending Condition

The Display Flip Pending condition is defined as the period starting with the execution of a “flip” (MI_DISPLAY_BUFFER_INFO) command and ending with the completion of that flip request. Note that the MI_DISPLAY_BUFFER_INFO command can specify whether the flip should be synchronized to vertical refresh or completed “as soon as possible” (likely some number of horizontal refresh cycles later).



12.3.7.3.4 Overlay Flip Pending Condition

The Overlay Flip Pending condition is similar to the Display Flip Pending condition, with the exception that overlay flips are only performed synchronously with display refresh.

12.3.7.3.5 Display Pipe A,B Scan Line Window Conditions

The graphics device supports two conditions relating to the progress of refresh within a particular display stream. A “Scan Line Window” is defined as the period of time between the refresh of two specific display scan lines. The MI_WAIT_ON_EVENT command can be used to pause an command stream while a particular display refresh is inside or outside the Scan Line Window. (Actually, the MI_WAIT_EVENT command only supports waiting on the Scan Line Window condition, and the MI_LOAD_SCAN_LINES_INCL or MI_LOAD_SCAN_LINES_EXCL are used to define an “inclusive” or “exclusive” window).

If no Scan Line Window has been defined for the particular display stream, the MI_WAIT_EVENT specifying the Scan Line Window event will never introduce a wait.

12.3.7.3.6 Semaphore Wait Condition

One of the 8 defined condition codes contained within the Execute Condition Code (EXCC) Register can be selected as the source of a wait condition. While the selected condition code bit is set, the initiating command stream will be removed from arbitration (i.e., paused). Arbitration of that command stream will resume once the condition code bit is clear. If the selected condition code is clear when the WAIT_ON_EVENT is executed, the command is effectively ignored.

12.3.7.4 Command Arbitration Points

The CP performs arbitration for command execution at the following points:

- Upon execution of an MI_ARB_CHECK command
- When the ring buffer becomes empty

12.3.7.5 Command Arbitration Rules

- At an arbitration point, the CP will switch to the new head pointer contained in the UHPTR register if it is valid. Otherwise it will idle if empty, or continue execution in the current command flow if it arbitrated due to an MI_ARB_CHECK command.

12.3.7.6 Batch Buffer Protection

The CP employs a protection mechanism to help prevent random writes to system memory from occurring as a result of the execution of a batch buffer generated by a “non-secure” agent (e.g., client-mode library). Commands executed directly from a ring buffer, along with batch buffers initiated from a ring buffer and marked as “secure”, will not be subject to this protection mechanism as it is assumed they can only be generated by “secure” driver components.

This protection mechanism is enabled via a field in a Batch Buffer command that indicates whether the associated batch buffer is “secure” or “non-secure”. When the



CP processes a non-secure batch buffer from the ring buffer it does not allow any MI_STORE_DATA_IMM commands that reference physical addresses, as that would allow the non-secure source to perform writes to any random DWORD in the system. (Note that graphics engines will only write to graphics memory ranges, which by definition are virtual memory ranges mapped into physical memory pages using the GTT hardware). Placing an MI_STORE_DATA in a non-secure batch buffer will instead cause a Command Error. The CP will store the header of the command, the origin of the command, and an error code. In addition, such a Command Error can generate an interrupt or a hardware write to system memory (if these actions are enabled and unmasked in the IER and IMR registers respectively.) At this point the CP can be reactivated only by a **full reset**.

The security indication field of Batch Buffer instructions placed in batch buffers (i.e., “chaining” batch buffers) is ignored and the chained batch buffer will therefore inherit the security indication of the first Batch Buffer in the chain (i.e. the batch buffer that was initiated by an MI_BATCH_BUFFER_START command in the Ring Buffer).

12.3.8 Graphics Engine Synchronization

This table lists the cases where engine synchronization is required, and whether software needs to ensure synchronization with an explicit MI_FLUSH command or whether the device performs an implicit (automatic) flush instead. Note that a pipeline flush can be performed without flushing the render cache, but not vice versa.

Event	Implicit Flush or Requires Explicit Flush?
PIPELINE_SELECT	Requires explicit pipeline flush
Any Non-pipelined State Command	Device implicitly stalls the command until the pipeline has drained sufficiently to allow the state update to be performed without corrupting work-in-progress
MI_SET_CONTEXT	Device performs implicit flush
MI_DISPLAY_BUFFER_INFO ("display flip")	Requires explicit render cache flush
MI_OVERLAY_FLIP	Requires explicit render cache flush
3D color destination buffer (render target) used as texture (i.e., "rendered texture")	Requires explicit render cache flush
MEDIA_STATE_POINTERS	Requires explicit pipeline flush
MEDIA_OBJECT	Requires explicit pipeline flush
Media: Previous Destination Used as Source	Requires explicit render cache flush



12.3.9 Graphics Memory Coherency

Table 12-2 lists the various types of graphics memory coherency provided by the device, specifically where the CPU writes to a 64B cacheline, and the device then accesses that same cacheline. Note that the coherency policy depends on the address type (GM or MM) involved in the accesses.

Table 12-2. Graphics Memory Coherency

CPU Access	Subsequent Device Access	Example Operand	Coherency
Write GM	Read GM		TBD
Write MM	Read MM	Batch Buffer	TBD
Write GM	Write GM		Device ensures coherency following every Ring Buffer Tail Pointer write. (This can be made optional via a bit in the Tail Pointer data).
Write MM	Write MM		TBD “assumed to exclusive byte” ?
Write GM	Read MM		Device ensures coherency following every Ring Buffer Tail Pointer write. (This can be made optional via a bit in the Tail Pointer data).

12.3.10 Graphics Cache Coherency

There are several caches employed within the graphics device implementation. This section describes the impact of these caches on the programming model (i.e., if/when does software need to be concerned).

12.3.10.1 Rendering Cache

The rendering (frame buffer) cache is used by the blit and 3D rendering engines and caches portions of the frame buffer color and depth buffers. This cache is guaranteed to be flushed under the following conditions (note that the implementation may flush the cache under additional, implementation-specific conditions):

- Execution of an MI_FLUSH command with the **Render Flush Cache Inhibit** bit clear
- Execution of a PIPE_CONTROL instruction with the **Write Cache Flush Enable** bit set (Depth Stall must be clear).
- A SyncFlush handshake operation
- A change of rendering engines (e.g., going from 2D to 3D, 3D to 2D, etc.)
- Logical Context switch (via MI_SET_CONTEXT)
 - The render cache must be explicitly flushed using one of these mechanisms under certain conditions. See Graphics Engine Synchronization above.



12.3.10.2 Sampler Cache

The read-only sampler cache is used to cache texels and other data read by the Sampling Engine in the 3D pipeline. This cache can be enabled or disabled via the **Texture L2 Disable** bit of the Cache_Mode_0 register (see *Memory Interface Registers*). Note that, although there may be more than one level of sampler cache within the implementation, the sampler cache is exposed as a single entity at the programming interface.

The sampler cache is guaranteed to be invalidated under the following conditions (note that the implementation may invalidate the cache under additional, implementation-specific conditions):

- Execution of an MI_FLUSH command with the **Map Cache Invalidate** bit set
- Execution of PIPE_CONTROL with the **Depth Stall Enable** bit clear.
- A SyncFlush handshake operation

The sampler cache must be invalidated prior to reallocation of physical texture memory (i.e., software must guarantee that stale texture data is invalidated before reusing physical texture memory for a new or modified texture).

12.3.10.3 Instruction / State Cache

The read-only ISC is used to cache pipelined state and EU instructions read in from memory. It also functions as a prefetch cache by reading in additional state information and instructions beyond those immediately requested in order to decrease latency and improve performance. As with the sampler cache, there may be more than one level of ISC within the implementation. The ISC is exposed as a single entity at the programming interface.

The instruction/state cache is guaranteed to be invalidated under the following conditions (note that the implementation may invalidate the cache under additional, implementation-specific conditions):

- Execution of an MI_FLUSH command with the **State / Instruction Cache Invalidate** bit set
- Execution of PIPE_CONTROL with the Instruction/State Cache Flush Enable bit set.
- A SyncFlush handshake operation

The instruction/state cache must be invalidated prior to reallocation of physical state/instruction memory (i.e., software must guarantee that stale state/instruction data is invalidated before reusing physical state/instruction memory for new or modified state or instructions).



12.3.10.4 Vertex Cache

The vertex cache consists of 2 sub-caches: one that caches vertex buffer data based on address, and another that caches (possibly shaded) vertex attribute data based on index (see the *Vertex Fetch* chapter for vertex index details). The latter cache is always invalidated between primitive topologies.

Both vertex caches are guaranteed to be invalidated under the following conditions (note that the implementation may invalidate the cache under additional, implementation-specific conditions):

- Execution of an MI_FLUSH command
- Execution of a PIPE_CONTROL command
- A SyncFlush handshake operation
- Logical Context switch (via MI_SET_CONTEXT)

12.3.10.5 GTT TLBs

The following table summarizes when the various TLBs are invalidated.

TLB	Normal Invalidation Mechanism
Display	Refreshed on Vsync
Overlay	Refreshed on Vsync
Render/Blit	Internal Flush*
Host	Through a Page Table PTE write
Sampler Cache	Internal Flush*
Command Stream	Through a Page Table PTE write

NOTE: * -- Includes MI_FLUSH, Engine switch, and Context switch.

12.3.11 Command Synchronization

This section describes the hardware mechanisms that can be used by software to provide synchronization with command stream parsing and execution.

The key point here is distinguishing between command *parsing* and *retirement* – in that, for most commands, there is some finite delay between the parsing of a command and the retirement (coherent completion) of the operation it specifies.

Interrogation of the Ring Buffer Head Pointer only gives an indication of the progress of command parsing. This information is required to discern the availability of command data within the Ring Buffer or Batch Buffers. If the Head Pointer indicates the command data has been parsed, those locations can be reused; otherwise the commands must be considered still pending parsing and left alone.

Given the CP rules for command execution, it is possible to use the indication of command parsing progress to infer the retirement status of parsed commands. The



only indication of instruction retirement available from instruction parsing is that parsing of an MI instruction implies retirement of previous MI instructions with the following exceptions:

- The parsing of a Memory Interface (MI) command implies that all previously-parsed MI commands have completed, with the following exceptions:
 - Display and Overlay Flip commands: Only the submission of the flip request is guaranteed. The flip operation will occur some time later. Mechanisms to detect the actual completion of a flip operation are described below.
 - “Store-Data” type commands: Only the submission of the store operation is guaranteed. The write result will be complete (coherent) some time later (this is practically a finite period but there is no guaranteed latency).
 - Batch Buffer commands: There is no guarantee that the operations performed by the batch buffer have completed.

Other than the cases described above, additional measures must be taken to discern the progress of command retirement. These measures are described in the following subsections.

12.3.11.1 MI_FLUSH

The MI_FLUSH command pauses further command parsing until all drawing engines become idle and any internal rendering cache is flushed and invalidated. All previous rendering commands can therefore be considered retired.

This flush operation is considered complete once command parsing proceeds to the next command. Software can, for example, follow an MI_FLUSH command with an MI_STORE_DATA_IMM or MI_STORE_DATA_INDEX command – where the completion of the store operation implies that the flush operation has completed. (Note that if the last DWORD in a ring buffer is an MI_FLUSH instruction, there is no way by simply looking at the Ring Buffer registers to determine whether the flush operation is complete or still pending.)

The successful completion of an MI_FLUSH command does not guarantee that *all* previous operations have completed. Operations that may still be pending include:

- Store Data type commands (MI_STORE_DATA_IMM, MI_STORE_DATA_INDEX, MI_REPORT_HEAD)
- Display or Overlay Flip operations

See section 12.3.10.2 for more information on when the sampler cache should be invalidated.

12.3.11.2 Sync Flush

Inserting MI_FLUSH commands, while effective at determining or forcing the retirement of previous rendering commands, may negatively impact performance if not absolutely required. For example, if the knowledge of rendering command retirement is not known a priori, it is likely undesirable to insert MI_FLUSH commands at intervals in the command stream. However, it may not be acceptable to insert an MI_FLUSH command (and wait for its completion) at the point that rendering command retirement is required – as there may be a large number of commands pending in ring/batch buffers at that point and flushing the entire device (including waiting for completion of pending commands that have not yet been parsed) may be prohibitive. There is a mechanism, however, where command stream synchronization



can be performed on demand, without requiring earlier submitted commands and batch buffers to complete – it is called the “Sync Flush” mechanism.

Here's how it works:

- Software must (preferably at driver initialization time) unmask the Sync Status bit in the Hardware Status Mask Register (HWSTAM). This should be done unconditionally (at least whenever HW status writes are enabled), as any bandwidth increase due to Sync Status-initiated writes is negligible.
- At the point that synchronization is required, software must guarantee that command parsing has progressed past the point of interest in the command stream (i.e., past the last command whose retirement is required). Note that this step is required in any scheme.
- Software then reads the location where the Interrupt Status is reported in the Hardware Status Page (DWord offset 0) and saves that DWord in a temporary variable.
- Software then sets the Sync Enable bit of the Command Parser Mode Register (INSTPM) via an uncached write.
- The Command Parser will detect the Sync Enable bit set before it proceeds to the very next command (or immediately if the CP is idle). It will then perform an internal flush operation. This flush is identical to that performed by an MI_FLUSH command with all flush types enabled.
- Once this flush operation is complete, the CP will clear the Sync Enable bit of the INSTPM register and then *toggle* the Sync Status bit of the ISR register. This will initiate a write of the ISR register contents (with the toggled Sync Status) to DWord 0 of the Hardware Status page (as part the normal hardware status write mechanism).
- Software, following the write of the INSTPM register, should periodically poll the Hardware Status location. By comparing the current versus saved value of the Sync Status bit, software can then detect when the flush operation is complete. Note that the latency of this operation is typically small, as it will be initiated either immediately or at least before the next command is parsed (regardless of arbitration conditions).

12.4 Hardware Status

The graphics device supports a number of internal hardware status bits which can be used to detect and monitor hardware status conditions via polling or interrupts. This section will describe each hardware status bit. The following section describes the hardware status reporting (polling) mechanism. The mechanism to allow these status bits to generate interrupts is described in the Interrupts section. Note that the hardware status bits are actually reported in the Interrupt Status Register, so “hardware status” and “interrupt status” are used interchangeably here (though many hardware status bits won't necessarily ever be used to generate interrupts).

The following subsections describe the various hardware (interrupt) status bits, as defined in the Interrupt Status Register.



12.4.1 Hardware-Detected Errors (Master Error bit)

This interrupt status bit is generated whenever an “unmasked” hardware-detected error status is detected. See Errors.

12.4.2 Thermal Sensor Event

This interrupt status bit is generated by “thermal events” detected by the Thermal Sensor logic. The bit corresponding to this event in the HWSTAM register must always be masked (i.e., set to ‘1’) so that thermal sensor events do not generate HW status DWord writes. See Hardware Status Writes.

12.4.3 Sync Status

This bit should only be used as described in Sync Flush, and should not be used to generate interrupts (i.e., the corresponding interrupt should not be enabled in the IER).

12.4.4 Display Plane A, B, Flip Pending

These bits are used to report the status of “flip” operations on the corresponding Display Plane. Display Flip operations are requested via the MI_DISPLAY_BUFFER_INFO command. When that command is executed, the corresponding Display Flip Pending status in the ISR register will be set to ‘1’ indicating that a display flip has been requested but has not yet been performed. (Requesting a flip operation when one is already pending is UNDEFINED). This indicates that a flip is “pending”. At the appropriate time during the next vertical blank period (for that display stream), the flip operation will be performed (i.e., the display will switch to refreshing from the new display buffer). This causes the Display Flip Pending status to reset to ‘0’. When this occurs, and the Display Flip Pending status bit is unmasked by the Interrupt Mask Register (IMR), the Display Flip Pending status bit of the Interrupt Identity Register (IIR) is set. Note that this setting of an interrupt identity bit on the falling edge of the status bit is contrary to the general definition of interrupt status bits.

12.4.5 Overlay Flip Pending

This bit is similar to the Display Flip Pending bits. It is set to ‘1’ when the MI_OVERLAY_FLIP command is executed. It is cleared to ‘0’ after the overlay registers are read from memory during the next vertical blanking period.

12.4.6 Display Pipe A,B VBLANK

These bits are set on the leading edge of the selected Display Pipe’s VBLANK signal.



12.4.7 User Interrupt

This bit is set in response to the execution of an MI_USER_INTERRUPT command. The Command Parser will continue parsing after processing that command. If a user interrupt is currently outstanding (set in the ISR) this packet has no effect.

Programming Note: User interrupts can be used to notify software of the progress of instruction parsing past the MI_USER_INTERRUPT instruction. In particular, user interrupts can be inserted into the command stream but effectively disabled for “normal operation” via the IMR and HWSTAM registers. Whenever software requires the notification afforded by the user interrupts, it can unmask this bit.

12.4.8 PIPE_CONTROL Notify Interrupt

This bit is set when a PIPE_CONTROL command with the **Notify Enable** bit set reaches the end of the pipeline and all required cache flushes have occurred.

12.4.9 Display Port Interrupt

This bit is set on a hot plug event. See the *Display Registers* chapter for details.

12.5 Hardware Status Writes

The graphics device supports the writing of the hardware status (ISR) bits into memory for optimized access from software. Software can select which (if any) status bits will trigger the write of the ISR contents to memory using the Hardware Status Mask (HWSTAM) register. Writing a '0' to a defined bit position in the HWSTAM register will cause any change (0 → 1 or 1 → 0) in the corresponding ISR bit to trigger the write. The complete ISR contents will be written to DWord offset 0 of the hardware status page, located at the address programmed via the Hardware Status Page Address Register (HWS_PGA).

12.6 Interrupts

The graphics device supports the generation of an interrupt. This interrupt can be raised in response to one or more internal interrupt status conditions. Which interrupt status conditions are allowed to raise an interrupt is programmed via the Interrupt Mask Register (IMR) and Interrupt Enable Register (IER). The IMR is used to selectively “unmask” hardware status bits as to allow them to be reported in the Interrupt Identity Register (IIR). The IER holds a set of interrupt enable bits corresponding to each bit of the IIR – setting bits in the IER will allow interrupts to be generated by the corresponding bits in the IIR.



12.7 Errors

The graphics device supports the hardware detection of a number of *operational* and *debug-only* errors. Operational errors occur out of the immediate control of driver software and must be anticipated and tolerated to the extent required by the relevant APIs. Software must therefore support the detection and proper handling of all relevant operational errors. The (more numerous) debug-only errors are just that – detected to facilitate initial system debug but not intended to be tolerated during normal system operation. In many cases, debug-only errors are not recoverable. They require the use of debug registers to detect and diagnose.

12.7.1 Error Reporting

Regardless of the error classification, all errors funnel through the **Master Error** bit of the Interrupt Control Registers. This bit can be used to raise a device interrupt or trigger a hardware status write operation. (Needless to say it can also be polled directly, though this is clearly discouraged). Refer to Interrupt Control Registers in the *Memory Interface Registers* chapter for more information.

There are three registers dedicated to control, detect, and clear hardware error status conditions in a similar fashion to the Interrupt Control Registers. All three error registers share a common error status bit definition.

The Error Status Register (ESR) holds the actual error status bits (each of which may be the logical OR of “source” error bits in various functional registers). The Error Mask Register (EMR) is used to select which error status bit(s) are reported in the Error Identity Register (EIR). The EIR holds the “persistent” values of the unmasked error status bits, and is also used to clear error status conditions. Any bits set in the EIR will raise the Master Error interrupt status condition.

The error conditions corresponding to the error status bits include:

- **Page Table Error (*Debug only*)** – This is a summary of a number of possible errors associated with the mapping function of the GTT. See Table 12-3 for more information.
- **Display or Overlay Underrun (*Debug only*)** – This error is raised when a FIFO underrun condition is detected in the display or overlay isochronous streams. See the description of the Display/Overlay Status Register in the *Display Registers* chapter.
- **Command Error (*Debug Only*)** – This is a summary of a number of command data errors detected by the Command Parser. See Command Errors below for more information.



12.7.2 Page Table Errors

The following tables describe the various sources and types of Page Table Errors. Refer to the description of the PGTBL_ERR register in *Memory Interface Registers* for more details.

Table 12-3. Page Table Error Types

Error	Description	Streams
Invalid GTT PTE	In the process of mapping an address, the MI encountered a GTT PTE that was marked "Invalid". This would be the result of a programming error.	All (See Table 11-1)
Invalid TLB Miss	An unexpected TLB miss (detected at GTT request time) was encountered (e.g., during Display/Overlay/Sprite access).	Display, Overlay
Invalid PTE Data	Mapping to the physical page specified in the PTE is not permitted (e.g., a page in PAM, SMM or over the top of memory, etc.). This is the result of a programming error.	Host
Invalid Tiling	A tiling parameter was found inconsistent with the current operation. This includes the use of Y-Major tiling in the Render/Display/Overlay streams. This is the result of a programming error. This is detected during GTT request.	Blt, Display, Overlay

NOTE: Note that Page Table Errors cannot be cleared. A device reset is required.

12.7.3 Clearing Errors

For operational errors, software is responsible for taking the proper steps to recover from the error and then clearing the error indication. The actions required to recover from operational errors may be discussed in the various functional areas (not here). See the Hardware-detected Error Bit Definitions in *Memory Interface Registers* for more details. This subsection describes the actions required to clear the error indication.

In order to clear operational errors, software is responsible for clearing the error condition from the source, working back to the Master Error bit. Typically this will entail the following sequence.

- First the primary source of the error must be cleared. This requires clearing the functional register(s) containing the source error indication.
- Next, clear the particular error status bit by writing a '1' to the appropriate bit of the Error Identity Register (EIR). This will clear the error status bit in the Error Status Register (ESR). If multiple errors are present, all error status bits should be cleared simultaneously.
- Next, clear the Master Error interrupt status bit by writing a '1' to the Master Error bit of the Interrupt Identity Register (IIR).

Note: Page Table Errors cannot be cleared.



12.8 Rendering Context Management

The graphics device operation (rendering, etc.) is controlled via the settings of numerous hardware state variables. These state variables are divided into *global state* and *context state*.

There is only one copy of global state variables, and changing the settings of these variables requires explicit programming of the state variables. Examples of global state include:

- MI registers (HWSTAM, Ring Buffer, etc.) with the exception of those listed in the next paragraph (i.e, registers listed there *are* saved/restored)
- Configuration registers
- Display programming registers

On the other hand, context state is associated with a specific *context*, where switching to that context causes that context's state to be restored. While the associated context is active, the state variables and registers can be programmed via the command stream. Examples of context state include the PIPELINE_STATE_POINTERS command and most non-pipelined state. The following MI registers are considered part of context state and thus saved/restored with context:

- INSTPM
- CACHE_MODE_0
- CACHE_MODE_1
- MI_ARB_STATE
- 3D Software Visible Counter Registers

The graphics device supports both a *hardware context* and *logical contexts*. The multiple logical context support provides robust rendering context support by swapping contexts to/from memory.

12.8.1 Multiple Logical Rendering Contexts

The graphics device supports multiple *logical rendering contexts* stored in Main Memory. Logical rendering contexts are referenced via a 2KB-aligned *Logical Context Address*.

The maximum size of a logical context entry (which is information required by the driver to allocate contexts) is currently 2K bytes. For forward compatibility, the maximum size of a logical context entry should be supplied to the drivers via a VBIOS mechanism as opposed to being hardcoded in the driver.

The actual size of a logical rendering context is the amount of data stored/restored during a context switch and is measured in 64B cache lines. There is a debug mechanism that allows software/BIOS to program the actual size of the logical rendering context via the CXT_SIZE register. Note that this register will default to the correct value, so software should not have to modify it.



The format of the logical rendering context in memory is considered device-dependent; software must not attempt to modify the contents of a logical rendering context directly. This restriction is motivated by forward compatibility concerns because the location and definition of fields may change between implementations.

12.8.1.1 Current Context IDs

The ring buffer has an associated *Current Context ID* (CCID) register. The CCID includes a Logical Pipeline Context Address (LPCA).

The CCID for a ring buffer is set during the processing of the new MI_SET_CONTEXT command from that ring. The MI_SET_CONTEXT command provides a new CCID value (LPCA) to be loaded into the CCID register for the associated ring buffer. The MI_SET_CONTEXT command also contains a Restore Inhibit bit used to optionally inhibit the restoration (loading) of the new rendering context. This bit must be used during context initialization to avoid the loading of uninitialized (garbage) context data from memory. Failure to do so leads to UNDEFINED operation.

The initial values of the CCIDs are UNDEFINED. The first time a valid CCID is set from a ring buffer, the normal context save operation will be suppressed, as the previous CCID is invalid.

12.8.1.2 Intra-Ring Context Switch

Within a specific ring buffer, a new logical rendering context is specified via the MI_SET_CONTEXT command. Note that MI_SET_CONTEXT commands are permitted only within a ring buffer (not within a batch buffer).

As part of the execution of the MI_SET_CONTEXT command from within a ring buffer, the Logical Pipeline Context Address fields of the CCID register and MI_SET_CONTEXT command are compared. If they differ (or the CCID register is uninitialized), a rendering context switch operation will be performed, which includes:

1. If the CCID contents are valid, a context save operation will be performed. The contents of the HW context will be saved in memory starting at the Logical Pipeline Context Address specified in the CCID.
2. If the Restore Inhibit command field is not set, a context restore operation will be performed. Here the logical context values are read starting from the Logical Pipeline Context Address field of the command and used to set the internal HW context.
3. The relevant contents of the command will be loaded into the appropriate CCID register. (This occurs irrespective of the LPCA comparison result). At this point, the ring buffer has switched to using the new logical rendering context.



12.8.1.3 Logical Rendering Context Creation and Initialization

12.8.1.3.1 Rendering Context Creation Rules

1. Software only knows the **size** of the logical rendering context (2KB), for allocation purposes.
2. Given (1), software does **not** know the format of the context, and therefore is not allowed to write any portion of a logical rendering context. Software can, however, copy/move entire logical context blocks.
3. Given (2), software must never restore (load) a logical rendering context from memory that has not been previously stored by HW. I.e., software must never attempt to initialize a context itself and then cause it to be loaded. Breaking this rule causes UNDEFINED operation (as in the hang seen in BDG validation).
4. Initialization software must write **all** HW context variables with legal values before the first rendering context can be saved (this must be done before you can perform any rendering anyways). Given this, and the obvious rule that software must never program illegal state values, guarantees that the HW context will forever remain valid (and therefore be available to store into a logical rendering context). Note that software-visible context variables include 3D state, Blt register state, etc.

12.8.1.3.2 Context Initialization

Logical Rendering Contexts can be initialized (in memory) by software in the following way:

1. Issue an MI_SET_CONTEXT command w/ the **Restore Inhibit** bit set and the about-to-be-initialized logical pipeline context address. This will save the current rendering context and then change the LPCA to the new context (without loading it).
2. Use state commands to modify the context as desired.
3. Issue another MI_SET_CONTEXT command specifying some other LPCA (e.g., the previous one). This will cause the new context to be stored (initialized) in memory

12.8.1.4 Context Save

A context save will occur anytime all of the following apply:

- A rendering context switch occurs as a result of the execution of MI_SET_CONTEXT
- the CCID of the current context (CCID register of current ring) and the new CCID (the CCID register of the newly selected ring or the new CCID in the MI_SET_CONTEXT command) differ OR an MI_SET_CONTEXT with the “Force Restore” bit set initiated the context switch
- the current CCID is valid (has been previously set)

The current rendering context will be written out to memory starting at the LPCA in the format described by Logical Context Layout in *Memory Data Formats*. Note that this includes a limited number of Memory Interface Registers whose values are saved by embedding them in an MI_LOAD_REGISTER_IMM command that is written out to memory.



The Optional Extended Context will also be written if the Extended Save Enable bit is set in the current CCID register. Context saves DO NOT modify pipelined state stored in memory.

12.9 Reset State

This section describes the state of the programming interface following a hardware reset. Refer to the individual register definitions for details on reset (default) settings.

- The settings of the hardware context state variables are UNDEFINED. Software must program all state variables prior to their use in rendering.
- The ring buffer is disabled.
- All interrupts and error status bits are “masked” (disabled). All interrupts are disabled via IER. There will be no HW activity to cause any hardware/interrupt status bits to be set.
- The Hardware Status Page is located at 1FFFF000h (though HW status writes are effectively disabled)
- All FENCE registers are INVALID
- The GTT is disabled (accesses other than CPU reads, cursor and VGA reads will generate an error).
- All INSTDONE bits are set (“DONE”).
- The NOPID register is 0.
- All command groupings are enabled (via INSTPM)



13 *Frame Buffer Compression* (*[DevCL] Only*)

13.1 Overview

The Run-Length Encoded Frame Buffer Compression (RLE-FBC) function is a mechanism to reduce display refresh memory traffic. By reducing memory reads required for display refresh, power consumption is reduced (thus extending battery life for mobile systems).

The conditions under which the RLE-FBC is most effective are:

- Display images that are well suited to RLE compression. Good examples are text windows, slide shows, etc. Poor examples are 3D games - rich in textured and smooth-shaded objects.
- Screens that are fairly static. Good examples are screens with significant portions of the background showing, 2D apps (reading mail, etc.), CPU benchmarks, etc., or conditions when the CPU is idle. Poor examples are full-screen 3D games and benchmarks that flip the display image at or near display refresh rates.

Note that this compression function is different from, and mutually exclusive with, Discard Alpha Frame Buffer Compression – which is effective for 32bpp 3D environments.

The RLE-FBC function is comprised of three subfunctions:

- A **Compressor** that attempts to compress the display buffer as a background task.
- A **Decompressor** in the Display engine that uses compressed lines for display refresh, if available.
- A **Frame Buffer Write Detector** that snoops writes to the uncompressed frame buffer and invalidates the corresponding compressed lines.

The RLE-FBC **Compressor** periodically compresses lines of Display Plane A (an uncompressed display source image) using run-length encoding and stores the results into a pre-allocated compressed frame buffer. During subsequent display refreshes, the Display engine **Decompressor** attempts to refresh Display A from the compressed frame buffer. Lines that were not compressed or lines that were modified since the last compression – as detected by the **Frame Buffer Write Detector** – are displayed from the uncompressed buffer.



13.2 Programming Interface

13.2.1 FBC unit programming interface

The following table summarizes the register programming interface to the RLE-FBC function. Refer to the *Memory Interface Registers* chapter for details on the individual registers provided in the programming interface.

Register	Field(s)	Description
FBC_CFB_BASE	Compressed Frame Buffer Address	Specifies the location of the compressed frame buffer
FBC_LL_BASE	Compressed Line Length Buffer Address	Specifies the location of the compressed line length buffer
FBC_CONTROL	Enable	Turns the RLE-FBC function on/off
	Mode Select	Specifies Single or Periodic compression mode
	Interval	Specifies time period (in display refreshes) used in periodic mode
	Stop Compressing on Modification (DEBUG)	Specifies that the compression pass should be aborted if a line is modified during compression.
	Uncompressible Enable	Enable Uncompressible state for the tag RAM. if ENABLE compressor will mark the uncompressible scan line to prevent future compressing attempt
	Compressed Frame Buffer Stride	Specifies the stride (pitch) of the compressed frame buffer 64-byte unit
	Fence Number	Specifies the FENCE register associated with the uncompressed source frame buffer
FBC_CONTROL2		
	FBC Cx state mode	Specifies FBC behavior when PM signals CPU goes to Cx (non C0)
	CPU Fence Enable	If ENABLE the display buffer is existed within CPU fence
	Display Plane Select	Select Plane A or B for Frame Buffer Compression
FBC_YFENCE_DISP	Fence Display Buffer Y offset	Y offset from the CPU fence to the Display Buffer base
FBC_MOD_CTR	FBC modification Counter for Recompression	Recompress the Display Buffer only after the programmed number of modifications to the display buffer



Register	Field(s)	Description
FBC_COMMAND	Compression Request	Used to request compression passes in Single compression mode
FBC_STATUS	Compressing (RO)	Status indicating if the compressor is running.
	Compressed (RO, R/W for DEBUG)	Status indicating if the compressed frame buffer is available for display
	Any Modified (RO, R/W for DEBUG)	Indicates whether any lines of the uncompressed frame buffer have been modified since the last compression pass.
	Current Line Compressing (RO)	Indicates the progress of the compressor
FBC_TAG[0..N]	Tag[i+0..i+48] (DEBUG)	Status indication for each pair of display lines.

13.2.2 Programming interface from Display Engine

The following table summarizes the indirect register programming interface to the RLE-FBC function from Display Engine. These registers are programmed in Display Engine for Display function, but they are passed to FBC unit to use for Frame Buffer Compression operation. Depend on how FBC_CONTROL2 < **Display Plane Select** > is set Display Plane A or B registers are passed to FBC unit. Refer to the *Memory Interface Registers* chapter for details on the individual registers provided in the programming interface.

FBC used these registers when reading uncompressed frame buffer and building a compressed buffer that is identical to uncompressed buffer of Display Plane A or B.

Register	Field(s)	Description
DSPA(B)CNTR	Display A(B) Source Pixel Format	4-bit source Pixel format- FBC can only works with 16-bit or 32-bit Source pixel format that organize in 8-bit chunk (not 10:10:10:2 format)
DSPA(B)STRIDE	Display A (B) Stride	This value is used to determine the line to line increment for the display. FBC can work with non-power-of-two stride from 2KB to 16KB with increment of 512bytes
DSPA(B)SURF	Display A (B) Surface Base Address	This address specifies the surface base address. When the surface is tiled, panning is specified using (x, y) offsets in the DSPA (B) TILEOFF register. This address must be 4K aligned.



Register	Field(s)	Description
DSPA(B)LINOFF	Plane Start Y-Position	These 12 bits specify the vertical position in lines of the beginning of the active display plane relative to the display surface.
	Plane Start X-Position	These 12 bits specify the horizontal offset in pixels of the beginning of the active display plane relative to the display surface.
HTOTAL(B)	Pipe A (B) Horizontal Active Display Pixels	This 12-bit field provides Horizontal Active Display resolutions up to 4096 pixels. Note that the first horizontal active display pixel is considered pixel number 0. The value programmed should be the (active pixels/line – 1).
VTOTAL(B)	Pipe A (B) Vertical Active Display Lines	This 12-bit field provides vertical active display resolutions up to 4096 lines. It should be programmed with the desired number of lines minus one.

13.3 Operating Modes

13.3.1 RLE-FBC Function Modes

The RLE-FBC function (compression and decompression) is enabled or disabled via the **Enable** bit of the FBC_CONTROL register.

In order to request the disabling of the function software must set **Enable** to DISABLED. The function does not subsequently become disabled until the **Compressing** status bit of FBC_STATUS is clear. Software must ensure that the function is in fact disabled (via interrogation of the **Compressing** status bit) before re-enabling the RLE-FBC function and under the following conditions:

- Prior to changing the contents of the FBC_CFB_BASE or FBC_LL_BASE registers
- Prior to changing the contents of the following fields of the FBC_CONTROL register:
 - Mode Select
 - Interval
 - Stop Compressing on Modification
 - Uncompressible Enable
 - Compressed Frame Buffer Stride
 - Fence Number
- Prior to changing the contents of the following fields of the FBC_CONTROL2 register:
 - FBC Cx state mode
 - CPU fence Enable
 - Frame Buffer Compression Display Plane Select A/B
- Prior to changing the contents of the FBC_Fence_Display_Y_Offset register:



- Prior to changing the contents of the following fields of the FBC_MOD_CTR register:
 - FBC_mod_ctr
 - FBC_mod_ctr_valid
- Prior to changing the display_mode of the source frame buffer (Display Plane A) including display pixel format, dimensions, and pitch (stride).
- Prior to entering/use of any modes listed under *Restrictions* below

Modification of DEBUG-mode controls is implementation dependent.

13.3.2 Compression Modes

The RLE-FBC compressor is capable of operating in one of two modes, Single or Periodic Compression, as specified by the **Mode Select** field of the FBC_CONTROL register.

13.3.2.1 Single Compression Mode

In this mode software can request a single compression pass via the **Compression Request** bit of the FBC_COMMAND register. The compression results will be used until another compression is requested or the RLE-FBC function is disabled. Note that subsequent modifications to the uncompressed frame buffer will invalidate corresponding compressed lines – diminishing the benefits of the function.

Single compression mode is preferred when software has knowledge that significant portions of the frame buffer lines will remain static for a period of time – where memory bandwidth would not be wasted further recompressing the static frame buffer data.

13.3.2.2 Periodic Compression Mode

In Periodic mode, recompression is attempted at a programmed rate in units of display refreshes. The time period is programmed via the **Interval** field of the FBC_CONTROL register. The RLE-FBC compressor will not initiate a periodic compression if there have been no modifications to the source frame buffer since the last compression.

This mode is preferred when software expects significant portions of the frame buffer line to be written on a frequent basis (or at least cannot guarantee that this will not occur). The time period can be adjusted according to the refresh rate and/or frequency and extent of (expected) frame buffer modifications.

If Uncompressible **Enable** is set to ENABLED the compressor will mark a tag line uncompressible if both scan lines of a tag line are uncompressible so compressor won't attempt to compress these scan lines again in subsequent compression run unless these lines are modified by CPU or RC.

If FBC_mod_ctr_valid is SET the compressor will only attempt to recompress if the number of tag lines were modified since last compression run is greater or equal the value of FBC_mod_ctr.



13.4 Usage Restrictions

RLE Frame Buffer compression must not be enabled unless the following conditions are met:

1. If Display A is selected DSPACNTR—Display A Plane Control Register[Pixel Multiply] = No line duplication and Display A Plane Control Register[Horizontal Pixel Multiply] = 1x
2. If Display B is selected DSPBCNTR—Display B Plane Control Register[Pixel Multiply] = No line duplication and Display B Plane Control Register[Horizontal Pixel Multiply] = 1x
3. Panning of Selected Display Plane is permitted. If FBC is enabled and a compressed buffer is available when a panning event happened FBC will invalidate the current compressed buffer and recompress if necessary using the current FBC control parameters. If new uncompressed buffer required a new set of FBC control parameters then RLE-FBC must be first disabled.
4. Sync flips of Selected Display Plane are permitted. If FBC is enabled and a compressed buffer is available when sync flips event happened FBC will invalidate the current compressed buffer and recompress if necessary using the current FBC control parameters. If new uncompressed buffer required a new set of FBC control parameters then RLE-FBC must be first disabled
5. The display pixel format is 15-bit, 16-bit or 32-bit xRGB_8888 mode (as the alpha channel is removed as part of the compression).
6. Discard Alpha Frame Buffer Compression is DISABLED.
7. The uncompressed frame buffer is tiled with pitch from 2KB to 16KB in step of 0.5KB
8. The Line Width (in pixels) of the uncompressed frame buffer is a multiple of 8 in the range [640, 2048].
9. Number of lines of the uncompressed frame buffer is a multiple of 2 in the range [480, 1536].
10. Dual-wide display is not active.
11. If the pipe A is selected (i.e., DSPACNTR—Display A Plane Control Register [Display Pipe A Select] = Select Pipe A), then Pipe A Configuration Register [Interlaced modes] must be in Progressive mode.
12. If the pipe B is selected (i.e., DSPBCNTR—Display B Plane Control Register [Display Pipe B Select] = Select Pipe B), then Pipe B Configuration Register [Interlaced modes] must be in Progressive mode.
13. Compressed Frame Buffer Stride in bytes is equal or smaller than Uncompressed Frame Buffer Stride in bytes to prevent unintended buffer expansion in 16bpp frame.
14. Both Regular and SR display watermarks for 16bpp must equal 32bpp as calculated



15. Compressed Frame Buffer and Line Length buffers must reside entirely in stolen memory segment. If hardware tried to access compressed buffer or line length buffer outside of stolen memory FBC unit will be invalidate compressed buffers and makes unavailable to DISPLAY.
16. Display 180 degree rotation using gen4 hardware is turned off. This feature is not compatible with FBC scanline addressing. Software rotation can be enabled at the same time with FBC.
17. Async Flips are not permitted. FBC must be disabled when async flips are in use.

13.5 Power Management Interface

At the system level the amount of saving power of Frame Buffer Compression may be offset by power consumed by other units including CPU and memory subsystem when waiting for Frame Buffer Compression complete its pass. Device-specific power management modes need to add in to basic Frame Buffer Compression operation.

For [DevCL], different Cx state modes are used to provide a tuning mechanism between CPU low-power states (or Cx state) and FBC operation. Power Management Unit will signal to FBC that CPU is in low power state and wait for FBC to signal back that FBC is idle and no longer accessing external memory. Power Management unit then can implement global power saving scheme like putting external memory in self-refresh or clock gating FBC and/or other related units.

In DevCL, Cx state mode are specified as following:

- FBC_CONTROL2<**Cx state mode**>=IMMEDIATE IDLENESS. FBC blocks its requests to memory (read and write) and waits for all read returns to complete before asserting FBC-idle (default)
- FBC_CONTROL2<**Cx state mode**>=NORMAL IDLENESS. FBC finishes current compression pass before asserting FBC-idle
- FBC_CONTROL2<**Cx state mode**>=SCANLINE IDLENESS FBC completes the current line/line pair and skips remaining lines and makes the compressed buffer available for display before asserting FBC-idle.
- FBC_CONTROL2<**Cx state mode**>=IMMEDIATE DEBUG IDLENESS. FBC asserting FBC-idle immediately, more memory transactions may be still underway. This allows PM to find the fastest path to go to lower power state regardless of FBC operation.

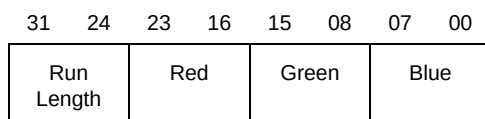
13.6 Memory Data Structures

13.6.1 RLE Pixel Runs

A compressed line contains one or more *pixel runs* of identical pixel values. A pixel run is stored as a DWord containing (1) an RGB *pixel value* and (2) a *run length* that specifies the number of times (minus one) that the pixel value is to be replicated.

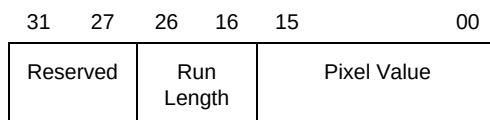
For 32bpp pixel formats, the run length is encoded in Bits 31:24 of the run Dword. This permits run lengths of 1 to 256 pixels. Any alpha value stored in Bits 31:24 is discarded. The remaining 24-bit RGB pixel value is left in place (in Bits 23:0).

Figure 13-1. 32bpp Pixel Run



For 16bpp pixel formats, the run length is encoded in Bits 26:16 of the run Dword. This permits run lengths of 1 to 2048 pixels. The 16-bit RGB pixel value is stored in Bits 15:0 (for 15bpp formats, Bit 15 is Reserved).

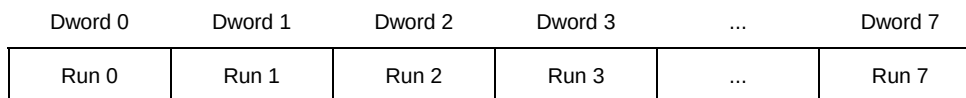
Figure 13-2. 16bpp Pixel Run



13.6.2 RLE Pixel Run Sets

The RLE-FBC function groups 8 consecutive pixel runs into 32-byte (Sword) *pixel run sets*. This matches the granularity used to read the compressed frame buffer.

Figure 13-3. Pixel Run Set



13.6.3 RLE-Compressed Line

An RLE-compressed *line* is comprised of a horizontal series of pixel run sets corresponding to a scan line in the uncompressed frame buffer.

Note that there is no encoding for “unused” Dwords in the last pixel run set. During display the Display engine will end the decompression of pixel runs when the number of decompressed pixels per line is satisfied.



13.6.4 RLE Compressed Frame and Line Length Buffers

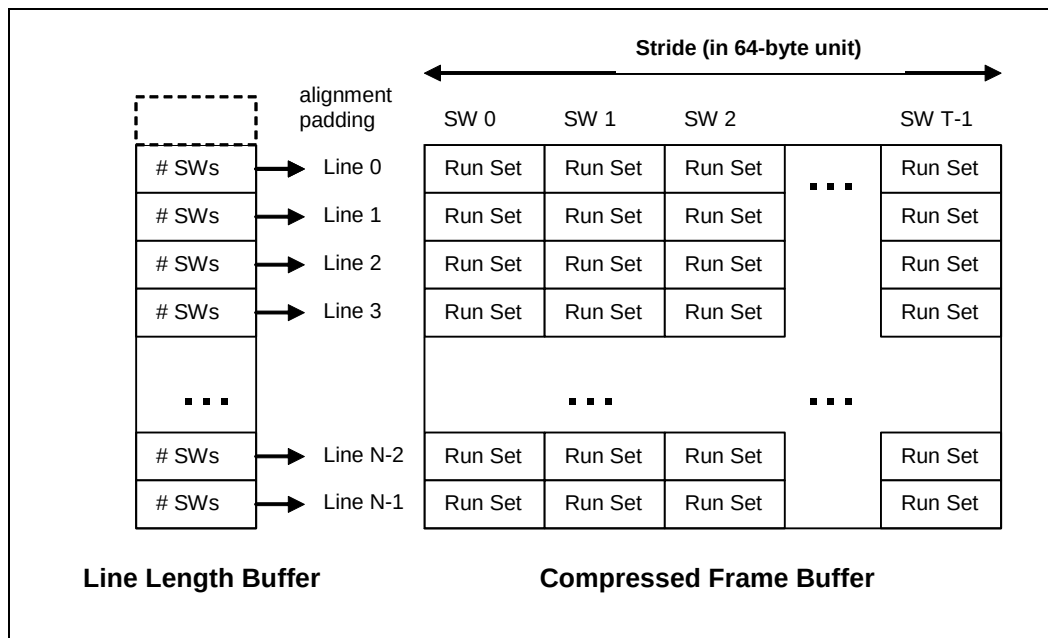
The RLE-compressed frame buffer and the Compressed Line Length Buffer must be in locked, fixed, contiguous, and uncacheable physical memory.

The RLE-Compressed Frame Buffer is a 4KB-aligned rectangular array of pixel run sets residing in physically contiguous memory (it is not mapped by the GTT). The physical address of the buffer is programmed via the FBC_CFB_BASE register.

The stride (width) of the buffer in Swords (run sets) is programmed via the **Compressed Frame Buffer Stride** field of the FBC_CONTROL register.

Different lines will typically compress to a different number of Pixel Runs. In order to record how many Swords needs to be fetched from the RLE-Compressed Frame Buffer, a Compressed Line Length Buffer is used. The Compressed Line Length Buffer is a (1536+32)-byte, 4KB-aligned list in physically contiguous memory (it is not mapped by the GTT). The physical address of the buffer is programmed via the FBC_LL_BASE register. Each byte in the buffer specifies the number of Swords (minus one) valid for the corresponding line in the RLE-Compressed Frame Buffer.

Figure 13-4. RLE-Compression Buffers



The byte in the Compressed Line Length Buffer that corresponds to Line 0 of the Compressed Frame Buffer is offset according to the alignment of the uncompressed display buffer. The Compressor and Decompressor both use the 6 least significant bit of y offset from Display Base as starting offset for line 0.



13.7 Tuning Parameters

13.7.1 Stride

The **Compressed Frame Buffer Stride** field of the FBC_CONTROL register specifies the distance (in 64-byte unit) between consecutive lines in the compressed frame buffer. If a source line cannot be compressed to fit within a compressed line, it will remain uncompressed.

The maximum compression ratio can be achieved by setting the compressed frame buffer stride to correspond with the uncompressed frame buffer line length. The stride can be set to a smaller number if there is not enough memory available for the compressed frame buffer.

13.7.2 Interval

As previously mentioned, the interval with which periodic compression passes are attempted can be adjusted as desired (e.g., as a function of refresh rate and/or expected frequency/extent of frame buffer modifications). The interval is programmed via the **Interval** field of the FBC_CONTROL register.

13.7.3 FBC Modification Counter

As previously mentioned, the FBC modification Counter can be used to reduce the number of recompression attempts if the number of modification since last attempt is small. At **Interval** expiry compressor will compare the number of accumulated tag line modifications (tag line modification counter) with the value of **FBC_mod_ctr** if the latter is larger the compressor will be back to sleep and tag line modification counter will continue counting.



13.8 Implementation (DEBUG)

This section describes the implementation of RLE_FBC function. Information in this section is not required for operational drivers – it is only required for debug activities.

13.8.1 Tag Array

A tag associated with every two sequential lines and indicates the current status of the lines. The tag states are defined as follows:

Tag Encoding	Definition	Description
'00'	Modified	At least one of the lines of the pair has been modified since the last compression pass, or a compression pass has not been made since (a) the source buffer address has changed, (b) RLE-FBC has been enabled, or (c) Reset
'01'	Uncompressed	One of the lines has not been compressed successfully.
'10'	Uncompressible	Both of the lines are uncompressible (compressed length is larger than compressed stride)
'11'	Compressed	Both of the lines are compressed

If the first line of the uncompressed source frame buffer is in an odd address, the first tag entry is associated with only one line, the first line; the second entry is associated with the second and third frame buffer line and so on. The last line will be also alone in this case.

13.8.1.1 Transitions

The following table describes the valid transitions of the Tag value. All tags start at the Modified state upon reset.

From	To	Conditions
Modified	Uncompressed	Unconditionally at the start of a compression pass.
Uncompressed	Modified	One of the lines is modified, or the source frame buffer base address was changed, or when compression becomes enabled.
Uncompressed	Compressed	Both lines were successfully compressed.
Uncompressed	Uncompressible	Both lines were unsuccessfully compressed in the previous pass
Compressed	Modified	Line was modified, or the source frame buffer base address was changed, or when compression becomes enabled.
Uncompressible	Modified	Line was modified, or the source frame buffer base address was changed, or when compression becomes enabled.



13.8.2 Compressor

The compressor will compress only if the display is on.

```

START:
if (Display Plane)
    return
on (Start of Display Vblank)
    Sample the FBC address and configuration registers
    if (Mode == Periodic)
        Interval counter = interval counter-- % Interval

    if ((Mode == Periodic AND Interval == 0) OR Compression Request)
AND
    Display is ON AND
    (At least one line pair is Modified) AND
    (!Compressing) AND
    (Local cache and write posting buffers are empty) AND
    (Display buffer is tiled)
        goto COMPRESSION
    else goto START

COMPRESSION:
{
    Change Modified to Uncompressed // One cycle
    Set FBC_CONTROL<Compressing>
    Reset FBC_CONTROL<Compressed>
    Reset FBC_CONTROL<Modified>

    for (each and every Uncompressed line pair)
    {
        /* By first marking and then compressing we guarantee that
        modification to this line will be marked as Modified and will
        not be overridden when compression is completed */
        Mark the pair as Compressed
        Compress first line
        if (Stride exceeded)
            Mark pair as Uncompressed
        else
            Write the compressed line length to the
line-length buffer
            Compress second line
            if (Stride exceeded)
                Mark pair as
Uncompressed
            else
                Write the
compressed line length to the line-length buffer
                Mark pair as
Compressed
                Set FBC_CONTROL<Compressed>
    } // end for each uncompressed line pair
    Reset the "Compression in progress" bit
    Set Compressed-buffer-avail bit
} // end compression

// If we succeeded to compress or not
if (Mode == Periodic)
    Reset the interval-counter
goto START

```



13.8.3 Decompressor

When the display streamer gets the first line request it checks for the following condition:

- FBC_CONTROL<**Enable**> is set
- FBC_CONTROL<**Compressing**> is clear (compression not in progress)
- FBC_CONTROL<**Compressed**> is set (a compression pass has completed)

If any of these conditions are not met, only the uncompressed source buffer will be used for refresh.

If all these conditions are met, the Decompressor will, for every line:

- If the line marked as **Compressed** the display streamer will read the compressed line length from the compressed line length buffer, and then read the compressed line data according to this length. If the line is not marked as **Compressed**, the display streamer reads the line from the uncompressed frame buffer. In both cases the pixel data is posted to the display FIFO.
- If the line is **Compressed** the Decompressor reads Dwords from the FIFO and sends on the pixel data multiple times according to the run length, 1 – 256 in 32-bit mode and 1 – 2048 in 16-bit modes. The Decompressor keeps track of the number of pixels and stops when it reaches the line width (in pixels) and discards any remaining Dwords.

13.8.4 Frame Buffer Write Detector

The Frame Buffer Write Detector snoops all frame buffers writes from the CPU and render engines, and marks the modified line pairs as **Modified**.

- If Display buffer is a subset of the render buffer and cpu path is enabled via a fence, where the fence is a superset of the render buffer then frame buffers lines might be modified by both cpu write and render cache write.
- If Display buffer is a subset of render buffer and fence cpu path is disable then frame buffers might be modified by render cache line only.
- If CPU path is disabled and Render and Display are independent buffers then no modified should be happened.

In order to detect CPU write the following FBC registers need to be programmed before the FBC is enable

- FBC_CONTROL2 <**CPU Fence Enable**> is set.
- FBC_CONTROL <**Fence Number**> set to match the fence that render target and Frame Buffer reside in.
- FBC_YFENCE_DISP is set to the distance from fence base address to DSPA(**B**)SURF

Chipset unit passes CPU writes that are within Graphic Aperture to FBC. FBC write detector decode the line number and marked affected line as **modified**.



There are no register programming needed for render cache write monitor. Render cache unit pass each write to its cache to FBC. If Render Target Address match with DSPA (B) SURF, and the render cache line has the same offset with active display then the affected line pair is marked as **Modified**.

All lines will be marked as modified whenever:

- The uncompressed source Frame Buffer base address changes (this is only permitted to happen as a result of a direct register write – flips of Selected Display Plane are not allowed when RLE-FBC is enabled)
- RLE-FBC is enabled
- Reset

If the FBC_CONTROL<**Stop Compressing on Modification**> (DEBUG) bit is set, and a source frame buffer write is detected during a compression pass, the compression is aborted and the current line pair is marked as **Modified**. Compression will be reattempted at the next periodic compression or when the next single compression pass is requested.

13.8.5 Coherency

The display coherency is kept by keeping the following rules:

- The compressed frame buffer is not displayed during compression.
- The Compressor will only compress lines that are marked as **Uncompressed**.
- Lines state changes from **Modified** to **Uncompressed** can only when there are no display reads or pending display writes. This is achieved by waiting for Vblank start and then starting the compression only if the render cache is empty.
- Marking a line as **Modified** takes precedence over the (simultaneous) transition from **Modified** to **Uncompressed**.
- Before a line pair is compressed, the tag is changed from **Uncompressed** to **Compressed**. This will guarantee that if a line is modified while being compressed it will transition to the **Modified** state.
- Compressor frame buffer reads push CPU writes to memory.
- At the end of each compression path FBC issues dummy reads to push Compressed Buffer writes to memory.



14 *BLT Engine*

14.1 Introduction

2D Rendering can be divided into 2 categories: classical BLTs, described here, and 3D BLTs. 3D BLTs are operations which can take advantage of the 3D drawing engine's functionality and access patterns.

Functions such as Alpha BLTs, arithmetic (bilinear) stretch BLTs, rotations, transposing pixel maps, color space conversion, and DIBs are all considered 3D BLTs and are covered in the 3D rendering section. DIBs can be thought of as an indexed texture which uses the texture palette for performing the data translation. All drawing engines have swappable context. The same hardware can be used by multiple driver threads where the current state of the hardware is saved to memory and the appropriate state is loaded from memory on thread switches.

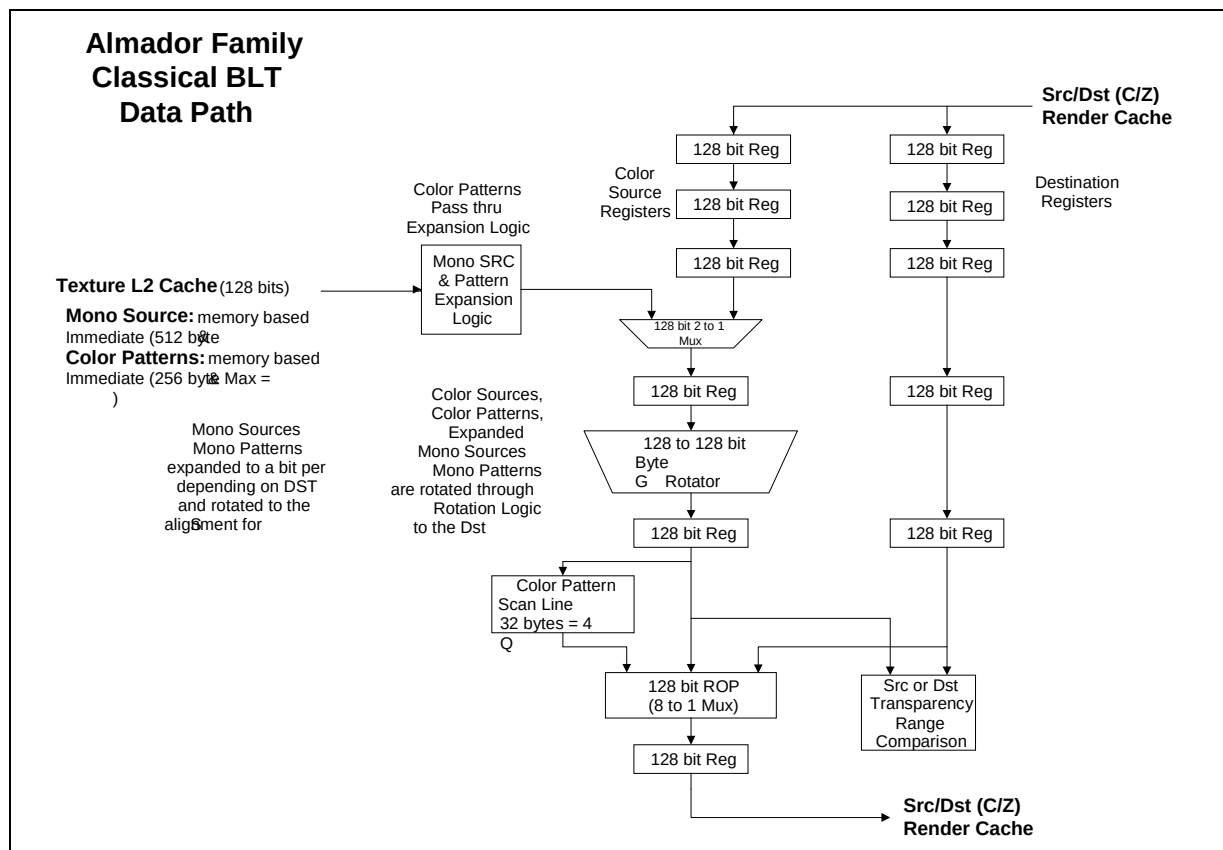
All operands for both 3D and classical BLTs can be in graphics aperture or cacheable system memory. Some operands can be immediates which are sent through the command stream. Immediate operands are: patterns, monochrome sources, DIB palettes, and DIB source operands. All non-monochrome operands which are not tiled have a stride granularity of a double-word (4 bytes).

The classical BLT commands support both linear addressing and X, Y coordinates with and without clipping. All X1 and Y1 destination and clipping coordinates are inclusive, while X2 and Y2 are exclusive. Currently, only destination coordinates can be negative. The source and clipping coordinates must be positive. If clipping is disabled, but a negative destination coordinate is specified, the negative coordinate is clipped to 0. Linear address BLT commands must supply a non-zero height and width. If either height or width = 0, then no accesses occur.

14.2 Classical BLT Engine Functional Description

The graphics controller provides a hardware-based BLT engine to off load the work of moving blocks of graphics data from the host CPU. Although the BLT engine is often used simply to copy a block of graphics data from the source to the destination, it also has the ability to perform more complex functions. The BLT engine is capable of receiving three different blocks of graphics data as input as shown in the figure below. The source data may exist in the frame buffer or the Graphics aperture. The pattern data always represents an 8x8 block of pixels that can be located in the frame buffer, Graphics aperture, or passed through a command packet. The pattern data must be located in linear memory. The data already residing at the destination may also be used as an input. The destination data can also be located in the frame buffer or graphics aperture.

Figure 14-1. Block Diagram and Data Paths of the BLT Engine



The BLT engine may use any combination of these three different blocks of graphics data as operands, in both bit-wise logical operations to generate the actual data to be written to the destination, and in per-pixel write-masking to control the writing of data to the destination. It is intended that the BLT engine will perform these bit-wise and per-pixel operations on color graphics data that is at the same color depth that the rest of the graphics system has been set. However, if either the source or pattern data is monochrome, the BLT engine has the ability to put either block of graphics data through a process called “color expansion” that converts monochrome graphics data to color. Since the destination is often a location in the on-screen portion of the frame buffer, it is assumed that any data already at the destination will be of the appropriate color depth.

14.2.1 Basic BLT Functional Considerations

14.2.1.1 Color Depth Configuration and Color Expansion

The graphics system and BLT engine can be configured for color depths of 8, 16, and 32 bits per pixel.

The configuration of the BLT engine for a given color depth dictates the number of bytes of graphics data that the BLT engine will read and write for each pixel while performing a BLT operation. It is assumed that any graphics data already residing at



the destination which is used as an input is already at the color depth to which the BLT engine is configured. Similarly, it is assumed that any source or pattern data used as an input has this same color depth, unless one or both is monochrome. If either the source or pattern data is monochrome, the BLT engine performs a process called “color expansion” to convert such monochrome data to color at the color depth to which the BLT engine has been set.

During “color expansion” the individual bits of monochrome source or pattern data that correspond to individual pixels are converted into 1, 2, or 4 bytes (which ever is appropriate for the color depth to which the BLT engine has been set). If a given bit of monochrome source or pattern data carries a value of 1, then the byte(s) of color data resulting from the conversion process are set to carry the value of a specified foreground color. If a given bit of monochrome source or pattern data carries a value of 0, the resulting byte(s) are set to the value of a specified background color or not written if transparency is selected.

The BLT engine is set to a default configuration color depth of 8, 16, or 32 bits per pixel through BLT command packets. Whether the source and pattern data are color or monochrome must be specified using command packets. Foreground and background colors for the color expansion of both monochrome source and pattern data are also specified through the command packets. The source foreground and background colors used in the color expansion of monochrome source data are specified independently of those used for the color expansion of monochrome pattern data.

14.2.1.2 Graphics Data Size Limitations

The BLT engine is capable of transferring very large quantities of graphics data. Any graphics data read from and written to the destination is permitted to represent a number of pixels that occupies up to 65,536 scan lines and up to 32,768 bytes per scan line at the destination. The maximum number of pixels that may be represented per scan line's worth of graphics data depends on the color depth.

Any source data used as an input must represent the same number of pixels as is represented by any data read from or written to the destination, and it must be organized so as to occupy the same number of scan lines and pixels per scan line.

The actual number of scan lines and bytes per scan line required to accommodate data read from or written to the destination are set in the destination width & height registers or using X and Y coordinates within the command packets. These two values are essential in the programming of the BLT engine, because the engine uses these two values to determine when a given BLT operation has been completed.

14.2.1.3 Bit-Wise Operations

The BLT engine can perform any one of 256 possible bit-wise operations using various combinations of the three previously described blocks of graphics data that the BLT engine can receive as input.

The choice of bit-wise operation selects which of the three inputs will be used, as well as the particular logical operation to be performed on corresponding bits from each of the selected inputs. The BLT engine automatically foregoes reading any form of graphics data that has not been specified as an input by the choice of bit-wise operation. An 8-bit code written to the raster operation field of the command packets



chooses the bit-wise operation. The following table lists the available bit-wise operations and their corresponding 8-bit codes.

Table 14-1. Bit-Wise Operations and 8-Bit Codes (00-3F)

Code	Value Written to Bits at Destination
00	writes all 0's
01	not(D or (P or S))
02	D and (not(P or S))
03	not(P or S)
04	S and (not(D or P))
05	not(D or P)
06	not(P or (not(D xor S)))
07	not(P or (D and S))
08	S and (D and (notP))
09	not(P or (D xor S))
0A	D and (notP)
0B	not(P or (S and (notD)))
0C	S and (notP)
0D	not(P or (D and (notS)))
0E	not(P or (not(D or S)))
0F	notP
10	P and (not(D or S))
11	not(D or S)
12	not(S or (not(D xor P)))
13	not(S or (D and P))
14	not(D or (not(P xor S)))
15	not(D or (P and S))
16	P xor (S xor (D and (not(P and S))))
17	not(S xor ((S xor P) and (D xor S)))
18	(S xor P) and (P xor D)
19	not(S xor (D and (not(P and S))))
1A	P xor (D or (S and P))
1B	not(S xor (D and (P xor S)))
1C	P xor (S or (D and P))
1D	not(D xor (S and (P xor D)))
1E	P xor (D or S)
1F	not(P and (D or S))

Code	Value Written to Bits at Destination
20	D and (P and (notS))
21	not(S or (D xor P))
22	D and (notS)
23	not(S or (P and (notD)))
24	(S xor P) and (D xor S)
25	not(P xor (D and (not(S and P))))
26	S xor (D or (P and S))
27	S xor (D or (not(P xor S)))
28	D and (P xor S)
29	not(P xor (S xor (D or (P and S))))
2A	D and (not(P and S))
2B	not(S xor ((S xor P) and (P xor D)))
2C	S xor (P and (D or S))
2D	P xor (S or (notD))
2E	P xor (S or (D xor P))
2F	not(P and (S or (notD)))
30	P and (notS)
31	not(S or (D and (notP)))
32	S xor (D or (P or S))
33	notS
34	S xor (P or (D and S))
35	S xor (P or (not(D xor S)))
36	S xor (D or P)
37	not(S and (D or P))
38	P xor (S and (D or P))
39	S xor (P or (notD))
3A	S xor (P or (D xor S))
3B	not(S and (P or (notD)))
3C	P xor S
3D	S xor (P or (not(D or S)))
3E	S xor (P or (D and (notS)))
3F	not(P and S)

Notes: S = Source Data
P = Pattern Data
D = Data Already Existing at the Destination



Table 14-2. Bit-Wise Operations and 8-bit Codes (40 - 7F)

Code	Value Written to Bits at Destination
40	$P \text{ and } (S \text{ and } (\text{not} D))$
41	$\text{not}(D \text{ or } (P \text{ xor } S))$
42	$(S \text{ xor } D) \text{ and } (P \text{ xor } D)$
43	$\text{not}(S \text{ xor } (P \text{ and } (\text{not}(D \text{ and } S))))$
44	$S \text{ and } (\text{not} D)$
45	$\text{not}(D \text{ or } (P \text{ and } (\text{not} S)))$
46	$D \text{ xor } (S \text{ or } (P \text{ and } D))$
47	$\text{not}(P \text{ xor } (S \text{ and } (D \text{ xor } P)))$
48	$S \text{ and } (D \text{ xor } P)$
49	$\text{not}(P \text{ xor } (D \text{ xor } (S \text{ or } (P \text{ and } D))))$
4A	$D \text{ xor } (P \text{ and } (S \text{ or } D))$
4B	$P \text{ xor } (D \text{ or } (\text{not} S))$
4C	$S \text{ and } (\text{not}(D \text{ and } P))$
4D	$\text{not}(S \text{ xor } ((S \text{ xor } P) \text{ or } (D \text{ xor } S)))$
4E	$P \text{ xor } (D \text{ or } (S \text{ xor } P))$
4F	$\text{not}(P \text{ and } (D \text{ or } (\text{not} S)))$
50	$P \text{ and } (\text{not} D)$
51	$\text{not}(D \text{ or } (S \text{ and } (\text{not} P)))$
52	$D \text{ xor } (P \text{ or } (S \text{ and } D))$
53	$\text{not}(S \text{ xor } (P \text{ and } (D \text{ xor } S)))$
54	$\text{not}(D \text{ or } (\text{not}(P \text{ or } S)))$
55	$\text{not} D$
56	$D \text{ xor } (P \text{ or } S)$
57	$\text{not}(D \text{ and } (P \text{ or } S))$
58	$P \text{ xor } (D \text{ and } (S \text{ or } P))$
59	$D \text{ xor } (P \text{ or } (\text{not} S))$
5A	$D \text{ xor } P$
5B	$D \text{ xor } (P \text{ or } (\text{not}(S \text{ or } D)))$
5C	$D \text{ xor } (P \text{ or } (S \text{ xor } D))$
5D	$\text{not}(D \text{ and } (P \text{ or } (\text{not} S)))$
5E	$D \text{ xor } (P \text{ or } (S \text{ and } (\text{not} D)))$
5F	$\text{not}(D \text{ and } P)$

Code	Value Written to Bits at Destination
60	$P \text{ and } (D \text{ xor } S)$
61	$\text{not}(D \text{ xor } (S \text{ xor } (P \text{ or } (D \text{ and } S))))$
62	$D \text{ xor } (S \text{ and } (P \text{ or } D))$
63	$S \text{ xor } (D \text{ or } (\text{not} P))$
64	$S \text{ xor } (D \text{ and } (P \text{ or } S))$
65	$D \text{ xor } (S \text{ or } (\text{not} P))$
66	$D \text{ xor } S$
67	$S \text{ xor } (D \text{ or } (\text{not}(P \text{ or } S)))$
68	$\text{not}(D \text{ xor } (S \text{ xor } (P \text{ or } (\text{not}(D \text{ or } S))))$
69	$\text{not}(P \text{ xor } (D \text{ xor } S))$
6A	$D \text{ xor } (P \text{ and } S)$
6B	$\text{not}(P \text{ xor } (S \text{ xor } (D \text{ and } (P \text{ or } S))))$
6C	$S \text{ xor } (D \text{ and } P)$
6D	$\text{not}(P \text{ xor } (D \text{ xor } (S \text{ and } (P \text{ or } D))))$
6E	$S \text{ xor } (D \text{ and } (P \text{ or } (\text{not} S)))$
6F	$\text{not}(P \text{ and } (\text{not}(D \text{ xor } S)))$
70	$P \text{ and } (\text{not}(D \text{ and } S))$
71	$\text{not}(S \text{ xor } ((S \text{ xor } D) \text{ and } (P \text{ xor } D)))$
72	$S \text{ xor } (D \text{ or } (P \text{ xor } S))$
73	$\text{not}(S \text{ and } (D \text{ or } (\text{not} P)))$
74	$D \text{ xor } (S \text{ or } (P \text{ xor } D))$
75	$\text{not}(D \text{ and } (S \text{ or } (\text{not} P)))$
76	$S \text{ xor } (D \text{ or } (P \text{ and } (\text{not} S)))$
77	$\text{not}(D \text{ and } S)$
78	$P \text{ xor } (D \text{ and } S)$
79	$\text{not}(D \text{ xor } (S \text{ xor } (P \text{ and } (D \text{ or } S))))$
7A	$D \text{ xor } (P \text{ and } (S \text{ or } (\text{not} D)))$
7B	$\text{not}(S \text{ and } (\text{not}(D \text{ xor } P)))$
7C	$S \text{ xor } (P \text{ and } (D \text{ or } (\text{not} S)))$
7D	$\text{not}(D \text{ and } (\text{not}(P \text{ xor } S)))$
7E	$(S \text{ xor } P) \text{ or } (D \text{ xor } S)$
7F	$\text{not}(D \text{ and } (P \text{ and } S))$

Notes: S = Source Data
P = Pattern Data
D = Data Already Existing at the Destination



Table 14-3. Bit-Wise Operations and 8-bit Codes (80 - BF)

Code	Value Written to Bits at Destination
80	D and (P and S)
81	not((S xor P) or (D xor S))
82	D and (not(P xor S))
83	not(S xor (P and (D or (notS))))
84	S and (not(D xor P))
85	not(P xor (D and (S or (notP))))
86	D xor (S xor (P and (D or S)))
87	not(P xor (D and S))
88	D and S
89	not(S xor (D or (P and (notS))))
8A	D and (S or (notP))
8B	not(D xor (S or (P xor D)))
8C	S and (D or (notP))
8D	not(S xor (D or (P xor S)))
8E	S xor ((S xor D) and (P xor D))
8F	not(P and (not(D and S)))
90	P and (not(D xor S))
91	not(S xor (D and (P or (notS))))
92	D xor (P xor (S and (D or P)))
93	not(S xor (P and D))
94	P xor (S xor (D and (P or S)))
95	not(D xor (P and S))
96	D xor (P xor S)
97	P xor (S xor (D or (not(P or S))))
98	not(S xor (D or (not(P or S))))
99	not(D xor S)
9A	D xor (P and (notS))
9B	not(S xor (D and (P or S)))
9C	S xor (P and (notD))
9D	not(D xor (S and (P or D)))
9E	D xor (S xor (P or (D and S)))
9F	not(P and (D xor S))

Code	Value Written to Bits at Destination
A0	D and P
A1	not(P xor (D or (S and (notP))))
A2	D and (P or (notS))
A3	not(D xor (P or (S xor D)))
A4	not(P xor (D or (not(S or P))))
A5	not(P xor D)
A6	D xor (S and (notP))
A7	not(P xor (D and (S or P)))
A8	D and (P or S)
A9	not(D xor (P or S))
AA	D
AB	D or (not(P or S))
AC	S xor (P and (D xor S))
AD	not(D xor (P or (S and D)))
AE	D or (S and (notP))
AF	D or (notP)
B0	P and (D or (notS))
B1	not(P xor (D or (S xor P)))
B2	S xor ((S xor P) or (D xor S))
B3	not(S and (not(D and P)))
B4	P xor (S and (notD))
B5	not(D xor (P and (S or D)))
B6	D xor (P xor (S or (D and P)))
B7	not(S and (D xor P))
B8	P xor (S and (D xor P))
B9	not(D xor (S or (P and D)))
BA	D or (P and (notS))
BB	D or (notS)
BC	S xor (P and (not(D and S)))
BD	not((S xor D) and (P xor D))
BE	D or (P xor S)
BF	D or (not(P and S))

Notes: S = Source Data
P = Pattern Data
D = Data Already Existing at the Destination



Table 14-4. Bit-Wise Operations and 8-bit Codes (C0 - FF)

Code	Value Written to Bits at Destination
C0	P and S
C1	not(S xor (P or (D and (notS)))))
C2	not(S xor (P or (not(D or S)))))
C3	not(P xor S)
C4	S and (P or (notD))
C5	not(S xor (P or (D xor S)))
C6	S xor (D and (notP))
C7	not(P xor (S and (D or P)))
C8	S and (D or P)
C9	not(S xor (P or D))
CA	D xor (P and (S xor D))
CB	not(S xor (P or (D and S)))
CC	S
CD	S or (not(D or P))
CE	S or (D and (notP))
CF	S or (notP)
D0	P and (S or (notD))
D1	not(P xor (S or (D xor P)))
D2	P xor (D and (notS))
D3	not(S xor (P and (D or S)))
D4	S xor ((S xor P) and (P xor D))
D5	not(D and (not(P and S)))
D6	P xor (S xor (D or (P and S)))
D7	not(D and (P xor S))
D8	P xor (D and (S xor P))
D9	not(S xor (D or (P and S)))
DA	D xor (P and (not(S and D)))
DB	not((S xor P) and (D xor S))
DC	S or (P and (notD))
DD	S or (notD)
DE	S or (D xor P)
DF	S or (not(D and P))

Code	Value Written to Bits at Destination
E0	P and (D or S)
E1	not(P xor (D or S))
E2	D xor (S and (P xor D))
E3	not(P xor (S or (D and P)))
E4	S xor (D and (P xor S))
E5	not(P xor (D or (S and P)))
E6	S xor (D and (not(P and S)))
E7	not((S xor P) and (P xor D))
E8	S xor ((S xor P) and (D xor S))
E9	not(D xor (S xor (P and (not(D and S))))))
EA	D or (P and S)
EB	D or (not(P xor S))
EC	S or (D and P)
ED	S or (not(D xor P))
EE	D or S
EF	S or (D or (notP))
F0	P
F1	P or (not(D or S))
F2	P or (D and (notS))
F3	P or (notS)
F4	P or (S and (notD))
F5	P or (notD)
F6	P or (D xor S)
F7	P or (not(D and S))
F8	P or (D and S)
F9	P or (not(D xor S))
FA	D or P
FB	D or (P or (notS))
FC	P or S
FD	P or (S or (notD))
FE	D or (P or S)
FF	writes all 1's

Notes: S = Source Data
P = Pattern Data
D = Data Already Existing at the Destination



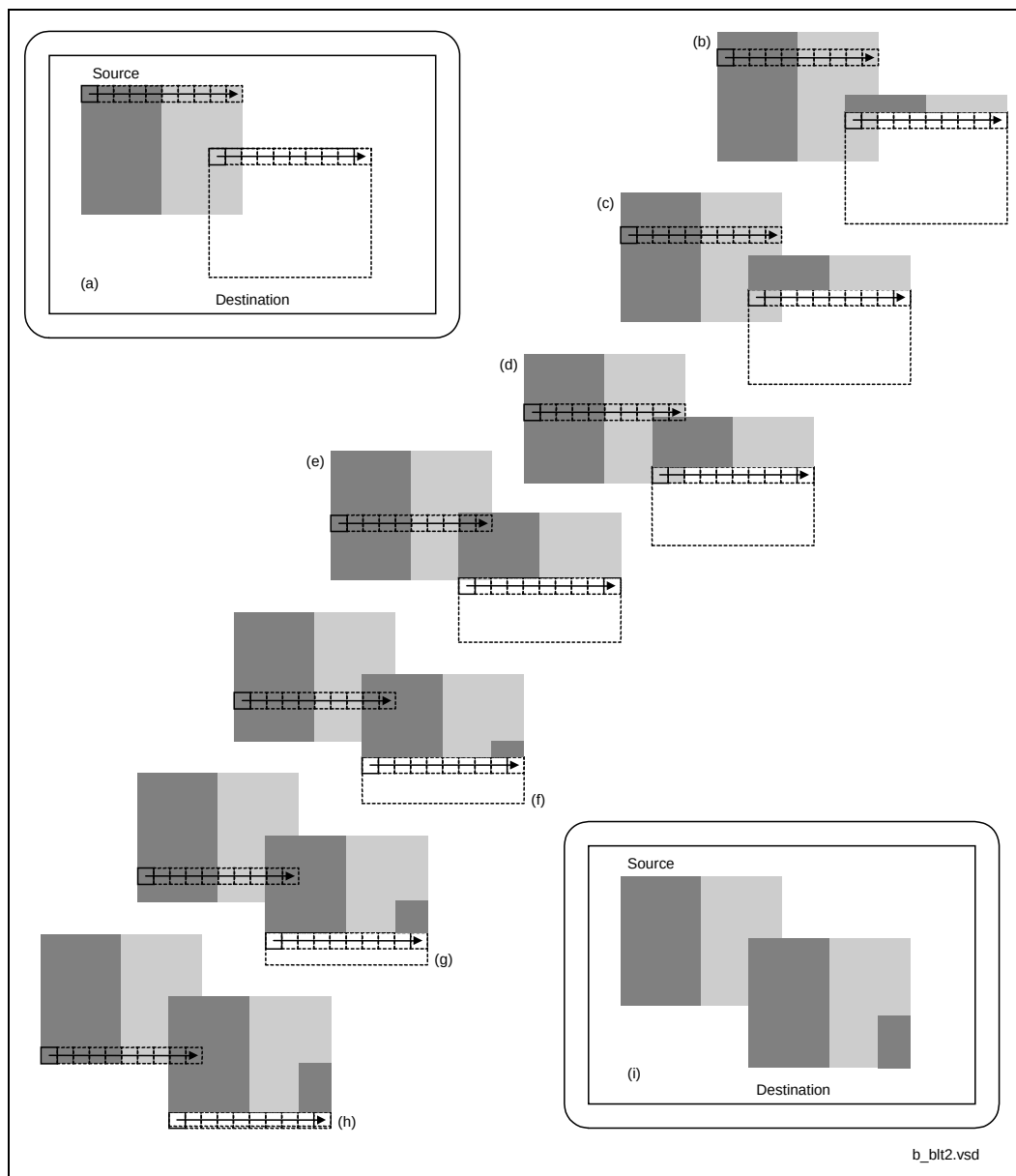
The 3-bit field, destination transparency mode, within the command packets can select per-pixel write-masking with a mask based on the results of color comparisons. The monochrome source background and foreground are range compared with either the bytes for the pixels at the destination or the source operand. This operation is described in the BLT command packet and register descriptions.

14.2.1.5 When the Source and Destination Locations Overlap

It is possible to have BLT operations in which the locations of the source and destination data overlap. This frequently occurs in BLT operations where a user is shifting the position of a graphical item on the display by only a few pixels. In these situations, the BLT engine must be programmed so that destination data is not written into destination locations that overlap with source locations before the source data at those locations has been read. Otherwise, the source data will become corrupted. The XY commands determine whether there is an overlap and perform the accesses in the proper direction to avoid data corruption.

The following figure shows how the source data can be corrupted when a rectangular block is copied from a source location to an overlapping destination location. The BLT engine typically reads from the source location and writes to the destination location starting with the left-most pixel in the top-most line of both, as shown in step (a). As shown in step (b), corruption of the source data has already started with the copying of the top-most line in step (a) — part of the source that originally contained lighter-colored pixels has now been overwritten with darker-colored pixels. More source data corruption occurs as steps (b) through (d) are performed. At step (e), another line of the source data is read, but the two right-most pixels of this line are in the region where the source and destination locations overlap, and where the source has already been overwritten as a result of the copying of the top-most line in step (a). Starting in step (f), darker-colored pixels can be seen in the destination where lighter-colored pixels should be. This errant effect occurs repeatedly throughout the remaining steps in this BLT operation. As more lines are copied from the source location to the destination location, it becomes clear that the end result is not what was originally intended.

Figure 14-3. Source Corruption in BLT with Overlapping Source and Destination Locations

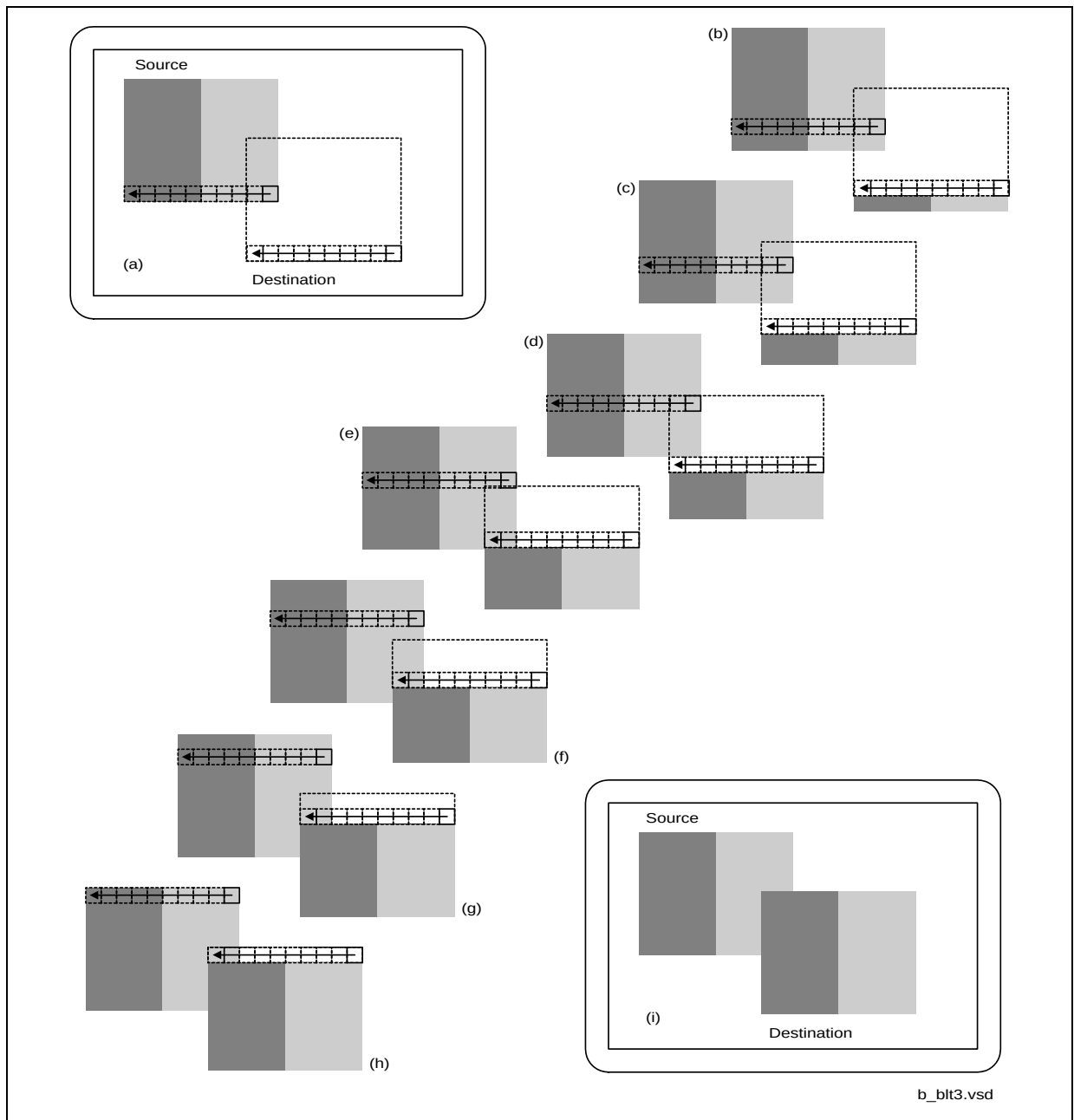


The BLT engine can alter the order in which source data is read and destination data is written when necessary to avoid source data corruption problems when the source and destination locations overlap. The command packets provide the ability to change the point at which the BLT engine begins reading and writing data from the upper left-hand corner (the usual starting point) to one of the other three corners. The BLT engine may be set to read data from the source and write it to the destination starting at any of the four corners of the panel.

The XY command packets perform the necessary comparisons and start at the proper corner of each operand which avoids data corruption.

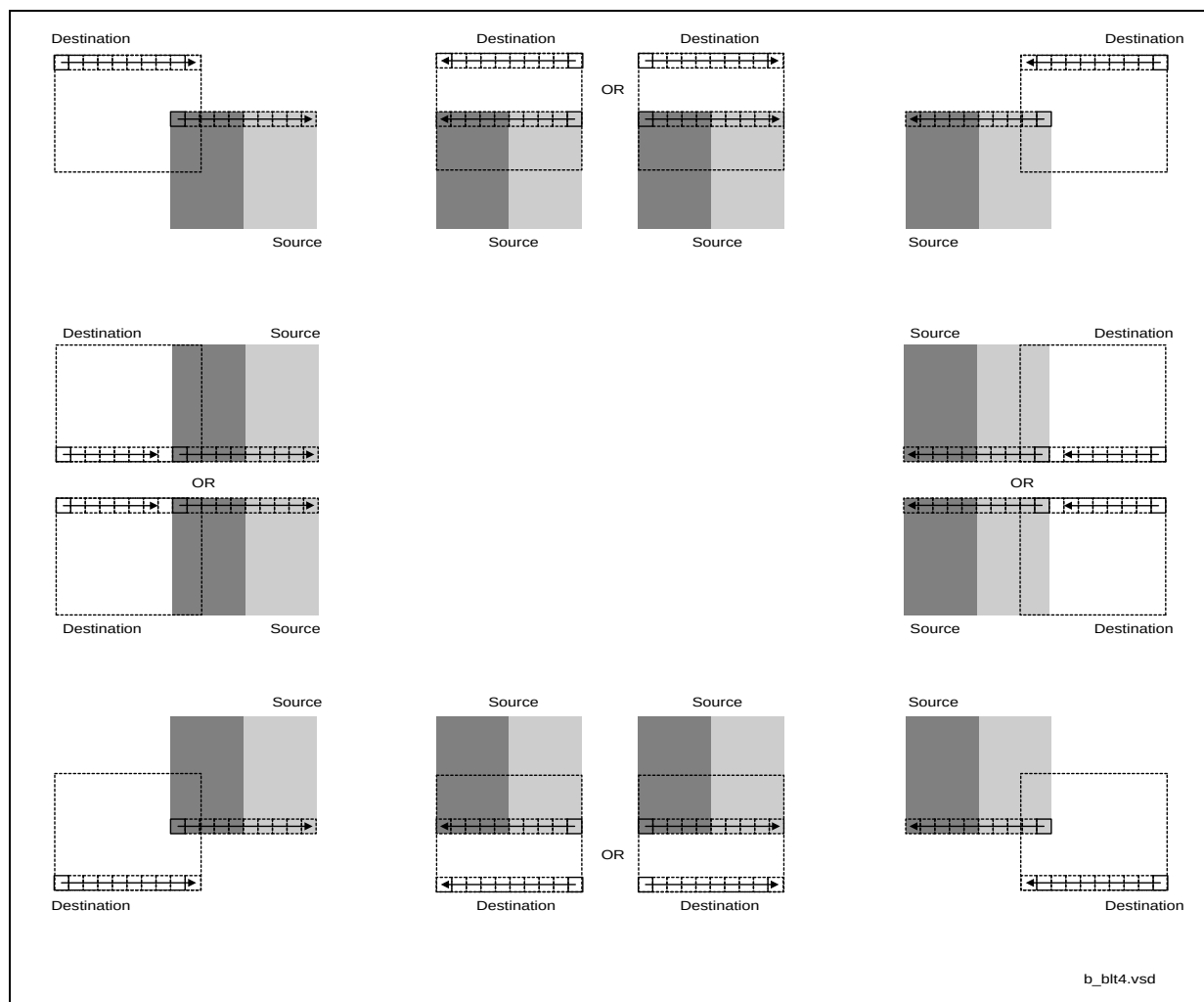


Figure 14-4. Correctly Performed BLT with Overlapping Source and Destination Locations



The following figure illustrates how this feature of the BLT engine can be used to perform the same BLT operation as was illustrated in the figure above, while avoiding the corruption of source data. As shown in the figure below, the BLT engine reads the source data and writes the data to the destination starting with the right-most pixel of the bottom-most line. By doing this, no pixel existing where the source and destination locations overlap will ever be written to before it is read from by the BLT engine. By the time the BLT operation has reached step (e) where two pixels existing where the source and destination locations overlap are about to be over written, the source data for those two pixels has already been read.

Figure 14-5. Suggested Starting Points for Possible Source and Destination Overlap Situations



The figure above shows the recommended lines and pixels to be used as starting points in each of 8 possible ways in which the source and destination locations may overlap. In general, the starting point should be within the area in which the source and destination overlap.

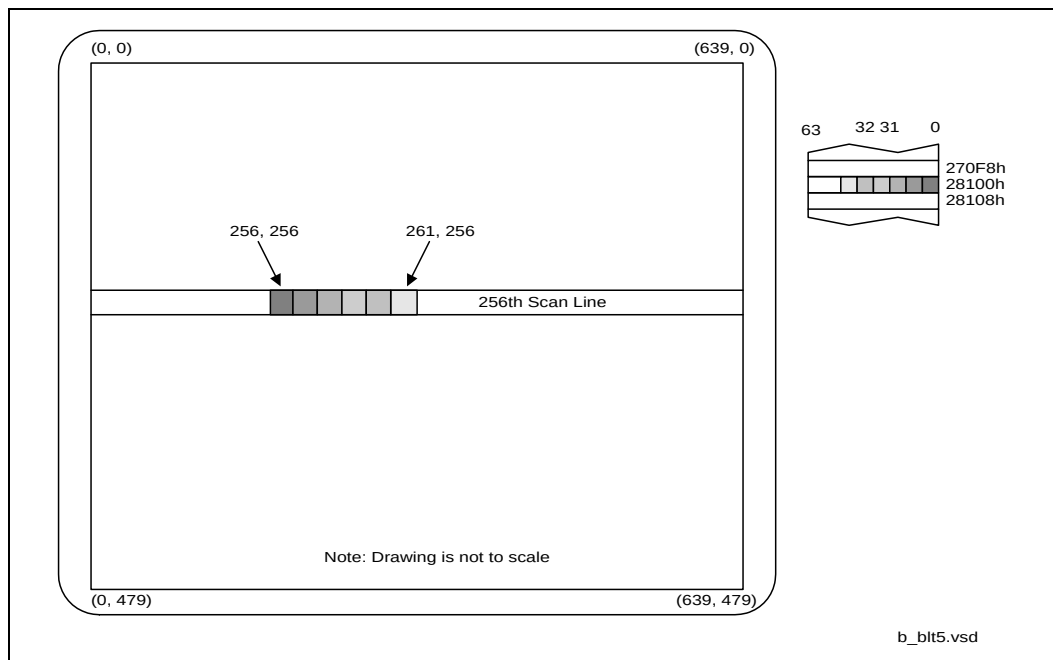


14.2.2 Basic Graphics Data Considerations

14.2.2.1 Contiguous vs. Discontinuous Graphics Data

Graphics data stored in memory, particularly in the frame buffer of a graphics system, has organizational characteristics that often distinguish it from other varieties of data. The main distinctive feature is the tendency for graphics data to be organized in a discontinuous block of graphics data made up of multiple sub-blocks of bytes, instead of a single contiguous block of bytes.

Figure 14-6. Representation of On-Screen Single 6-Pixel Line in the Frame Buffer

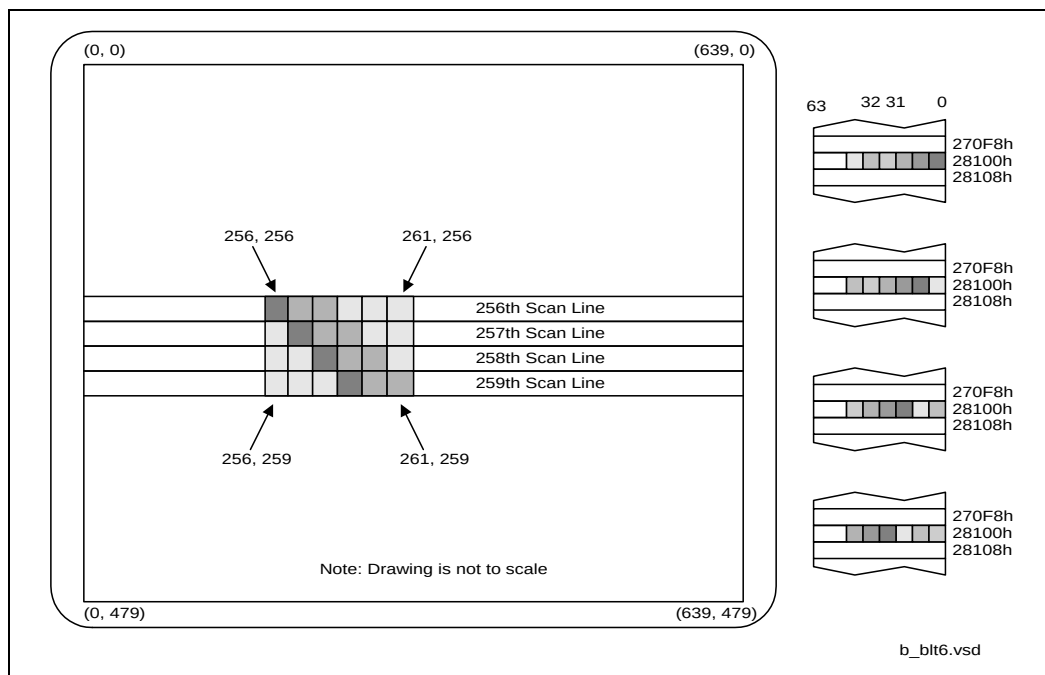


The figure above shows an example of contiguous graphics data — a horizontal line made up of six adjacent pixels within a single scan line on a display with a resolution of 640x480. Presuming that the graphics system driving this display has been set to 8 bits per pixel and that the frame buffer's starting address of 0h corresponds to the upper left-most pixel of this display, then the six pixels that make this horizontal line starting at coordinates (256, 256) occupies the six bytes starting at frame buffer address 28100h, and ending at address 28105h.

In this case, there is only one scan line's worth of graphics data in this single horizontal line, so the block of graphics data for all six of these pixels exists as a single, contiguous block comprised of only these six bytes. The starting address and the number of bytes are the only pieces of information that a BLT engine would require to read this block of data.

The simplicity of the above example of a single horizontal line contrasts sharply to the example of discontinuous graphics data depicted in the figure below. The simple six-pixel line of the figure above is now accompanied by three more six-pixel lines placed on subsequent scan lines, resulting in the 6x4 block of pixels shown.

Figure 14-7. Representation of On-Screen 6x4 Array of Pixels in the Frame Buffer



Since there are other pixels on each of the scan lines on which this 6x4 block exists that are not part of this 6x4 block, what appears to be a single 6x4 block of pixels on the display must be represented by a discontinuous block of graphics data made up of 4 separate sub-blocks of six bytes apiece in the frame buffer at addresses 28100h, 28380h, 28600h, and 28880h. This situation makes the task of reading what appears to be a simple 6x4 block of pixels more complex. However, there are two characteristics of this 6x4 block of pixels that help simplify the task of specifying the locations of all 24 bytes of this discontinuous block of graphics data: all four of the sub-blocks are of the same length, and the four sub-blocks are separated from each other at equal intervals.

The BLT engine is designed to make use of these characteristics of graphics data to simplify the programming required to handle discontinuous blocks of graphics data. For such a situation, the BLT engine requires only four pieces of information: the starting address of the first sub-block, the length of a sub-block, the offset (in bytes), pitch, of the starting address of each subsequent sub-block, and the quantity of sub-blocks.

14.2.2.2 Source Data

The source data may exist in the frame buffer or elsewhere in the graphics aperture where the BLT engine may read it directly, or it may be provided to the BLT engine by the host CPU through the command packets. The block of source graphics data may be either contiguous or discontinuous, and may be either in color (with a color depth that matches that to which the BLT engine has been set) or monochrome.

The source select bit in the command packets specifies whether the source data exists in the frame buffer or is provided through the command packets. Monochrome source data is always specified as being supplied through an immediate command packet.



If the color source data resides within the frame buffer or elsewhere in the graphics aperture, then the Source Address Register, specified in the command packets is used to specify the address of the source.

In cases where the host CPU provides the source data, it does so by writing the source data to ring buffer directly after the BLT command that requires the data or uses an IMMEDIATE_INDIRECT_BLT command packet which has a size and pointer to the operand in Graphics aperture.

The block of bytes sent by the host CPU through the command packets must be quadword-aligned and the source data contained within the block of bytes must also be aligned.

To accommodate discontinuous source data, the source and destination pitch registers can be used to specify the offset in bytes from the beginning of one scan line's worth source data to the next. Otherwise, if the source data is contiguous, then an offset equal to the length of a scan line's worth of source data should be specified.

14.2.2.3 Monochrome Source Data

The opcode of the command packet specifies whether the source data is color or monochrome. Since monochrome graphics data only uses one bit per pixel, each byte of monochrome source data typically carries data for 8 pixels which hinders the use of byte-oriented parameters when specifying the location and size of valid source data. Some additional parameters must be specified to ensure the proper reading and use of monochrome source data by the BLT engine. The BLT engine also provides additional options for the manipulation of monochrome source data versus color source data.

The various bit-wise logical operations and per-pixel write-masking operations were designed to work with color data. In order to use monochrome data, the BLT engine converts it into color through a process called color expansion, which takes place as a BLT operation is performed. In color expansion the single bits of monochrome source data are converted into one, two, or four bytes (depending on the color depth) of color data that are set to carry value corresponding to either the foreground or background color that have been specified for use in this conversion process. If a given bit of monochrome source data carries a value of 1, then the byte(s) of color data resulting from the conversion process will be set to carry the value of the foreground color. If a given bit of monochrome source data carries a value of 0, then the resulting byte(s) will be set to the value of the background color. The foreground and background colors used in the color expansion of monochrome source data can be set in the source expansion foreground color register and the source expansion background color register.

The BLT Engine requires that the bit alignment of each scan line's worth of monochrome source data be specified. Each scan line's worth of monochrome source data is word aligned but can actually start on any bit boundary of the first byte. Monochrome text is special cased and it is bit or byte packed, where in bit packed there are no invalid pixels (bits) between scan lines. There is a 3 bit field which indicates the starting pixel position within the first byte for each scan line, Mono Source Start.

The BLT engine also provides various clipping options for use with specific BLT commands (BLT_TEXT) with a monochrome source. Clipping is supported through: Clip rectangle Y addresses or coordinates and X coordinates along with scan line



starting and ending addresses (with Y addresses) along with X starting and ending coordinates.

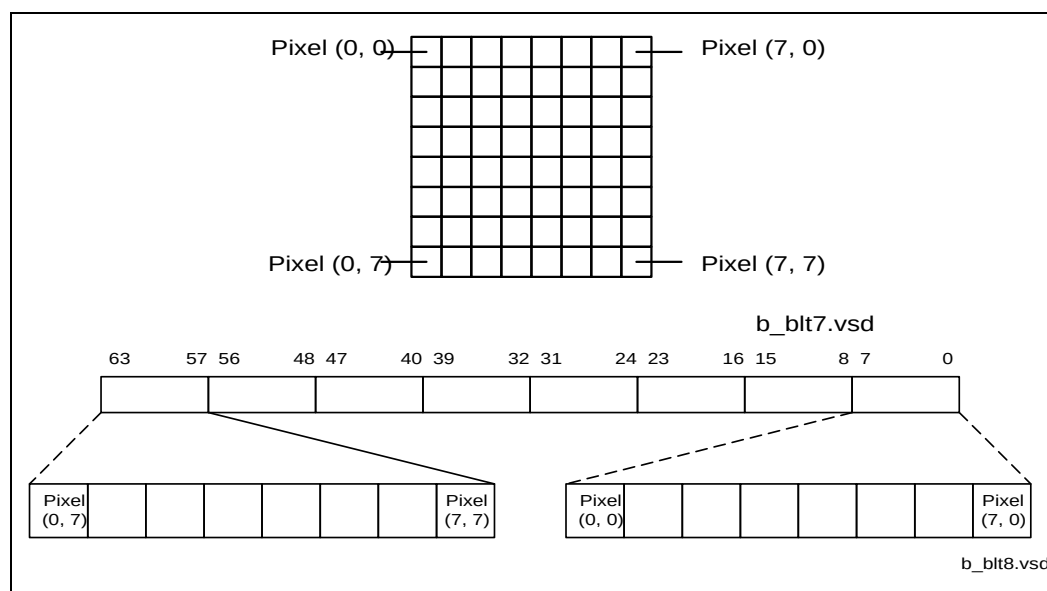
The maximum immediate source size is 128 bytes.

14.2.2.4 Pattern Data

The color pattern data must exist within the frame buffer or Graphics aperture where the BLT engine may read it directly or it can be sent through the command stream. The pattern data must be located in linear memory. Monochrome pattern data is supplied by the command packet when it is to be used. As shown in figure below, the block of pattern graphics data always represents a block of 8x8 pixels. The bits or bytes of a block of pattern data may be organized in the frame buffer memory in only one of three ways, depending upon its color depth which may be 8, 16, or 32 bits per pixel (whichever matches the color depth to which the BLT engine has been set), or monochrome.

The maximum color pattern size is 256 bytes.

Figure 14-8. Pattern Data -- Always an 8x8 Array of Pixels



The Pattern Address Register is used to specify the address of the color pattern data at which the block of pattern data begins. The three least significant bits of the address written to this register are ignored, because the address must be in terms of quadwords. This is because the pattern must always be located on an address boundary equal to its size. Monochrome patterns take up 8 bytes, or a single quadword of space, and are loaded through the command packet that uses it. Similarly, color patterns with color depths of 8, 16, and 32 bits per pixel must start on 64-byte, 128-byte and 256-byte boundaries, respectively. The next 3 figures show how monochrome, 8bpp, 16bpp, and 32bpp pattern data, respectively, is organized in memory.



Figure 14-9. 8bpp Pattern Data -- Occupies 64 Bytes (8 quadwords)

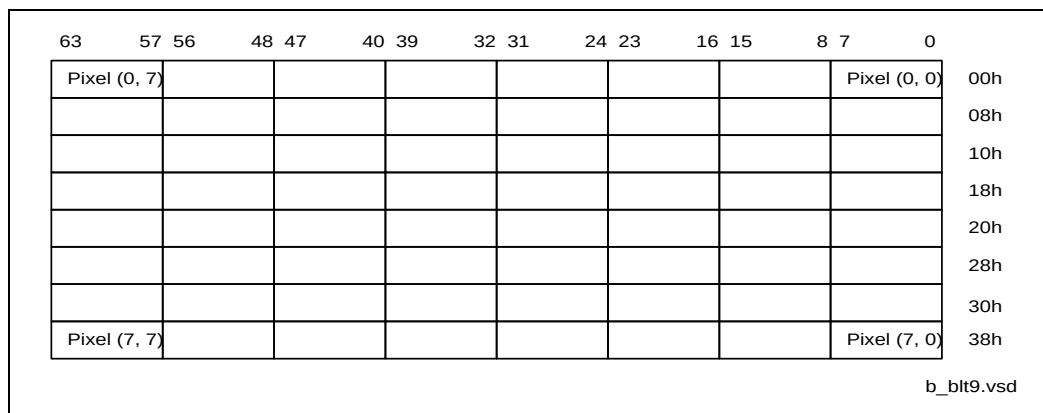


Figure 14-10. 16bpp Pattern Data -- Occupies 128 Bytes (16 quadwords)

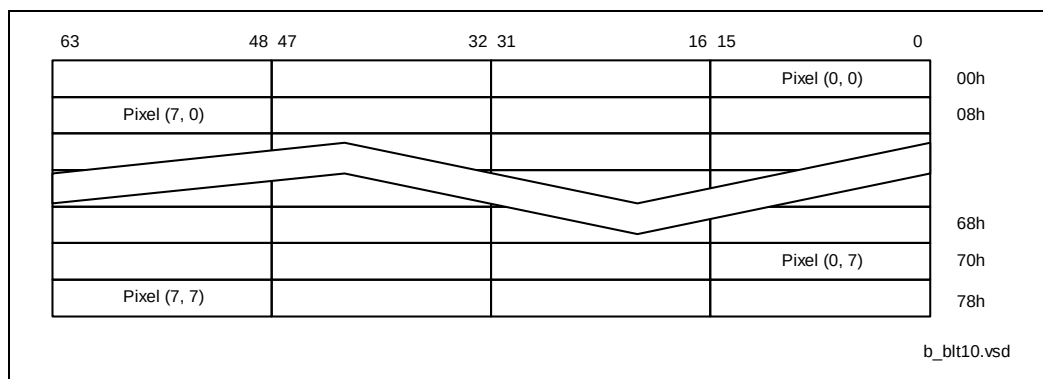
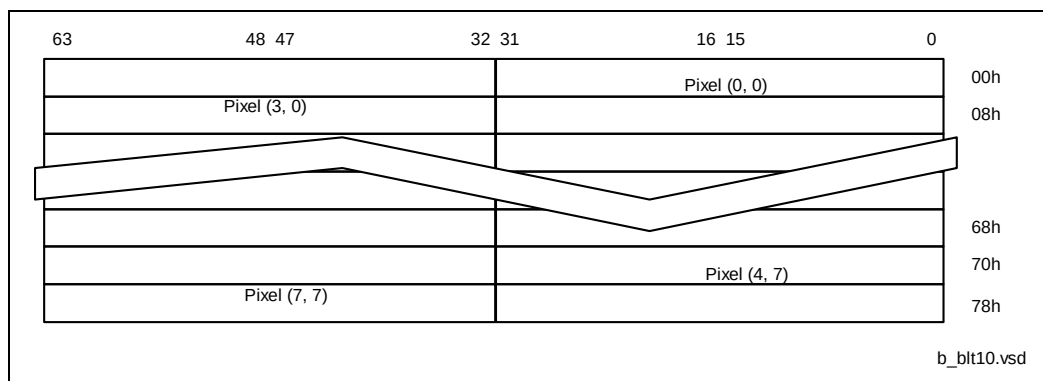


Figure 14-11. 32bpp Pattern Data -- Occupies 256 Bytes (32 quadwords)



The opcode of the command packet specifies whether the pattern data is color or monochrome. The various bit-wise logical operations and per-pixel write-masking operations were designed to work with color data. In order to use monochrome pattern data, the BLT engine is designed to convert it into color through a process called “color expansion” which takes place as a BLT operation is performed. In color expansion, the single bits of monochrome pattern data are converted into one, two, or four bytes (depending on the color depth) of color data that are set to carry values



corresponding to either the foreground or background color that have been specified for use in this process. The foreground color is used for pixels corresponding to a bit of monochrome pattern data that carry the value of 1, while the background color is used where the corresponding bit of monochrome pattern data carries the value of 0. The foreground and background colors used in the color expansion of monochrome pattern data can be set in the Pattern Expansion Foreground Color Register and Pattern Expansion Background Color Register.

14.2.2.5 Destination Data

There are actually two different types of “destination data”: the graphics data already residing at the location that is designated as the destination, and the data that is to be written into that very same location as a result of a BLT operation.

The location designated as the destination must be within the frame buffer or Graphics aperture where the BLT engine can read from it and write to it directly. The blocks of destination data to be read from and written to the destination may be either contiguous or discontinuous. All data written to the destination will have the color depth to which the BLT engine has been set. It is presumed that any data already existing at the destination which will be read by the BLT engine will also be of this same color depth — the BLT engine neither reads nor writes monochrome destination data.

The Destination Address Register is used to specify the address of the destination.

To accommodate discontinuous destination data, the Source and Destination Pitch Registers can be used to specify the offset in bytes from the beginning of one scan line's worth of destination data to the next. Otherwise, if the destination data is contiguous, then an offset equal to the length of a scan line's worth of destination data should be specified.

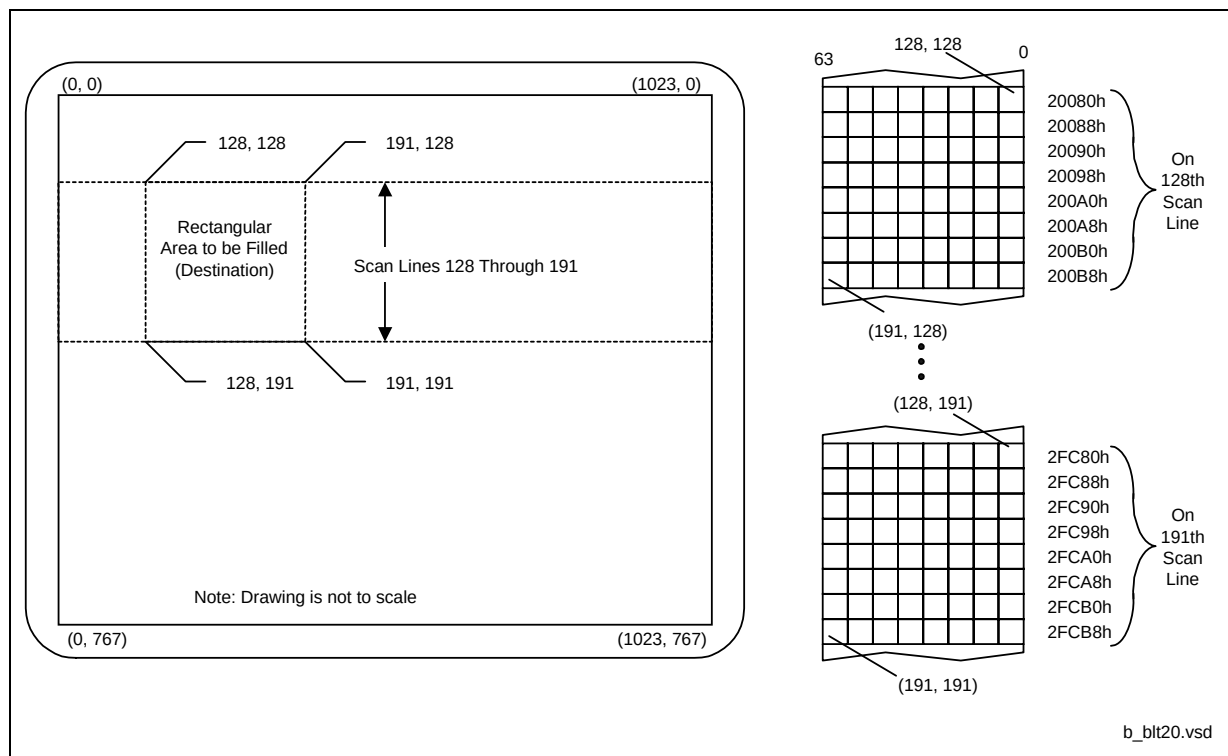


14.2.3 BLT Programming Examples

14.2.3.1 Pattern Fill — A Very Simple BLT

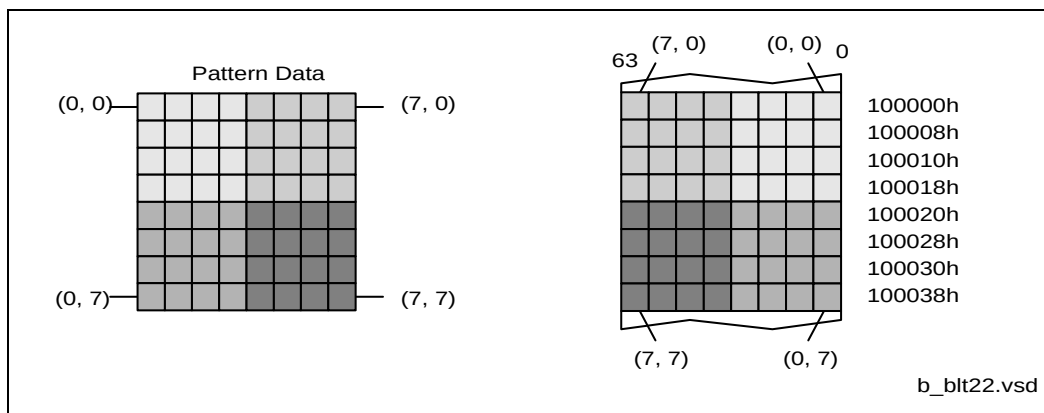
In this example, a rectangular area on the screen is to be filled with a color pattern stored as pattern data in off-screen memory. The screen has a resolution of 1024x768 and the graphics system has been set to a color depth of 8 bits per pixel.

Figure 14-12. On-Screen Destination for Example Pattern Fill BLT



As shown in the figure above, the rectangular area to be filled has its upper left-hand corner at coordinates (128, 128) and its lower right-hand corner at coordinates (191, 191). These coordinates define a rectangle covering 64 scan lines, each scan line's worth of which is 64 pixels in length — in other words, an array of 64x64 pixels. Presuming that the pixel at coordinates (0, 0) corresponds to the byte at address 00h in the frame buffer memory, the pixel at (128, 128) corresponds to the byte at address 20080h.

Figure 14-13. Pattern Data for Example Pattern Fill BLT



As shown in figure above, the pattern data occupies 64 bytes starting at address 100000h. As always, the pattern data represents an 8x8 array of pixels.

The BLT command packet is used to select the features to be used in this BLT operation, and must be programmed carefully. The vertical alignment field should be set to 0 to select the top-most horizontal row of the pattern as the starting row used in drawing the pattern starting with the top-most scan line covered by the destination. The pattern data is in color with a color depth of 8 bits per pixel, so the dynamic color enable should be asserted with the dynamic color depth field should be set to 0. Since this BLT operation does not use per-pixel write-masking (destination transparency mode), this field should be set to 0. Finally, the raster operation field should be programmed with the 8-bit value of F0h to select the bit-wise logical operation in which a simple copy of the pattern data to the destination takes place. Selecting this bit-wise operation in which no source data is used as an input causes the BLT engine to automatically forego either reading source data from the frame buffer.

The Destination Pitch Register must be programmed with number of bytes in the interval from the start of one scan line's worth of destination data to the next. Since the color depth is 8 bits per pixel and the horizontal resolution of the display is 1024, the value to be programmed into these bits is 400h, which is equal to the decimal value of 1024.

Bits [31:3] of the Pattern Address Register must be programmed with the address of the pattern data.

Similarly, bits [31:0] of the Destination Address Register must be programmed with the byte address at the destination that will be written to first. In this case, the address is 20080h, which corresponds to the byte representing the pixel at coordinates (128, 128).

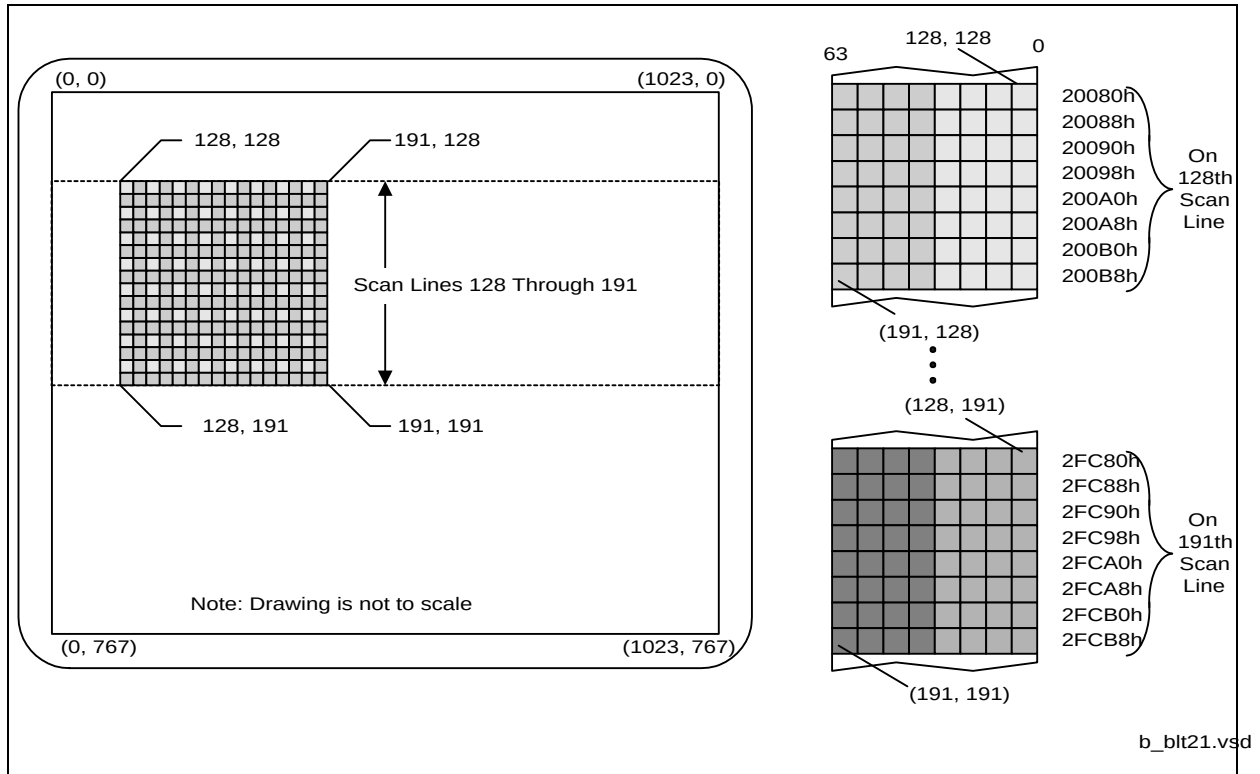
This BLT operation does not use the values in the Source Address Register or the Source Expansion Background or Foreground Color Registers.

The Destination Width and Height Registers (or the Destination X and Y Coordinates) must be programmed with values that describe to the BLT engine the 64x64 pixel size of the destination location. The height should be set to carry the value of 40h, indicating that the destination location covers 64 scan lines. The width should be set to carry the value of 40h, indicating that each scan line's worth of destination data



occupies 64 bytes. All of this information is written to the ring buffer using the PAT_BLT (or XY_PAT_BLT) command packet.

Figure 14-14. Results of Example Pattern Fill BLT

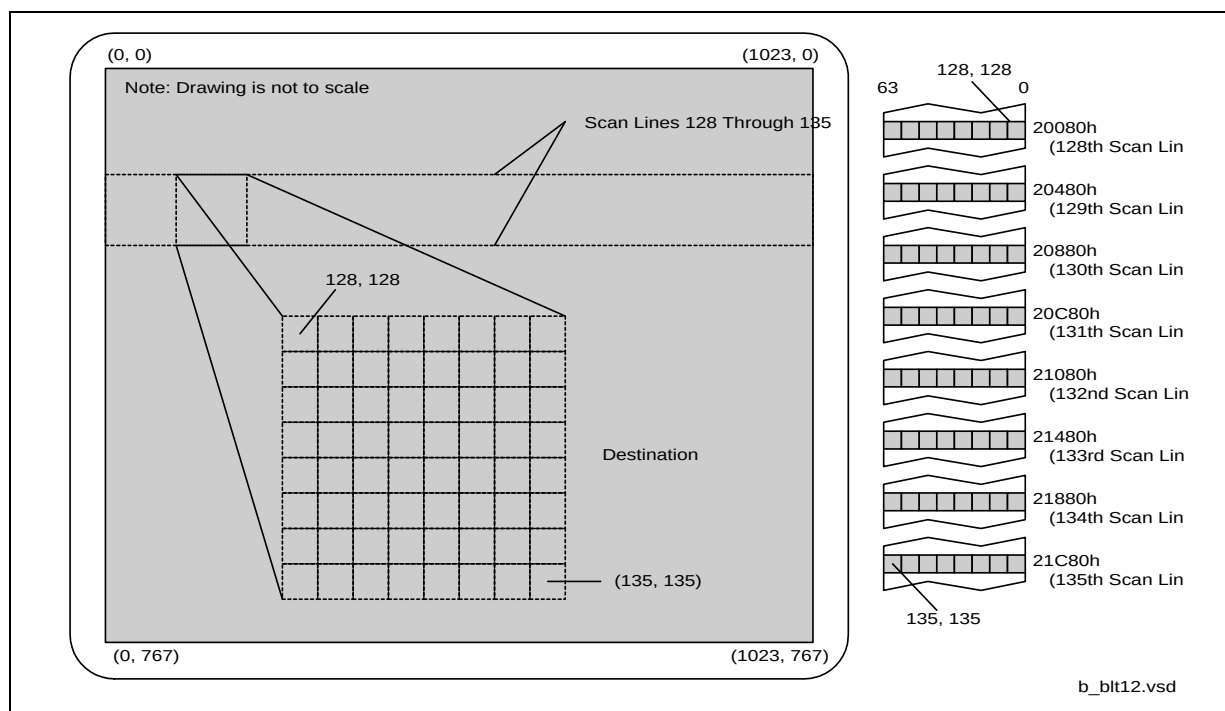


The figure above shows the end result of performing this BLT operation. The 8x8 pattern has been repeatedly copied ("tiled") into the entire 64x64 area at the destination.

14.2.3.2 Drawing Characters Using a Font Stored in System Memory

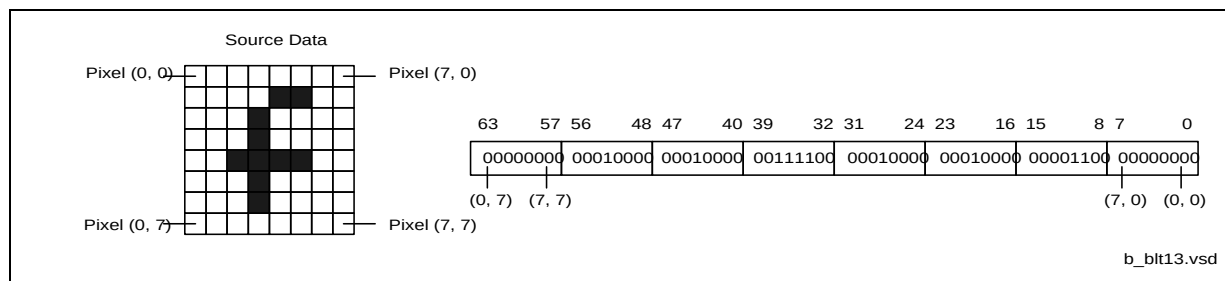
In this example BLT operation, a lowercase letter “f” is to be drawn in black on a display with a gray background. The resolution of the display is 1024x768, and the graphics system has been set to a color depth of 8 bits per pixel.

Figure 14-15. On-Screen Destination for Example Character Drawing BLT



The figure above shows the display on which this letter “f” is to be drawn. As shown in this figure, the entire display has been filled with a gray color. The letter “f” is to be drawn into an 8x8 region on the display with the upper left-hand corner at the coordinates (128, 128).

Figure 14-16. Source Data in System Memory for Example Character Drawing BLT



The figure above shows both the 8x8 pattern making up the letter “f” and how it is represented somewhere in the host’s system memory — the actual address in system memory is not important. The letter “f” is represented in system memory by a block of



monochrome graphics data that occupies 8 bytes. Each byte carries the 8 bits needed to represent the 8 pixels in each scan line's worth of this graphics data. This type of pattern is often used to store character fonts in system memory.

During this BLT operation, the host CPU will read this representation of the letter "f" from system memory, and write it to the BLT engine by performing memory writes to the ring buffer as an immediate monochrome BLT operand following the BLT_TEXT command. The BLT engine will receive this data through the command stream and use it as the source data for this BLT operation. The BLT engine will be set to the same color depth as the graphics system — 8 bits per pixel, in this case. Since the source data in this BLT operation is monochrome, color expansion must be used to convert it to an 8 bpp color depth. To ensure that the gray background behind this letter "f" is preserved, per-pixel write masking will be performed, using the monochrome source data as the pixel mask.

The BLT Setup and Text_immediate command packets are used to select the features to be used in this BLT operation. Only the fields required by these two command packets must be programmed carefully. The BLT engine ignores all other registers and fields. The source select field in the Text_immediate command must be set to 1, to indicate that the source data is provided by the host CPU through the command packet. Finally, the raster operation field should be programmed with the 8-bit value CCh to select the bit-wise logical operation that simply copies the source data to the destination. Selecting this bit-wise operation in which no pattern data is used as an input, causes the BLT engine to automatically forego reading pattern data from the frame buffer.

The Setup Pattern/Source Expansion Foreground Color Register to specify the color with which the letter "f" will be drawn. There is no Source address. All scan lines of the glyph are bit packed and the clipping is controlled by the ClipRect registers from the SETUP_BLT command and the Destination Y1, Y2, X1, and X2 registers in the TEXT_BLT command. Only the pixels that are within (inclusive comparisons) the clip rectangle are written to the destination surface.

The Destination Pitch Register must be programmed with a value equal to the number of bytes in the interval between the first bytes of each adjacent scan line's worth of destination data. Since the color depth is 8 bits per pixel and the horizontal resolution of the display is 1024 pixels, the value to be programmed into these bits is 400h, which is equal to the decimal value of 1024. Since the source data used in this BLT operation is monochrome, the BLT engine will not use a byte-oriented pitch value for the source data.

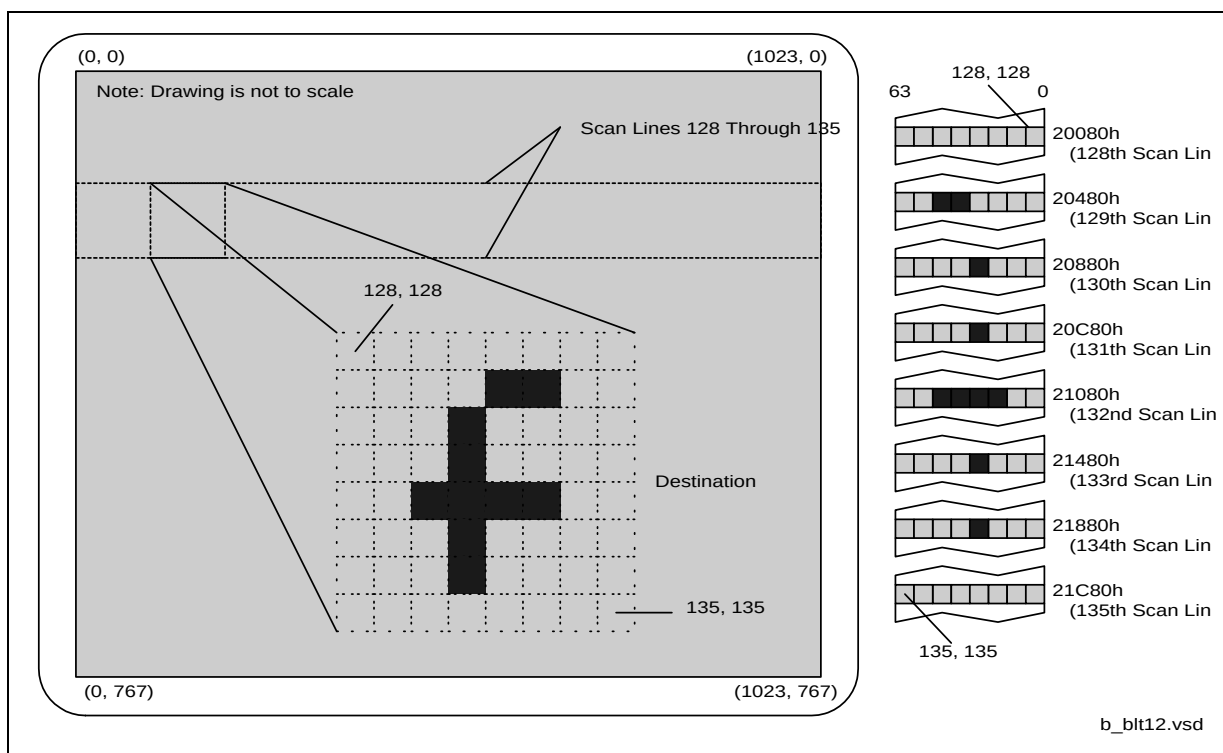
Since the source data is monochrome, color expansion is required to convert it to color with a color depth of 8 bits per pixel. Since the Setup Pattern/Source Expansion Foreground Color Register is selected to specify the foreground color of black to be used in drawing the letter "f", this register must be programmed with the value for that color. With the graphics system set for a color depth of 8 bits per pixel, the actual colors are specified in the RAMDAC palette, and the 8 bits stored in the frame buffer for each pixel actually specify the index used to select a color from that palette. This example assumes that the color specified at index 00h in the palette is black, and therefore bits [7:0] of this register should be set to 00h to select black as the foreground color. The BLT engine ignores bits [31:8] of this register because the selected color depth is 8 bits per pixel. Even though the color expansion being performed on the source data normally requires that both the foreground and background colors be specified, the value used to specify the background color is not important in this example. Per-pixel write-masking is being performed with the

monochrome source data as the pixel mask, which means that none of the pixels in the source data that will be converted to the background color will ever be written to the destination. Since these pixels will never be seen, the value programmed into the Pattern/Source Expansion Background Color Register to specify a background color is not important.

The Destination Width and Height Registers are not used. The Y1, Y2, X1, and X2 are used to describe to the BLT engine the 8x8 pixel size of the destination location. The Destination Y1 and Y2 address (or coordinate) registers must be programmed with the starting and ending scan line address (or Y coordinates) of the destination data. This address is specified as an offset from the start of the frame buffer of the scan line at the destination that will be written to first. The destination X1 and X2 registers must be programmed with the starting and ending pixel offsets from the beginning of the scan line.

This BLT operation does not use the values in the Pattern Address Register, the Source Expansion Background Color Register, or the Source Expansion Foreground Color Register.

Figure 14-17. Results of Example Character Drawing BLT



The preceding shows the end result of performing this BLT operation. Only the pixels that form part of the actual letter "f" have been drawn into the 8x8 destination location on the display, leaving the other pixels within the destination with their original gray color.



14.3 BLT Instruction Overview

This chapter defines the instructions used to control the 2D (BLT) rendering function.

The instructions detailed in this chapter are used across devices. However, slight changes may be present in some instructions (i.e., for features added or removed), or some instructions may be removed entirely. Refer to the *Device Dependencies* chapter for summary information regarding device-specific behaviors/interfaces/features.

The XY instructions offload the drivers by providing X and Y coordinates and taking care of the access directions for overlapping BLTs without fields specified by the driver.

Color pixel sizes supported are 8, 16, and 32 bits per pixel (bpp). All pixels are naturally aligned.

14.4 BLT Engine State

Most of the BLT instructions are state-free, which means that all states required to execute the command is within the instruction. If clipping is not used, then there is no shared state for many of the BLT instructions. This allows the BLT Engine to be shared by many drivers with minimal synchronization between the drivers.

Instructions which share state are:

- All instructions that are X,Y commands and use the Clipping Rectangle by asserting the Clip Enable field
- All XY_Setup Commands (XY_SETUP_BLT and XY_SETUP_MONO_PATTERN_SL_BLT) load the shared state for the following commands:
 - XY_PIXEL_BLT (Negative Stride (=Pitch) Not Allowed)
 - XY_SCANLINES_BLT
 - XY_TEXT_BLT (Negative Stride (=Pitch) Not Allowed)
 - XY_TEXT_IMMEDIATE_BLT (Negative Stride (=Pitch) Not Allowed)

State registers that are saved & restored in the Logical Context:

BR1+	Setup Control (Solid Pattern Select, Clipping Enable, Mono Source Transparency Mode, Mono Pattern Transparency Mode, Color Depth[1:0], Raster Operation[7:0], & Destination Pitch[15:0]) + 32bpp Channel Mask[1:0], Mono / Color Pattern
BR05	Setup Background Color
BR06	Setup Foreground Color
BR07	Setup Pattern Base Address
BR09	Setup Destination Base Address
BR20	DW0 for a Monochrome Pattern
BR21	DW1 for a Monochrome Pattern
BR24	ClipRectY1'X1
BR25	ClipRectY2'X2



14.5 Cacheable Memory Support

The BLT Engine can be used to transfer data between cacheable (“system”) memory and uncached (“main”, or “UC”) graphics memory using the BLT instructions. The GTT must be properly programmed to map memory pages as cacheable or UC. Only linear-mapped (not tiled) surfaces can be mapped as cacheable.

Transfers between cacheable sources and cacheable destinations are not supported. Patterns and monochrome sources cannot be located in cacheable memory.

Cacheable write operands do not snoop the processor’s cache nor update memory until evicted from the render cache. Cacheable read or write operands are not snooped (nor invalidated) from either internal cache by external (processor, hublink,...) accesses.

14.6 Device Cache Coherency: Render and Texture Caches

Software must initiate cache flushes to enforce coherency between the render and texture caches, i.e., both the render and texture caches must be flushed before a BLT destination surface can be reused as a texture source. Color sources and destinations use the render cache, while patterns and monochrome sources use the texture cache.



14.7 BLT Engine Instructions

The Instruction Target field is used as an opcode by the BLT Engine state machine to qualify the control bits that are relevant for executing the instruction. The descriptions for each DWord and bit field are contained in the *BLT Engine Instruction Field Definition* section. Each DWord field is described as a register, but none of these registers can be written or read through a memory mapped location – they are internal state only.

14.7.1 Blt Programming Restrictions

- **Overlapping Source/ Destination BLTs:** The following condition must be avoided when programming the Blt engine: Linear surfaces with a cache line in scan line Y for the source stream overlapping with a cache line in scan line Y-1 for the dest stream ($\Rightarrow$ non-aligned surface pitches). The cache coherency rules combined with the Blitter data consumption rules result in UNDEFINED operation. (Note that this restriction will likely follow forward to future products due to architectural complexities.) There are two suggested software workarounds:
 - In order to perform coherent overlapping Blts, (a) the Source and Destination Base Address registers must hold the same value (without alignment restriction), and (b) the Source and Destination Pitch registers (BR11, BR13) must both be a multiple of **64 bytes**.
 - If (a) isn't possible, do overlapping source copy BLTs as two blits, using a separate intermediate surface.
- All reserved fields must be programmed to 0s.
- When using monosource or text data (bit/byte/word aligned): do not program pixel widths greater than 32,745 pixels.

14.8 Fill / Move Instructions

These instructions use linear addresses with width and height. BLT clipping is not supported.



14.8.1 COLOR_BLT (Fill)

COLOR_BLT is the simplest BLT operation. It performs a color fill to the destination (with a possible ROP). The only operand is the destination operand which is written dependent on the raster operation. The solid pattern color is stored in the pattern background register.

This instruction is optimized to run at the maximum memory write bandwidth.

The typical Raster operation code = F0 which performs a copy of the pattern background register to the destination.

DWord	Bit	Description
0 = BR00	31:29	Client: 02h - 2D Processor
	28:22	Instruction Target (Opcode) : 40h
	21:20	32 bpp byte mask: (21 =1= write alpha channel; 20=1= write RGB channels)
	19:05	Reserved. Note no tiling specification allowed for this non-XY blit command. Only linear blits are allowed.
	04:00	DWord Length: 03h
1 = BR13	31:26	Reserved.
	25:24	Color Depth: 00 = 8 bit color. 01 = 16 bit color (656). 10 = 16 bit color (1555). 11 = 32 bit color
	23:16	Raster Operation:
	15:00	Destination Pitch (signed): Destination pitch in bytes (Same as before).
2 = BR14	31:16	Destination Height (in scan lines):
	15:00	Destination Width (in bytes):
3 = BR09	31:00	Destination Address: Address of the first byte to be written
4 = BR16	31:00	Solid Pattern Color: 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]



14.8.2 SRC_COPY_BLT (Move)

This BLT instruction performs a color source copy where the only operands involved is a color source and destination of the same bit width.

The source and destination operands may overlap. The command must indicate the horizontal and vertical directions: either forward or backwards to avoid data corruption. The X direction (horizontal) field applies to both the destination and source operands. The source and destination pitches (stride) are signed.

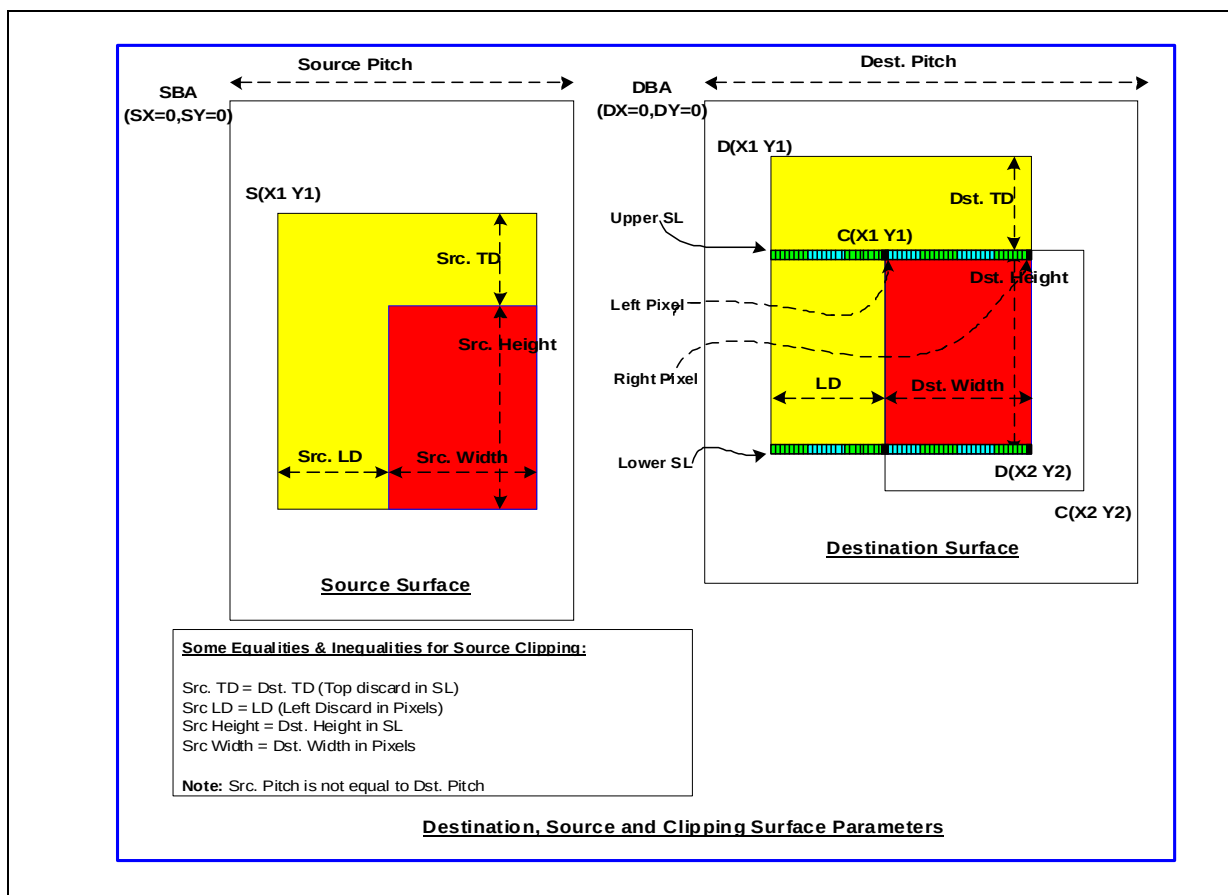
DWord	Bit	Description
0 = BR00	31:29	Client: 02h – 2D Processor
	28:22	Instruction Target (Opcode) : 43h
	21:20	32 bpp byte mask: (21 =1= write alpha channel; 20=1= write RGB channels)
	19:05	Reserved. Note no tiling specification allowed for this non-XY blit command. Only linear blits are allowed.
	04:00	Dword Length: 04h
1 = BR13	31	Reserved.
	30	X Direction (1 = written from right to left (decrementing = backwards); 0 = incrementing)
	29:26	Reserved.
	25:24	Color Depth: 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color
	23:16	Raster Operation:
	15:00	Destination Pitch (signed): Destination pitch in bytes (Same as before).
2 = BR14	31:16	Destination Height (in scan lines):
	15:00	Destination Width (in bytes):
3 = BR09	31:00	Destination Address: Address of the first byte to be written
4 = BR11	31:14	Reserved.
	15:00	Source Pitch: (double word aligned and signed)
5 = BR12	31:00	Source Address: Address of the first byte to be read.

14.9 2D (X,Y) BLT Instructions

Most BLT instructions (prefixed with “XY_”) use 2D X,Y coordinate specifications vs. lower-level linear addresses. These instructions also support simple 2D clipping against a clip rectangle.

The top and left Clipping coordinates are inclusive. The bottom and right coordinates are exclusive. The BLT Engine performs a trivial reject for all CLIP BLT instructions before performing any accesses.

Negative destination and source coordinates are supported. In the case of negative source coordinates, the destination X1 and Y1 are modified by the absolute value of the negative source coordinate before the destination clip checking and final drawing coordinates are calculated. The absolute value of the source negative coordinate is added to the corresponding destination coordinate. The BLT engine clipping also checks for $(DX2 \leq DX1)$ or $(DY2 \leq DY1)$ after this calculation and if true, then the BLT is totally rejected.





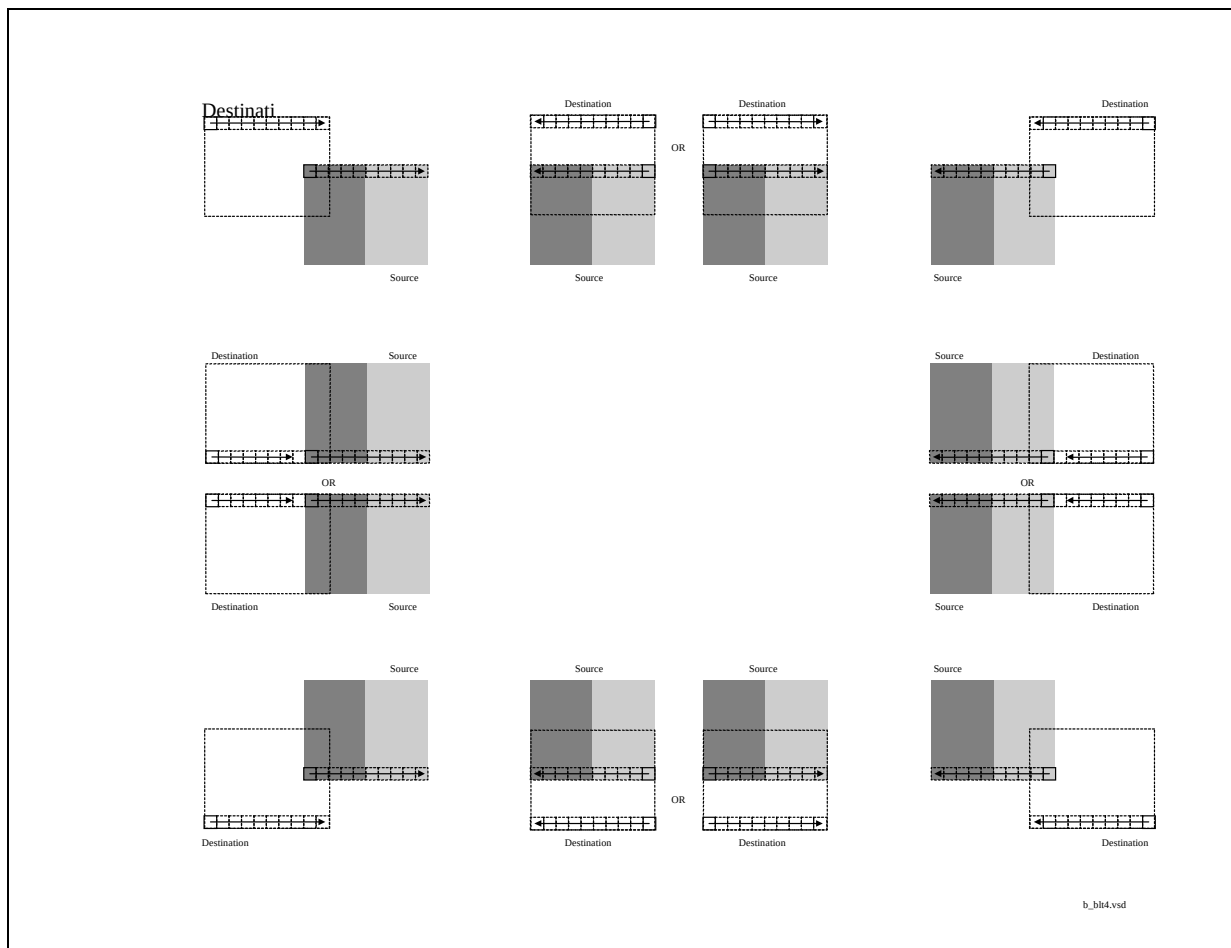
DX1, DY1, CX1, and CY1 are inclusive, while DX2, DY2, CX2, and CY2 are exclusive.

Destination pixel address = (Destination Base Address + (Destination Y coordinate * Destination pitch) + (Destination X coordinate * bytes per pixel)).

Source pixel address = (Source Base Address + (Source Y coordinate * Source pitch) + (Source X coordinate * bytes per pixel)).

Since there is 1 set of Clip Rectangle registers, the Interrupt Ring BLT commands either MUST NEVER enable clipping with these command and never use the XY_Pixel_BLT, XY_Scanline_BLT, nor XY_Text_BLT commands or it must use context switching. The Interrupt rings can also use the non-clipped, linear address commands specified before this section.

The base addresses plus the X and Y coordinates determine if there is an overlap between the source and destination operands. If the base addresses of the source and destination are the same and the Source X1 is **less than** Destination X1, then the BLT Engine performs the accesses in the X-backwards access pattern. There is no need to look for an actual overlap. If the base addresses are the same and Source Y1 is **less than** Destination Y1, then the scan line accesses are performed backwards.





14.9.1 XY_SETUP_BLT

This setup instruction supplies common setup information including clipping coordinates used by the XY commands: XY_PIXEL_BLT, XY_SCANLINE_BLT, XY_TEXT_BLT, and XY_TEXT_BLT_IMMEDIATE.

These are the only instructions that require that state be saved between instructions other than the Clipping parameters. There are 5 dedicated registers to contain the state for these 3 instructions. All other BLTs use a temporary version of these. The 5 double word registers are: DW1 (Setup Control), DW6 (Setup Foreground color), DW5 (Setup Background color), DW7 (Setup Pattern address), and DW4 (Setup Destination Base Address).

DWord	Bit	Description
0 = BR00	31:29	Client: 02h - 2D Processor
	28:22	Instruction Target (Opcode): 01h
	21:20	32 bpp byte mask: (21 =1= write alpha channel; 20=1= write RGB channels)
	19:12	Reserved.
	11	Tiling Enable: 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10: 08	Reserved
	07:00	Dword Length: 06h
1 = BR01	31	Reserved.
	30	Clipping Enable: (1 = enabled; 0 = disabled)
	29	Mono Source Transparency Mode: (1 = transparency enabled; 0 = use background)
	28:26	Reserved.
	25:24	Color Depth: All 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color
	23:16	Raster Operation:
	15:00	Destination Pitch in DWords: [15:00] 2's complement (Negative Pitch Not allowed for Pixel nor Text) For Tiled surfaces (bit_11 enabled) this ptich is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
2 = BR24	31:16	ClipRect Y1 Coordinate (Top): (30:16 = 15 bit positive number)
	15:00	ClipRect X1 Coordinate (Left): (14:00 = 15 bit positive number)
3 = BR25	31:16	ClipRect Y2 Coordinate (Bottom): (30:16 = 15 bit positive number)
	15:00	ClipRect X2 Coordinate (Right): (14:00 = 15 bit positive number)
4 = BR09	31:00	Setup Destination Base Address: (base address of the destination surface: X=0, Y=0) When Tiling is enabled (Bit_11 enabled), this address is limited to 4Kbytes.
5 = BR05	31:00	Setup Background Color: 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0] All
6 = BR06	31:00	Setup Foreground Color: 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0] (SLB & TB only)
7 = BR07	31:00	Setup Pattern Base Address for Color Pattern: (26:06 are implemented) (SLB only) (Note no NPO2 change here). The pattern data must be located in linear memory.



14.9.2 XY_SETUP_MONO_PATTERN_SL_BLT

This setup instruction supplies common setup information including clipping coordinates used exclusively with the following instruction: XY_SCANLINE_BLT (SLB)
- 1 scan line of monochrome pattern and destination are the only operands allowed.

DWord	Bit	Description
0 = BR00	31:29	Client: 02h - 2D Processor
	28:22	Instruction Target (Opcode): 11h
	21:20	32 bpp byte mask: (21 = 1= write alpha channel; 20=1= write RGB channels)
	19:12	Reserved.
	11	Tiling Enable: 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10: 08	Reserved
	07:00	Dword Length: 07h
1 = BR01	31	Solid Pattern Select: (1 = solid pattern; 0 = no solid pattern) - (SLB & Pixel only)
	30	Clipping Enable: (1 = enabled; 0 = disabled)
	29	Reserved.
	28	Mono Pattern Transparency Mode: (1 = transparency enabled; 0 = use background)
	27:26	Reserved.
	25:24	Color Depth: 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color
	23:16	Raster Operation:
	15:00	Destination Pitch in DWords: [15:00] 2's complement (Negative Pitch Not allowed for Pixel nor Text) For Tiled surfaces (bit_11 enabled) this pitch is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
2 = BR24	31:16	ClipRect Y1 Coordinate (Top): (30:16 = 15 bit positive number)
	15:00	ClipRect X1 Coordinate (Left): (14:00 = 15 bit positive number)
3 = BR25	31:16	ClipRect Y2 Coordinate (Bottom): (30:16 = 15 bit positive number)
	15:00	ClipRect X2 Coordinate (Right): (14:00 = 15 bit positive number)
4 = BR09	31:00	Setup Destination Base Address: (base address of the destination surface: X=0, Y=0) When Tiling is enabled (Bit_11 enabled), this address is limited to 4Kbytes.
5 = BR05	31:00	Setup Background Color: 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]
6 = BR06	31:00	Setup Foreground Color: 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]
7 = BR20	31:00	DW0 (least significant) for a Monochrome Pattern:
8 = BR21	31:00	DW1 (most significant) for a Monochrome Pattern:



14.9.3 XY_SETUP_CLIP_BLT

This command is used to only change the clip coordinate registers. These are the same clipping registers as the Setup clipping registers above.

DWord	Bit	Description
0 = BR00	31:29	Client: 02h - 2D Processor
	28:22	Instruction Target (Opcode): 03h
	21:12	Reserved.
	11	Tiling Enable: 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10: 08	Reserved
	07:00	Dword Length: 01h
1 = BR24	31:16	ClipRect Y1 Coordinate (Top): (30:16 = 15 bit positive number)
	15:00	ClipRect X1 Coordinate (Left): (14:00 = 15 bit positive number)
2 = BR25	31:16	ClipRect Y2 Coordinate (Bottom): (30:16 = 15 bit positive number)
	15:00	ClipRect X2 Coordinate (Right): (14:00 = 15 bit positive number)

14.9.4 XY_PIXEL_BLT

The Destination X coordinate and Destination Y coordinate is compared with the ClipRect registers. If it is within all 4 comparisons, then the pixel supplied in the XY_SETUP_BLT instruction is written with the raster operation to (Destination Y Address + (Destination Y coordinate * Destination pitch) + (Destination X coordinate * bytes per pixel)).

ROP field must specify pattern or fill with 0's or 1's. There is no source operand.

Negative Stride (= Pitch) specified in the Setup command is Not Allowed

DWord	Bit	Description
0 = BR00	31:29	Client: 02h - 2D Processor
	28:22	Instruction Target (Opcode): 24h
	21:12	Reserved.
	11	Tiling Enable: 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10: 08	Reserved
	07:00	Dword Length : 00h
1 = BR22	31:16	Destination Y1 Coordinate (Top): (31:16 = 16 bit signed number)
	15:00	Destination X1 Coordinate (Left): (15:00 = 16 bit signed number)



14.9.5 XY_SCANLINES_BLT

All scan lines and pixels that fall within the ClipRect Y and X coordinates are written. Only pixels within the ClipRectX coordinates and the Destination X coordinates are written using the raster operation.

The Pattern Seeds correspond to Destination X = 0 (horizontal) and Y = 0 (vertical). The alignment is relative to the destination coordinates. The pixel of the pattern used / scan line is the (destination X coordinate + horizontal seed) modulo 8. The scan line of the pattern used is the (destination Y coordinate + vertical seed) modulo 8.

Solid pattern should use the XY_SETUP_MONO_PATTERN_SL_BLT instruction.

ROP field must specify pattern or fill with 0's or 1's. There is no source operand.

DWord	Bit	Description
0 = BR00	31:29	Client: 02h - 2D Processor
	28:22	Instruction Target (Opcode): 25h
	21:15	Reserved.
	14:12	Pattern Horizontal Seed: (pixel of the scan line to start on corresponding to DST X=0)
	11	Tiling Enable: 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10:08	Pattern Vertical Seed: (scan line of the 8x8 pattern to start on corresponding to DST Y=0)
	07:00	Dword Length: 01h
1 = BR22	31:16	Destination Y1 Coordinate (Top): (31:16 = 16 bit signed number)
	15:00	Destination X1 Coordinate (Left): (15:00 = 16 bit signed number)
2 = BR23	31:16	Destination Y2 Coordinate (Bottom): (31:16 = 16 bit signed number)
	15:00	Destination X2 Coordinate (Right): (15:00 = 16 bit signed number)



14.9.6 XY_TEXT_BLT

All source scan lines and pixels that fall within the ClipRect Y and X coordinates are written. The source address corresponds to Destination X1 and Y1 coordinate.

Text is either bit or byte packed. Bit packed means that the next scan line starts 1 pixel after the end of the current scan line with no bit padding. Byte packed means that the next scan line starts on the first bit of the next byte boundary after the last bit of the current line.

Source expansion color registers are always in the SETUP_BLT.

Negative Stride (= Pitch) is NOT ALLOWED.

DWord	Bit	Description
0 = BR00	31:29	Client: 02h - 2D Processor
	28:22	Instruction Target (Opcode): 26h
	21:17	Reserved.
	16	Bit (0) / Byte (1) packed: Byte packed is for the NT driver
	15:12	Reserved.
	11	Tiling Enable: 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10: 08	Reserved
	07:00	Dword Length: 02h
1 = BR22	31:16	Destination Y1 Coordinate (Top): (31:16 = 16 bit signed number)
	15:00	Destination X1 Coordinate (Left): (15:00 = 16 bit signed number)
2 = BR23	31:16	Destination Y2 Coordinate (Bottom): (31:16 = 16 bit signed number)
	15:00	Destination X2 Coordinate (Right): (15:00 = 16 bit signed number)
3 = BR12	31:00	Source Address: (address of the first byte on scan line corresponding to Dst X1,Y1) (Note no NPO2 change here)



14.9.7 XY_TEXT_IMMEDIATE_BLT

This instruction allows the Driver to send data through the instruction stream that eliminates the read latency of reading a source from memory. If an operand is in system cacheable memory and either small or only accessed once, it can be copied directly to the instruction stream versus to graphics accessible memory.

The IMMEDIATE_BLT data MUST transfer an even number of doublewords. The BLT engine will hang if it does not get an even number of doublewords.

All source scan lines and pixels that fall within the ClipRect X and Y coordinates are written. The source data corresponds to Destination X1 and Y1 coordinate.

Source expansion color registers are always in the SETUP_BLT.

NEGATIVE STRIDE (= PITCH) IS NOT ALLOWED.

DWord	Bit	Description
0 = BR00	31:29	Client: 02h – 2D Processor
	28:22	Instruction Target (Opcode): 31h
	21:17	Reserved.
	16	Bit (0) / Byte (1) packed: Byte packed is for the NT driver
	15:12	Reserved.
	11	Tiling Enable: 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10: 08	Reserved
	07:00	Dword Length : 01+ DWL = (Number of Immediate double words)h
1 = BR22	31:16	Destination Y1 Coordinate (Top): (31:16 = 16 bit signed number)
	15:00	Destination X1 Coordinate (Left): (15:00 = 16 bit signed number)
2 = BR23	31:16	Destination Y2 Coordinate (Bottom): (31:16 = 16 bit signed number)
	15:00	Destination X2 Coordinate (Right): (15:00 = 16 bit signed number)
3	31:00	Immediate Data DW 0:
4	31:00	Immediate Data DW 1:
5 thru DWL+3	S	Immediate Data DWs 2 through DWORD_LENGTH (DWL):



14.9.8 XY_COLOR_BLT

COLOR_BLT is the simplest BLT operation. It performs a color fill to the destination (with a possible ROP). The only operand is the destination operand which is written dependent on the raster operation. The solid pattern color is stored in the pattern background register.

This instruction is optimized to run at the maximum memory write bandwidth.

The typical (and fastest) Raster operation code = F0 which performs a copy of the pattern background register to the destination.

DWord	Bit	Description
0 = BR00	31:29	Client: 02h - 2D Processor
	28:22	Instruction Target (Opcode): 50h
	21:20	32 bpp byte mask: (21 =1= write alpha channel; 20=1= write RGB channels)
	19:12	Reserved.
	11	Tiling Enable: 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10: 08	Reserved
	07:00	Dword Length: 04h
1 = BR13	31	Reserved.
	30	Clipping Enable: (1 = enabled; 0 = disabled)
	29:26	Reserved.
	25:24	Color Depth: 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color
	23:16	Raster Operation:
	15:00	Destination Pitch in DWords: [15:00] 2's complement For Tiled surfaces (bit_11 enabled) this ptich is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
2 = BR22	31:16	Destination Y1 Coordinate (Top): (31:16 = 16 bit signed number)
	15:00	Destination X1 Coordinate (Left): (15:00 = 16 bit signed number)
3 = BR23	31:16	Destination Y2 Coordinate (Bottom): (31:16 = 16 bit signed number)
	15:00	Destination X2 Coordinate (Right): (15:00 = 16 bit signed number)
4 = BR09	31:00	Destination Base Address: (base address of the destination surface: X=0, Y=0) When Tiling is enabled (Bit_11 enabled), this address is limited to 4Kbytes.
5 = BR16	31:00	Solid Pattern Color: 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]



14.9.9 XY_PAT_BLT

PAT_BLT is used when there is no source and the color pattern is not trivial (is not a solid color only).

If clipping is enabled, all scan lines and pixels that fall within the ClipRect Y and X coordinates are written. Only pixels within the ClipRectX coordinates and the Destination X coordinates are written using the raster operation.

The Pattern Seeds correspond to Destination X = 0 (horizontal) and Y = 0 (vertical). The alignment is relative to the destination coordinates. The pixel of the pattern used / scan line is the (destination X coordinate + horizontal seed) modulo 8. The scan line of the pattern used is the (destination Y coordinate + vertical seed) modulo 8.

DWord	Bit	Description
0 = BR00	31:29	Client: 02h - 2D Processor
	28:22	Instruction Target (Opcode): 51h
	21:20	32 bpp byte mask: (21 =1= write alpha channel; 20=1= write RGB channels)
	19:15	Reserved.
	14:12	Pattern Horizontal Seed (pixel of the scan line to start on corresponding to DST X=0)
	11	Tiling Enable: 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10:08	Pattern Vertical Seed: (Starting Scan line of the 8x8 pattern corresponding to DST Y=0)
	07:00	Dword Length: 04h
1 = BR13	31	Reserved.
	30	Clipping Enable: (1 = enabled; 0 = disabled)
	29:26	Reserved.
	25:24	Color Depth: 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color
	23:16	Raster Operation:
	15:00	Destination Pitch in DWords: [15:00] 2's complement For Tiled surfaces (bit_11 enabled) this pitch is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
2 = BR22	31:16	Destination Y1 Coordinate (Top): (31:16 = 16 bit signed number)
	15:00	Destination X1 Coordinate (Left): (15:00 = 16 bit signed number)
3 = BR23	31:16	Destination Y2 Coordinate (Bottom): (31:16 = 16 bit signed number)
	15:00	Destination X2 Coordinate (Right): (15:00 = 16 bit signed number)
4 = BR09	31:00	Destination Base Address: (base address of the destination surface: X=0, Y=0) When Tiling is enabled (Bit_11 enabled), this address is limited to 4Kbytes.
5 = BR15	31:00	Pattern Base Address: (28:06 are implemented) (Note no NPO2 change here). The pattern data must be located in linear memory.



14.9.10 XY_PAT_CHROMA_BLT

PAT_BLT is used when there is no source and the color pattern is not trivial (is not a solid color only).

All scan lines and pixels that fall within the ClipRect Y and X coordinates are written. Only pixels within the ClipRectX coordinates and the Destination X coordinates are written using the raster operation.

The Pattern Seeds correspond to Destination X = 0 (horizontal) and Y = 0 (vertical). The alignment is relative to the destination coordinates. The pixel of the pattern used / scan line is the (destination X coordinate + horizontal seed) modulo 8. The scan line of the pattern used is the (destination Y coordinate + vertical seed) modulo 8.

DWord	Bit	Description
0 = BR00	31:29	Client: 02h - 2D Processor
	28:22	Instruction Target (Opcode): 76h
	21:20	32 bpp byte mask: (21 =1= write alpha channel; 20=1= write RGB channels)
	19:17	Transparency Range Mode: (chroma-key) – Dst Chroma-key modes ONLY (SRC ILLEGAL)
	16:15	Reserved.
	14:12	Pattern Horizontal Seed (pixel of the scan line to start on corresponding to DST X=0)
	11	Tiling Enable: 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10:08	Pattern Vertical Seed: (Starting Scan line of the 8x8 pattern corresponding to DST Y=0)
	07:00	Dword Length: 06h
1 = BR13	31	Reserved.
	30	Clipping Enable: (1 = enabled; 0 = disabled)
	29:26	Reserved.
	25:24	Color Depth: 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color
	23:16	Raster Operation:
	15:00	Destination Pitch in DWords: [15:00] 2's complement For Tiled surfaces (bit_11 enabled) this pitch is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
2 = BR22	31:16	Destination Y1 Coordinate (Top): (31:16 = 16 bit signed number)
	15:00	Destination X1 Coordinate (Left): (15:00 = 16 bit signed number)
3 = BR23	31:16	Destination Y2 Coordinate (Bottom): (31:16 = 16 bit signed number)
	15:00	Destination X2 Coordinate (Right): (15:00 = 16 bit signed number)
4 = BR09	31:00	Destination Base Address: (base address of the destination surface: X=0, Y=0) When Tiling is enabled (Bit_11 enabled), this address is limited to 4Kbytes.
5 = BR15	31:00	Pattern Base Address: (26:06 are used, other bits are ignored) (Note no NPO2 change here). The pattern data must be located in linear memory.
6 = BR18	31:00	Transparency Color Low: (Chroma-key Low = Pixel Greater or Equal)
7 = BR19	31:00	Transparency Color High: (Chroma-key High = Pixel Less or Equal)



14.9.11 XY_PAT_BLT_IMMEDIATE

PAT_BLT_IMMEDIATE is used when there is no source and the color pattern is not trivial (is not a solid color only) and the pattern is pulled through the command stream. The immediate data sizes are 64 bytes (16 DWs), 128 bytes (32 DWs), or 256 (64DWs) for 8, 16, and 32 bpp color patterns.

All scan lines and pixels that fall within the ClipRect Y and X coordinates are written. Only pixels within the ClipRectX coordinates and the Destination X coordinates are written using the raster operation.

The Pattern Seeds correspond to Destination X = 0 (horizontal) and Y = 0 (vertical). The alignment is relative to the destination coordinates. The pixel of the pattern used / scan line is the (destination X coordinate + horizontal seed) modulo 8. The scan line of the pattern used is the (destination Y coordinate + vertical seed) modulo 8.

DWord	Bit	Description
0 = BR00	31:29	Client: 02h - 2D Processor
	28:22	Instruction Target (Opcode): 72h
	21:20	32 bpp byte mask: (21 =1= write alpha channel; 20=1= write RGB channels)
	19:15	Reserved.
	14:12	Pattern Horizontal Seed (pixel of the scan line to start on corresponding to DST X=0)
	11	Tiling Enable: 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10:08	Pattern Vertical Seed: (starting scan line of the 8x8 pattern corresponding to DST Y=0)
	07:00	Dword Length: 03+ DWL = (Number of Immediate double)h
1 = BR13	31	Reserved
	30	Clipping Enable: (1 = enabled; 0 = disabled)
	29:26	Reserved.
	25:24	Color Depth: 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color
	23:16	Raster Operation:
	15:00	Destination Pitch in DWords: [15:00] 2's complement For Tiled surfaces (bit_11 enabled) this pitch is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
2 = BR22	31:16	Destination Y1 Coordinate (Top): (31:16 = 16 bit signed number)
	15:00	Destination X1 Coordinate (Left): (15:00 = 16 bit signed number)
3 = BR23	31:16	Destination Y2 Coordinate (Bottom): (31:16 = 16 bit signed number)
	15:00	Destination X2 Coordinate (Right): (15:00 = 16 bit signed number)
4 = BR09	31:00	Destination Base Address: (base address of the destination surface: X=0, Y=0) When Tiling is enabled (Bit_11 enabled), this address is limited to 4Kbytes.
5	31:00	Immediate Data DW 0:
6	31:00	Immediate Data DW 1:
7 thru DWL+3	S	Immediate Data DWs 2 through DWORD_LENGTH (DWL):



14.9.12 XY_PAT_CHROMA_BLT_IMMEDIATE

PAT_BLT_IMMEDIATE is used when there is no source and the color pattern is not trivial (is not a solid color only) and the pattern is pulled through the command stream. The immediate data sizes are 64 bytes (16 DWs), 128 bytes (32 DWs), or 256 (64DWs) for 8, 16, and 32 bpp color patterns.

All scan lines and pixels that fall within the ClipRect Y and X coordinates are written. Only pixels within the ClipRectX coordinates and the Destination X coordinates are written using the raster operation.

The Pattern Seeds correspond to Destination X = 0 (horizontal) and Y = 0 (vertical). The alignment is relative to the destination coordinates. The pixel of the pattern used / scan line is the (destination X coordinate + horizontal seed) modulo 8. The scan line of the pattern used is the (destination Y coordinate + vertical seed) modulo 8.

DWord	Bit	Description
0 = BR00	31:29	Client: 02h - 2D Processor
	28:22	Instruction Target (Opcode): 77h
	21:20	32 bpp byte mask: (21 =1= write alpha channel; 20=1= write RGB channels)
	19:17	Transparency Range Mode: (chroma-key) – Dst Chroma-key modes ONLY (SRC ILLEGAL)
	16:15	Reserved.
	14:12	Pattern Horizontal Seed (pixel of the scan line to start on corresponding to DST X=0)
	11	Tiling Enable: 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10:08	Pattern Vertical Seed: (starting scan line of the 8x8 pattern corresponding to DST Y=0)
	07:00	Dword Length: 05+ DWL = (Number of Immediate double)h
1 = BR13	31	Reserved.
	30	Clipping Enable (1 = enabled; 0 = disabled)
	29:26	Reserved.
	25:24	Color Depth: 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color
	23:16	Raster Operation:
	15:00	Destination Pitch in DWords: [15:00] 2's complement For Tiled surfaces (bit_11 enabled) this pitch is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
2 = BR22	31:16	Destination Y1 Coordinate (Top): (31:16 = 16 bit signed number)
	15:00	Destination X1 Coordinate (Left): (15:00 = 16 bit signed number)
3 = BR23	31:16	Destination Y2 Coordinate (Bottom): (31:16 = 16 bit signed number)
	15:00	Destination X2 Coordinate (Right): (15:00 = 16 bit signed number)
4 = BR09	31:00	Destination Base Address: (base address of the destination surface: X=0, Y=0) When Tiling is enabled (Bit_11 enabled), this address is limited to 4Kbytes.
5 = BR18	31:00	Transparency Color Low: (Chroma-key Low = Pixel Greater or Equal)
6 = BR19	31:00	Transparency Color High: (Chroma-key High = Pixel Less or Equal)
7	31:00	Immediate Data DW 0:
8	31:00	Immediate Data DW 1:
9 thru DWL+3	S	Immediate Data DWs 2 through DWORD_LENGTH (DWL):



14.9.13 XY_MONO_PAT_BLT

MONO_PAT_BLT is used when we have no source and the monochrome pattern is not trivial (is not a solid color only). The monochrome pattern is loaded from the instruction stream.

All scan lines and pixels that fall within the ClipRect Y and X coordinates are written. Only pixels within the ClipRectX coordinates and the Destination X coordinates are written using the raster operation.

The Pattern Seeds correspond to Destination X = 0 (horizontal) and Y = 0 (vertical). The alignment is relative to the destination coordinates. The pixel of the pattern used / scan line is the (destination X coordinate + horizontal seed) modulo 8. The scan line of the pattern used is the (destination Y coordinate + vertical seed) modulo 8.

The monochrome pattern transparency mode indicates whether to use the pattern background color or de-assert the write enables when the bit in the pattern is 0. When the pattern bit is 1, then the pattern foreground color is used in the ROP operation.

DWord	Bit	Description
0 = BR00	31:29	Client: 02h - 2D Processor
	28:22	Instruction Target (Opcode) : 52h
	21:20	32 bpp byte mask: (21 =1= write alpha channel; 20=1= write RGB channels)
	19:15	Reserved.
	14:12	Pattern Horizontal Seed: (pixel of the scan line to start on corresponding to DST X=0)
	11	Tiling Enable: 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10:08	Pattern Vertical Seed: (starting scan line of the 8x8 pattern corresponding to DST Y=0)
	07:00	Dword Length: 07h
1 = BR13	31	Reserved.
	30	Clipping Enable (1 = enabled; 0 = disabled)
	29	Reserved.
	28	Mono Pattern Transparency Mode: (1 = transparency enabled; 0 = use background)
	27:26	Reserved.
	25:24	Color Depth: 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color
	23:16	Raster Operation:
	15:00	Destination Pitch in DWords: [15:00] 2's complement For Tiled surfaces (bit_11 enabled) this pitch is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
2 = BR22	31:16	Destination Y1 Coordinate (Top): 31:16 = 16 bit signed number)
	15:00	Destination X1 Coordinate (Left): (15:00 = 16 bit signed number)



DWord	Bit	Description
3 = BR23	31:16	Destination Y2 Coordinate (Bottom): (31:16 = 16 bit signed number)
	15:00	Destination X2 Coordinate (Right): (15:00 = 16 bit signed number)
4 = BR09	31:00	Destination Base Address: (base address of the destination surface: X=0, Y=0) When Tiling is enabled (Bit_11 enabled), this address is limited to 4Kbytes.
5 = BR16	31:00	Pattern Background Color: 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]
6 = BR17	31:00	Pattern Foreground Color: 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]
7 = BR20	31:00	Pattern Data 0: (least significant DW)
8 = BR21	31:00	Pattern Data 1: (most significant DW)



14.9.14 XY_MONO_PAT_FIXED_BLT

MONO_PAT_FIXED_BLT is used when we have no source and the monochrome pattern is not trivial (is not a solid color only). The monochrome pattern is one of 10 fixed patterns described below. The pattern seeds can still be used with the fixed patterns, creating even more fixed patterns. This eliminates 2 doublewords compared to the XY_MONO_PAT_BLT command packet.

All scan lines and pixels that fall within the ClipRect Y and X coordinates are written. Only pixels within the ClipRectX coordinates and the Destination X coordinates are written using the raster operation.

The Pattern Seeds correspond to Destination X = 0 (horizontal) and Y = 0 (vertical). The alignment is relative to the destination coordinates. The pixel of the pattern used / scan line is the (destination X coordinate + horizontal seed) modulo 8. The scan line of the pattern used is the (destination Y coordinate + vertical seed) modulo 8.

The monochrome pattern transparency mode indicates whether to use the pattern background color or de-assert the write enables when the bit in the pattern is 0. When the pattern bit is 1, then the pattern foreground color is used in the ROP operation.

DWord	Bit	Description
0 = BR00	31:29	Client: 02h - 2D Processor
	28:22	Instruction Target (Opcode): 59h
	21:20	32 bpp byte mask: (21 =1= write alpha channel; 20=1= write RGB channels)
	19	Reserved.
	18:15	Fixed Pattern: 0000 HS_HORIZONTAL 0001 HS_VERTICAL 0010 HS_FDIAGONAL 0011 HS_BDIAGONAL 0100 HS_CROSS 0101 HS_DIAGCROSS 0110 Reserved 0111 Reserved 1000 Screen Door 1001 SD Wide 1010 Walking Bit (one) 1011 Walking Zero 1100 Reserved 1101 Reserved 1110 Reserved
	14:12	Pattern Horizontal Seed: (pixel of the scan line to start on corresponding to DST X=0)
	11	Tiling Enable: 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10:08	Pattern Vertical Seed: (starting scan line of the 8x8 pattern corresponding to DST Y=0)
	07:00	Dword Length: 05h
1 = BR13	31	Reserved.

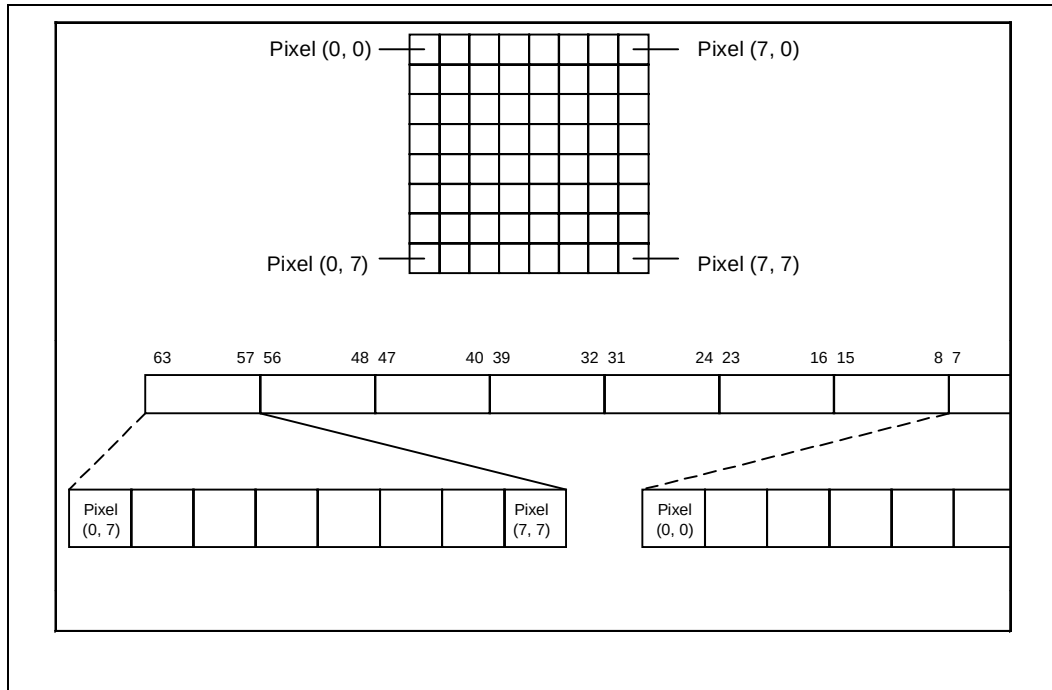


DWord	Bit	Description
	30	Clipping Enable (1 = enabled; 0 = disabled)
	29	Reserved.
	28	Mono Pattern Transparency Mode: (1 = transparency enabled; 0 = use background)
	27	Bit Mask Enable: (1 = use bit mask register for bit writes; 0 = disabled)
	27:26	Reserved.
	25:24	Color Depth: 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color
	23:16	Raster Operation:
	15:00	Destination Pitch in DWords: [15:00] 2's complement For Tiled surfaces (bit_11 enabled) this pitch is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
2 = BR22	31:16	Destination Y1 Coordinate (Top): (31:16 = 16 bit signed number)
	15:00	Destination X1 Coordinate (Left): (15:00 = 16 bit signed number)
3 = BR23	31:16	Destination Y2 Coordinate (Bottom): (31:16 = 16 bit signed number)
	15:00	Destination X2 Coordinate (Right): (15:00 = 16 bit signed number)
4 = BR09	31:00	Destination Base Address: (base address of the destination surface: X=0, Y=0) When Tiling is enabled (Bit_11 enabled), this address is limited to 4Kbytes.
5 = BR16	31:00	Pattern Background Color: 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]
6 = BR17	31:00	Pattern Foreground Color: 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]



14.9.14.1 Monochrome Pattern Memory Format

The monochrome pattern is made of 8 bytes that correspond to the 8 pixels per scan line and 8 scan lines. Byte 0 corresponds to scan line 0, byte 1 corresponds to scan line 1,..., and byte 7 corresponds to scan line 7. The bits within each byte are transposed. Pixel 0 is bit 7, pixel 1 is bit 6,..., pixel 7 is bit 0. The diagram below illustrates the byte and bit relationship to the pixels of the pattern.



14.9.14.2 HS_HORIZONTAL 0

Bit 7								0
0,0				7,0				
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

14.9.14.3 HS_VERTICAL 1

Bit 7								0
0,0				7,0				
0	0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0	0

14.9.14.4 HS_FDIAGONAL 2

Bit 7								0
0,0				7,0				
1	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0
0	0	0	0	1	0	0	0	0
0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	1

14.9.14.5 HS_BDIAGONAL 3

Bit 7								0
0,0				7,0				
0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	1	0	0
0	0	0	0	0	1	0	0	0
0	0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0	0
0	0	1	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0



14.9.14.6 HS_CROSS 4

Bit 7 0							
0,0				7,0			
0	0	0	0	1	0	0	0
0	0	0	0	1	0	0	0
0	0	0	0	1	0	0	0
1	1	1	1	1	1	1	1
0	0	0	0	1	0	0	0
0	0	0	0	1	0	0	0
0	0	0	0	1	0	0	0
0	0	0	0	1	0	0	0

14.9.14.7 HS_DIAGCROSS 5

Bit 7 0							
0,0				7,0			
1	0	0	0	0	0	0	1
0	1	0	0	0	0	1	0
0	0	1	0	0	1	0	0
0	0	0	1	1	0	0	0
0	0	0	1	1	0	0	0
0	0	1	0	0	1	0	0
0	1	0	0	0	0	1	0
1	0	0	0	0	0	0	1

14.9.14.8 Screen Door 8

Bit 7 0							
0,0				7,0			
0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0

14.9.14.9 SD Wide 9

Bit 7 0							
0,0				7,0			
1	1	0	0	1	1	0	0
0	0	1	1	0	0	1	1
1	1	0	0	1	1	0	0
0	0	1	1	0	0	1	1
1	1	0	0	1	1	0	0
0	0	1	1	0	0	1	1
1	1	0	0	1	1	0	0
0	0	1	1	0	0	1	1



14.9.14.10 Walking Bit (One) A

				Bit 7			
				0,0			0
					7,0		
1	0	0	0	1	0	0	0
0	1	0	0	0	1	0	0
0	0	1	0	0	0	1	0
0	0	0	1	0	0	0	1
1	0	0	0	1	0	0	0
0	1	0	0	0	1	0	0
0	0	1	0	0	0	1	0
0	0	0	1	0	0	0	1

14.9.14.11 Walking Zero B

				Bit 7			
				0,0			0
					7,0		
0	1	1	1	0	1	1	1
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	0	1	1	1	0
0	1	1	1	0	1	1	1
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	0	1	1	1	0

14.9.15 XY_SRC_COPY_BLT

This BLT instruction performs a color source copy where the only operands involved is a color source and destination of the same bit width.

The source and destination operands may overlap, which means that the X and Y directions can be either forward or backwards. The BLT Engine takes care of all situations. The base addresses plus the X and Y coordinates determine if there is an overlap between the source and destination operands. If the base addresses of the source and destination are the same and the Source X1 is **less than** Destination X1, then the BLT Engine performs the accesses in the X-backwards access pattern. There is no need to look for an actual overlap. If the base addresses are the same and Source Y1 is **less than** Destination Y1, then the scan line accesses start at Destination Y2 with the corresponding source scan line and the strides are subtracted for every scan line access.

The ROP value chosen must involve source and no pattern data in the ROP operation.

DWord	Bit	Description
0 = BR00	31:29	Client: 02h - 2D Processor
	28:22	Instruction Target (Opcode): 53h
	21:20	32 bpp byte mask: (21 =1= write alpha channel; 20=1= write RGB channels)
	19:16	Reserved.



DWord	Bit	Description
	15	Src Tiling Enable: 0 = Tiling Disabled (Linear) 1 = Tiling enabled (Tile-X only)
	14:12	Reserved
	11	Dest Tiling Enable: 0 = Tiling Disabled (Linear) 1 = Tiling enabled (Tile-X only)
	10: 8	Reserved
	7:0	Dword Length: 06h
1 = BR13	31	Reserved.
	30	Clipping Enable: (1 = enabled; 0 = disabled)
	29:26	Reserved.
	25:24	Color Depth: 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color
	23:16	Raster Operation:
	15:00	Destination Pitch in DWords: [15:00] 2's complement For Tiled Dest (bit 11 enabled) this ptich is of 512Byte granularity and can be up to 128Kbytes (or 32KDwords).
2 = BR22	31:16	Destination Y1 Coordinate (Top): (31:16 = 16 bit signed number)
	15:00	Destination X1 Coordinate (Left): (15:00 = 16 bit signed number)
3 = BR23	31:16	Destination Y2 Coordinate (Bottom): (31:16 = 16 bit signed number)
	15:00	Destination X2 Coordinate (Right): (15:00 = 16 bit signed number)
4 = BR09	31:00	Destination Base Address: (base address of the destination surface: X=0, Y=0) When Dest Tiling is enabled (Bit 11 enabled), this address is limited to 4Kbytes.
5 = BR26	31:16	Source Y1 Coordinate (Top): (31:16 = 16 bit signed number)
	15:00	Source X1 Coordinate (Left): (15:00 = 16 bit signed number)
6 = BR11	31:16	Reserved
	15:00	Source Pitch (double word aligned) and in DWords: [15:00] 2's complement. For Tiled Src (bit 15 enabled) this ptich is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
7 = BR12	31:00	Source Base Address: (base address of the source surface: X=0, Y=0) When Src Tiling is enabled (Bit 15 enabled), this address is limited to 4Kbytes.



14.9.16 XY_SRC_COPY_CHROMA_BLT

This BLT instruction performs a color source copy with chroma-keying where the only operands involved is a color source and destination of the same bit width.

The source and destination operands may overlap, which means that the X and Y directions can be either forward or backwards. The BLT Engine takes care of all situations. The base addresses plus the X and Y coordinates determine if there is an overlap between the source and destination operands. If the base addresses of the source and destination are the same and the Source X1 is **less than** Destination X1, then the BLT Engine performs the accesses in the X-backwards access pattern. There is no need to look for an actual overlap. If the base addresses are the same and Source Y1 is **less than** Destination Y1, then the scan line accesses start at Destination Y2 with the corresponding source scan line and the strides are subtracted for every scan line access.

The ROP value chosen must involve source and no pattern data in the ROP operation.

DWord	Bit	Description
0 = BR00	31:29	Client: 02h - 2D Processor
	28:22	Instruction Target (Opcode): 73h
	21:20	32 bpp byte mask: (21 =1= write alpha channel; 20=1= write RGB channels)
	19:17	Transparency Range Mode: (chroma-key)
	16	Reserved
	15	Src Tiling Enable: 0 = Tiling Disabled (Linear) 1 = Tiling enabled (Tile-X only)
	14:12	Reserved
	11	Dest Tiling Enable: 0 = Tiling Disabled (Linear) 1 = Tiling enabled (Tile-X only)
	10: 08	Reserved
	07:00	Dword Length: 08h
1 = BR13	31	Reserved.
	30	Clipping Enable: (1 = enabled; 0 = disabled)
	29:26	Reserved.
	25:24	Color Depth: 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color
	23:16	Raster Operation:



DWord	Bit	Description
	15:00	Destination Pitch in DWords: [15:00] 2's complement For Tiled Dest (bit 11 enabled) this pitch is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
2 = BR22	31:16	Destination Y1 Coordinate (Top): (31:16 = 16 bit signed number)
	15:00	Destination X1 Coordinate (Left): (15:00 = 16 bit signed number)
3 = BR23	31:16	Destination Y2 Coordinate (Bottom): (31:16 = 16 bit signed number)
	15:00	Destination X2 Coordinate (Right): (15:00 = 16 bit signed number)
4 = BR09	31:00	Destination Base Address: (base address of the destination surface: X=0, Y=0) When Dest Tiling is enabled (Bit 11 enabled), this address is limited to 4Kbytes.
5 = BR26	31:16	Source Y1 Coordinate (Top): (31:16 = 16 bit signed number)
	15:00	Source X1 Coordinate (Left): (15:00 = 16 bit signed number)
6 = BR11	31:16	Reserved.
	15:00	Source Pitch (double word aligned) and in DWords: [15:00] 2's complement. For Tiled Src (bit 15 enabled) this pitch is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
7 = BR12	31:00	Source Base Address: (base address of the source surface: X=0, Y=0) When Src Tiling is enabled (Bit 15 enabled), this address is limited to 4Kbytes.
8 = BR18	31:00	Transparency Color Low: (Chroma-key Low = Pixel Greater or Equal)
9 = BR19	31:00	Transparency Color High: (Chroma-key High = Pixel Less or Equal)

14.9.17 XY_MONO_SRC_COPY_BLT

This BLT instruction performs a monochrome source copy where the only operands involved is a monochrome source and destination. The source and destination operands cannot overlap therefore the X and Y directions are always forward.

All non-text monochrome sources are word aligned. At the end of a scan line of monochrome source, all bits until the next word boundary are ignored. The monochrome source data bit position field [2:0] indicates the bit position within the first byte of the scan line that should be used as the first source pixel which corresponds to the destination X1 coordinate.

The monochrome source transparency mode indicates whether to use the source background color or de-assert the write enables when the bit in the source is 0. When the source bit is 1, then the source foreground color is used in the ROP operation. The ROP value chosen must involve source and no pattern data in the ROP operation. Negative Stride (= Pitch) is NOT ALLOWED.



DWord	Bit	Description
0 = BR00	31:29	Client: 02h - 2D Processor
	28:22	Instruction Target (Opcode): 54h
	21:20	32 bpp byte mask: (21 =1= write alpha channel; 20=1= write RGB channels)
	19:17	Monochrome source data bit position of the first pixel within a byte per scan line.
	16:12	Reserved.
	11	Tiling Enable: 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10: 08	Reserved
	07:00	Doubleword Length: 06h
1 = BR13	31	Reserved.
	30	Clipping Enable: (1 = enabled; 0 = disabled)
	29	Mono Source Transparency Mode: (1 = transparency enabled; 0 = use background)
	28:26	Reserved.
	25:24	Color Depth: 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color
	23:16	Raster Operation:
	15:00	Destination Pitch in DWords: [15:00] 2's complement For Tiled surfaces (bit_11 enabled) this pitch is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
2 = BR22	31:16	Destination Y1 Coordinate (Top): (31:16 = 16 bit signed number)
	15:00	Destination X1 Coordinate (Left): (15:00 = 16 bit signed number)
3 = BR23	31:16	Destination Y2 Coordinate (Bottom): (31:16 = 16 bit signed number)
	15:00	Destination X2 Coordinate (Right): (15:00 = 16 bit signed number)
4 = BR09	31:00	Destination Base Address: (base address of the destination surface: X=0, Y=0) When Tiling is enabled (Bit_11 enabled), this address is limited to 4Kbytes.
5 = BR12	31:00	Source Address: (address corresponding to DST X1,Y1) (Note no NPO2 change here)
6 = BR18	31:00	Source Background Color: 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]
7 = BR19	31:00	Source Foreground Color: 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]



14.9.18 XY_MONO_SRC_COPY_IMMEDIATE_BLT

This instruction allows the Driver to send monochrome data through the instruction stream, eliminating the read latency of the source during command execution.

The IMMEDIATE_BLT data MUST transfer an even number of doublewords and the exact number of quadwords.

All non-text monochrome sources are word aligned. At the end of a scan line of monochrome source, all bits until the next word boundary are ignored. The Monochrome source data bit position field [2:0] indicates the bit position within the first byte of the scan line that should be used as the first source pixel which corresponds to the destination X1 coordinate.

The monochrome source transparency mode indicates whether to use the source background color or de-assert the write enables when the bit in the source is 0. When the source bit is 1, then the source foreground color is used in the ROP operation. The ROP value chosen must involve source and no pattern data in the ROP operation.

The monochrome source data supplied corresponds to the Destination X1 and Y1 coordinates.

Negative Stride (= Pitch) is NOT ALLOWED.

DWord	Bit	Description
0 = BR00	31:29	Client: 02h - 2D Processor
	28:22	Instruction Target (Opcode): 71h
	21:20	32 bpp byte mask: (21 =1= write alpha channel; 20=1= write RGB channels)
	19:17	Monochrome source data bit position of the first pixel within a byte per scan line.
	16:12	Reserved.
	11	Tiling Enable: 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10: 08	Reserved
	07:00	Dword Length: 05+ DWL = (Number of Immediate double words)h
1 = BR13	31	Reserved.
	30	Clipping Enable: (1 = enabled; 0 = disabled)
	29	Mono Source Transparency Mode: (1 = transparency enabled; 0 = use background)
	28:26	Reserved.
	25:24	Color Depth: 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color
	23:16	Raster Operation:



DWord	Bit	Description
	15:00	Destination Pitch in DWords: [15:00] 2's complement For Tiled surfaces (bit_11 enabled) this ptich is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
2 = BR22	31:16	Destination Y1 Coordinate (Top): (31:16 = 16 bit signed number)
	15:00	Destination X1 Coordinate (Left): (15:00 = 16 bit signed number)
3 = BR23	31:16	Destination Y2 Coordinate (Bottom): (31:16 = 16 bit signed number)
	15:00	Destination X2 Coordinate (Right): (15:00 = 16 bit signed number)
4 = BR09	31:00	Destination Base Address: (base address of the destination surface: X=0, Y=0) When Tiling is enabled (Bit_11 enabled), this address is limited to 4Kbytes.
5 = BR18	31:00	Source Background Color: 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]
6 = BR19	31:00	Source Foreground Color: 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]
7	31:00	Immediate Data DW 0:
8	31:00	Immediate Data DW 1:
9 thru DWL+4	S	Immediate Data DWs 2 through DWORD_LENGTH (DWL):



14.9.19 XY_FULL_BLT

The full BLT is the most comprehensive BLT instruction. It provides the ability to specify all 3 operands: destination, source, and pattern. The source and pattern operands are the same bit width as the destination operand.

The source and destination operands may overlap, which means that the X and Y directions can be either forward or backwards. The BLT Engine takes care of all situations. The base addresses plus the X and Y coordinates determine if there is an overlap between the source and destination operands. If the base addresses of the source and destination are the same and the Source X1 is **less than** Destination X1, then the BLT Engine performs the accesses in the X-backwards access pattern. There is no need to look for an actual overlap. If the base addresses are the same and Source Y1 is **less than** Destination Y1, then the scan line accesses start at Destination Y2 with the corresponding source scan line and the strides are subtracted for every scan line access.

All scan lines and pixels that fall within the ClipRect Y and X coordinates are written. Only pixels within the ClipRectX coordinates and the Destination X coordinates are written using the raster operation.

The Pattern Seeds correspond to Destination X = 0 (horizontal) and Y = 0 (vertical). The alignment is relative to the destination coordinates. The pixel of the pattern used / scan line is the (destination X coordinate + horizontal seed) modulo 8. The scan line of the pattern used is the (destination Y coordinate + vertical seed) modulo 8.

DWord	Bit	Description
0 = BR00	31:29	Client: 02h - 2D Processor
	28:22	Instruction Target (Opcode): 55h
	21:20	32 bpp byte mask: (21 =1= write alpha channel; 20=1= write RGB channels)
	19:16	Reserved.
	15	Src Tiling Enable: 0 = Tiling Disabled (Linear) 1 = Tiling enabled (Tile-X only)
	14:12	Pattern Horizontal Seed (pixel of the scan line to start on corresponding to DST X=0)
	11	Dest Tiling Enable: 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10:08	Pattern Vertical Seed: (starting scan line of the 8x8 pattern corresponding to DST Y=0)
	07:00	Doubleword Length: 07h
1 = BR13	31	Reserved.
	30	Clipping Enable: (1 = enabled; 0 = disabled)
	29:26	Reserved.



DWord	Bit	Description
	25:24	Color Depth: 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color
	23:16	Raster Operation:
	15:00	Destination Pitch in DWords: [15:00] 2's complement For Tiled Dest (bit 11 enabled) this pitch is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
2 = BR22	31:16	Destination Y1 Coordinate (Top): (31:16 = 16 bit signed number)
	15:00	Destination X1 Coordinate (Left): (15:00 = 16 bit signed number)
3 = BR23	31:16	Destination Y2 Coordinate (Bottom): (31:16 = 16 bit signed number)
	15:00	Destination X2 Coordinate (Right): (15:00 = 16 bit signed number)
4 = BR09	31:00	Destination Base Address: (base address of the destination surface: X=0, Y=0) When Dest Tiling is enabled (Bit 11 enabled), this address is limited to 4Kbytes.
5 = BR11	31:16	Reserved.
	15:00	Source Pitch (double word aligned and signed) and in DWords: [15:00] 2's complement. For Tiled Src (bit 15 enabled) this pitch is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
6 = BR26	31:16	Source Y1 Coordinate (Top): (31:16 = 16 bit signed number)
	15:00	Source X1 Coordinate (Left): (15:00 = 16 bit signed number)
7 = BR12	31:00	Source Base Address: (base address of the source surface: X=0, Y=0) When Src Tiling is enabled (Bit 15 enabled), this address is limited to 4Kbytes.
8 = BR15	31:00	Pattern Base Address: (28:06 are implemented) (Note no NPO2 change here). The pattern data must be located in linear memory.



14.9.20 XY_FULL_IMMEDIATE_PATTERN_BLT

The full BLT is the most comprehensive BLT instruction. It provides the ability to specify all 3 operands: destination, source, and pattern. The source and immediate pattern operands are the same bit width as the destination operand. The immediate data sizes are 64 bytes (16 DWs), 128 bytes (32 DWs), or 256 (64 DWs) for 8, 16, and 32 bpp color patterns.

The source and destination operands may overlap, which means that the X and Y directions can be either forward or backwards. The BLT Engine takes care of all situations. The base addresses plus the X and Y coordinates determine if there is an overlap between the source and destination operands. If the base addresses of the source and destination are the same and the Source X1 is **less than** Destination X1, then the BLT Engine performs the accesses in the X-backwards access pattern. There is no need to look for an actual overlap. If the base addresses are the same and Source Y1 is **less than** Destination Y1, then the scan line accesses start at Destination Y2 with the corresponding source scan line and the strides are subtracted for every scan line access.

All scan lines and pixels that fall within the ClipRect Y and X coordinates are written. Only pixels within the ClipRectX coordinates and the Destination X coordinates are written using the raster operation.

The Pattern Seeds correspond to Destination X = 0 (horizontal) and Y = 0 (vertical). The alignment is relative to the destination coordinates. The pixel of the pattern used / scan line is the (destination X coordinate + horizontal seed) modulo 8. The scan line of the pattern used is the (destination Y coordinate + vertical seed) modulo 8.

DWord	Bit	Description
0 = BR00	31:29	Client: 02h - 2D Processor
	28:22	Instruction Target (Opcode): 74h
	21:20	32 bpp byte mask: (21 = 1= write alpha channel; 20=1= write RGB channels)
	19:16	Reserved.
	15	Src Tiling Enable: 0 = Tiling Disabled (Linear) 1 = Tiling enabled (Tile-X only)
	14:12	Pattern Horizontal Seed: (pixel of the scan line to start on corresponding to DST X=0)
	11	Dest Tiling Enable: 0 = Tiling Disabled (Linear) 1 = Tiling enabled (Tile-X only)
	10:8	Pattern Vertical Seed: (starting scan line of the 8x8 pattern corresponding to DST Y=0)
	7:0	Doubleword Length: 06+ DWL = (Number of Immediate double words)h
1 = BR13	31	Reserved.
	30	Clipping Enable: (1 = enabled; 0 = disabled)
	29:26	Reserved.



DWord	Bit	Description
	25:24	Color Depth: 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color
	23:16	Raster Operation:
	15:00	Destination Pitch in DWords: [15:00] 2's complement For Tiled Dest (bit 11 enabled) this pitch is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
2 = BR22	31:16	Destination Y1 Coordinate (Top): (31:16 = 16 bit signed number)
	15:00	Destination X1 Coordinate (Left): (15:00 = 16 bit signed number)
3 = BR23	31:16	Destination Y2 Coordinate (Bottom): (31:16 = 16 bit signed number)
	15:00	Destination X2 Coordinate (Right): (15:00 = 16 bit signed number)
4 = BR09	31:00	Destination Base Address: (base address of the destination surface: X=0, Y=0) When Dest Tiling is enabled (Bit 11 enabled), this address is limited to 4Kbytes.
5 = BR11	31:16	Reserved.
	15:00	Source Pitch (double word aligned and signed) and in DWords: [15:00] 2's complement. For Tiled Src (bit 15 enabled) this pitch is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
6 = BR26	31:16	Source Y1 Coordinate (Top): (31:16 = 16 bit signed number)
	15:00	Source X1 Coordinate (Left): (15:00 = 16 bit signed number)
7 = BR12	31:00	Source Base Address: (base address of the source surface: X=0, Y=0) When Src Tiling is enabled (Bit 15 enabled), this address is limited to 4Kbytes.
8	31:00	Immediate Data DW 0:
9	31:00	Immediate Data DW 1:
A thru DWL+4	S	Immediate Data DWs 2 through DWORD_LENGTH (DWL):



14.9.21 XY_FULL_MONO_SRC_BLT

The full BLT is the most comprehensive BLT instruction. It provides the ability to specify all 3 operands: destination, source, and pattern. The source operand is monochrome and the pattern operand is the same bit width as the destination.

The monochrome source transparency mode indicates whether to use the source background color or de-assert the write enables when the bit in the source is 0. When the source bit is 1, then the source foreground color is used in the ROP operation.

All non-text and non-immediate monochrome sources are word aligned. At the end of a scan line the monochrome source, the remaining bits until the next word boundary are ignored. The Monochrome source data bit position field [2:0] indicates which bit position within the first byte should be used as the first source pixel which corresponds to the Destination X1 coordinate.

All scan lines and pixels that fall within the ClipRect Y and X coordinates are written. Only pixels within the ClipRectX coordinates and the Destination X coordinates are written using the raster operation.

The Pattern Seeds correspond to Destination X = 0 (horizontal) and Y = 0 (vertical). The alignment is relative to the destination coordinates. The pixel of the pattern used / scan line is the (destination X coordinate + horizontal seed) modulo 8. The scan line of the pattern used is the (destination Y coordinate + vertical seed) modulo 8.

Negative Stride (= Pitch) is NOT ALLOWED

DWord	Bit	Description
0 = BR00	31:29	Client: 02h - 2D Processor
	28:22	Instruction Target (Opcode): 56h
	21:20	32 bpp byte mask: (21 =1= write alpha channel; 20=1= write RGB channels)
	19:17	Monochrome source data bit position of the first pixel within a byte per scan line.
	16:15	Reserved.
	14:12	Pattern Horizontal Seed: (pixel of the scan line to start on corresponding to DST X=0)
	11	Tiling Enable: 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10:08	Pattern Vertical Seed: (starting address of the 8x8 pattern corresponding to DST Y=0)
	07:00	Doubleword Length : 07h
1 = BR13	31	Reserved.
	30	Clipping Enable: (1 = enabled; 0 = disabled)
	29	Mono Source Transparency Mode: (1 = transparency enabled; 0 = use background)
	28:27	Reserved.
	26	Reserved.



DWord	Bit	Description
	25:24	Color Depth: 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color
	23:16	Raster operation:
	15:00	Destination Pitch in DWords: [15:00] 2's complement For Tiled surfaces (bit_11 enabled) this pitch is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
2 = BR22	31:16	Destination Y1 Coordinate (Top): (31:16 = 16 bit signed number)
	15:00	Destination X1 Coordinate (Left): (15:00 = 16 bit signed number)
3 = BR23	31:16	Destination Y2 Coordinate (Bottom): (31:16 = 16 bit signed number)
	15:00	Destination X2 Coordinate (Right): (15:00 = 16 bit signed number)
4 = BR09	31:00	Destination Base Address: (base address of the destination surface: X=0, Y=0) When Tiling is enabled (Bit_11 enabled), this address is limited to 4Kbytes.
5 = BR12	31:00	Mono Source Address: (address corresponds to DST X1, Y1) (Note no NPO2 change here)
6 = BR18	31:00	Source Background Color: 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]
7 = BR19	31:00	Source Foreground Color: 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]
8 = BR15	31:00	Pattern Base Address: (28:06 are implemented) (Note no NPO2 change here). The pattern data must be located in linear memory.



14.9.22 XY_FULL_MONO_SRC_IMMEDIATE_PATTERN_BLT

The full BLT is the most comprehensive BLT instruction. It provides the ability to specify all 3 operands: destination, source, and pattern. The source operand is a monochrome and the immediate pattern operand is the same bit width as the destination. The immediate data sizes are 64 bytes (16 DWs), 128 bytes (32 DWs), or 256 (64DWs) for 8, 16, and 32 bpp color patterns.

The monochrome source transparency mode indicates whether to use the source background color or de-assert the write enables when the bit in the source is 0. When the source bit is 1, then the source foreground color is used in the ROP operation.

All non-text monochrome sources are word aligned. At the end of a scan line the monochrome source, the remaining bits until the next word boundary are ignored. The Monochrome source data bit position field [2:0] indicates which bit position within the first byte should be used as the first source pixel which corresponds to the destination X1 coordinate.

All scan lines and pixels that fall within the ClipRect Y and X coordinates are written. Only pixels within the ClipRectX coordinates and the Destination X coordinates are written using the raster operation.

The Pattern Seeds correspond to Destination X = 0 (horizontal) and Y = 0 (vertical). The alignment is relative to the destination coordinates. The pixel of the pattern used / scan line is the (destination X coordinate + horizontal seed) modulo 8. The scan line of the pattern used is the (destination Y coordinate + vertical seed) modulo 8.

Negative Stride (= Pitch) is NOT ALLOWED.

DWord	Bit	Description
0 = BR00	31:29	Client: 02h - 2D Processor
	28:22	Instruction Target (Opcode): 75h
	21:20	32 bpp byte mask: (21 =1= write alpha channel; 20=1= write RGB channels)
	19:17	Monochrome source data bit position of the first pixel within a byte per scan line.
	16:15	Reserved.
	14:12	Pattern Horizontal Seed: (pixel of the scan line to start on corresponding to DST X=0)
	11	Tiling Enable: 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10:08	Pattern Vertical Seed: (starting address of the 8x8 pattern corresponding to DST Y=0)
	07:00	Doubleword Length : 06+ DWL = (Number of Immediate double words)h
1 = BR13	31	Reserved.
	30	Clipping Enable: (1 = enabled; 0 = disabled)
	29	Mono Source Transparency Mode: (1 = transparency enabled; 0 = use background)



DWord	Bit	Description
	28:26	Reserved.
	25:24	Color Depth: 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color
	23:16	Raster operation:
	15:00	Destination Pitch in DWords: [15:00] 2's complement For Tiled surfaces (bit_11 enabled) this pitch is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
2 = BR22	31:16	Destination Y1 Coordinate (Top): (31:16 = 16 bit signed number)
	15:00	Destination X1 Coordinate (Left): (15:00 = 16 bit signed number)
3 = BR23	31:16	Destination Y2 Coordinate (Bottom): (31:16 = 16 bit signed number)
	15:00	Destination X2 Coordinate (Right): (15:00 = 16 bit signed number)
4 = BR09	31:00	Destination Base Address: (base address of the destination surface: X=0, Y=0) When Tiling is enabled (Bit_11 enabled), this address is limited to 4Kbytes.
5 = BR12	31:00	Mono Source Address: (address corresponds to DST X1, Y1) (Note no NPO2 change here)
6 = BR18	31:00	Source Background Color: 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]
7 = BR19	31:00	Source Foreground Color: 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]
8	31:00	Immediate Data DW 0:
9	31:00	Immediate Data DW 1:
A thru DWL+4	S	Immediate Data DWs 2 through DWORD_LENGTH (DWL):



14.9.23 XY_FULL_MONO_PATTERN_BLT

The full BLT is the most comprehensive BLT instruction. It provides the ability to specify all 3 operands: destination, source, and pattern. The pattern operand is monochrome and the source operand is the same bit width as the destination operand.

The source and destination operands may overlap, which means that the X and Y directions can be either forward or backwards. The BLT Engine takes care of all situations. The base addresses plus the X and Y coordinates determine if there is an overlap between the source and destination operands. If the base addresses of the source and destination are the same and the Source X1 is **less than** Destination X1, then the BLT Engine performs the accesses in the X-backwards access pattern. There is no need to look for an actual overlap. If the base addresses are the same and Source Y1 is **less than** Destination Y1, then the scan line accesses start at Destination Y2 with the corresponding source scan line and the strides are subtracted for every scan line access.

The monochrome pattern transparency mode indicates whether to use the pattern background color or de-assert the write enables when the bit in the source is 0. When the source bit is 1, then the pattern foreground color is used in the ROP operation.

All scan lines and pixels that fall within the ClipRect Y and X coordinates are written. Only pixels within the ClipRectX coordinates and the Destination X coordinates are written using the raster operation.

The Pattern Seeds correspond to Destination X = 0 (horizontal) and Y = 0 (vertical). The alignment is relative to the destination coordinates. The pixel of the pattern used / scan line is the (destination X coordinate + horizontal seed) modulo 8. The scan line of the pattern used is the (destination Y coordinate + vertical seed) modulo 8.

Setting both Solid Pattern Select =1 & Mono Pattern Transparency = 1 is mutually exclusive. The device implementation results in NO PIXELs DRAWN.

DWord	Bit	Description
0 = BR00	31:29	Client: 02h - 2D Processor
	28:22	Instruction Target (Opcode): 57h
	21:20	32 bpp byte mask: (21 =1= write alpha channel; 20=1= write RGB channels)
	19:16	Reserved.
	15	Src Tiling Enable: 0 = Tiling Disabled (Linear) 1 = Tiling enabled (Tile-X only)
	14:12	Pattern Horizontal Seed: (pixel of the scan line to start on corresponding to DST X=0)
	11	Dest Tiling Enable: 0 = Tiling Disabled (Linear) 1 = Tiling enabled (Tile-X only)
	10:08	Pattern Vertical Seed: (starting scan line of the 8x8 pattern corresponding to DST Y=0)
	07:00	Dword Length : 0Ah



DWord	Bit	Description
1 = BR13	31	Solid Pattern Select: (1 = solid pattern; 0 = no solid pattern)
	30	Clipping Enable: (1 = enabled; 0 = disabled)
	29	Reserved.
	28:27	Mono Pattern Transparency Mode: (1 = transparency enabled; 0 = use background)
	26	Reserved.
	25:24	Color Depth: 00 = 8 bit color 01 = 16 bit color 10 = 16 bit color (1555) 11 = 32 bit color (565)
	23:16	Raster Operation:
	15:00	Destination Pitch in DWords: [15:00] 2's complement For Tiled Dest (bit 11 enabled) this pitch is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
2 = BR22	31:16	Destination Y1 Coordinate (Top): (31:16 = 16 bit signed number)
	15:00	Destination X1 Coordinate (Left): (15:00 = 16 bit signed number)
3 = BR23	31:16	Destination Y2 Coordinate (Bottom): (31:16 = 16 bit signed number)
	15:00	Destination X2 Coordinate (Right): (15:00 = 16 bit signed number)
4 = BR09	31:00	Destination Base Address: (base address of the destination surface: X=0, Y=0) When Dest Tiling is enabled (Bit 11 enabled), this address is limited to 4Kbytes.
5 = BR11	31:16	Reserved.
	15:00	Source Pitch (double word aligned and signed) and in DWords: [15:00] 2's complement. For Tiled Src (bit 15 enabled) this pitch is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
6 = BR26	31:16	Source Y1 Coordinate (Top): (31:16 = 16 bit signed number)
	15:00	Source X1 Coordinate (Left): (15:00 = 16 bit signed number)
7 = BR12	31:00	Source Base Address: (base address of the source surface: X=0, Y=0) When Src Tiling is enabled (Bit 15 enabled), this address is limited to 4Kbytes.
8 = BR16	31:00	Pattern Background Color: 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]
9 = BR17	31:00	Pattern Foreground Color: 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]
A = BR20	31:00	Pattern Data 0: (least significant DW)
B = BR21	31:00	Pattern Data 1: (most significant DW)



14.9.24 XY_FULL_MONO_PATTERN_MONO_SRC_BLT

The full BLT provides the ability to specify all 3 operands: destination, source, and pattern. The pattern and source operands are monochrome.

The monochrome source transparency mode indicates whether to use the source background color or de-assert the write enables when the bit in the source is 0. When the source bit is 1, then the source foreground color is used in the ROP operation.

All non-text monochrome sources are word aligned. At the end of a scan line the monochrome source, the remaining bits until the next word boundary are ignored. The Monochrome source data bit position field [2:0] indicates which bit position within the first byte should be used as the first source pixel which corresponds to the destination X1 coordinate.

The monochrome pattern transparency mode indicates whether to use the pattern background color or de-assert the write enables when the bit in the pattern is 0. When the source bit is 1, then the pattern foreground color is used in the ROP operation. The monochrome source transparency mode works identical to the pattern transparency mode.

All scan lines and pixels that fall within the ClipRect Y and X coordinates are written. Only pixels within the ClipRectX coordinates and the Destination X coordinates are written using the raster operation.

The Pattern Seeds correspond to Destination X = 0 (horizontal) and Y = 0 (vertical). The alignment is relative to the destination coordinates. The pixel of the pattern used / scan line is the (destination X coordinate + horizontal seed) modulo 8. The scan line of the pattern used is the (destination Y coordinate + vertical seed) modulo 8.

Setting both Solid Pattern Select = 1 & Mono Pattern Transparency = 1 is mutually exclusive. The device implementation results in NO PIXELS DRAWN.

Negative Stride (= Pitch) is NOT ALLOWED.

DWord	Bit	Description
0 = BR00	31:29	Client: 02h - 2D Processor
	28:22	Instruction Target (Opcode): 58h
	21:20	32 bpp byte mask: (21 = 1 = write alpha channel; 20 = 1 = write RGB channels)
	19:17	Monochrome source data bit position of the first pixel within a byte per scan line.
	16:15	Reserved.
	14:12	Pattern Horizontal Seed: (pixel of the scan line to start on corresponding to DST X = 0)
	11	Tiling Enable: 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only)
	10:08	Pattern Vertical Seed: (starting scan line of the 8x8 pattern corresponding to DST Y = 0)
	07:00	Doubleword Length : 0Ah



DWord	Bit	Description
1 = BR13	31	Solid Pattern Select: (1 = solid pattern; 0 = no solid pattern)
	30	Clipping Enable (1 = enabled; 0 = disabled)
	29	Mono Source Transparency Mode: (1 = transparency enabled; 0 = use background)
	28	Mono Pattern Transparency Mode: (1 = transparency enabled; 0 = use background)
	27:26	Reserved.
	25:24	Color Depth: 00 = 8 bit color 01 = 16 bit color (565) 10 = 16 bit color (1555) 11 = 32 bit color
	23:16	Raster Operation:
	15:00	Destination Pitch in DWords: [15:00] 2's complement For Tiled surfaces (bit_11 enabled) this ptich is of 512Byte granularity and can be upto 128Kbytes (or 32KDwords).
2 = BR22	31:16	Destination Y1 Coordinate (Top): (31:16 = 16 bit signed number)
	15:00	Destination X1 Coordinate (Left): (15:00 = 16 bit signed number)
3 = BR23	31:16	Destination Y2 Coordinate (Bottom): (31:16 = 16 bit signed number)
	15:00	Destination X2 Coordinate (Right): (15:00 = 16 bit signed number)
4 = BR09	31:00	Destination Base Address: (base address of the destination surface: X=0, Y=0) When Tiling is enabled (Bit_11 enabled), this address is limited to 4Kbytes.
5 = BR12	31:00	Source Address: (address corresponding to Dst X1,Y1) (Note no NPO2 change here)
6 = BR18	31:00	Source Background Color: 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]
7 = BR19	31:00	Source Foreground Color: 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]
8 = BR16	31:00	Pattern Background Color: 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]
9 = BR17	31:00	Pattern Foreground Color: 8 bit = [7:0], 16 bit = [15:0], 32 bit = [31:0]
A = BR20	31:00	Pattern Data 0: (least significant DW)
B = BR21	31:00	Pattern Data 1: (most significant DW)



14.10 BLT Engine Instruction Field Definitions

This section describes the BLT Engine instruction fields. These descriptions are in the format of register descriptions. These registers are internal and are not readable. Some of these registers are state that is saved and restored for supporting separate software threads.

14.10.1 BR00—BLT Opcode & Control

Memory Offset Address: none
 Default: 0000 0000
 Attributes: not accessible

BR00 is the last executed instruction DWord 0. Bits [22:5] are written by every DWord of every instruction. Bits [31:30] and [4:0] are status bits. Bits [28:27] are written from the DWord [15:14] of a Setup instruction and Bit 29 is written with a 1 when ever a Setup instruction is written. Bit 29 is a decode of the Setup instruction Opcode.

31	30	29	28	24
Rsvd	Clip Inst	Setup Mono Pattern	Instruction Target (Opcode)	

23	22	21	20	19	17	16
Instruction Target (Opcode)		32 bpp byte mask		Monochrome Source Start		Bit (0) / Byte (1) Packed

15	14	12	11	10	8
Rsvd	Pattern Horizontal Seed		Tiling Enable	Transparency Range Mode	

7	5	4	3	2	1	0
PatternVertical Seed		DST RMW	Color Source	Mono Source	Color Pattern	Mono Pattern



Bit	Descriptions
31	BLT Engine Busy. This bit indicates whether the BLT Engine is busy (1) or idle (0). This bit is replicated in the SETUP BLT Opcode & Control register. 1 = Busy 0 = Idle
30	Setup Instruction Instruction. The current instruction performs clipping (1).
29	Setup Monochrome Pattern. This bit is decoded from the Setup instruction opcode to identify whether a color (0) or monochrome (1) pattern is used with the SCANLINE_BLT instruction. 1 = Monochrome 0 = Color
28:22	Instruction Target (Opcode). This is the contents of the Instruction Target field from the last BLT instruction. This field is used by the BLT Engine state machine to identify the BLT instruction it is to perform. The opcode specifies whether the source and pattern operands are color or monochrome.
21:20	32 bpp byte mask: 21 = 1 = write alpha channel [31:24]; 20 = 1 = write RGB channels [23:00]. This field is only used for 32bpp.
19:17	Monochrome Source Start. This field indicates the starting monochrome pixel bit position within a byte per scan line of the source operand. The monochrome source is word aligned which means that at the end of the scan line all bits should be discarded until the next word boundary.
16	Bit/Byte Packed. Byte packed is for the NT driver 0 = Bit 1 = Byte
15	Src Tiling Enable: 0 = Tiling Disabled (Linear) 1 = Tiling enabled (Tile-X only)
14:12	Horizontal Pattern Seed. This field indicates the pattern pixel position which corresponds to X = 0.
11	Dest Tiling Enable: 0 = Tiling Disabled (Linear blit) 1 = Tiling enabled (Tile-X only) When set to '1', this means that Blitter is executing in Tiled-X mode. If '0' it means that Blitter is in Linear mode. Blitter never executes in Tiled-Y mode. On reset, this bit will be '0'. This definition applies to only X,Y Blits. Non-XY blits (COLOR_BLT, SRC_COPY_BLT), will support only linear mode and will not support tiling and for them this bit will remain reserved.



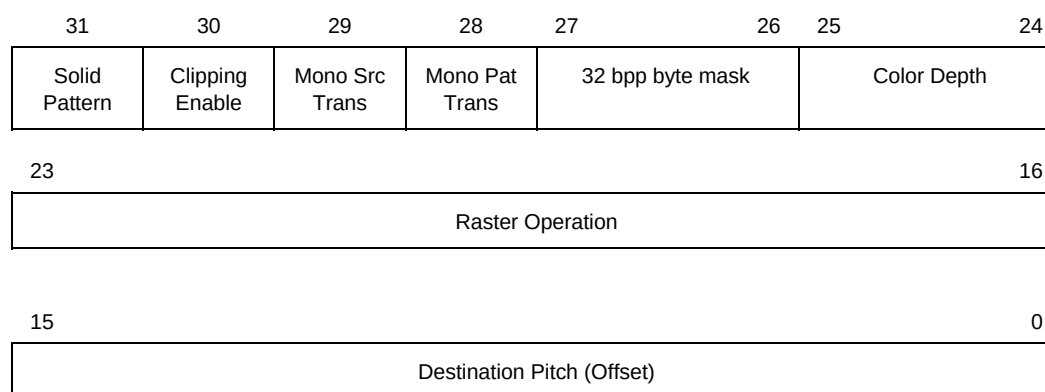
Bit	Descriptions
10:8	Transparency Range Mode. These bits control whether or not the byte(s) at the destination corresponding to a given pixel will be conditionally written, and what those conditions are. This feature can make it possible to perform various masking functions in order to selectively write or preserve graphics data already at the destination. XX0 = No color transparency mode enabled. This causes normal operation with regard to writing data to the destination. 001 = [Source color transparency] The Transparency Color Low: (Pixel Greater or Equal) (source background register) and the Transparency Color High: (Pixel Less or Equal) (source foreground register) are compared to the source pixels. The range comparisons are done on each component (R,G,B) and then logically ANDed. If the source pixel components are not within the range defined by the Transparency Color registers, then the byte(s) at the destination corresponding to the current pixel are written with the result of the bit-wise operation. 011 = [Source and Alpha color transparency] The Transparency Color Low: (Pixel Greater or Equal) (source background register) and the Transparency Color High: (Pixel Less or Equal) (source foreground register) are compared to the source pixels. The range comparisons are done on each component (A,R,G,B) and then logically ANDed. If the source pixel components are not within the range defined by the Transparency Color registers, then the byte(s) at the destination corresponding to the current pixel are written with the result of the bit-wise operation. 101 = [Destination and Alpha color transparency] The Transparency Color Low: (Pixel Greater or Equal) (source background register) and the Transparency Color High: (Pixel Less or Equal) (source foreground register) are compared to the destination pixels. The range comparisons are done on each component (A,R,G,B) and then logically ANDed. If the destination pixels are within the range, then the byte(s) at the destination corresponding to the current pixel are written with the result of the bit-wise operation. 111 = [Destination color transparency] The Transparency Color Low: (Pixel Greater or Equal) (source background register) and the Transparency Color High: (Pixel Less or Equal) (source foreground register) are compared to the destination pixels. The range comparisons are done on each component (R,G,B) and then logically ANDed. If the destination pixels are within the range, then the byte(s) at the destination corresponding to the current pixel are written with the result of the bit-wise operation.
7:5	Pattern Vertical Seed. This field specifies the pattern scan line which corresponds to Y=0.
4	Destination Read Modify Write. This bit is decoded from the last instruction's opcode field and Destination Transparency Mode to identify whether a Destination read is needed.
3	Color Source. This bit is decoded from the last instructions opcode field to identify whether a color (1) source is used.
2	Monochrome Source. This bit is decoded from the last instructions opcode field to identify whether a monochrome (1) source is used.
1	Color Pattern. This bit is decoded from the last instructions opcode field to identify whether a color (1) pattern is used.
0	Monochrome Pattern. This bit is decoded from the last instructions opcode field to identify whether a monochrome (1) pattern is used.



14.10.2 BR01—Setup BLT Raster OP, Control, and Destination Offset

Memory Offset Address: none
 Default: 0000 xxxx
 Attributes: State accessible

BR01 contains the contents of the last Setup instruction DWord 1. It is identical to the BLT Raster OP, Control, and Destination Offset definition, but it is used with the following instructions: PIXEL_BLT, SCANLINE_BLT, and TEXT_BLT.



Bit	Descriptions
31	Solid Pattern Select. This bit applies only when the pattern data is monochrome. This bit determines whether or not the BLT Engine actually performs read operations from the frame buffer in order to load the pattern data. Use of this feature to prevent these read operations can increase BLT Engine performance, if use of the pattern data is indeed not necessary. The BLT Engine is configured to accept either monochrome or color pattern data via the opcode field. 0 = This causes normal operation with regard to the use of the pattern data. The BLT Engine proceeds with the process of reading the pattern data, and the pattern data is used as the pattern operand for all bit-wise operations. 1 = The BLT Engine forgoes the process of reading the pattern data, the presumption is made that all of the bits of the pattern data are set to 0, and the pattern operand for all bit-wise operations is forced to the background color specified in the Color Expansion Background Color Register.
30	Clipping Enabled: 1 = Enabled; 0 = Disabled



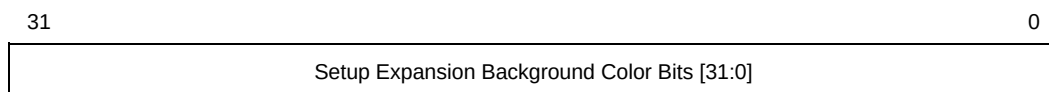
Bit	Descriptions
29	Monochrome Source Transparency Mode. This bit applies only when the source data is in monochrome. This bit determines whether or not the byte(s) at the destination corresponding to the pixel to which a given bit of the source data also corresponds will actually be written if that source data bit has the value of 0. This feature can make it possible to use the source as a transparency mask. The BLT Engine is configured to accepted either monochrome or color source data via the opcode field. 0 = This causes normal operation with regard to the use of the source data. Wherever a bit in the source data has the value of 0, the color specified in the background color register is used as the source operand in the bit-wise operation for the pixel corresponding to the source data bit, and the bytes at the destination corresponding to that pixel are written with the result. 1 = Wherever a bit in the source data has the value of 0, the byte(s) at the destination corresponding to the pixel to which the source data bit also corresponds are simply not written, and the data at those byte(s) at the destination are allowed to remain unchanged.
28	Monochrome Pattern Transparency Mode. This bit applies only when the pattern data is monochrome. This bit determines whether or not the byte(s) at the destination corresponding to the pixel to which a given bit of the pattern data also corresponds will actually be written if that pattern data bit has the value of 1. This feature can make it possible to use the pattern as a transparency mask. The BLT Engine is configured to accepted either monochrome or color pattern data via the opcode field. 0 = This causes normal operation with regard to the use of the pattern data. Wherever a bit in the pattern data has the value of 0, the color specified in the background color register is used as the pattern operand in the bit-wise operation for the pixel corresponding to the pattern data bit, and the bytes at the destination corresponding to that pixel are written with the result. 1 = Wherever a bit in the pattern data has the value of 0, the byte(s) at the destination corresponding to the pixel to which the pattern data bit also corresponds are simply not written, and the data at those byte(s) at the destination are allowed to remain unchanged.
27:26	32 bpp byte mask. 21 = 1 = write alpha channel [31:24]; 20 = 1 = write RGB channels [23:00]. This field is only used for 32bpp.
25:24	Color Depth. 00 = 8 Bit Color Depth 01 = 16 Bit Color Depth 10 = 16 Bit Color Depth 11 = 32 Bit Color Depth
23:16	Raster Operation Select. These 8 bits are used to select which one of 256 possible raster operations is to be performed by the BLT Engine. The opcode field must indicate a monochrome source if ROP = F0.



Bit	Descriptions
15:0	Destination Pitch (Offset). For non-XY Blits, the signed 16bit field allows for specifying upto ± 32Kbytes signed pitches in bytes (same as before). For X, Y Blits with tiled (X) surfaces, the pitch for Destination will be 512Byte aligned and should be programmable upto ± 128Kbytes. In this case, this 16bit signed pitch field is used to specify upto ± 32KDWords. For X, Y blits with nontiled surfaces (linear surfaces), this 16bit field can be programmed to byte specification of upto ± 32Kbytes (same as before). These 16 bits store the signed memory address offset value by which the destination address originally specified in the Destination Address Register is incremented or decremented as each scan line's worth of destination data is written into the frame buffer by the BLT Engine, so that the destination address will point to the next memory address to which the next scan line's worth of destination data is to be written. If the intended destination of a BLT operation is within on-screen frame buffer memory, this offset is normally set so that each subsequent scan line's worth of destination data lines up vertically with the destination data in the scan line, above. However, if the intended destination of a BLT operation is within off-screen memory, this offset can be set so that each subsequent scan line's worth of destination data is stored at a location immediately after the location where the destination data for the last scan line ended, in order to create a single contiguous block of bytes of destination data at the destination.

14.10.3 BR05—Setup Expansion Background Color

Memory Offset Address: none
Default: None
Attributes: State accessible



Bit	Descriptions
31:0	Setup Expansion Background Color Bits [31:0]. These bits provide the one, two, or four bytes worth of color data that select the background color to be used in the color expansion of monochrome pattern or source data for either the SCANLINE_BLT or TEXT_BLT instructions. BR05 is also used as the solid pattern for the PIXEL_BLT instruction. Whether one, two, or three bytes worth of color data is needed depends upon the color depth to which the BLT Engine has been set. For a color depth of 32bpp, 16bpp and 8bpp, bits [31:0], [15:0] and [7:0], respectively, are used.



14.10.4 BR06—Setup Expansion Foreground Color

Memory Offset Address: none
Default: None
Attributes: State accessible

31 0

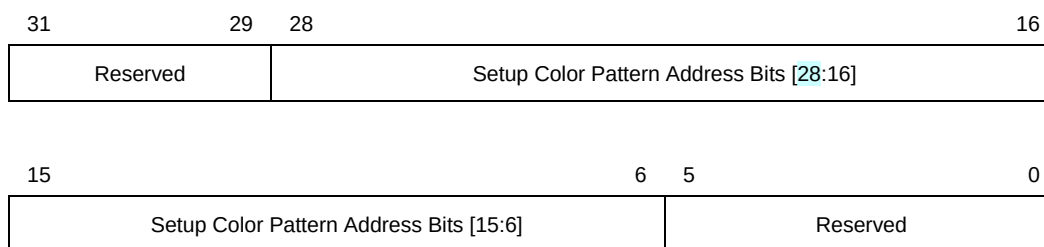
Setup Expansion Foreground Color Bits [31:0]

Bit	Descriptions
31:24	Reserved.
31:0	Setup Expansion Foreground Color Bits [31:0]. These bits provide the one, two, or four bytes worth of color data that select the foreground color to be used in the color expansion of monochrome pattern or source data for either the SCANLINE_BLT or TEXT_BLT instructions. Whether one, two, or three bytes worth of color data is needed depends upon the color depth to which the BLT Engine has been set. For a color depth of 32bpp, 16bpp and 8bpp, bits [31:0], [15:0] and [7:0], respectively, are used.



14.10.5 BR07—Setup Color Pattern Address

Memory Offset Address: none
 Default: None
 Attributes: State accessible

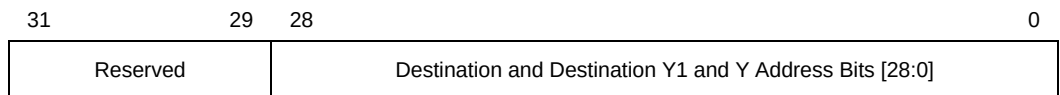


Bit	Descriptions
31:2 9	Reserved. The maximum GC graphics address is 512 MBs.
28:6	Pattern Address. These 23 bits specify the starting address of the color pattern from the SETUP_BLT instruction. This register works identically to the Pattern Address register, but this version is only used with the SCANLINE_BLT instruction execution. The pattern data must be located in linear memory. The pattern data must be located on a pattern-size boundary. The pattern is always of 8x8 pixels, and therefore, its size is dependent upon its pixel depth. The pixel depth may be 8, 16, or 32 bits per pixel if the pattern is in color (the pixel depth of a color pattern must match the pixel depth to which the graphics system has been set). Monochrome patterns require 8 bytes and are supplied through the instruction. Color patterns of 8, 16, and 32 bits per pixel color depth must start on 64-byte, 128-byte and 256-byte boundaries, respectively.
5:0	Reserved. These bits always return 0 when read.



14.10.6 BR09—Destination Address

Memory Offset Address: None
 Default: None
 Attributes: State accessible

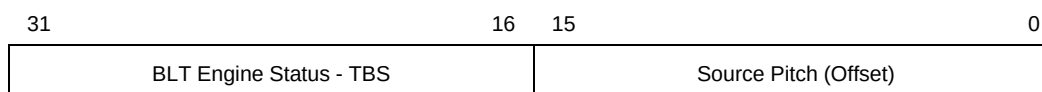


Bit	Descriptions
31:29	Reserved.
28:0	Destination Address Bits. When tiling is enabled for XY-blits, this base address should be limited to 4KB. Otherwise for XY blits, there is no restriction and it is same as before. These 29 bits specify the starting pixel address of the destination data. This register is also the working destination address register and changes as the BLT Engine performs the accesses. Used as the scan line address (Destination Y Address & Destination Y1 Address) for BLT instructions: PIXEL_BLT, SCANLINE_BLT, and TEXT_BLT. In this case the address points to the first pixel in a scan line and is compared with the ClipRect Y1 & Y2 address registers to determine whether the scan line should be written or not. The Destination Y1 address is the top scan line to be written for text. Note that for non-XY blits (COLOR_BLT, SRC_COPY_BLT), this address points to the first byte to be written. This register is always the last register written for a BLT drawing instruction. Writing BR09 starts the BLT engine execution. Note: Some instructions affect only one scan line (requiring only one coordinate); other instructions affect multiple scan lines and need both coordinates.



14.10.7 BR11—BLT Source Pitch (Offset)

Memory Offset Address: None
 Default: None
 Attributes: Not accessible

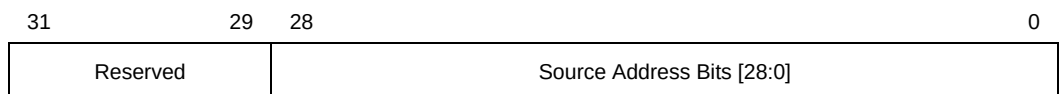


Bit	Descriptions
31:16	BLT Engine Status. This field is used to read back important debug status. It will be specified in the future.
15:0	Source Pitch (Offset) For non-XY Blits with color source operand (SRC_COPY_BLT), the signed 16bit field allows for specifying upto $\pm$ 32Kbytes signed pitch in bytes (same as before). For X, Y Blits with tiled (X) surfaces, the pitch for Color Source will be 512Byte aligned and should be programmable upto $\pm$ 128Kbytes. In this case, this 16bit signed pitch field is used to specify upto $\pm$ 32KDWords. For X, Y blits with nontiled color source surfaces (linear surfaces), this 16bit field can be programmed to byte specification of upto $\pm$ 32Kbytes (same as before). When the color source data is located within the frame buffer or AGP aperture, these signed 16 bits store the memory address offset (pitch) value by which the source address originally specified in the Source Address Register is incremented or decremented as each scan line's worth of source data is read from the frame buffer by the BLT Engine, so that the source address will point to the next memory address from which the next scan line's worth of source data is to be read. Note that if the intended source of a BLT operation is within on-screen frame buffer memory, this offset is normally set to accommodate the fact that each subsequent scan line's worth of source data lines up vertically with the source data in the scan line, above. However, if the intended source of a BLT operation is within off-screen memory, this offset can be set to accommodate a situation in which the source data exists as a single contiguous block of bytes where in each subsequent scan line's worth of source data is stored at a location immediately after the location where the source data for the last scan line ended.



14.10.8 BR12—Source Address

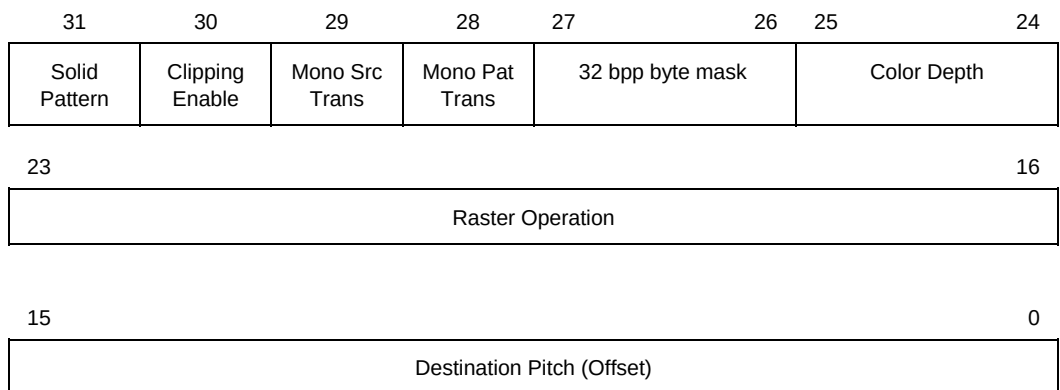
Memory Offset Address: None
 Default: None
 Attributes: Not accessible



Bit	Descriptions
31:29	Reserved. The maximum GC Graphics address is 512 MBs.
28:0	Source Address Bits [28:0]. When tiling is enabled for XY-blits with Color source surfaces, this base address should be limited to 4KB. Otherwise for XY blits, there is no restriction and it is same as before, including for monosource and text blits. Note that for non-XY blit with Color Source (SRC_COPY_BLT), this address points to the first byte to be read. These 29 bits are used to specify the starting pixel address of the color source data. The lower 3 bits are used to indicate the position of the first valid byte within the first Quadword of the source data.

14.10.9 BR13—BLT Raster OP, Control, and Destination Pitch

Memory Offset Address: None
 Default: 0000 xxxx
 Attributes: Not accessible





Bit	Descriptions
31	Solid Pattern Select. This bit applies only when the pattern data is monochrome. This bit determines whether or not the BLT Engine actually performs read operations from the frame buffer in order to load the pattern data. Use of this feature to prevent these read operations can increase BLT Engine performance, if use of the pattern data is indeed not necessary. The BLT Engine is configured to accept either monochrome or color pattern data via the opcode field. 0 = This causes normal operation with regard to the use of the pattern data. The BLT Engine proceeds with the process of reading the pattern data, and the pattern data is used as the pattern operand for all bit-wise operations. 1 = The BLT Engine forgoes the process of reading the pattern data, the presumption is made that all of the bits of the pattern data are set to 0, and the pattern operand for all bit-wise operations is forced to the background color specified in the Color Expansion Background Color Register.
30	Clipping Enabled: 1 = Enabled; 0 = Disabled
29	Monochrome Source Transparency Mode. This bit applies only when the source data is in monochrome. This bit determines whether or not the byte(s) at the destination corresponding to the pixel to which a given bit of the source data also corresponds will actually be written if that source data bit has the value of 0. This feature can make it possible to use the source as a transparency mask. The BLT Engine is configured to accepted either monochrome or color source data via the opcode field. 0 = This causes normal operation with regard to the use of the source data. Wherever a bit in the source data has the value of 0, the color specified in the background color register is used as the source operand in the bit-wise operation for the pixel corresponding to the source data bit, and the bytes at the destination corresponding to that pixel are written with the result. 1 = Where a bit in the source data has the value of 0, the byte(s) at the destination corresponding to the pixel to which the source data bit also corresponds are simply not written, and the data at those byte(s) at the destination are allowed to remain unchanged.
28	Monochrome Pattern Transparency Mode. This bit applies only when the pattern data is monochrome. This bit determines whether or not the byte(s) at the destination corresponding to the pixel to which a given bit of the pattern data also corresponds will actually be written if that pattern data bit has the value of 1. This feature can make it possible to use the pattern as a transparency mask. The BLT Engine is configured to accepted either monochrome or color pattern data via the opcode in the Opcode and Control register. 0 = This causes normal operation with regard to the use of the pattern data. Where a bit in the pattern data has the value of 0, the color specified in the background color register is used as the pattern operand in the bit-wise operation for the pixel corresponding to the pattern data bit, and the bytes at the destination corresponding to that pixel are written with the result. 1= Wherever a bit in the pattern data has the value of 0, the byte(s) at the destination corresponding to the pixel to which the pattern data bit also corresponds are simply not written, and the data at those byte(s) at the destination are allowed to remain unchanged.
25:24	Color Depth. 00 = 8 Bit Color Depth 01 = 16 Bit Color Depth 10 = 24 Bit Color Depth 11 = Reserved
23:16	Raster Operation Select. These 8 bits are used to select which one of 256 possible raster operations is to be performed by the BLT Engine. The opcode must indicate a monochrome source operand if ROP = F0.

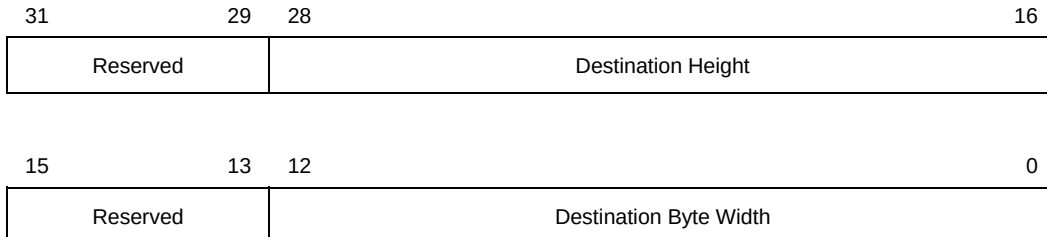


Bit	Descriptions
15:0	Destination Pitch (Offset). These 16 bits store the signed memory address offset value by which the destination address originally specified in the Destination Address Register is incremented or decremented as each scan line's worth of destination data is written into the frame buffer by the BLT Engine, so that the destination address will point to the next memory address to which the next scan line's worth of destination data is to be written. If the intended destination of a BLT operation is within on-screen frame buffer memory, this offset is normally set so that each subsequent scan line's worth of destination data lines up vertically with the destination data in the scan line, above. However, if the intended destination of a BLT operation is within off-screen memory, this offset can be set so that each subsequent scan line's worth of destination data is stored at a location immediately after the location where the destination data for the last scan line ended, in order to create a single contiguous block of bytes of destination data at the destination.

14.10.10 BR14—Destination Width & Height

Memory Offset Address: None
Default: None
Attributes: Not accessible

BR14 contains the values for the height and width of the data to be BLT. If these values are not correct, such that the BLT Engine is either expecting data it does not receive or receives data it did not expect, the system can hang.

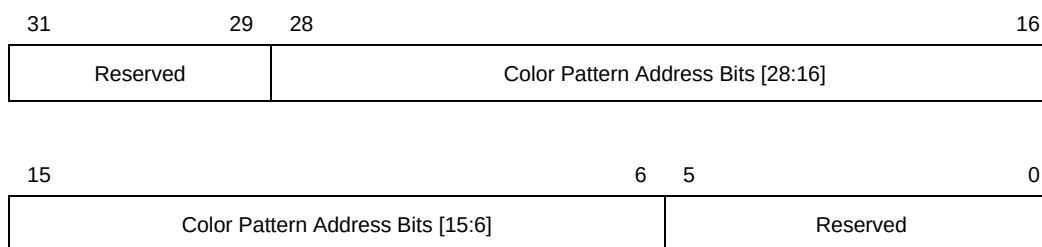


Bit	Descriptions
31:2 9	Reserved.
28:1 6	Destination Height. These 13 bits specify the height of the destination data in terms of the number of scan lines. This is a working register.
15:1 3	Reserved.
12:0	Destination Byte Width. These 13 bits specify the width of the destination data in terms of the number of bytes per scan line. The number of pixels per scan line into which this value translates depends upon the color depth to which the graphics system has been set.



14.10.11 BR15—Color Pattern Address

Memory Offset Address: None
 Default: None
 Attributes: Not accessible

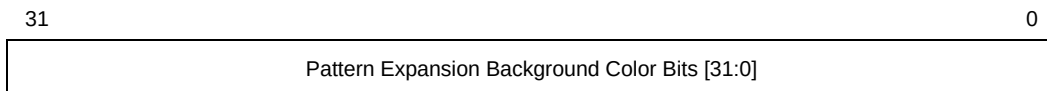


Bit	Descriptions
31:29	Reserved. The maximum GC graphics address is 512 MBs.
28:6	Color Pattern Address. There is no change to the Color Pattern address specification due to Non-Power-of-2 change. It remains the same as before. The pattern data must be located in linear memory. These 23 bits specify the starting address of the pattern. The pattern data must be located on a pattern-size boundary. The pattern is always of 8x8 pixels, and therefore, its size is dependent upon its pixel depth. The pixel depth may be 8, 16, or 32 bits per pixel if the pattern is in color (the pixel depth of a color pattern must match the pixel depth to which the graphics system has been set). Monochrome patterns require 8 bytes and are applied through the instruction. Color patterns of 8, 16, and 32 bits per pixel color depth must start on 64-byte, 128-byte and 256-byte boundaries, respectively.
5:0	Reserved. These bits always return 0 when read.



14.10.12 BR16—Pattern Expansion Background & Solid Pattern Color

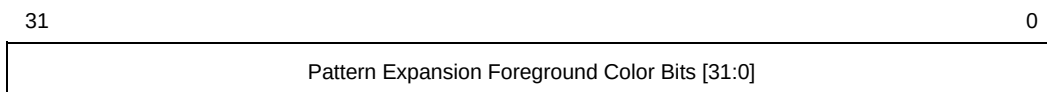
Memory Offset Address: 40040h
 Default: None
 Attributes: RO; DWord accessible



Bit	Descriptions
31:0	Pattern Expansion Background Color Bits [31:0]. These bits provide the one, two, or four bytes worth of color data that select the background color to be used in the color expansion of monochrome pattern data during BLT operations. Whether one, two, or four bytes worth of color data is needed depends upon the color depth to which the BLT Engine has been set. For a color depth of 32bpp, 16bpp and 8bpp, bits [31:0], [15:0] and [7:0], respectively, are used.

14.10.13 BR17—Pattern Expansion Foreground Color

Memory Offset Address: None
 Default: None
 Attributes: Not accessible

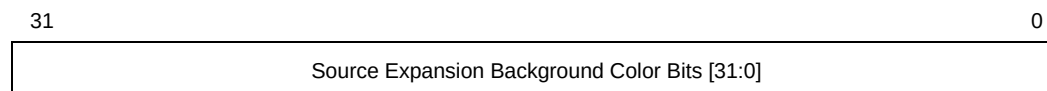


Bit	Descriptions
31:0	Pattern Expansion Foreground Color Bits [31:0]. These bits provide the one, two, or four bytes worth of color data that select the foreground color to be used in the color expansion of monochrome pattern data during BLT operations. Whether one, two, or four bytes worth of color data is needed depends upon the color depth to which the BLT Engine has been set. For a color depth of 32bpp, 16bpp and 8bpp, bits [31:0], [15:0] and [7:0], respectively, are used.



14.10.14 BR18—Source Expansion Background, and Destination Color

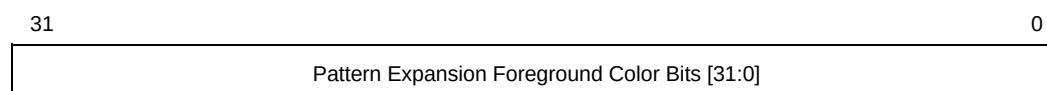
Memory Offset Address: None
Default: None
Attributes: Not accessible



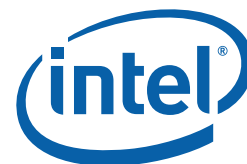
Bit	Descriptions
31:0	Source Expansion Background Color Bits [31:0]. These bits provide the one, two, or four bytes worth of color data that select the background color to be used in the color expansion of monochrome source data during BLT operations. This register is also used to support destination transparency mode and Solid color fill. Whether one, two, three, or four bytes worth of color data is needed depends upon the color depth to which the BLT Engine has been set. For a color depth of 32bpp, 16bpp and 8bpp, bits [31:0], [15:0] and [7:0], respectively, are used.

14.10.15 BR19—Source Expansion Foreground Color

Memory Offset Address: None
Default: None
Attributes: Not accessible



Bit	Descriptions
31:0	Pattern / Source Expansion Foreground Color Bits [31:0]. These bits provide the one, two, or four bytes worth of color data that select the foreground color to be used in the color expansion of monochrome source data during BLT operations. Whether one, two, or four bytes worth of color data is needed depends upon the color depth to which the BLT Engine has been set. For a color depth of 32bpp, 16bpp and 8bpp, bits [31:0], [15:0] and [7:0], respectively, are used.



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.

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, or life sustaining applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

The Intel® 965 Express Chipset family and Intel® G35 Express Chipset may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

I2C is a two-wire communications bus/protocol developed by Philips. SMBus is a subset of the I2C bus/protocol and was developed by Intel. Implementations of the I2C bus/protocol may require licenses from various entities, including Philips Electronics N.V. and North American Philips Corporation.

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

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

Copyright © 2008, Intel Corporation. All rights reserved.



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 refers 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 <i>3D Pipeline</i> .
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 “perspective 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 <i>Strips and Fans</i> .
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 <i>Graphics Processing Engine</i> .
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 <i>Geometry Shader</i> .
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 discontinuous 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, <i>r2</i> , is 256-bit wide, 256-bit aligned register. A sub-register, <i>r2.3:d</i> , is the fourth dword of GRF register <i>r2</i> .
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.

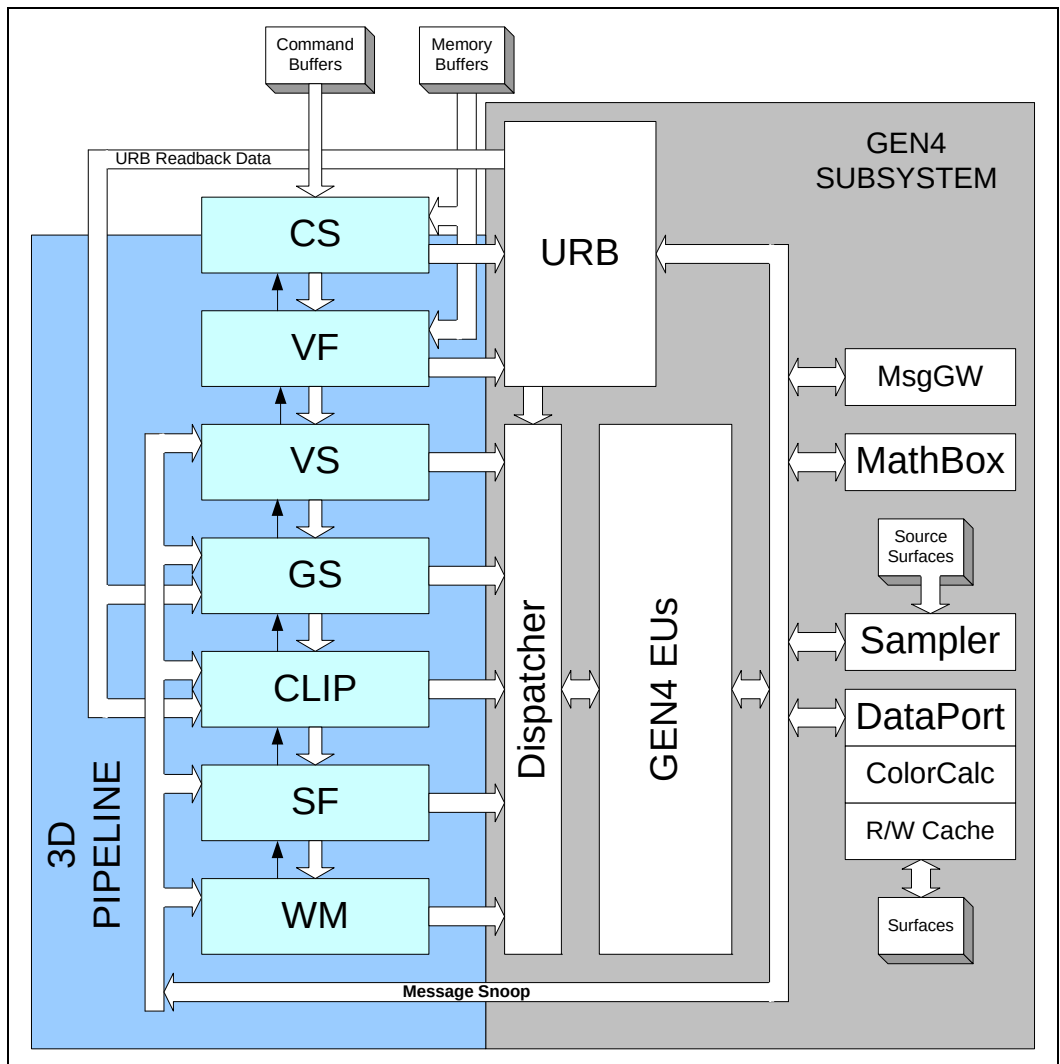


2 3D Pipeline

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 topologies.
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).

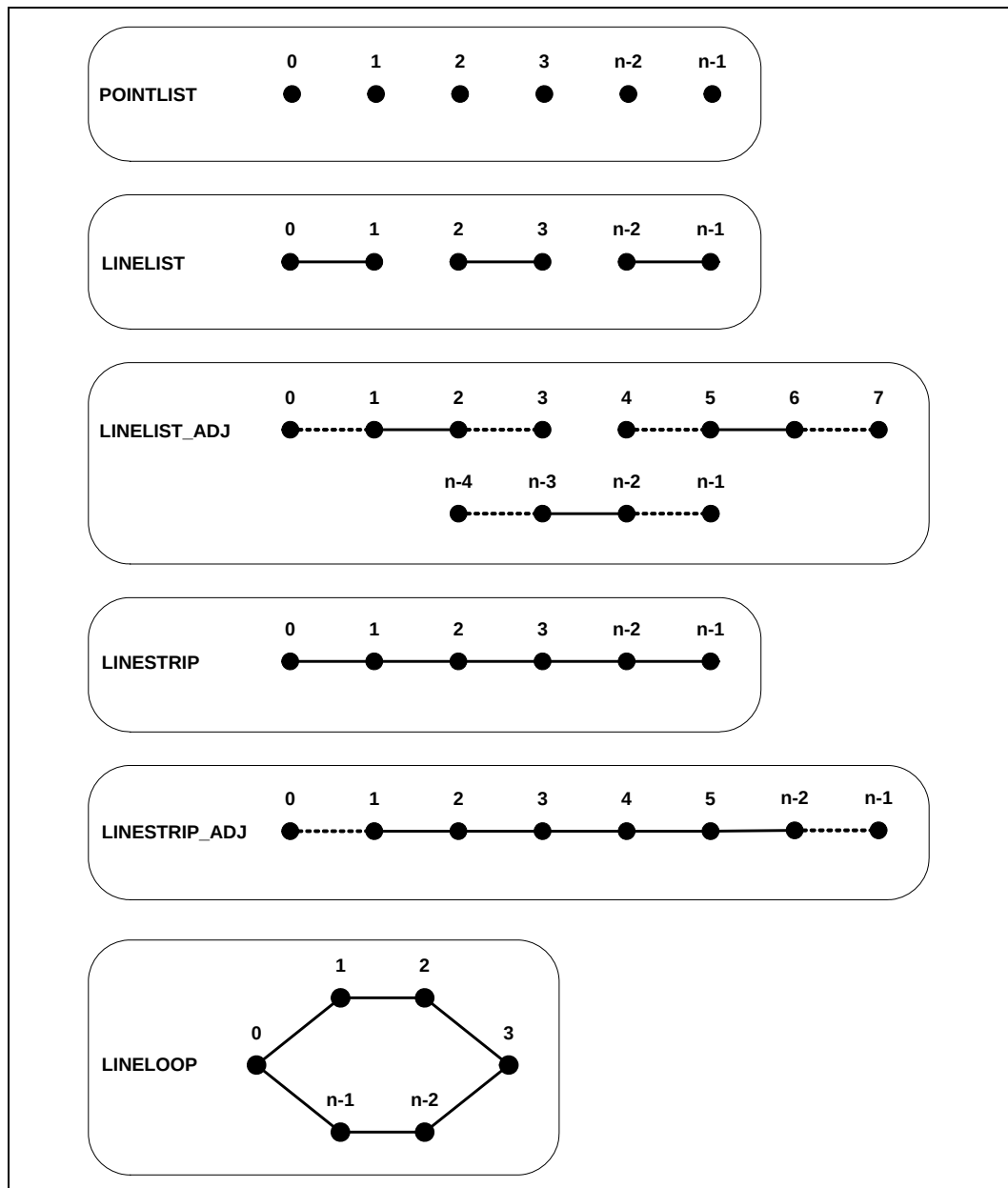


3D Primitive Topology Type (ordered alphabetically)	Description
RECTLIST	A list of independent rectangles, where only 3 vertices are provided per rectangle object, with the fourth vertex implied by the definition of a rectangle. V0=LowerRight, V1=LowerLeft, V2=UpperLeft. Implied V3 = V0-V1+V2. Programming 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 polygon stipple is not applied (even if enabled). This can be used to support “point” polygon fill mode, under the combination of the following conditions: (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.

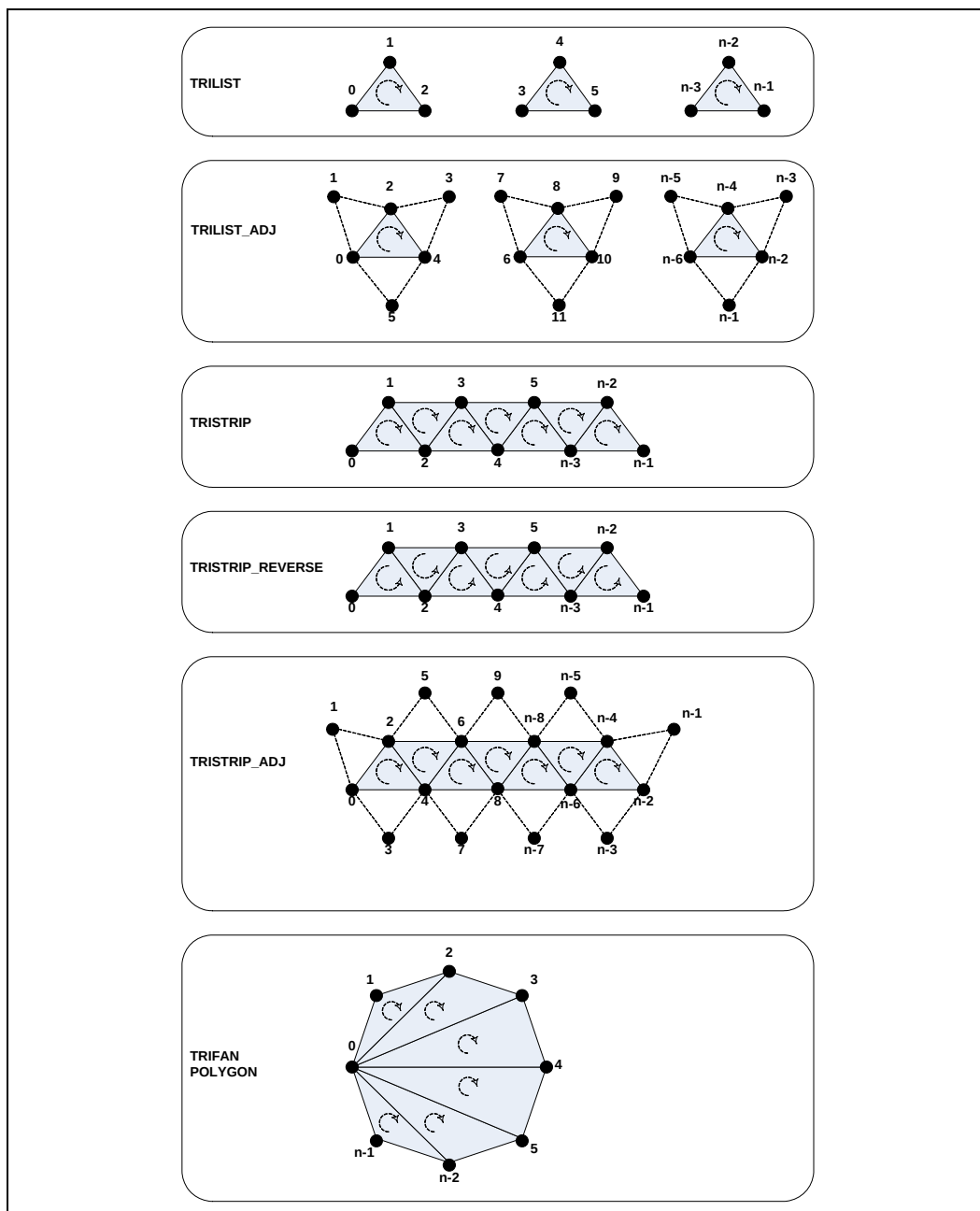
3D Primitive Topology Type (ordered alphabetically)	Description
TRISTRIP	A list of vertices connected such that, after the first two vertices, each additional vertex is associated with the last two vertices to define a connected triangle object. Programming Restrictions: • Normal usage expects at least 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 tristrrips into smaller pieces and still maintain correct face orientations.

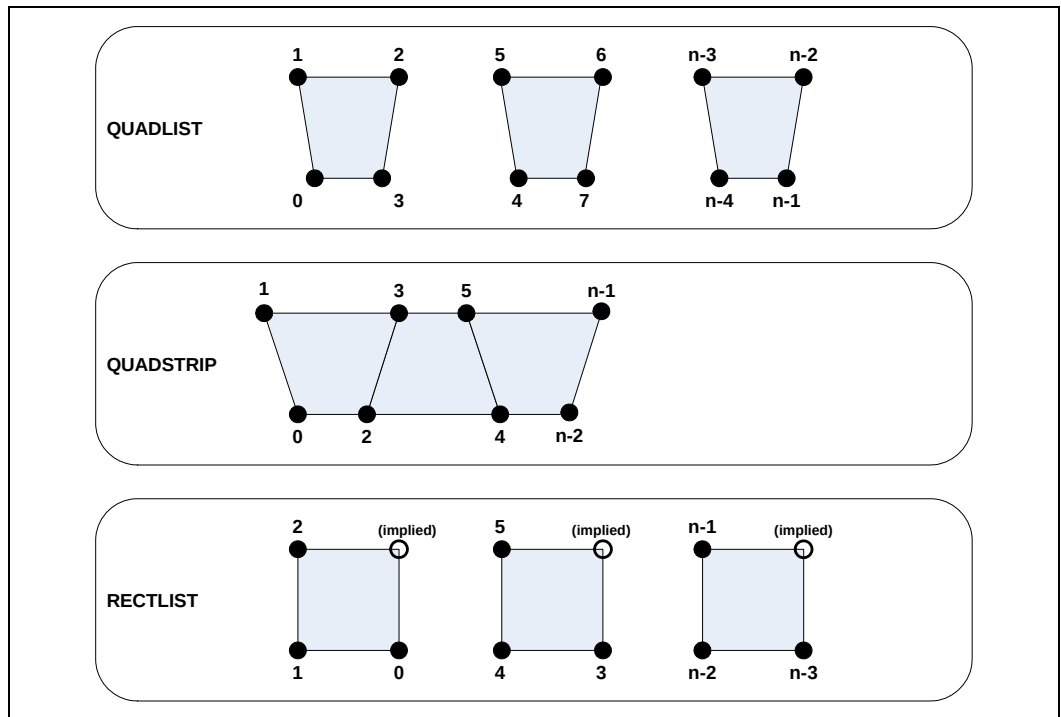


The following diagrams illustrate the basic 3D primitive topologies. (Variants are not shown if they have the same definition with respect to the information provided in the diagrams).



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 <i>VF Stage</i> chapter (as it is executed there), though the processing of this command includes the entire 3D pipeline.



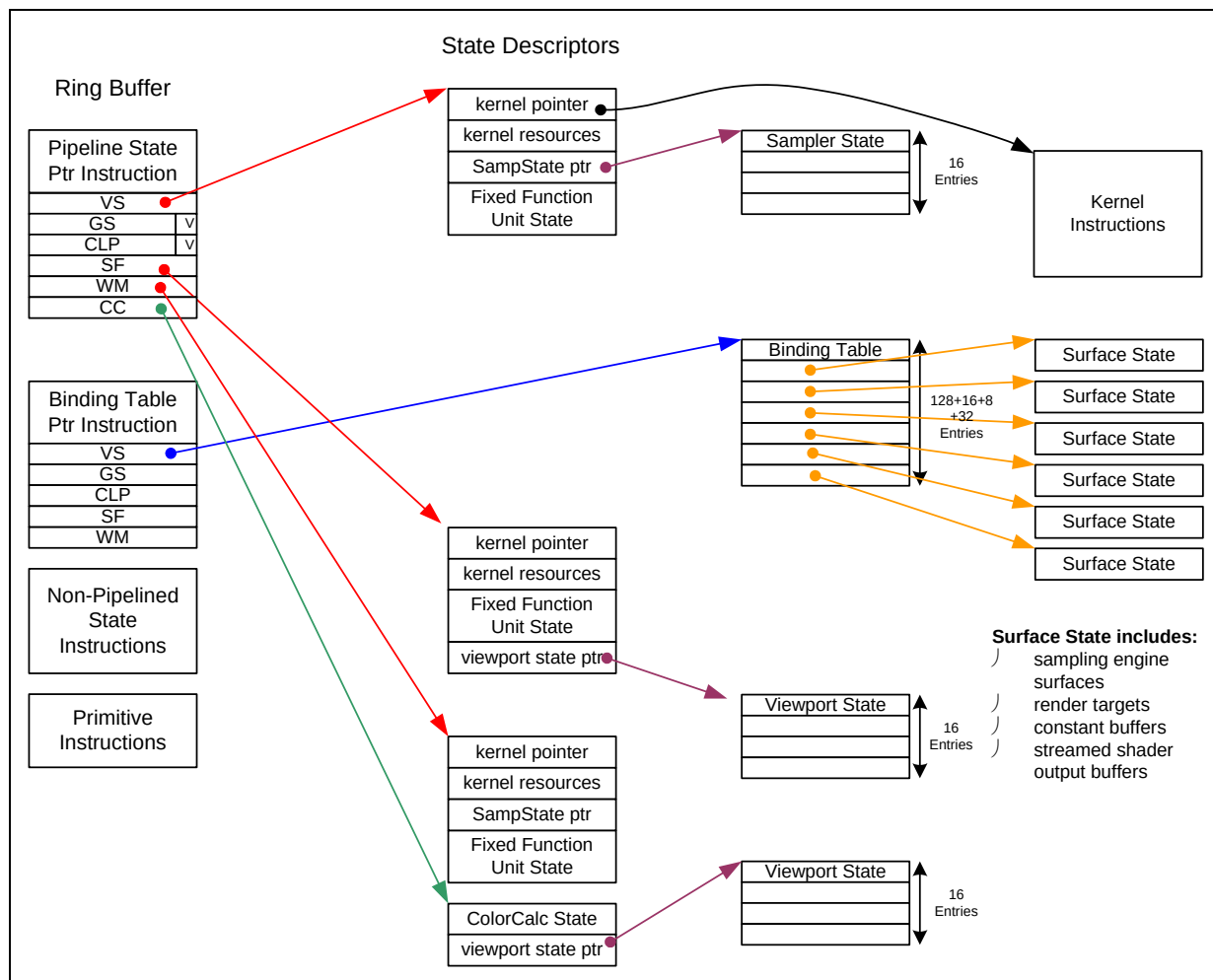
Command	Description
Control Operation Commands	
PIPE_CONTROL	This control operation command allows software to synchronize 3D pipeline operations as seen by the CPU. For example, this command can be used to inform the CPU (via a CPU-snoopable memory write or CPU interrupt) when previously-issued commands 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 <i>Clipper</i> 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 <i>Clipper</i> 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 following table defines the Vertex Header. The Position fields are described in further detail below.

Table 2-2. VUE 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 correctly, the clip-space X,Y,Z coordinates are multiplied by the reciprocal of the clip space W coordinate).
 - Note that the X,Y,Z coordinates may contain INFINITY or NaN values (see below).
- The NDC RHW coordinate is the reciprocal of the clip-space W coordinate and therefore, under normal perspective projections, it is the reciprocal of the view-space Z coordinate. Note that NDC space is really a 3D coordinate space, where this RHW coordinate is retained in order to perform perspective-correct interpolation, *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 = $X \cdot RHW$	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 position 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:

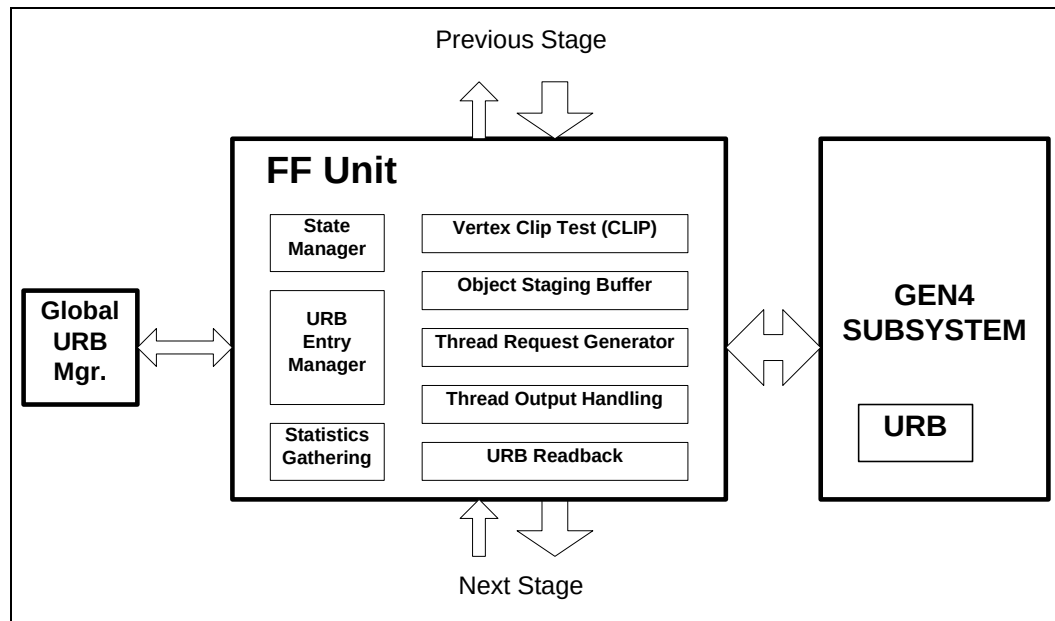
- X,Y coordinates are in absolute screen space (pixel coordinates, upper left origin). See Vertex X,Y Clamping and Quantization in the SF section for a discussion of the limitations/restrictions placed on screenspace X,Y coordinates.
- Z coordinate has been mapped into the range used for DepthTest.
 - 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 <i>3D Control Operations</i>.
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. (<i>Implementation Note:</i> 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 sent 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 <u>stage</u> 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

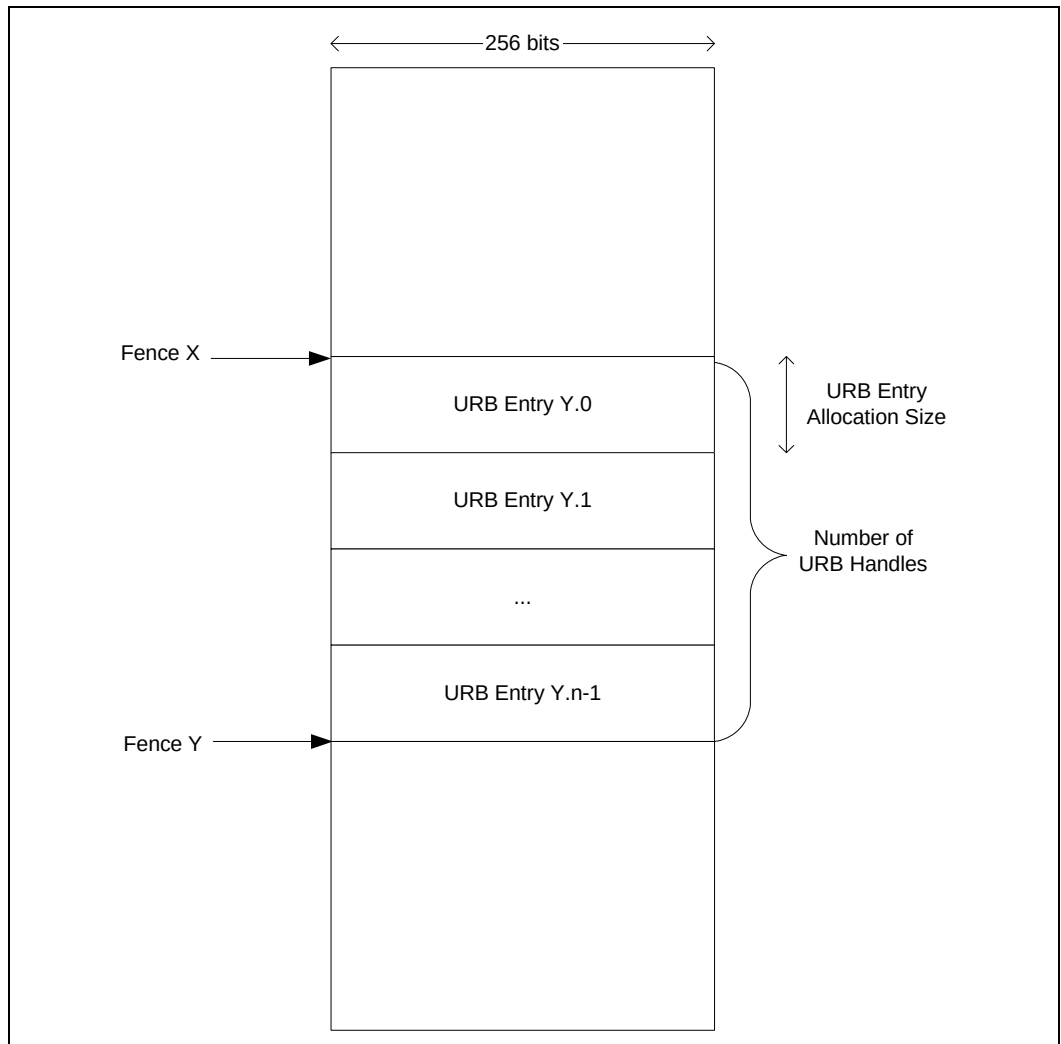
The **Number of URB Handles** state variable specifies how many URB entries are allocated for output by the FF unit.

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**:
 - o VS: For each input URB entry, an output URB entry must be available.
 - o 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.
 - o CLIP: Same considerations as GS (above)
 - o SF: An output URB entry must be available to store the results of the SETUP thread.
 - o 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 (1 st GEN4 core instruction) of the kernel program run by threads spawned by this FF unit. It is specified as a 64-byte-granular offset from the General State Pointer .	All FFs spawning threads (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 (SIMD _{Nxm} with $m = 1$) or multiple program flows (SIMD _{Nxm} with $m > 1$). See CR0 description in <i>ISA Execution Environment</i> .	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 handling logic in the EU. (See ISA.)	All FFs spawning threads (VS, GS, CLIP, SF, WM)
Sampler Count	This is a <u>hint</u> which specifies how many indirect SAMPLER_STATE structures should be prefetched concurrent with thread initiation. It is recommended that software program this field to equal the number of samplers, though there may be some minor performance impact if this number gets large. This value should not exceed the number of samplers accessed by the thread as there would be no performance advantage. Note that the data prefetch is treated as any other memory fetch (with respect to page faults, etc.).	All stages supporting sampling (VS, GS, WM)
Binding Table Entry Count	This is a <u>hint</u> which specifies how many indirect BINDING_TABLE_STATE structures should be prefetched concurrent with thread initiation. (The 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.

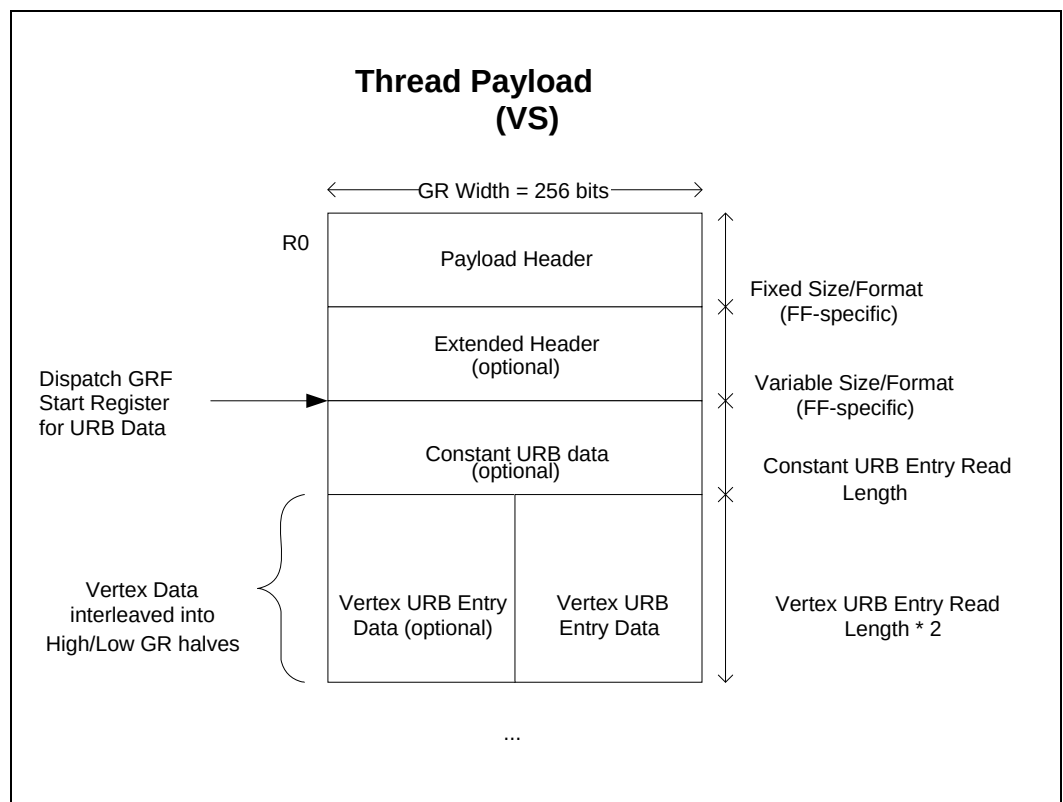
Table 2-4. Payload Sizes

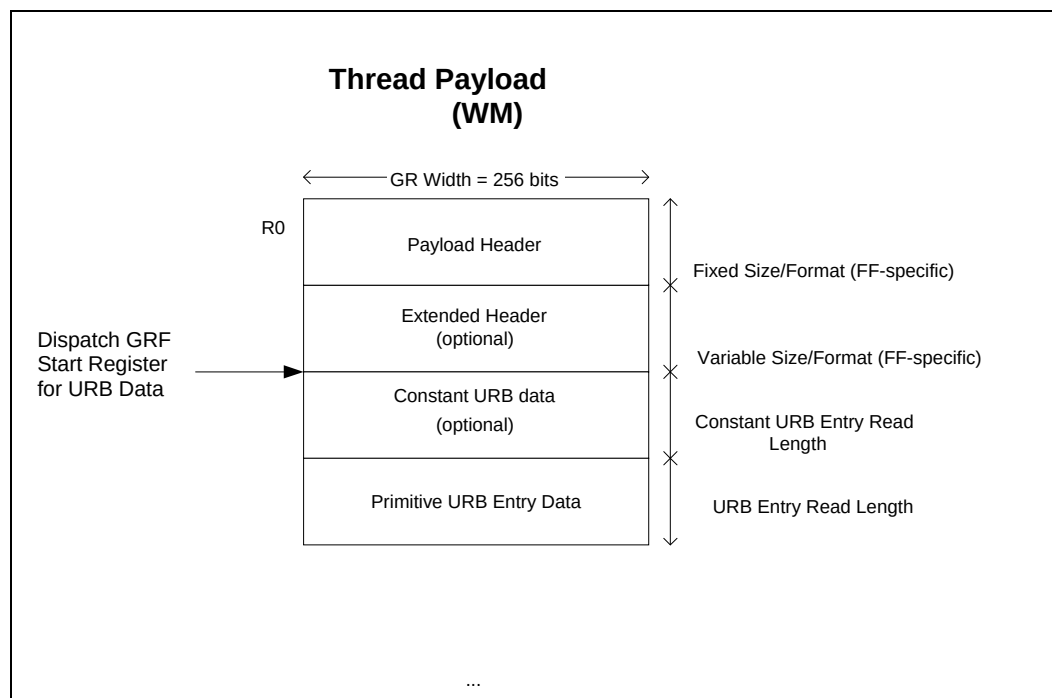
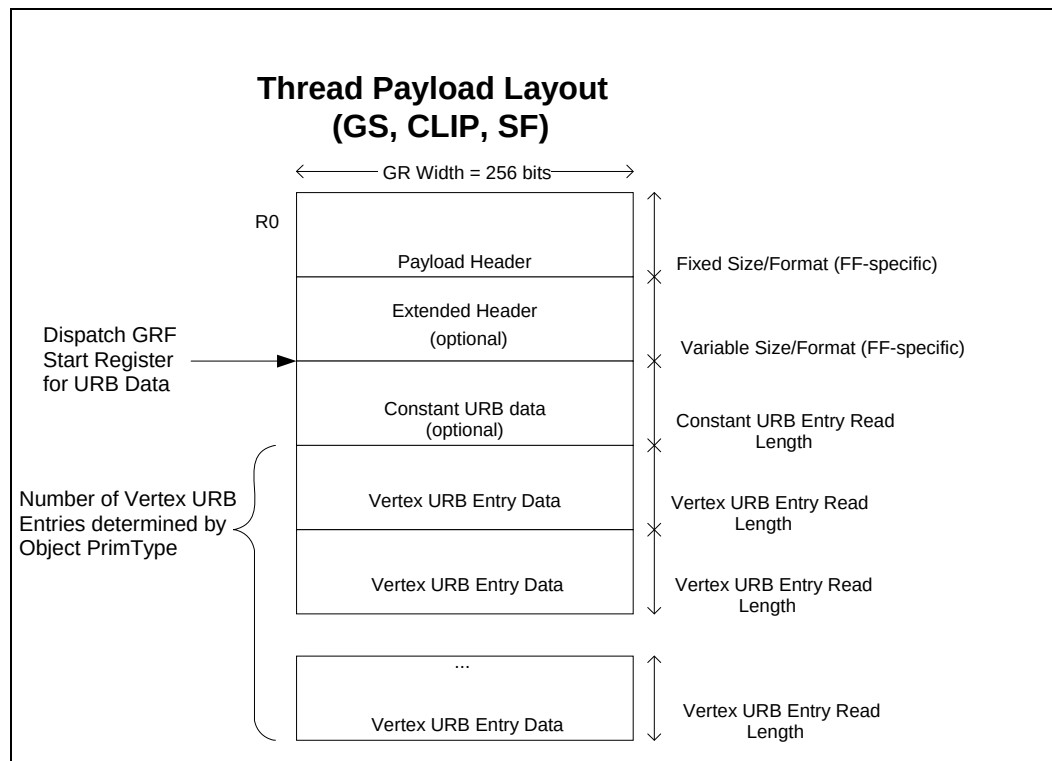
FF Stage	Fixed Payload Header Size (# GRF Regs)	Extended Payload Header Size (# GRF Regs)	URB Data
CS	N/A	N/A	N/A
VF	N/A	N/A	N/A
VS	1 (R0 only)	0 (typically, can be non-zero if padding required)	Optional Constant URB data + 2 (interleaved) Vertex URB Entries
GS	1 (R0 only)	0 (typically, can be non-zero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







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 <i>Debugging</i> 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 <u>not</u> uniquely identify the thread over any significant period of time. <i>Implementation Note:</i> 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 <i>Shared Functions</i> 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 <i>Shared Functions</i> for a description of a Sampler State Table.	All FFs spawning threads which sample (VS, GS, WM)
Per Thread Scratch Space	This field specifies the amount of scratch space allocated to each thread spawned by the FF unit. The driver must allocate enough contiguous scratch space, starting at the Scratch Space Base Pointer, to ensure that the Maximum Number of Threads can each get Per-Thread Scratch Space size without exceeding the driver-allocated scratch space.	All FFs spawning threads
Handle ID <n>	This ID is assigned by the FF unit and links the thread to a specific entry within the FF unit. The FF unit will use this information upon detecting a URB_WRITE message issued by the thread. Threads spawned by the GS, CLIP, and SF units are provided with a single Handle ID / URB Return Handle pair. Threads spawned by the VS are provided with one or two pairs (depending on how many vertices are to be processed). Threads spawned by the WM do not write to URB entries, and therefore this info is not supplied.	VS,GS,CLIP, SF
URB Return Handle <n>	This is an initial destination URB handle passed to the thread. If the thread does output URB entries, this identifies the destination URB entry. Threads spawned by the GS, CLIP, and SF units are provided with a single Handle ID / URB Return Handle pair. Threads spawned by the VS are provided with one or two pairs (depending on how many vertices are to be processed). Threads spawned by the WM do not write to URB entries, and therefore this info is not supplied.	VS,GS,CLIP, SF



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 associated with the primitive vertices as received by the FF unit. Refer to the individual FF unit chapters for cases where the FF unit modifies the value before passing it to the thread. (E.g., certain units perform toggling of TRIANGLESTRIP and TRIANGLESTRIP_REV).	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 (optionally) 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 $\min(128, 16 * \text{GRF Register Block Count})$. Software is responsible for ensuring this maximum size is not exceeded, taking into account:
 - o The size of the Fixed and Extended Payload Header associated with the FF unit.
 - o The **Dispatch GRF Start Register for URB Data SV**.
 - o The amount of CURBE data included (via **Constant URB Entry Read Length**)
 - o The number of VUEs included (as a function of FF unit, it's state programming, and incoming primitive types)
 - o The amount of VUE data included for each vertex (via **Vertex URB Entry Read Length**)
 - o (For WM-spawned PS threads) The amount of Primitive URB Entry data.
- For any type of URB Entry reads:
 - o 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 outputting 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'd?	Write (Render) Cache Flushed?	Inst/State Cache Inv'd?	Index-Based Vertex Cache Inv'd?	VF Cache Inv'd?	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. <table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td></td><td>No write occurs as a result of this instruction. This can be used to implement a “trap” operation, etc.</td><td>All</td></tr><tr><td>1h</td><td></td><td>Write the QWord containing Immediate Data Low, High DWs to the Destination Address</td><td>All</td></tr><tr><td>2h</td><td></td><td>Write the 64-bit PS_DEPTH_COUNT register to the Destination Address</td><td>All</td></tr><tr><td>3h</td><td></td><td>Write the 64-bit TIMESTAMP register to the Destination Address</td><td>All</td></tr></table> <table><tr><th>Errata</th><th>Description</th><th>Project</th></tr><tr><td>#</td><td>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.</td><td>BW-A,B</td></tr></table>			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.
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 <i>after</i> 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 <i>ignored</i> 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 <i>Memory Interface Registers</i> 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. <table><thead><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr></thead><tbody><tr><td>0h</td><td></td><td>Use process local PGTT</td><td>All</td></tr><tr><td>1h</td><td></td><td>Use Global GTT; valid only for privileged commands</td><td>All</td></tr></tbody></table> Programming Notes Ignored if "No write" is the selected in Operation . Project All			Value	Name	Description	Project	0h		Use process local PGTT	All	1h		Use Global GTT; valid only for privileged commands	All
	Value	Name	Description	Project												
	0h		Use process local PGTT	All												
1h		Use Global GTT; valid only for privileged commands	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 <i>VF Stage</i> 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 <i>Index Buffer</i> 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 <i>Vertex Buffers</i> 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 <i>Input Vertex Data</i> 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 where 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	



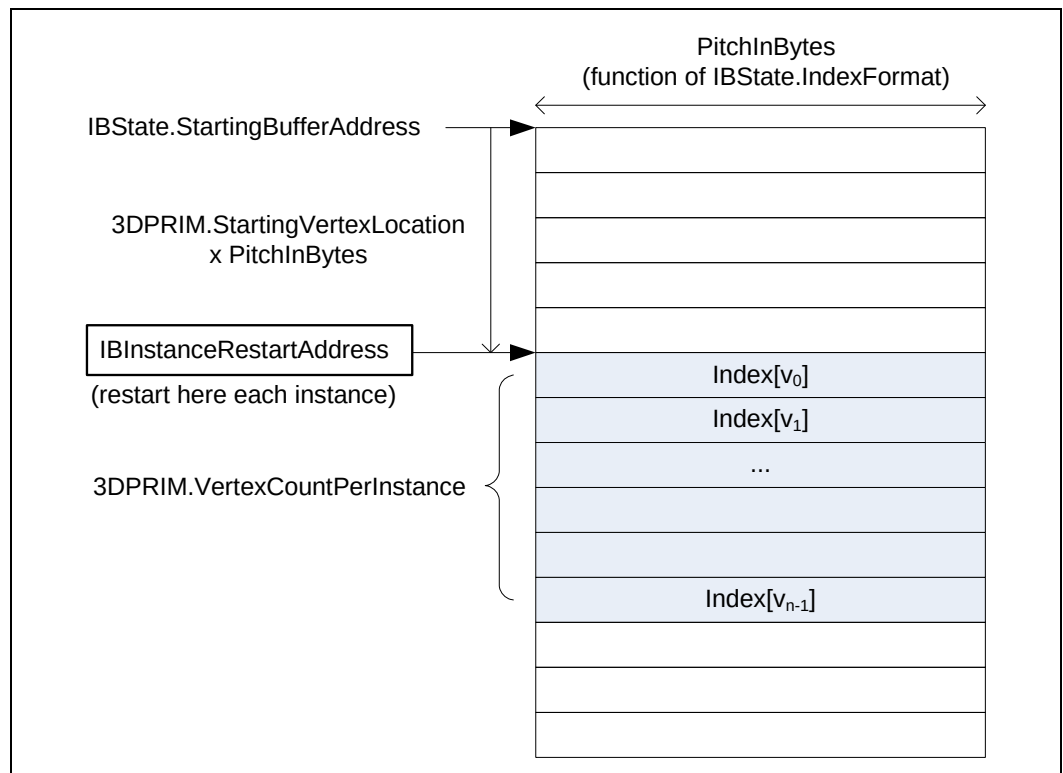
3DSTATE_INDEX_BUFFER		
2	31:0	Buffer Ending Address Project: All Format: GraphicsAddress[31:0] FormatDesc This field contains the address of the last valid byte in the index buffer. Any index buffer reads past this address returns 0 for all vertex elements (as if the buffer was zero-extended). Software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

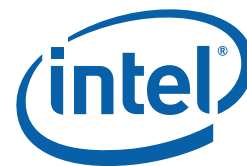
The following table lists which primitive topology types support the presence of Cut Indices. When the Index Buffer has **Cut Index Enable** set, it is UNDEFINED to issue a 3DPRIMITIVE with a primitive topology type not supporting a Cut Index (even if no cut indices are actually present in the index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





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 array of structures, where the size of the structure is 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.

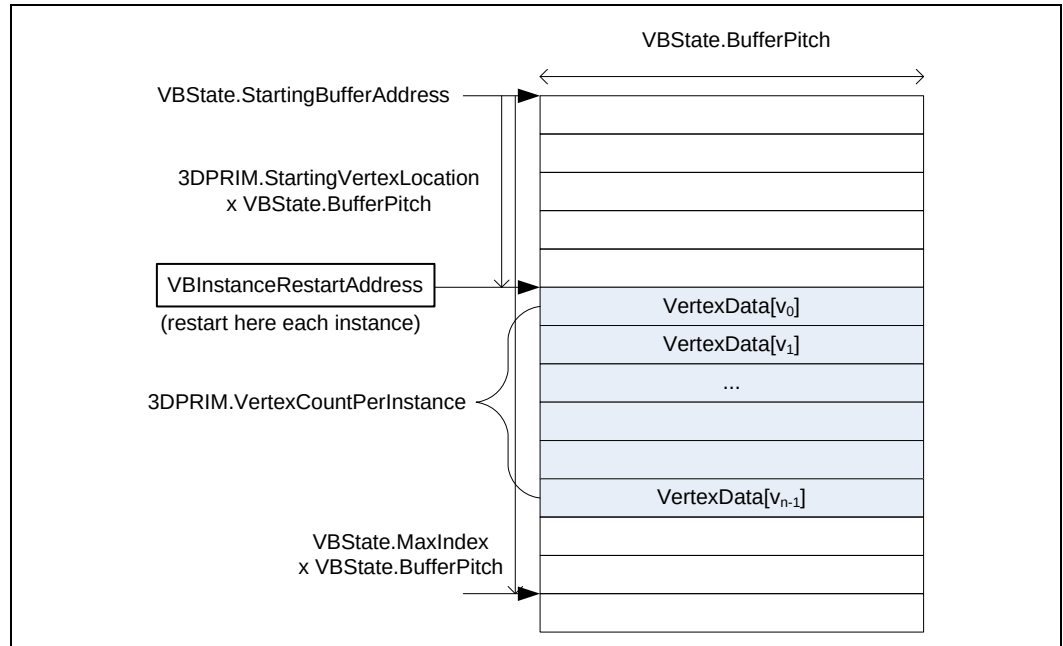
DWord	Bits(#)	Description
0	31:27	Vertex Buffer Index: This field contains an index value which selects the VB state being defined. Format: U5 index Range: [0,16]
	26	Buffer Access Type: This field determines how vertex element data is extracted from this VB. This control applies to all vertex elements 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



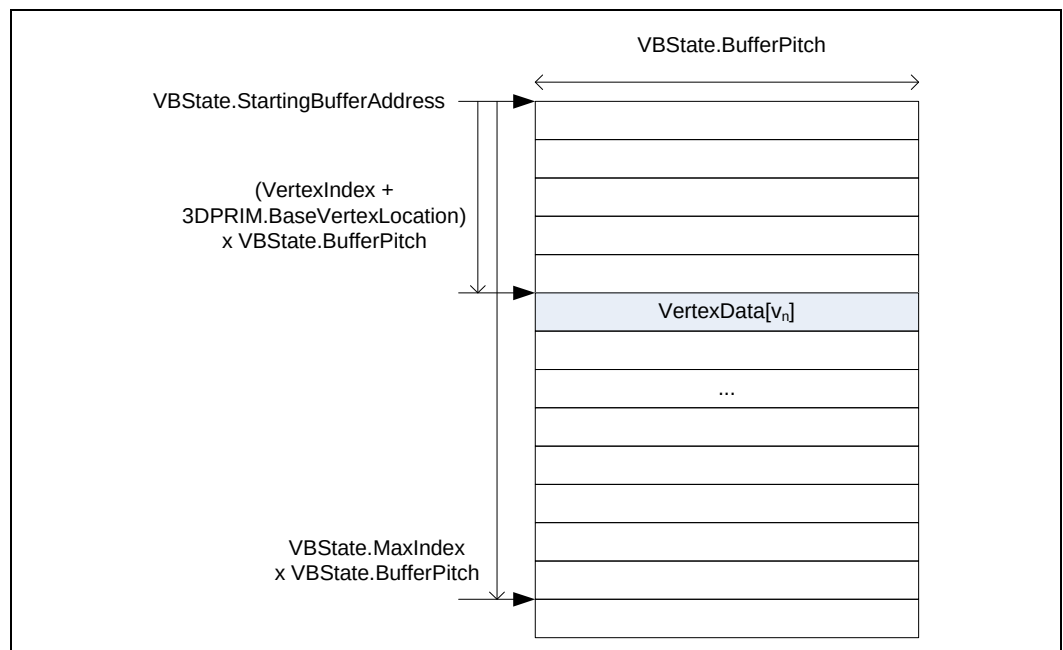
DWord	Bits(#)	Description
1	31:0	Buffer Starting Address: This field contains the byte-aligned Graphics Address of the first element of interest within the VB. Software must program this value with the combination (sum) of the base address of the memory resource and the byte offset from the base address 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 3DPRIMITIVE 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.

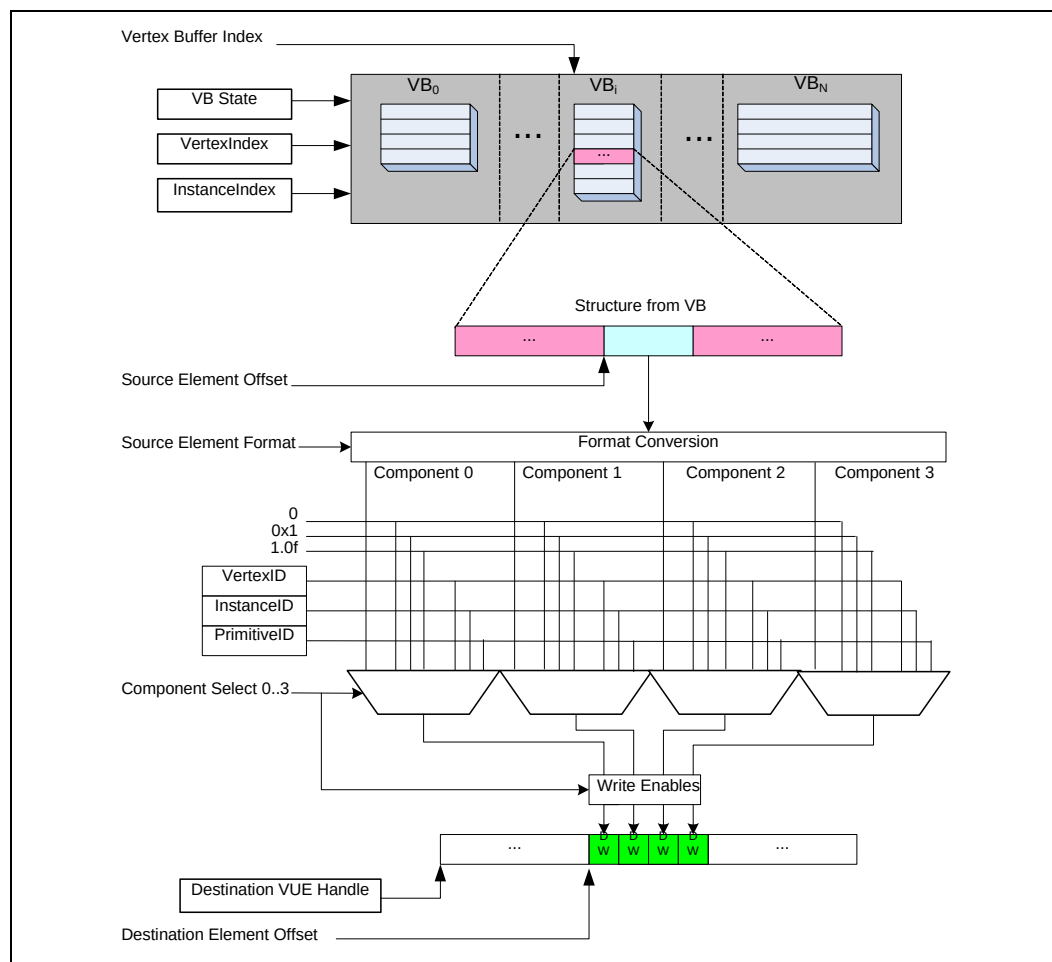
DWord	Bits(#)	Description
0	31:27	Vertex Buffer Index: This field specifies which vertex buffer the element is sourced from. Format: U5 Range: [0,16] (Up to 17 VBs are supported)
	26	Valid: If TRUE, this vertex element is used in vertex assembly If FALSE, this vertex element is not used. Format: Boolean
	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 <i>Sampler</i> chapter. Range: Valid encodings are those marked as “Y” in the “Vertex Buffer” column of the table of Surface Format encodings in the <i>Sampler</i> 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 WindowMasker 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 <i>topology type</i> 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 <u>for</u> 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 in 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 <u>signed</u> 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 suppress processing (shading) in the VS unit.

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



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_SSCELED	FLOAT	R	G	B	-
R16G16B16_USCELED	FLOAT	R	G	B	-
32 bits					
R10G10B10A2_UNORM	FLOAT	R	G	B	A
R10G10B10A2_UINT	UINT	R	G	B	A
R10G10B10X2_USCELED	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_SSCELED	FLOAT	R	G	B	A
R8G8B8A8_USCELED	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_SSCELED	FLOAT	R	G	-	-
R16G16_USCELED	FLOAT	R	G	-	-
R32_FLOAT	FLOAT	R	-	-	-
R32_SINT	SINT	R	-	-	-
R32_UINT	UINT	R	-	-	-
R32_SSCELED	FLOAT	R	-	-	-
R32_USCELED	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_SSCELED	FLOAT	R	G	B	-
R8G8B8_USCELED	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 (1 st GEN4 core instruction) of the kernel program run by threads spawned by this FF unit. It is specified as a 64-byte-granular offset from the General State 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 (SIMDn_{xm} with m = 1) or multiple program flows (SIMDn_{xm} with m > 1). If set, the VS unit will only dispatch 1-vertex thread payloads. See CR0 description in <i>ISA Execution Environment</i>. 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 <i>Exceptions</i> and <i>ISA Execution Environment</i>. 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 <i>Exceptions</i> and <i>ISA Execution Environment</i>. 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 <i>Exceptions</i> and <i>ISA Execution Environment</i>. 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 <u>for the Constant URB entry</u>, 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 <u>for each Vertex URB entry</u> , 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]
4	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]
	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 <i>Debug</i> 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 <i>Debug</i> for further description). Format: <u>Reserved for HW Implementation Use.</u>
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 <i>3D Pipeline</i> 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: <u>Reserved for HW Implementation Use.</u>
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 <i>3D Pipeline</i> for further description). Format = U4 power of two (in excess of 10) Range = [0,11] indicating [1K Bytes, 2M Bytes]
R0.2	31:0	Reserved : delivered as zeros (reserved for message header fields)
R0.1	31:25	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 <i>Generic FF Unit</i> for further description). Format = <u>Reserved for HW Implementation Use</u>.
	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 <i>Generic FF Unit</i> 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 <i>Generic FF Unit</i> for further description). Format = <u>Reserved for HW Implementation Use</u>.
	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 <i>Generic FF Unit</i> 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 (1 st GEN4 core instruction) of the kernel program run by threads spawned by this FF unit. It is specified as a 64-byte-granular offset from the General State 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 (SIMDn _{xm} with m = 1) or multiple program flows (SIMDn _{xm} with m > 1).													
		<table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td>Reserved</td><td></td><td>All</td></tr><tr><td>1h</td><td>Enable</td><td>Single Program Flow enabled</td><td>All</td></tr></table>	Value	Name	Description	Project	0h	Reserved		All	1h	Enable	Single Program Flow enabled	All	
	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. <table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td></td><td>Use IEEE-754 Rules</td><td>All</td></tr><tr><td>1h</td><td></td><td>Use alternate rules</td><td>All</td></tr></table>	Value	Name	Description	Project	0h		Use IEEE-754 Rules	All	1h		Use alternate rules	All	
	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 <i>Exceptions</i> and <i>ISA Execution Environment</i> .													
	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 <i>Exceptions</i> and <i>ISA Execution Environment</i> .													
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 <i>Exceptions</i> and <i>ISA Execution Environment</i> .														
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 <u>for the Constant URB entry</u> , 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 <u>for each Vertex URB entry</u> , 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 <u>dual-thread mode</u>, the Number of URB Entries field must contain an <u>even</u> 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 <i>Graphics Processing Engine</i> (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 All Graphics Processing Engine for Command Ordering Rules and a description of URB_FENCE.	
	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 <u>dual-thread mode</u> , the Number of URB Entries field must contain an <u>even</u> 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 All Graphics Processing Engine for Command Ordering Rules and a description of URB_FENCE.	
	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.																												
		<table> <tr> <th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr> <tr> <td>0h</td><td></td><td>no samplers used</td><td>All</td></tr> <tr> <td>1h</td><td></td><td>between 1 and 4 samplers used</td><td>All</td></tr> <tr> <td>2h</td><td></td><td>between 5 and 8 samplers used</td><td>All</td></tr> <tr> <td>3h</td><td></td><td>between 9 and 12 samplers used</td><td>All</td></tr> <tr> <td>4h</td><td></td><td>between 13 and 16 samplers used</td><td>All</td></tr> <tr> <td>5h-7h</td><td></td><td>Reserved</td><td>All</td></tr> </table>		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
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 <u>are not</u> reordered, and TRISTRIP_REV when the vertices <u>are</u> 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 VPIIndex Project: All Format: U4 index value (# of viewports -1) This field specifies the maximum valid VPIIndex value, corresponding to the number of active viewports. If the source of the VPIIndex exceeds this maximum value, a VPIIndex value of 0 is passed down the pipeline. Note that this clamping does not affect a VPIIndex 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 <i>Debug</i> 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 <i>Debug</i> for further description). Format: <u>Reserved for HW Implementation Use</u> .
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: <u>Reserved for Implementation Use</u>



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 <i>Generic Pipeline Stage</i> 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 <i>3D Pipeline</i>
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: <u>Reserved for Implementation Use</u>
	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 trisrips 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 trisrip 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 trisrip.)

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 Position (NDC or screen space), RTAIndex, VPIndex, PointWidth) and control information (PrimType, PrimStart, PrimEnd) while the remainder of the vertex data remains in the VUE in the URB.

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)	$\text{clip.x} < -\text{clip.w}$	$\text{ndc.x} < -1$
XMAX (NDC Right)	$\text{clip.w} < \text{clip.x}$	$\text{ndc.x} > 1$
YMIN (NDC Bottom)	$\text{clip.y} < -\text{clip.w}$	$\text{ndc.y} < -1$
YMAX (NDC top)	$\text{clip.w} < \text{clip.y}$	$\text{ndc.y} > 1$
ZMIN (NDC Near)	$\text{clip.z} < -\text{clip.w}$	$\text{ndc.z} < -1.0$
ZMAX (NDC Far)	$\text{clip.w} < \text{clip.z}$	$\text{ndc.z} > 1.0$

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 ($-\text{clip.w}$) 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 $-\text{clip.w}$. 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 $\text{clip.w}=0$ clip plane – a negative ndc.rhw value indicates the point is outside of the $\text{clip.w}=0$ plane. (See sections below for related errata in DevBW and DevCL devices)

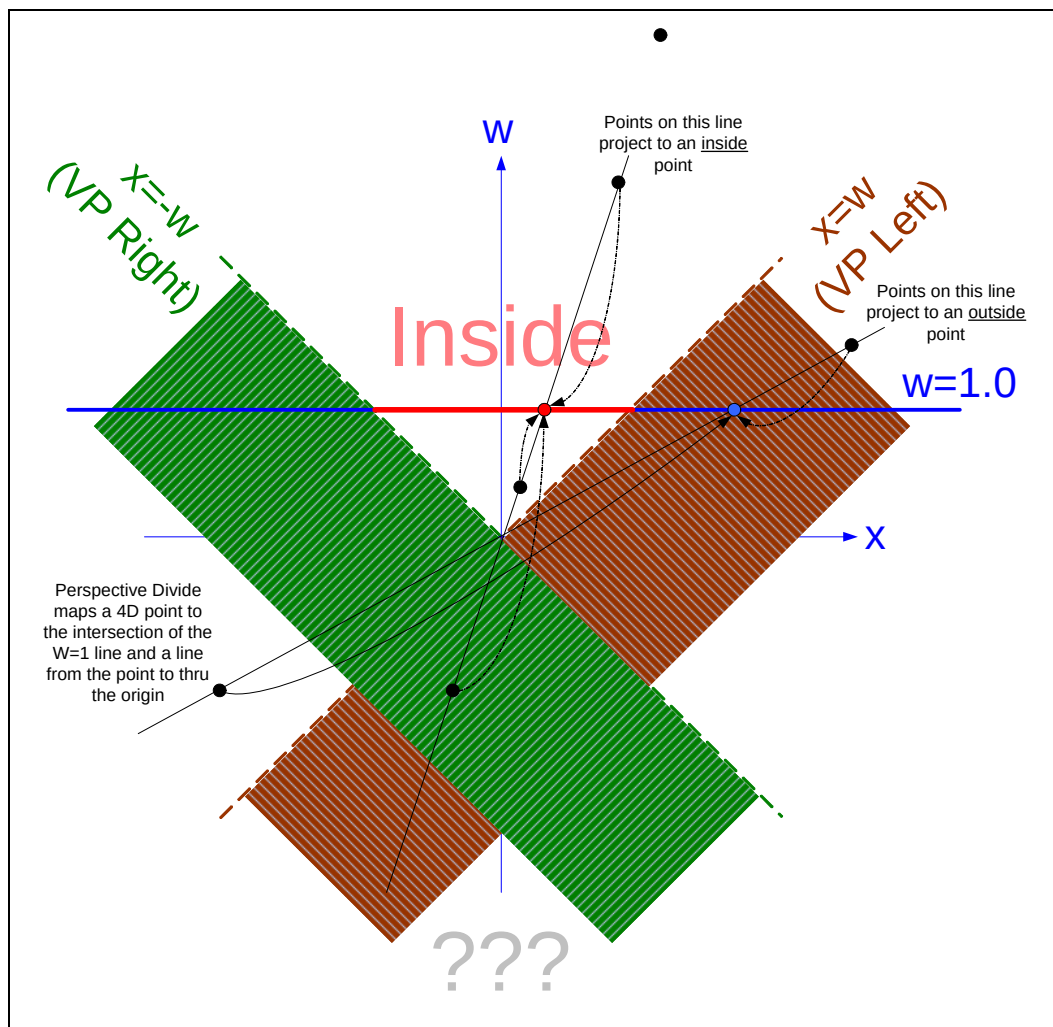
The assumption made in the Clip stage is that only the $w>0$ portion of clip space is considered visible. The VertexClipTest function tests each incoming $1/w$ value and, if



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-perspective-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:

$$(P.xyzw \text{ dot } UCP[n].xyzw) \geq 0, \quad 0 \leq n \leq 5$$

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 ($w < 0$) { $\text{npc.xyzw} = 0$ $\text{UC7} = 1$ }	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 ($w < 0$) { $\text{npc.xyz} = 0$ }	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 $\text{BAD} \leftarrow \text{UC7}$ 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 $\text{BAD} \leftarrow \text{UC7}$ 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 $\text{clip.w} = 0$ 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 occurring 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 UC7ClipTestEnableMask[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 $\text{rhw}=\text{NaN}$ 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 $\text{rhw} = \text{NaN}$, 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 primetype, a TRISTRIP primetype 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 these 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 trisrips 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-functionally as no culling will occur. One way to avoid some of this performance penalty would be for software 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.

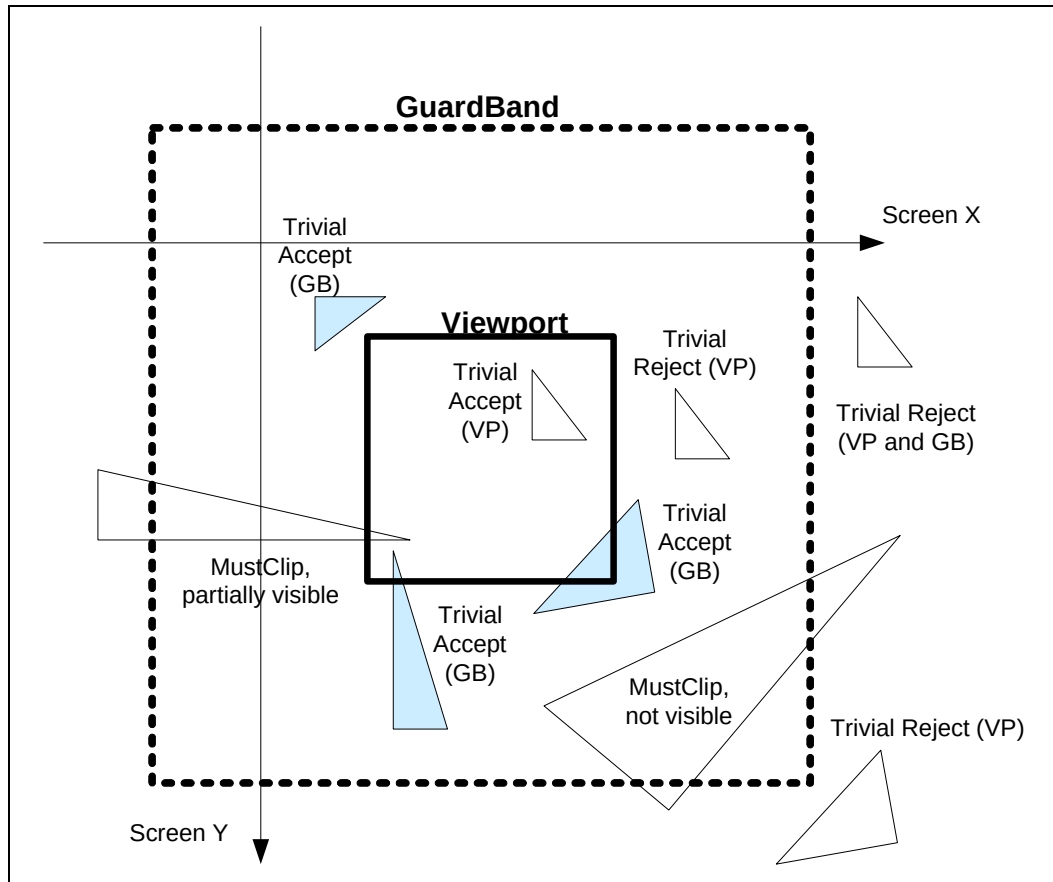
However, we must ensure that the 2D extent of these objects do not exceed the limitations of the renderer's coordinate space (see Vertex X,Y Clamping and Quantization in the SF section). Therefore we define a 2D *guardband* region corresponding to (though likely somewhat smaller than) the maximum 2D 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.

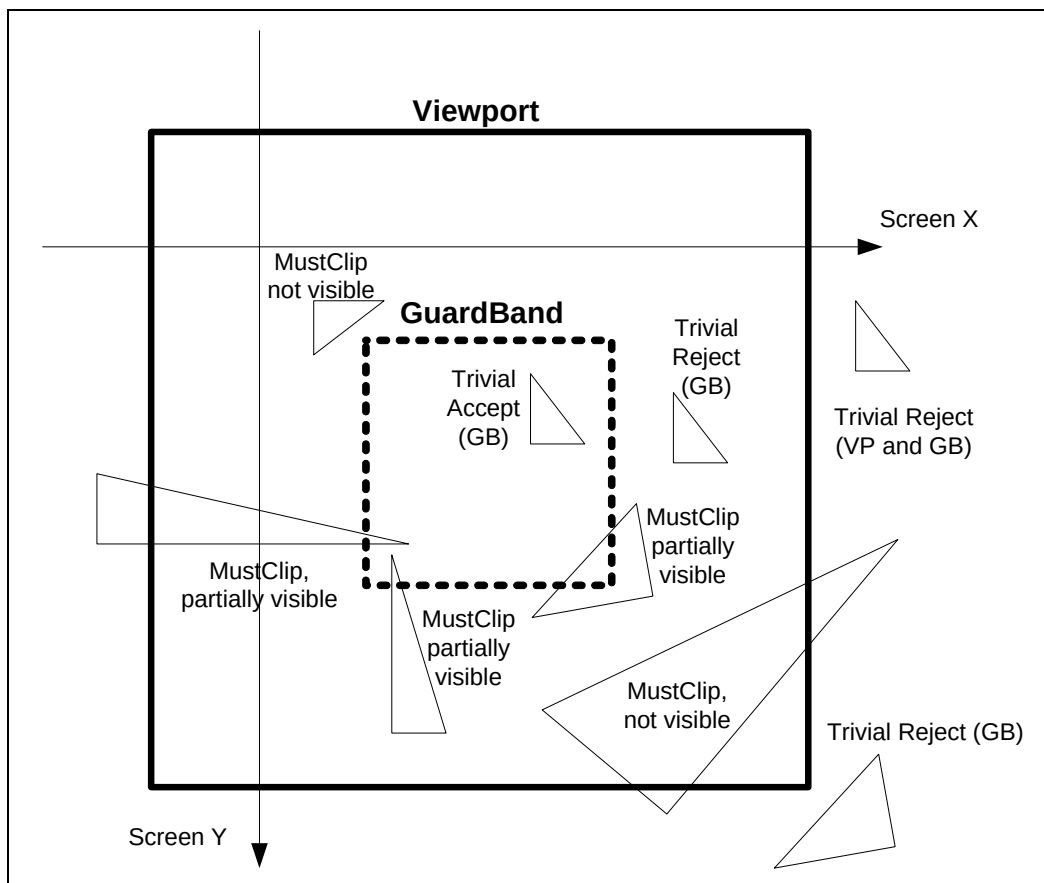


Figure 6-2. Normal Guardband Operation



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 be used to force explicit clipping in order to ensure legal coordinates are passed out of the CLIP stage. Therefore the CLIP unit supports a guardband that can be larger or smaller than the viewport (in any particular direction). The following diagram illustrates a case with a very large viewport, extending well beyond the guardband. Note that the only trivial accept case is where objects are completely within the guardband.

Figure 6-3. Very Large Viewport Case



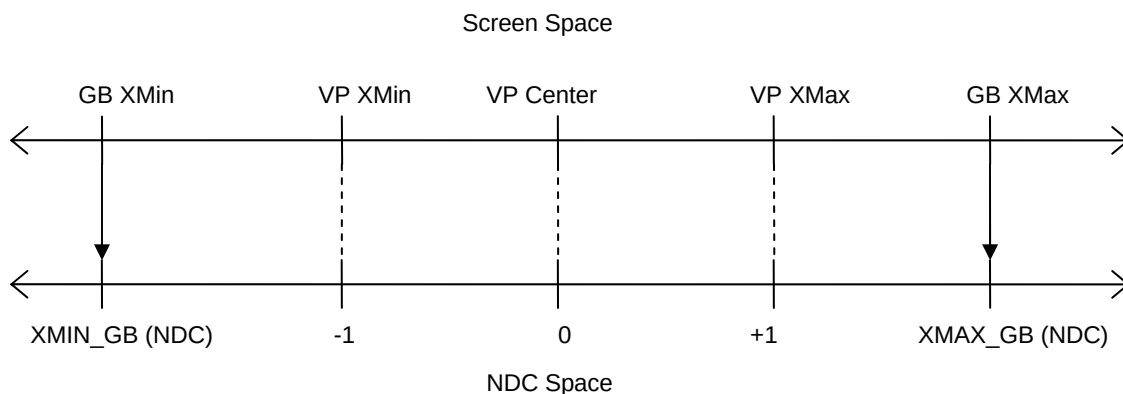
Programming Restriction: Varying ViewportIndex within Strip-based Primitives

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

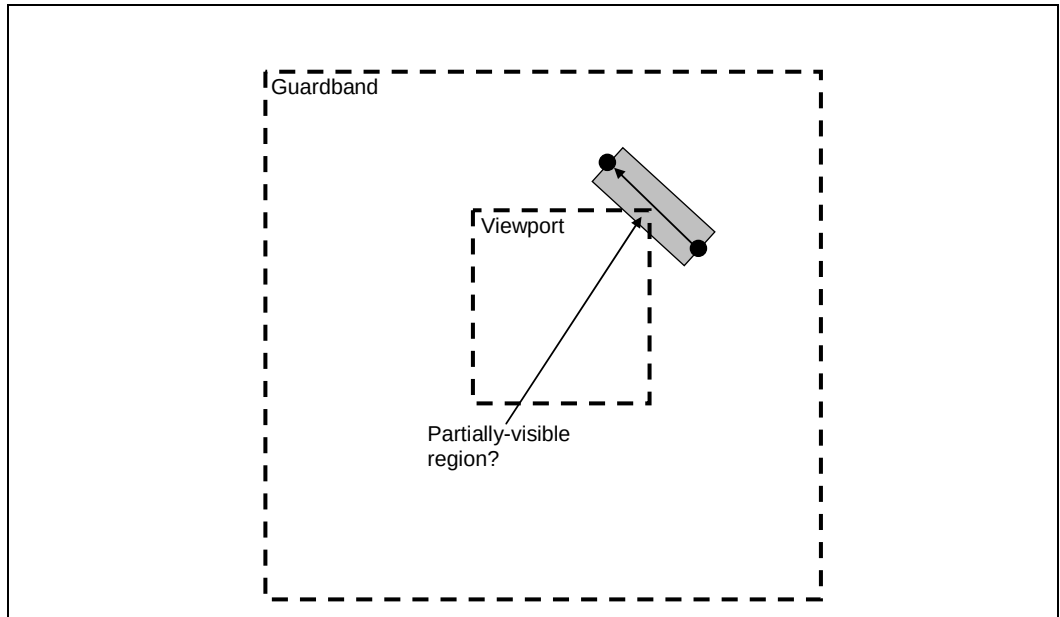
In the normal processing of non-wide, line-based primitives (linestrip/linelist/etc.), the footprint of each line is constrained to the 2D convex hull. I.e., the rendering of these lines will not produce pixels off of the line. Therefore the normal operation of the CLIP unit functions will support the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

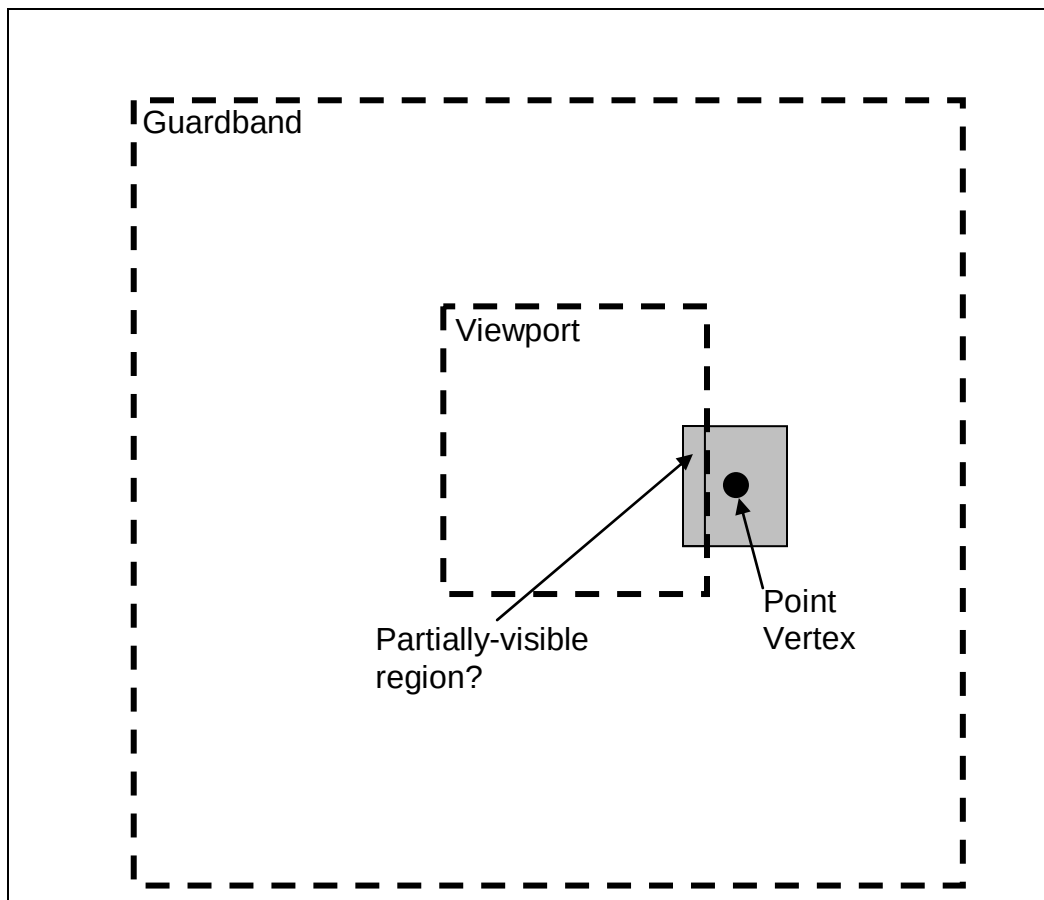


In general, OpenGL dictates that the partially-visible region must not be rendered. In this case the line must be clipped-out against the VPXY (not TA against the GB). To accomplish this, SW could disable the GB when drawing wide lines.

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 (1 st GEN4 core instruction) of the kernel program run by threads spawned by this FF unit. It is specified as a 64-byte-granular offset from the General State 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 (SIMDn _{xm} with m = 1) or multiple program flows (SIMDn _{xm} with m > 1). <table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td>Reserved</td><td></td><td>All</td></tr><tr><td>1h</td><td>Enable</td><td>Single Program Flow enabled</td><td>All</td></tr></table>			Value	Name	Description	Project	0h	Reserved		All	1h	Enable	Single Program Flow enabled	All
	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 <table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td></td><td>Normal Priority</td><td>All</td></tr></table>	Value	Name	Description	Project	0h		Normal Priority	All						
	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. <table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td></td><td>Use IEEE-754 Rules</td><td>All</td></tr><tr><td>1h</td><td></td><td>Use alternate rules</td><td>All</td></tr></table>	Value	Name	Description	Project	0h		Use IEEE-754 Rules	All	1h		Use alternate rules	All		
	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 <i>Exceptions and ISA Execution Environment</i> .													
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 <i>Exceptions and ISA Execution Environment</i> .													
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 <i>Exceptions</i> and <i>ISA Execution Environment</i> .
	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 <u>for the Constant URB entry</u> , 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 <u>for each Vertex URB entry</u> , 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 When running in <u>dual-thread mode</u>, the Number of URB Entries field must contain an <u>even</u> number. Each thread will be allocated one half the total number of entries. Project 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 <i>Graphics Processing Engine</i> (Command Ordering Rules) Project 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 Changing this value requires a subsequent URB_FENCE command. See Graphics Processing Engine for Command Ordering Rules and a description of URB_FENCE. Project 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 When running in <u>dual-thread mode</u> , the Number of URB Entries field must contain an <u>even</u> number. Each thread will be allocated one half the total number of entries. If ENABLED, the GS stage must be allocated at least one URB entry Changing this value requires a subsequent URB_FENCE command. See Graphics Processing Engine for Command Ordering Rules and a description of URB_FENCE. Project All 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. <table><thead><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr></thead><tbody><tr><td>0h</td><td>VPOS_NDCSP ACE</td><td>Vertex Position is in NDC space</td><td>All</td></tr><tr><td>1h</td><td>VPOS_SCREEN SPACE</td><td>Vertex Position is in Screen space</td><td>All</td></tr></tbody></table>			Value	Name	Description	Project	0h	VPOS_NDCSP ACE	Vertex Position is in NDC space	All	1h	VPOS_SCREEN SPACE	Vertex Position is in Screen space
Value	Name	Description	Project												
0h	VPOS_NDCSP ACE	Vertex Position is in NDC space	All												
1h	VPOS_SCREEN SPACE	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. <table><thead><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr></thead><tbody><tr><td>0h</td><td>CLIPMODE_NORMAL</td><td>TrivialAccept objects are passed down the pipeline, MustClip objects are passed to CLIP threads, TrivialReject and BAD objects are discarded</td><td>All</td></tr><tr><td>1h</td><td>CLIPMODE_ALL</td><td>All objects (including BAD objects & TrivReject) are passed to CLIP threads, regardless of classification</td><td>All</td></tr><tr><td>2h</td><td>CLIPMODE_CLIP_NON_REJECTED</td><td>TrivialAccept and MustClip objects are passed to CLIP threads, TrivReject and BAD objects are discarded</td><td>All</td></tr><tr><td>3h</td><td>CLIPMODE_REJECT_ALL</td><td>All objects are discarded</td><td>All</td></tr><tr><td>4h</td><td>CLIPMODE_ACCEPT_ALL</td><td>All objects (except BAD objects) are trivially accepted. This effectively disables the clip-test/clip-determination function.</td><td>All</td></tr><tr><td>5h</td><td>CLIPMODE_NORMAL_FFCLIP</td><td>Reserved</td><td>All</td></tr><tr><td>6h-7h</td><td></td><td>Reserved</td><td>All</td></tr><tr><td></td><td>Errata</td><td>Description</td><td>Project</td></tr><tr><td>#</td><td>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.</td><td>DevBW, DevCL-A, DevCL-B</td></tr></tbody></table>			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
	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 through 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 <u>left</u> 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 <u>left</u> 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 <u>right</u> 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 <u>right</u> 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 <u>bottom</u> 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 <u>top</u> 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 <u>top</u> 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 <u>bottom</u> 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
—     }
—     #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 against the corresponding GB boundary, so computing TR_GB is unnecessary.

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

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



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

```
—      //
—      // ClipDetermination
—      //
—      //
—      // Compute objANDCode and objORCode
—      //
—      switch (object type) {
—      case POINT:
—      {
—          objANDCode[...] = v0.outCode[...]
—          objORCode[...] = v0.outCode[...]
—      } break
—      case LINE:
—      {
—          objANDCode[...] = v0.outCode[...] & v1.outCode[...]
—          objORCode[...] = v0.outCode[...] | v1.outCode[...]
—      } break
—      case TRIANGLE:
—      {
—          objANDCode[...] = v0.outCode[...] & v1.outCode[...] &
v2.outCode[...]
—          objORCode[...] = v0.outCode[...] | v1.outCode[...] |
v2.outCode[...]
—      } break
—      //
—      // Determine TR/TA against interesting boundary subsets
—      //
—      TR_VPXY = (objANDCode[VP_L] | objANDCode[VP_R] |
objANDCode[VP_T] | objANDCode[VP_B])
—      TR_GB = (objANDCode[GB_L] | objANDCode[GB_R] |
objANDCode[GB_T] | objANDCode[GB_B])
—      TA_GB = !(objORCode[GB_L] | objORCode[GB_R] |
objORCode[GB_T] | objORCode[GB_B])
—      TA_VPZ = !(objORCode[VP_N] | objORCode[VP_Z])
—      TR_VPZ = (objANDCode[VP_N] | objANDCode[VP_Z])
—      TA_UC = !(objORCode[UC0] | objORCode[UC1] | ... |
objORCode[UC7])
—      TR_UC = (objANDCode[UC0] | objANDCode[UC1] | ... |
objANDCode[UC7])
—      BAD = objORCode[BAD]
—      //
—      // 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 and all other (TA, MC) objects are passed to CLIP threads. Usage of this mode assumes that the CLIP unit fixed functions (VertexClipTest, ClipDetermination) are sufficient at least in respect to determining trivial reject.

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 <u>do not</u> 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 <u>do not</u> 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 <i>Debug</i> 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 <i>Debug</i> for further description). Format: <u>Reserved for HW Implementation Use</u> .
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: <u>Reserved for Implementation Use</u>
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 <u>not</u> 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 <u>not</u> 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 <i>3D Pipeline</i>)
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: <u>Reserved for Implementation Use</u>
	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 <i>3D Pipeline</i> 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 position 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

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

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

Table 7-3. 3D Object Types

<i>objectType</i>	generated by <i>primType</i>	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 <u>object</u> vertices. Data in the array consists of X, Y, Z, invW and a pointer to the other vertex attributes. These additional attributes are not used by directly by the 3D fixed functions but are made available to the SF thread. The number of valid vertices depends on the object type. 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.

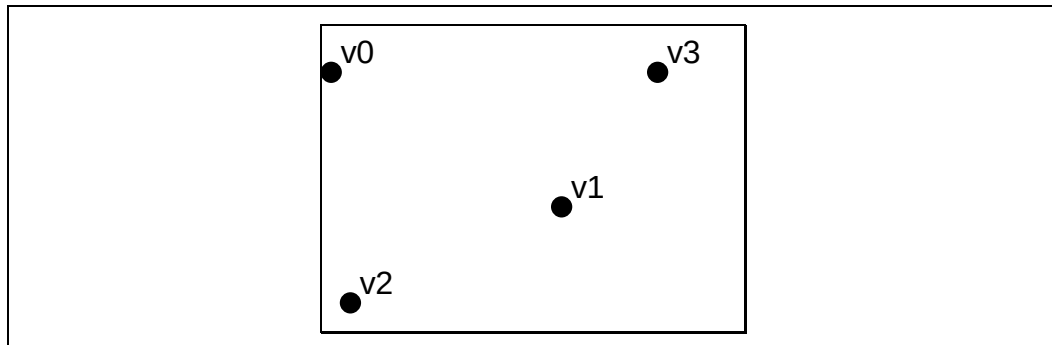
Table 7-5. VPIndex/RTAIndex Selection

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()
    }
}

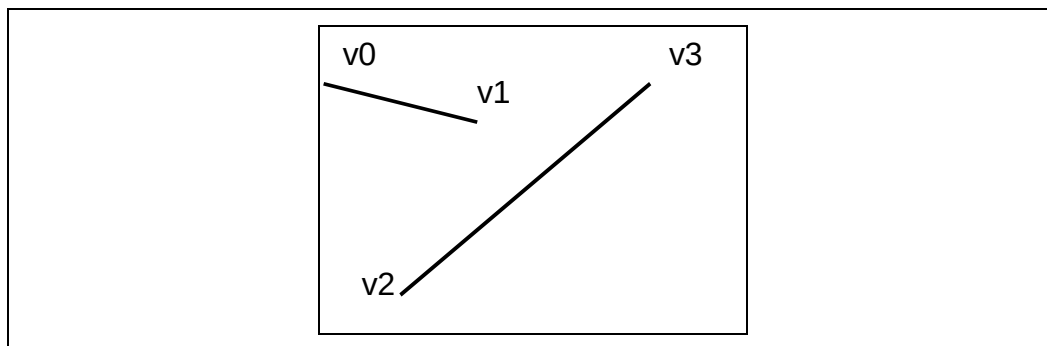
```



7.2.2 Line List Decomposition

The 3DPRIM_LINELIST primitive specifies a list of independent lines.

Figure 7-2. 3DPRIM_LINELIST Primitive



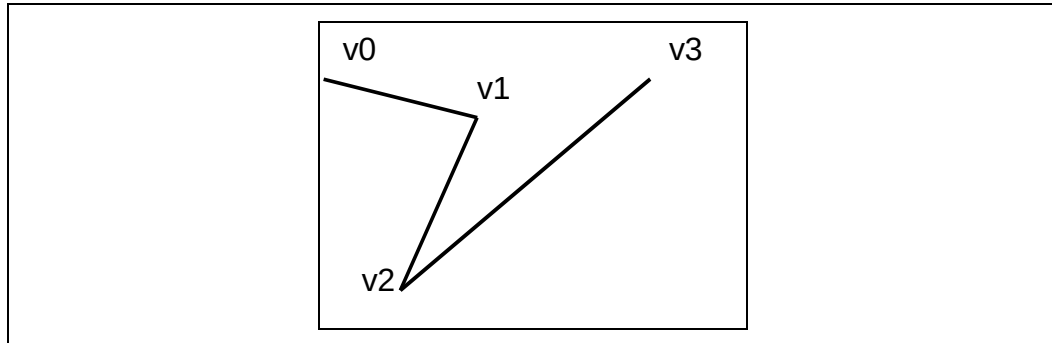
The decomposition process divides the list into a series of basic 3DOBJ_LINE objects that are then passed individually and in order to the Object Setup stage. The lines are generated with the following object vertex order: v0, v1; v2, v3; and so on. The *provokingVertex* of each object is taken from 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
    }
}

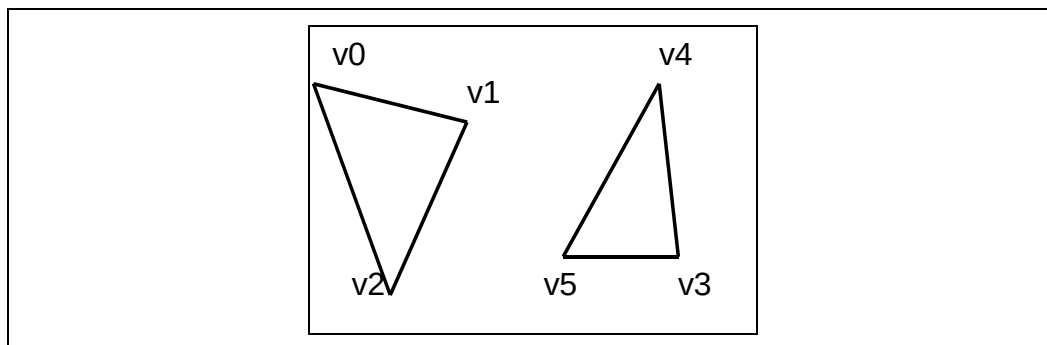
```



7.2.4 Triangle List Decomposition

The 3DPRIM_TRILIST primitive specifies a list of independent triangles.

Figure 7-4. 3DPRIM_TRILIST Primitive



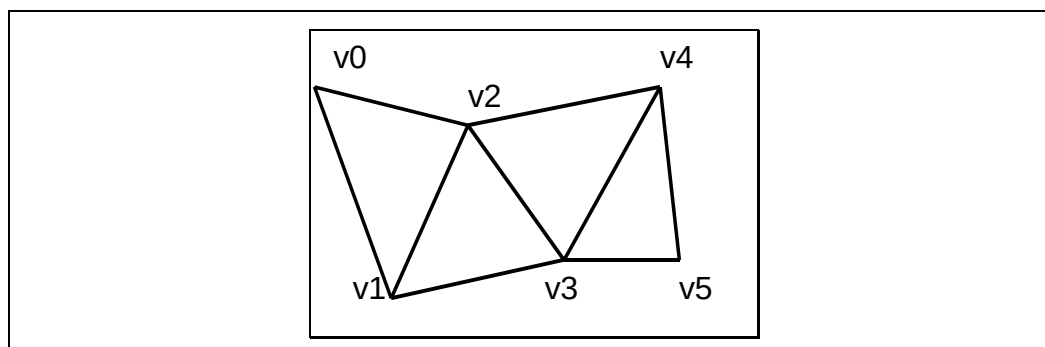
The decomposition process divides the list into a series of basic 3DOBJ_TRIANGLE objects that are then passed individually and in order to the Object Setup stage. The triangles are generated with the following object vertex order: v0,v1,v2; v3,v4,v5; and so on. The *provokingVertex* of each object is 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 tris strip primitive commands (e.g., due to ring or batch buffer size restrictions), two tris strip 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.

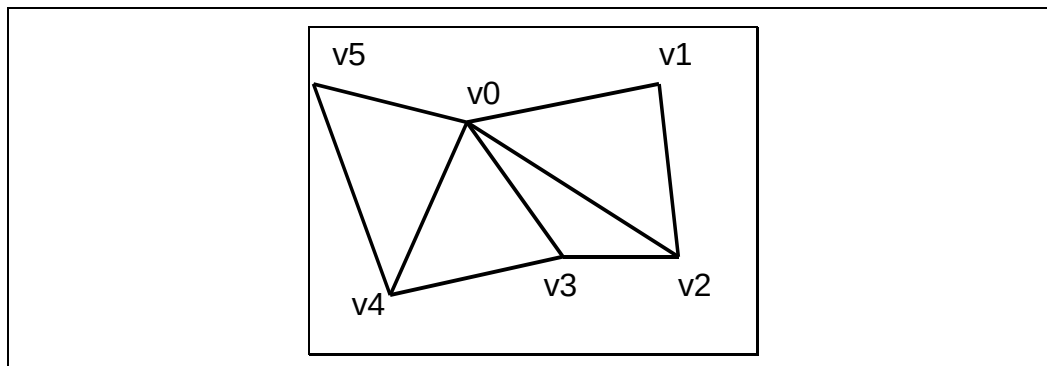
```
TriangleStripDecomposition() {
    objectType = 3DOBJ_TRIANGLE
    nV = 3
    provokingVtx = Triangle List/Strip Provoking Vertex Select
    if (primType == 3DPRIM_TRISTRIP)
        invertOrientation = FALSE
    else // primType == 3DPRIM_TRISTRIP_REVERSE
        invertOrientation = TRUE
    polyStippleEnable = 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()
    }
}
```

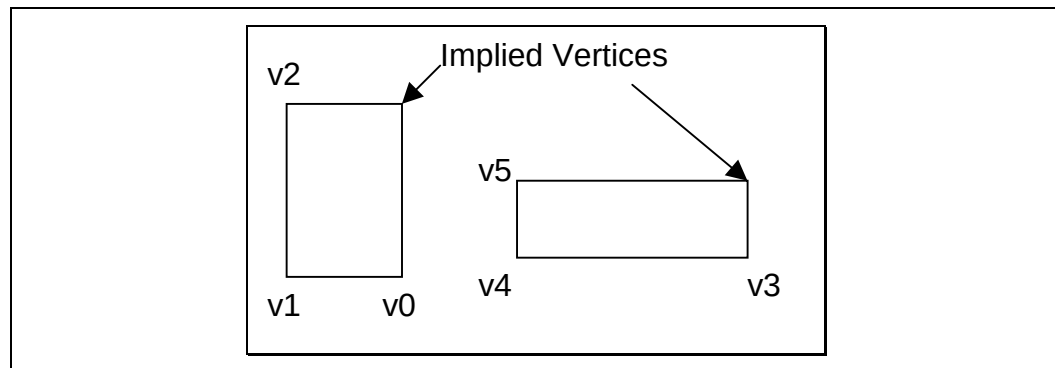
7.2.7 Polygon Decomposition

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.

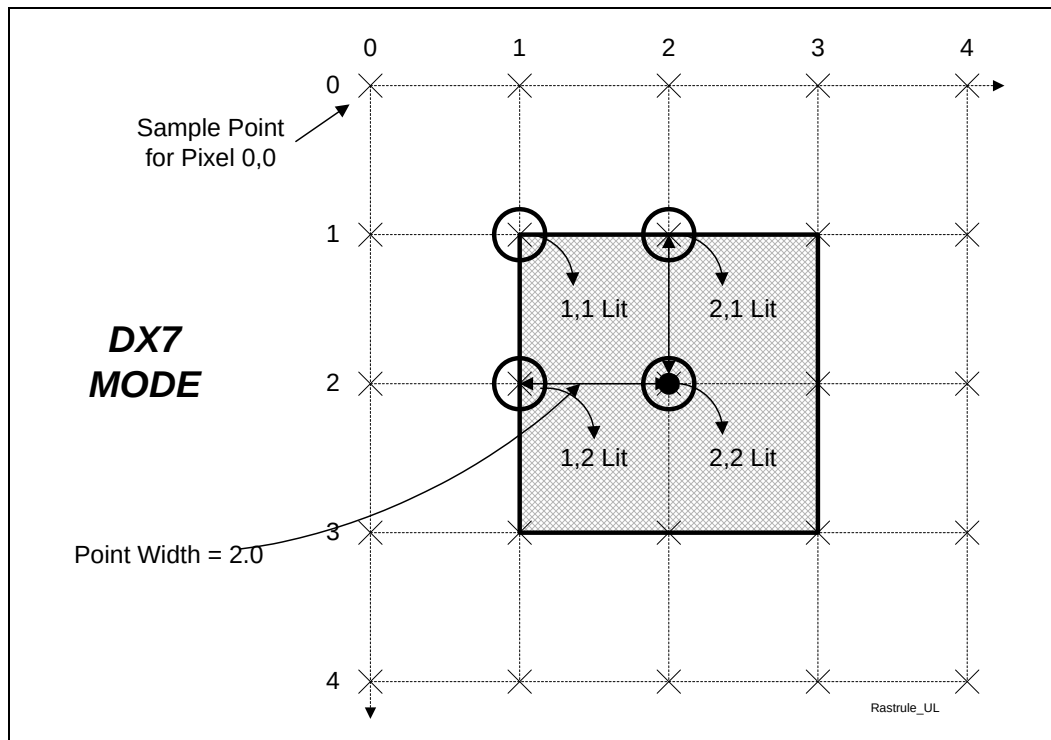
7.3.3 Destination Origin Bias

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



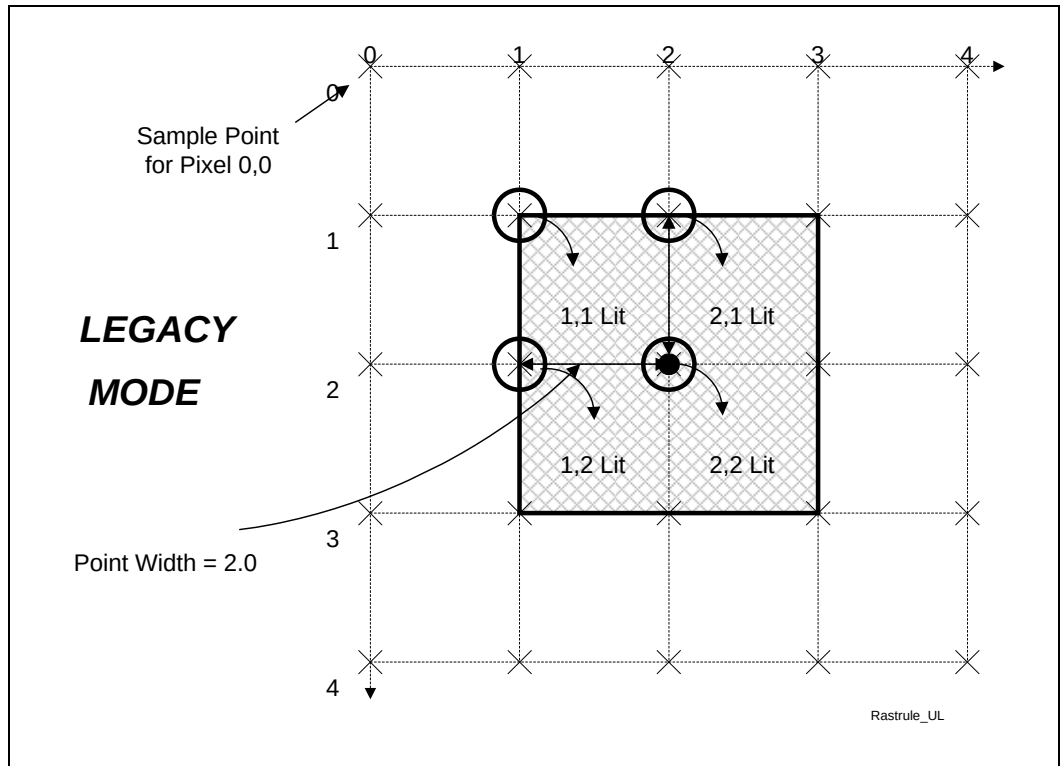
7.3.4 Point Rasterization Rule Adjustment

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.

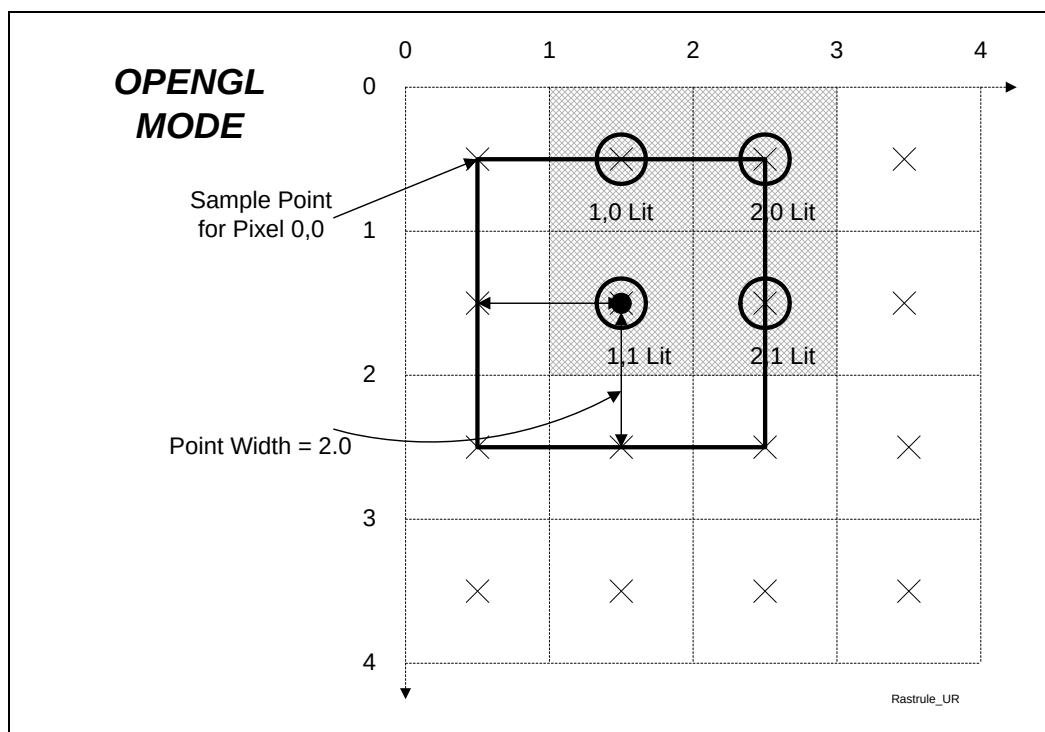


Figure 7-9. RASTRULE_UPPER_LEFT



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

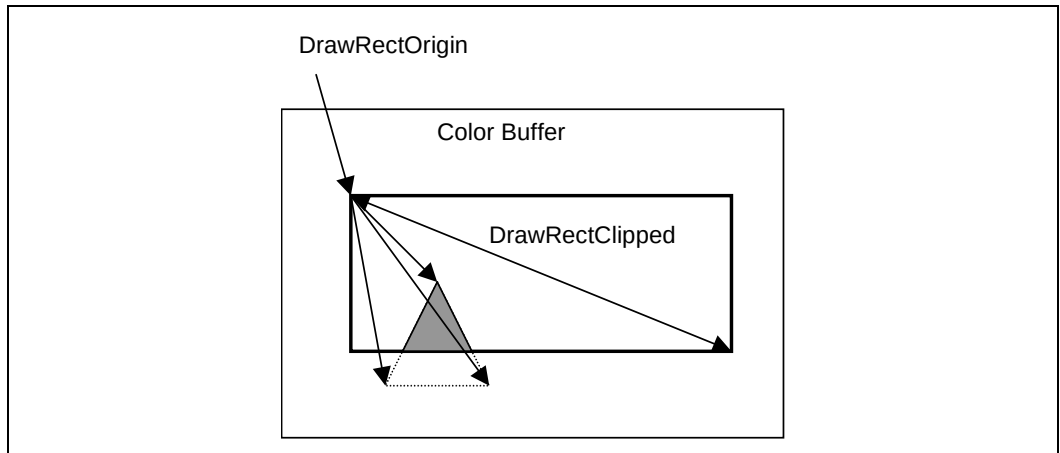
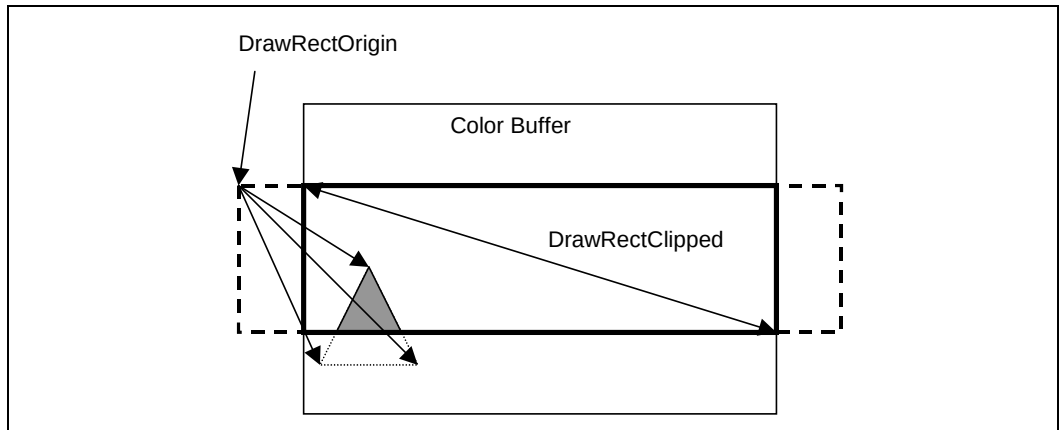


Figure 7-12. Partially-offscreen Draw Rectangle





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 <i>less than</i> Ymin will be clipped out. Programming Notes This value must be <i>less than or equal to</i> Clipped Drawing Rectangle Y Max . If Ymin==Ymax, the clipped drawing rectangle is 1 pixel wide in the Y direction.



3DSTATE_DRAWING_RECTANGLE		
2	15:0	Clipped Drawing Rectangle X Min Project: All Format: U16 in Pixels from Color Buffer origin FormatDesc (upper left corner) 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 <i>less than</i> Xmin will be clipped out. Programming Notes Project This value must be <i>less than or equal to</i> Clipped Drawing Rectangle X Max. If Xmin==Xmax, the clipped drawing rectangle is 1 pixel wide in the X direction. All
	31:16	Clipped Drawing Rectangle Y Max Project: All Format: U16 in Pixels from Color Buffer origin FormatDesc (upper left corner) 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 <i>greater than</i> Ymax will be clipped out. Programming Notes This value must be <i>greater than or equal to</i> 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 FormatDesc (upper left corner) 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 <i>greater than</i> Xmax will be clipped out. Programming Notes Project This value must be <i>greater than or equal to</i> Clipped Drawing Rectangle X Min. If Xmin==Xmax, the clipped drawing rectangle is 1 pixel wide in the X direction. All

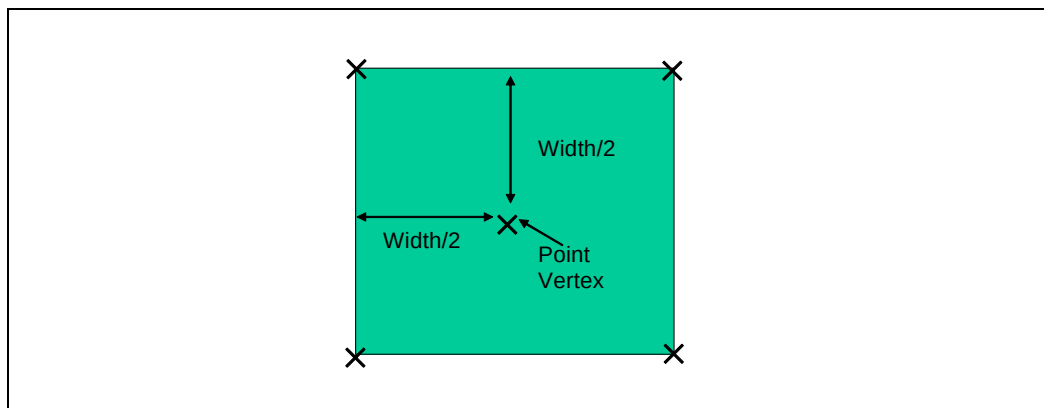
3DSTATE_DRAWING_RECTANGLE		
3	31:16	Drawing Rectangle Origin Y Project: All Format: S15 in Pixels from Color Buffer origin FormatDesc (upper left corner). Range [-8192,8191] (Bits 31:30 should be a sign extension) Specifies Y origin of Drawing Rectangle (in whole pixels) relative to origin of the Color 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 FormatDesc (upper left corner). 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



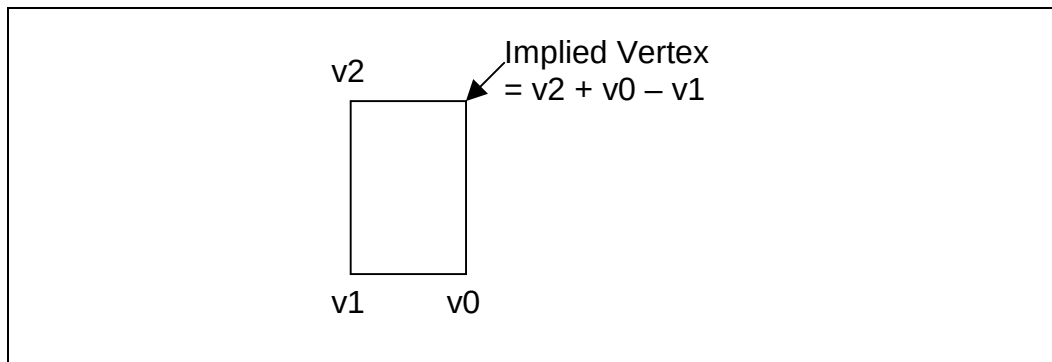
Z and W vertex attributes are copied from the single point center vertex to each of the four corner vertices.



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 applied.

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

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

<i>objType</i>	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.

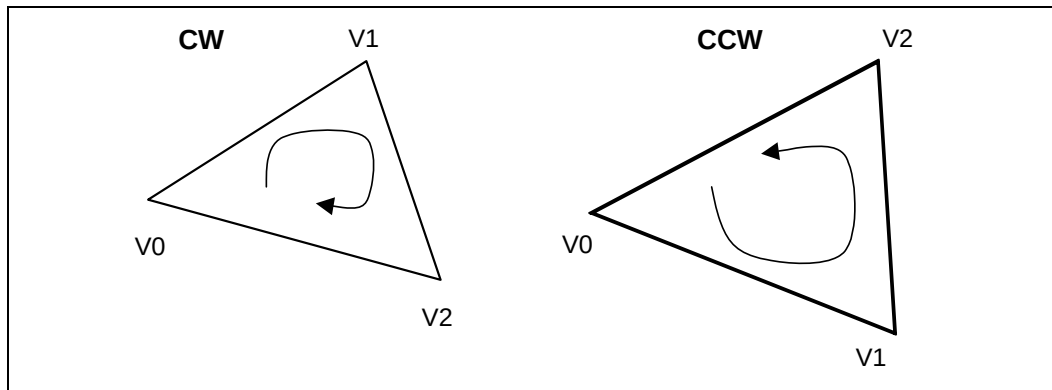
This operation is typically called “back face culling”, though front facing objects (or all 3DOBJ_TRIANGLE objects) can be selected to be discarded as well. Face culling is typically used to eliminate triangles facing away from the viewer, thus reducing rendering time.

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 from 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

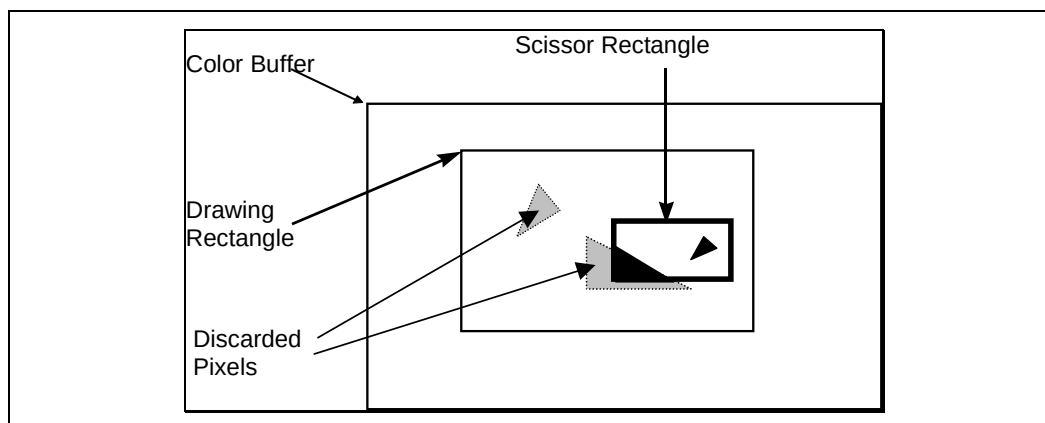
<i>CullMode</i>	<i>Definition</i>
CULLMODE_NONE	The face culling operation is disabled
CULLMODE_FRONT	Triangles with “front facing” orientation are discarded
CULLMODE_BACK	Triangles with “back facing” orientation are discarded
CULLMODE_BOTH	All triangles are discarded

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 $x_{min} > x_{max}$ or $y_{min} > y_{max}$ 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).

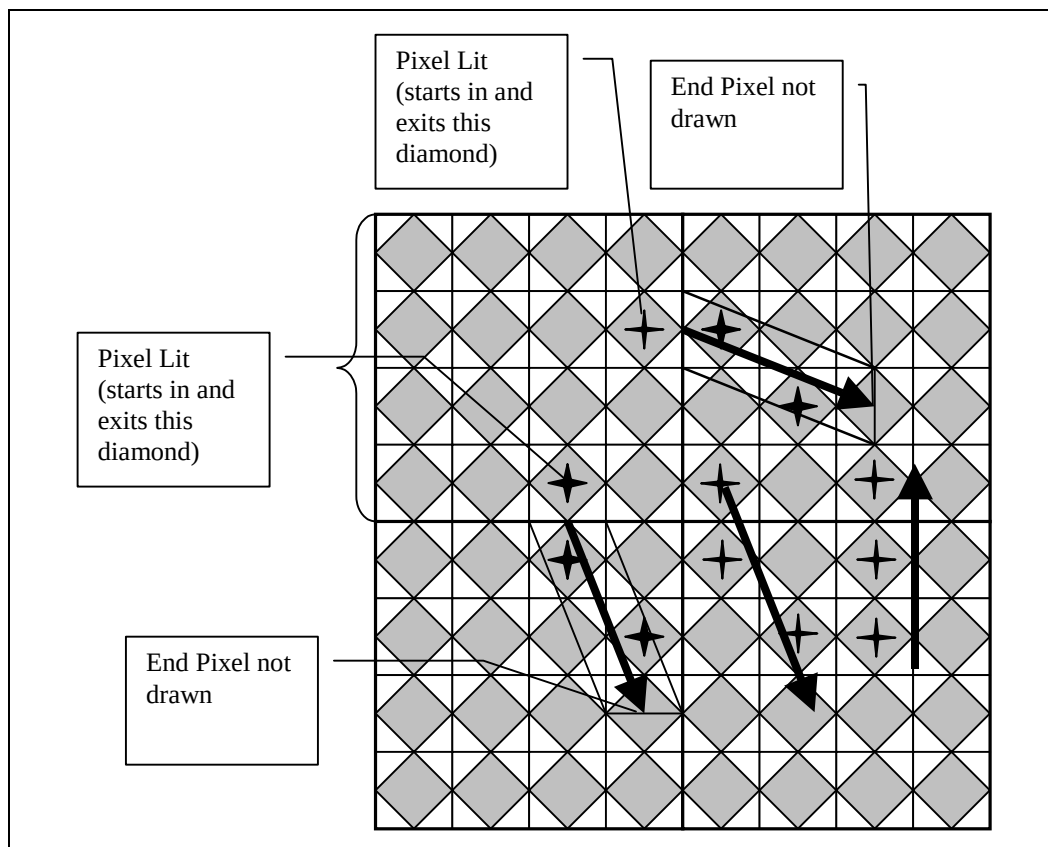
When the **Line Width** is set to zero, the device will use special rules to rasterize zero-width (“cosmetic”) lines. The **Anti-Aliasing Enable** state variable is ignored when **Line Width** is zero.



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.



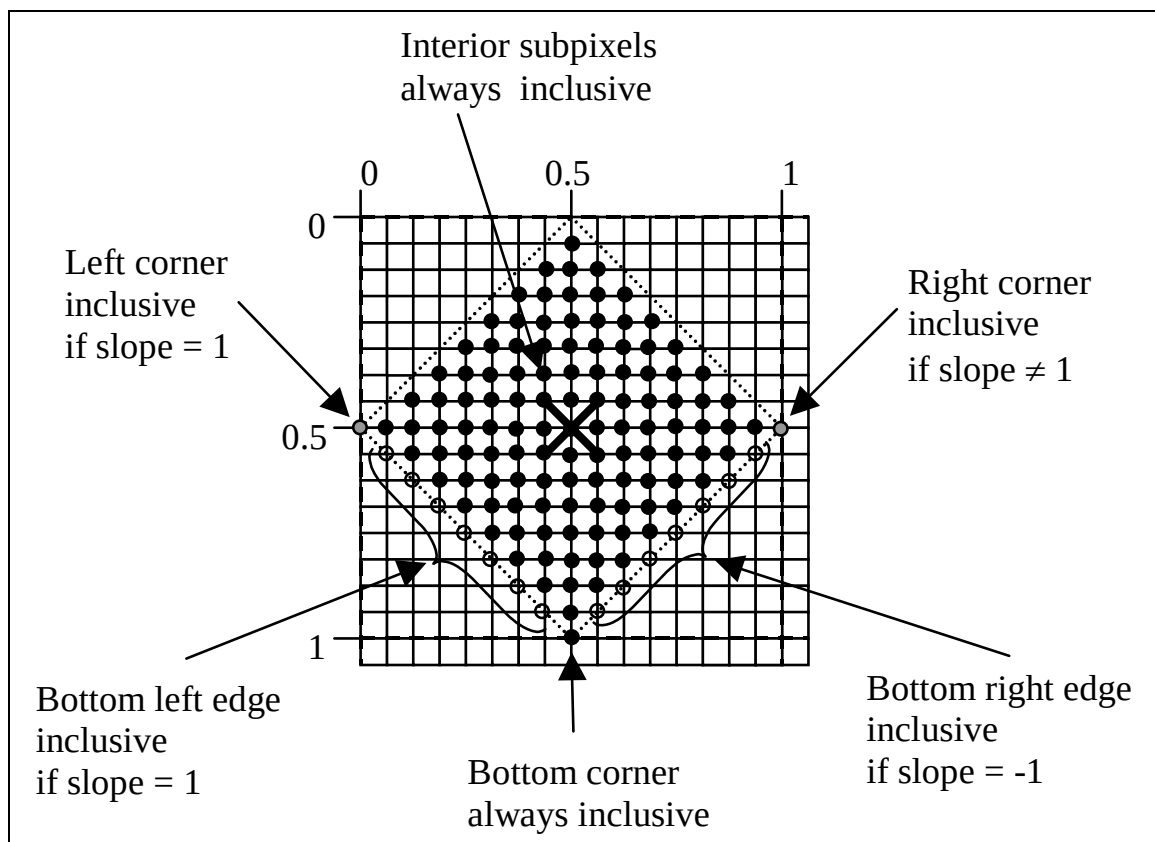
The following subsections describe the diamond exit rules in more detail.

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 $\frac{1}{2}$.

The subpixels falling on the edges of the diamond (Manhattan distance = $\frac{1}{2}$) are exclusive, with the following exceptions:

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



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

3. **The subpixels along the bottom left edge are inclusive only if the line slope = 1.** This is to correctly handle the case where a slope=1 line falls enters the diamond through a left or bottom corner and ends on the bottom left edge. One does not consider this “passing through” the diamond (where the normal rules would have us light the pixel). This is to avoid the following case: One slope=1 line segment enters through one corner and ends on the edge, and another (continuation) line segments starts at that point on the edge and exits through the other corner. If simply passing through a corner caused the pixel to be lit, this case would 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.

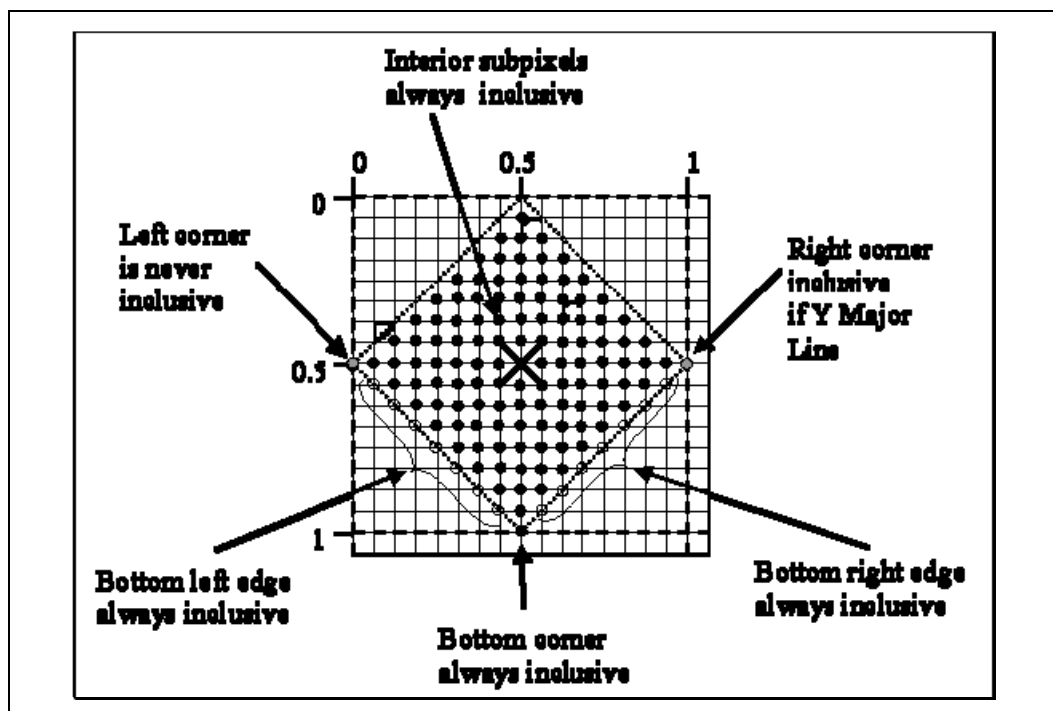
The following equation determines whether a point (point.x, point.y) is inside the diamond of the pixel sample point (sample.x, sample.y), given additional information about the slope (slopePosOne, slopeNegOne).

```
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 $\frac{1}{2}$.

The subpixels falling on the edges of the diamond (Manhattan distance = $\frac{1}{2}$) are exclusive, with the following exceptions:

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



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

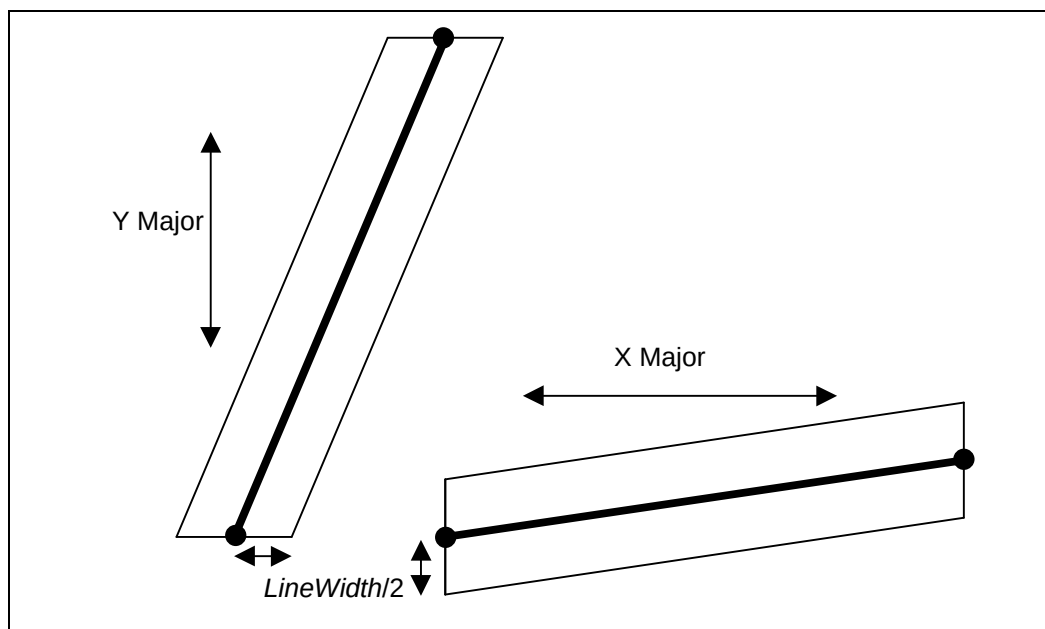
```
delta_x      = point.x - sample.x
delta_y      = point.y - sample.y
distance     = abs(delta_x) + abs(delta_y)
interior     = (distance < 0.5)
bottom_corner = (delta_x == 0.0) && (delta_y == 0.5)
left_corner  = (delta_x == -0.5) && (delta_y == 0.0)
right_corner  = (delta_x == 0.5) && (delta_y == 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

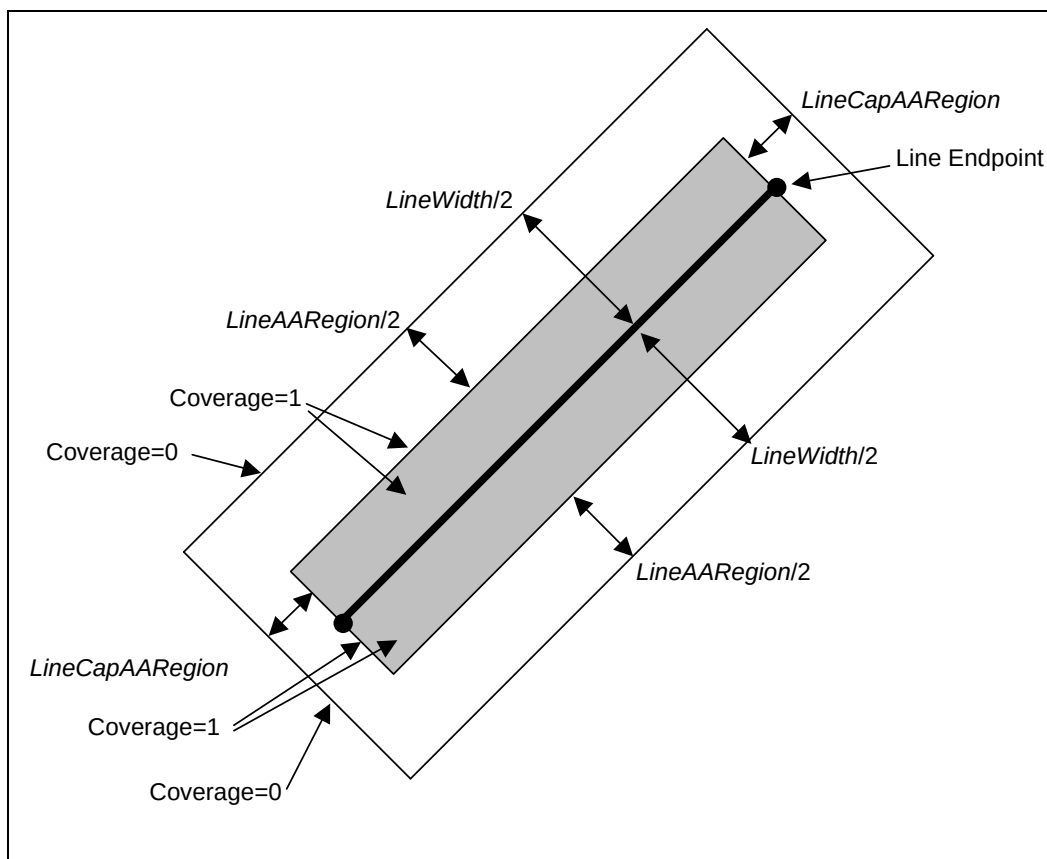


7.3.13.5 Anti-aliased Line Rasterization

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 its 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”:

$(\text{abs}(\text{delta\_x}) + \text{abs}(\text{delta\_y}))$

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 (1 st GEN4 core instruction) of the kernel program run by threads spawned by this FF unit. It is specified as a 64-byte-granular offset from the General State Base Address . See <i>3D Pipeline</i> 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 <i>3D Pipeline</i> 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 <i>ISA Execution Environment</i> 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 <i>3D Pipeline</i> 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 <i>Exceptions</i> and <i>ISA Execution Environment</i>. Format: Enable
	12	Reserved: MBZ
	11	MaskStack Exception Enable. This bit gets loaded into EU CR0.1[11]. See <i>Exceptions</i> and <i>ISA Execution Environment</i>. Format: Enable
	10:8	Reserved: MBZ
	7	Software Exception Enable. This bit gets loaded into EU CR0.1[13] (note the bit # difference). See <i>Exceptions</i> and <i>ISA Execution Environment</i>. 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 <i>Graphics Processing Engine</i>). 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 <i>Graphics Processing Engine</i>). 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 <i>AAEnable</i> (though the <i>AAEnable</i> 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 (0x0) 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 (0x0) 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 <u>as a hint</u> 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 <i>less than Y Min</i> 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 <i>less than X Min</i> 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 <i>greater than Y Max</i> 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 <i>greater than X Max</i> 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 <i>Debug</i> 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 <i>Debugging</i> 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 <u>reordered</u> 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 is 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 – <u>there is no other hardware function involved</u> . 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 <i>Vertex Fetch</i> 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 is less feasible to have the CLIP unit figure out how many objects have been output by a clip thread. It is easy for the SF unit to count the number of objects it receives from the CLIP stage since it is decomposing the output primitive topologies into objects anyway.

If the **Statistics Enable** bit is set in SF_STATE, then SF will increment the CL_PRIMITIVES_COUNT Register (see Memory Interface Registers in Volume 1a, *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 (SIMDn _{xm} with m = 1) or multiple program flows (SIMDn _{xm} with m > 1). See CR0 description in <i>ISA Execution Environment</i> . 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 <u>low order 128 bits</u> 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 <i>Strip and Fan Unit</i> and <i>URB</i> 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 <i>Exceptions</i> and <i>ISA Execution Environment</i> . Format: Enable
	3	Reserved : MBZ
	2	MaskStack Exception Enable . This bit gets loaded into EU CR0.1[11]. See <i>Exceptions</i> and <i>ISA Execution Environment</i> . Format: Enable



DWord	Bit	Description
	1	Software Exception Enable. This bit gets loaded into EU CR0.1[13] (note the bit # difference). See <i>Exceptions</i> and <i>ISA Execution Environment</i> . 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. <u>If MMIO register 21D0h bit 3 is set</u>, 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 <i>Statistics Gathering</i>. 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 <i>cleared</i> 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 <i>Graphics Processing Engine</i> 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 <i>Strips Fans</i> 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 <i>GlobalDepthOffsetScale</i> term used in the Global Depth Offset Function Format = IEEE_FP

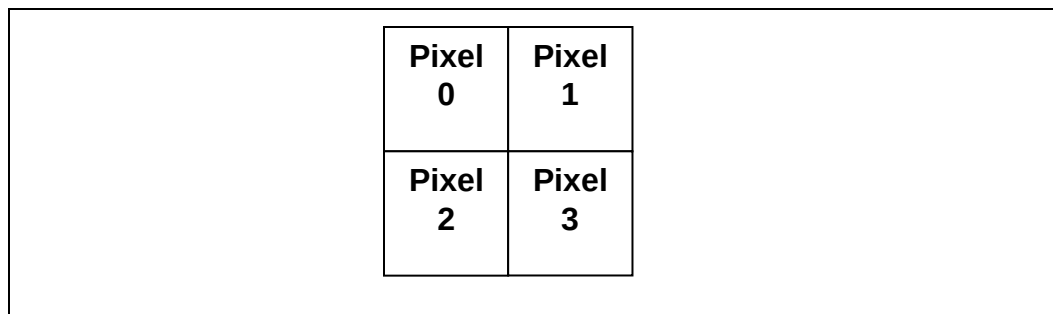


8.3 Rasterization

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**. Software must compute **Line Stipple Inverse Repeat Count** as $1.0f / \text{Line Stipple Repeat Count}$ and then converted from float to the required fixed point encoding (see 3DSTATE_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.

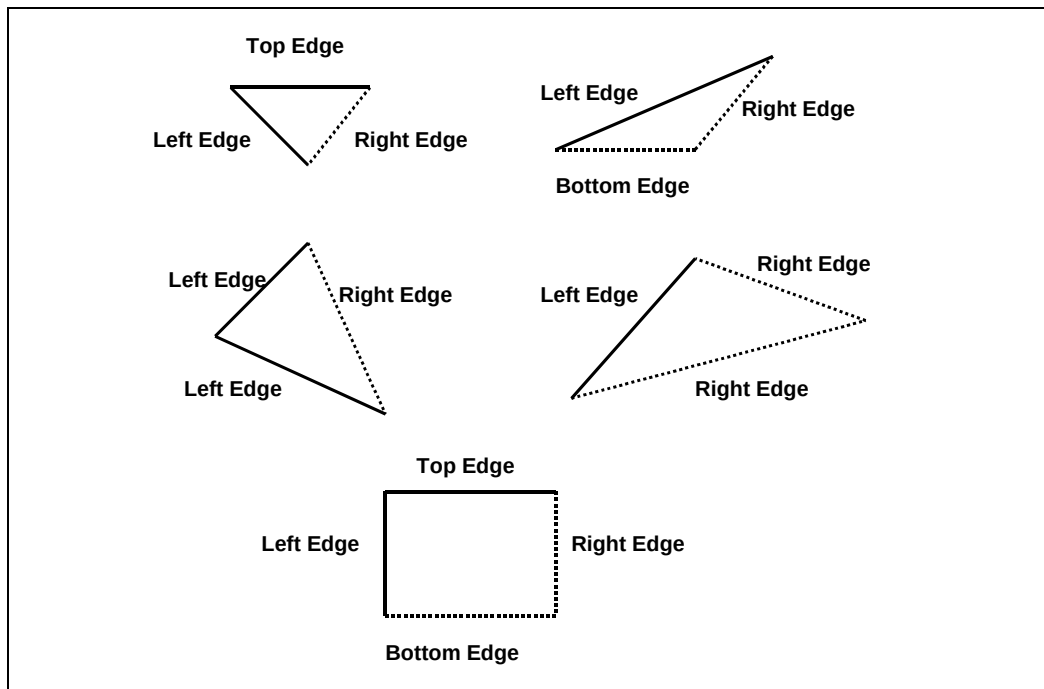
8.3.3 Polygon (Triangle and Rectangle) Rasterization

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 3h GFXPIPE Value: Format: OpCode	
	28:27	Command SubType Default 3h GFXPIPE_3D Value: Format: OpCode	
	26:24	3D Command Opcode Default 1h 3DSTATE_NONPIPELINED Value: Format: OpCode	
	23:16	3D Command Sub Opcode Default 06h 3DSTATE_POLY_STIPPLE_OFFSET Value: 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 <i>GlobalDepthOffsetClamp</i> 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:

$$\text{Depth Coefficient URB Read Offset} = \text{element\_entry} * 2 + 1$$

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:
 $\text{MaxDepthSlope} = \max(\text{abs}(dZ/dX), \text{abs}(dz/dy))$ // approximation of max depth slope for polygon

When UNORM Depth Buffer is at Output Merger (or no Depth Buffer):

$$\text{Bias} = \text{GlobalDepthOffsetConstant} * r + \text{GlobalDepthOffsetScale} * \text{MaxDepthSlope}$$



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

When Floating Point Depth Buffer at Output Merger:

$$\text{Bias} = \text{GlobalDepthOffsetConstant} * 2^{(\text{exponent}(\text{max } z \text{ in primitive}) - r)} + \text{GlobalDepthOffsetScale} * \text{MaxDepthSlope}$$

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

Adding Bias to z :

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

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 Compute 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 Compute d 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				
		Linear surfaces can be mapped to Main Memory (uncached) or System Memory (cacheable, snooped). Tiled surfaces can only be mapped to Main Memory.				
		The corresponding cache(s) must be invalidated before a previously accessed surface is accessed again with an altered state of this bit.				
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. <u>The Depth Buffer, if tiled, must use Y-Major tiling.</u> See <i>Memory Interface Functions</i> for details on memory tiling and restrictions. This field is ignored when the surface is linear. <table> <tr> <th>Value</th> <th>Name</th> <th>Description</th> <th>Project</th> </tr> <tr> <td>0h</td> <td>Reserved</td> <td></td> <td>All</td> </tr> <tr> <td>1h</td> <td>TILEWALK_YMAJOR</td> <td>Y major tiled</td> <td>All</td> </tr> </table>	Value	Name	Description	Project	0h	Reserved		All	1h	TILEWALK_YMAJOR	Y major tiled	All	
	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. <table> <tr> <th>Programming Notes</th> <th>Project</th> </tr> <tr> <td>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.</td> <td>All</td> </tr> <tr> <td>This bit must not be set when rendering to field-mode (interlaced) Color Buffers (i.e., when Surface State’s VerticalLineStride==1).</td> <td>All</td> </tr> <tr> <td>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)</td> <td>All</td> </tr> </table>	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					
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	



3DSTATE_DEPTH_BUFFER																																			
2	20:18	Surface Format Project: All Format: U3 enumerated type FormatDesc Specifies the format of the depth buffer. <table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td></td><td>D32_FLOAT_S8X24_UINT</td><td>All</td></tr><tr><td>1h</td><td></td><td>D32_FLOAT</td><td>All</td></tr><tr><td>2h</td><td></td><td>D24_UNORM_S8_UINT</td><td>All</td></tr><tr><td>3h</td><td></td><td>D24_UNORM_X8_UINT</td><td>Reserved</td></tr><tr><td>4h</td><td></td><td>Reserved</td><td>All</td></tr><tr><td>5h</td><td></td><td>D16_UNORM</td><td>All</td></tr><tr><td>6h-7h</td><td></td><td>Reserved</td><td>All</td></tr></table>		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
	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 <u>tiled</u> , the pitch specified must be a multiple of the tile pitch, All in the range [128B, 128KB]. If the surface is <u>linear</u> , the pitch can be any multiple of 64 bytes up to 128KB. All																																		
	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 <u>tiled</u> , the base address must conform to the Per-Surface Tiling Alignment Rules as documented in TBD. If the buffer is <u>linear</u> , 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



3DSTATE_DEPTH_BUFFER																
	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 <table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td></td><td>MIPLAYOUT_BELOW</td><td>All</td></tr><tr><td>1h</td><td></td><td>MIPLAYOUT_RIGHT</td><td>All</td></tr></table> Programming Notes MIPLAYOUT_RIGHT is legal only for 2D non-array surfaces All			Value	Name	Description	Project	0h		MIPLAYOUT_BELOW	All	1h		MIPLAYOUT_RIGHT	All
	Value	Name	Description	Project												
0h		MIPLAYOUT_BELOW	All													
1h		MIPLAYOUT_RIGHT	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 corresponds 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 multiply 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 <i>Debug</i> 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 <i>Debug</i> 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 <i>Debug</i> 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 synchronizing 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. <u>If MMIO register 21D0h bit 3 is set</u>, 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 executes in 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]



DWord	Bit	Description														
	26:16	Render Target Array Index:Specifies the array index to be used for the following surface types: SURFTYPE_1D: specifies the array index. Range = [0,511] SURFTYPE_2D: specifies the array index. Range = [0,511] SURFTYPE_3D: specifies the “r” coordinate. Range = [0,2047] SURFTYPE_CUBE: specifies the face identifier. Range = [0,5] <table><tr><th>face</th><th>Render Target Array Index</th></tr><tr><td>+x</td><td>0</td></tr><tr><td>-x</td><td>1</td></tr><tr><td>+y</td><td>2</td></tr><tr><td>-y</td><td>3</td></tr><tr><td>+z</td><td>4</td></tr><tr><td>-z</td><td>5</td></tr></table> Format = U11	face	Render Target Array Index	+x	0	-x	1	+y	2	-y	3	+z	4	-z	5
	face	Render Target Array Index														
+x	0															
-x	1															
+y	2															
-y	3															
+z	4															
-z	5															
	15:0	Reserved														
R1.6	31	Front / Back Facing Polygon:Determines whether the polygon is front or back facing. Used by the render cache to determine which stencil test state to use. 0 = Front Facing 1 = Back Facing														
	30	Source Depth Present:Indicates that source depth is included in the dispatch														
	29	Source Depth to Render Target: Indicates that source depth will be sent to the render target														
	28	Destination Depth Present: Indicates that destination depth is included in the dispatch and sent to the render target														
	27	Destination Stencil Present: Indicates that destination stencil is included in the dispatch and sent to the render target														
	26	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														
	25	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														
	24:5	Reserved														
	4:0	Primitive Topology Type:This field identifies the Primitive Topology Type associated with the primitive spawning this object. The WM unit does not modify this value (e.g., objects within POINTLIST topologies see POINTLIST). Format: (See 3DPRIMITIVE command in <i>3D Pipeline</i>)														



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)
R4.6	3:0	Antialias Alpha for Subspan 2, Pixel 0 (upper left)
	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)
R4.5:4	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 <i>32-pixel dispatch</i> .
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 Coefficients) 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):R _q		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

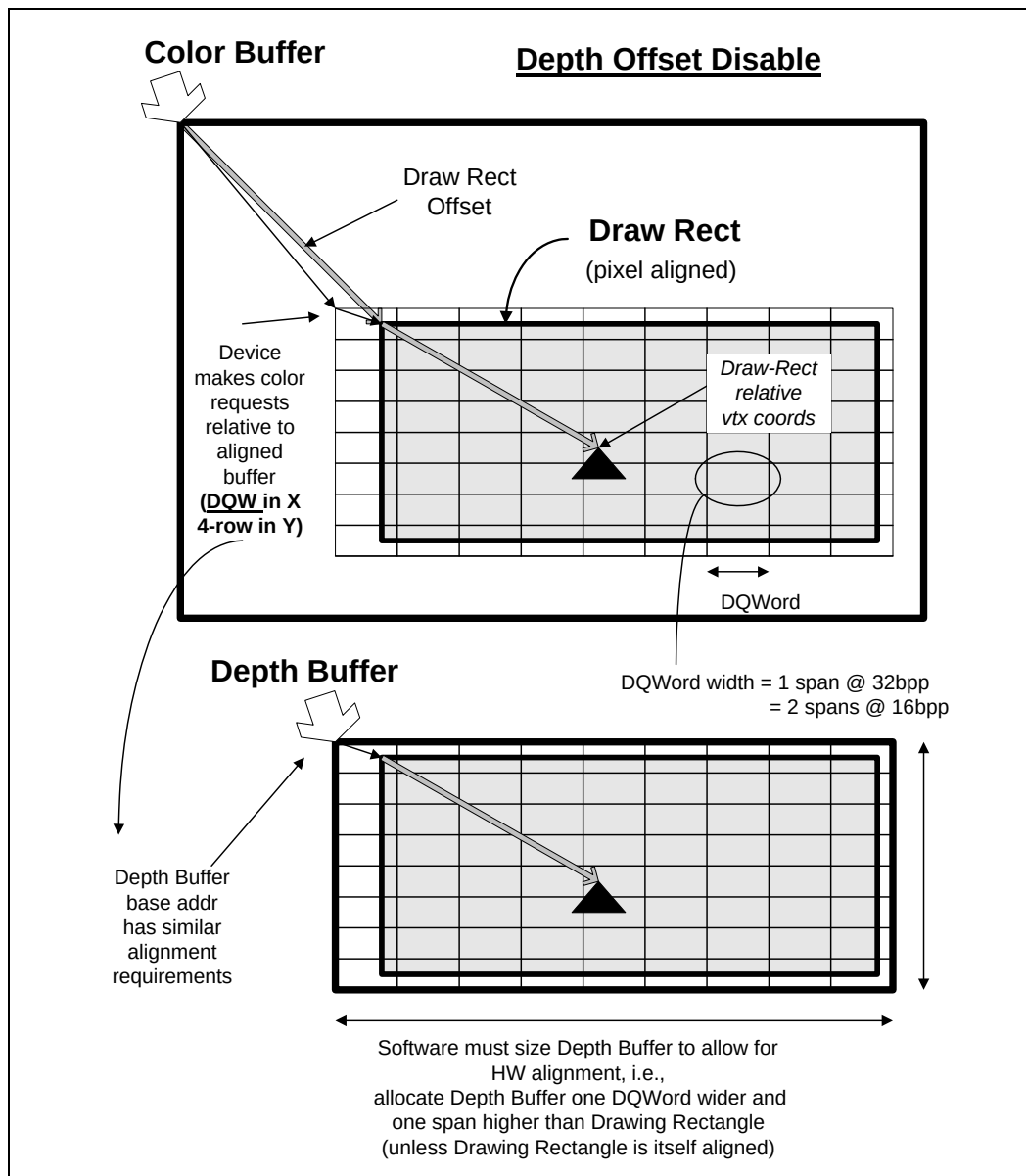
There is a capability to effectively disable the application of the Drawing Rectangle coordinate offset for accesses to the Depth Buffer. This is controlled via the **Depth Buffer Coordinate Offset Disable** state variable in the 3DSTATE_DEPTH_BUFFER command. This capability exists in order to better 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 propagated down the pipeline for use in the subsequent Depth Buffer Update function. The Depth Test function does not in itself discard pixels.

If **Depth Test Enable** is DISABLED, a result of “depth test passed” is propagated down the pipeline.

Programming 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 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 = 2 nd output from PS in Dual-Source mode only) (rtN = destination color from RT#N) (CC = Constant Color)
BLENDFACTOR_ZERO	0.0, 0.0, 0.0, 0.0
BLENDFACTOR_ONE	1.0, 1.0, 1.0, 1.0
BLENDFACTOR_SRC_COLOR	oN.r, oN.g, oN.b, oN.a
BLENDFACTOR_INV_SRC_COLOR	1.0-oN.r, 1.0-oN.g, 1.0-oN.b, 1.0-oN.a
BLENDFACTOR_SRC_ALPHA	oN.a, oN.a, oN.a, oN.a
BLENDFACTOR_INV_SRC_ALPHA	1.0-oN.a, 1.0-oN.a, 1.0-oN.a, 1.0-oN.a
BLENDFACTOR_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 (<u>for each color component</u>)
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

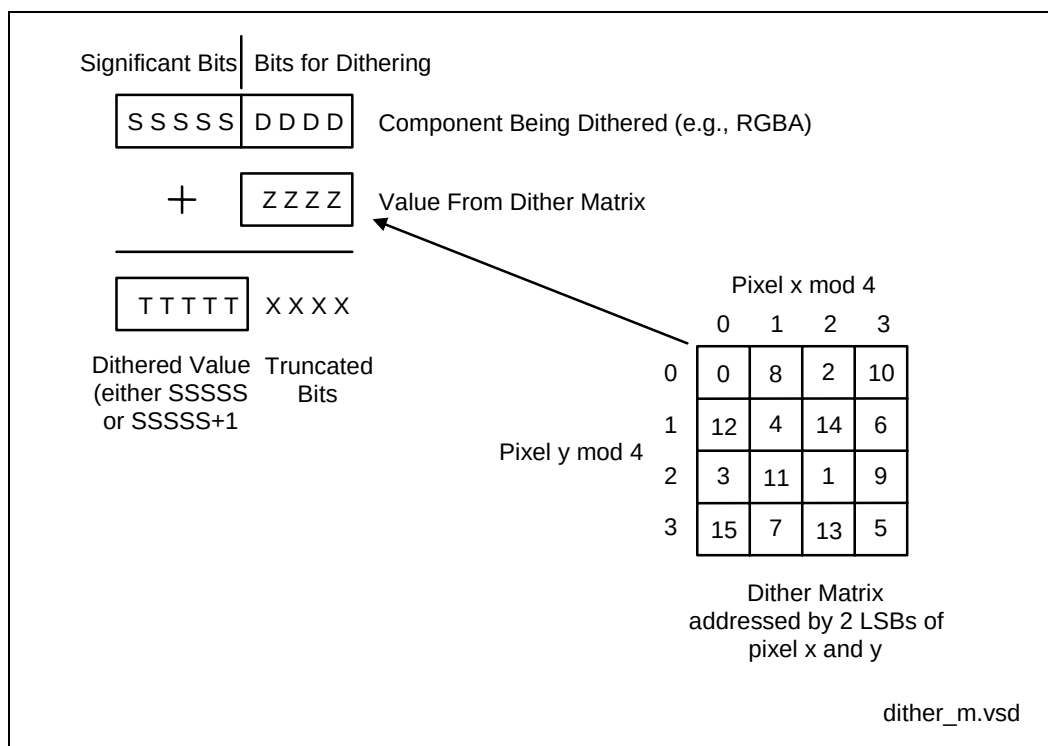
Dithering is used to give the illusion of a higher resolution when using low-bpp channels in color buffers (e.g., with 16bpp color buffer). By carefully choosing an arrangement of lower resolution colors, colors otherwise not representable can be approximated, especially when seen at a distance where 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 <i>StencilTestReference</i> reference value
STENCILOP_INCRSAT	Saturating increment (clamp to max value)
STENCILOP_DECRSAT	Saturating decrement (clamp to 0)
STENCILOP_INCR	Increment (possible wrap around to 0)
STENCILOP_DECR	Decrement (possible wrap to max value)
STENCILOP_INVERT	Logically invert the stencil value

Any and all writes to the stencil portion of the depth buffer are enabled by the **Stencil Buffer Write Enable** state variable.

When writes are enabled, the **Stencil Buffer Write Mask** and **Backface Stencil Buffer Write Mask** state variables provide an 8-bit mask that selects which bits of the stencil write value are modified. Masked-off bits (i.e., mask bit == 0) are left unmodified in the Stencil Buffer.

Programming 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 <i>Color Buffer Blending</i>). [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 suppressed only for that render target. The depth/stencil update will occur if alpha test passes for <i>any</i> render target. Format = Enable															
	10:8	Alpha Test Function : This field specifies the comparison function used in the AlphaTest function Format = 3D_CompareFunction <table><tr><td>0h: COMPAREFUNCTION_ALWAYS:</td><td>Always pass</td></tr><tr><td>1h: COMPAREFUNCTION_NEVER:</td><td>Never pass</td></tr><tr><td>2h: COMPAREFUNCTION_LESS:</td><td>Pass if the value is less than the reference</td></tr><tr><td>3h: COMPAREFUNCTION_EQUAL:</td><td>Pass if the value is equal to the reference</td></tr><tr><td>4h: COMPAREFUNCTION_LEQUAL:</td><td>Pass if the value is less than or equal to the reference</td></tr><tr><td>5h: COMPAREFUNCTION_GREATER:</td><td>Pass if the value is greater than the reference</td></tr><tr><td>6h: COMPAREFUNCTION_NOTEQUAL:</td><td>Pass if the value is not equal to the reference</td></tr><tr><td>7h: COMPAREFUNCTION_GEQUAL:</td><td>Pass if the value is greater than or equal to the reference</td></tr></table>	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:
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 <i>IndependentAlphaBlend</i> 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 <u>prior to blending</u>, <u>regardless of whether blending is enabled</u>. 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 <i>Pre-Blending Color Clamp</i> 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 <i>Pre-Blending Color Clamp</i> 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)



10.1.1.1 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 Compensation. It is part of the video decoding pipe.
MVFS	Motion Vector Field Selection – a four-bit field selecting reference fields for the motion vectors of the current macroblock.
PRT	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).

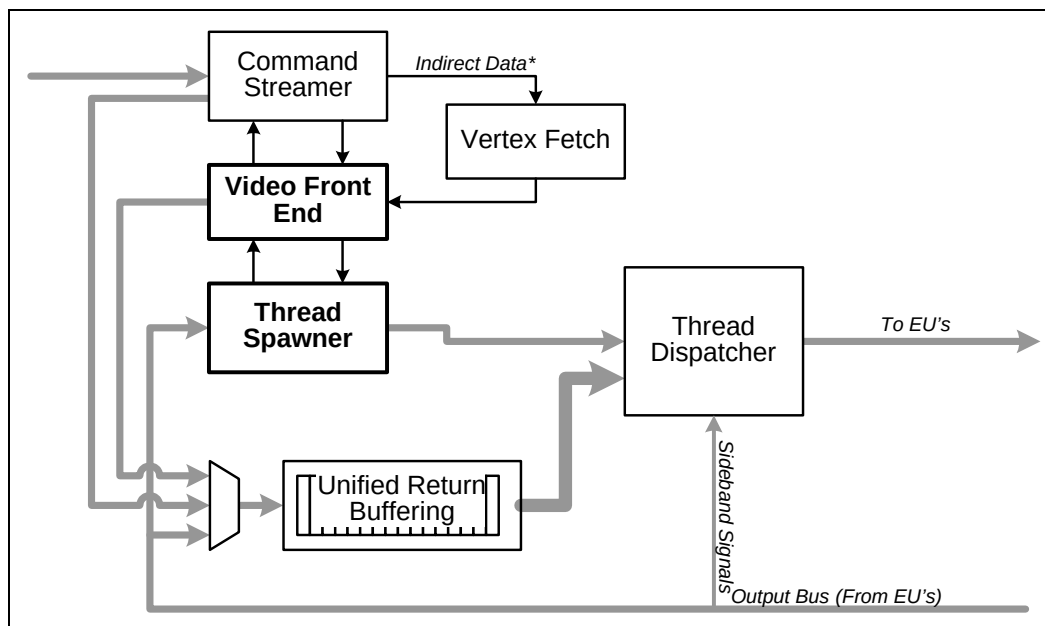
The fixed functions, VFE and TS, in the media pipeline, in most cases, share the same basic building blocks as the fixed functions in the 3D pipeline. However, there are some unique features in media fixed functions as highlighted by the followings.

- 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
 - o Always initialize the URB fence (with a URB_FENCE command) before the first pipeline select command (PIPELINE_SELECT).
 - o Always initialize the pipeline state pointer (with a STATE_BASE_ADDRESS command) before the first pipeline select command.



- Step 1: MI_FLUSH/PIPE_CONTROL
 - o This step is mandatory.
 - o Programmer may choose not to flush certain caches to improve performance.
 - o Multiple such commands in step 1 are allowed, but not recommended for performance reason.
 - o **[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**
 - o 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, and SF.
- Step 2: PIPELINE_SELECT
 - o 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.
 - o Multiple such commands in step 2 are allowed, but not recommended for performance reason.
 - o If this command is issued, it must be followed by a URB_FENCE command (step 3).
- Step 3: URB_FENCE
 - o 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.
 - o 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
 - o 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.*
 - o The following state commands can be issued in arbitrary order.
 - o 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.
 - o 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).
 - o STATE_PREFETCH
 - This command is optional for this step.
 - o STATE_SIP
 - This command is optional for this step. It is only required when SIP is used by the kernels.
 - o 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
 - o Vertex Buffer Index = 0
 - o Valid = True
 - o Surface Format = 0x002
 - o Source Element Offset = 0x0
 - o Component Control 0,1,2,3 = 0x1
 - o Destination Offset = 0x0
 - Vertex Element 1
 - o Vertex Buffer Index = 0
 - o Valid = True
 - o Surface Format = 0x002
 - o Source Element Offset = 0x10
 - o Component Control 0,1,2,3 = 0x1
 - o Destination Offset = 0x10
- o 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>
 - o 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.
 - o 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
 - o This step is optional. However, it is required (as a software workaround) when 3DPRIMITIVE commands are used subsequently to load indirect object data.



- o 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
 - o `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
 - o `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 achieved. 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 pose 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

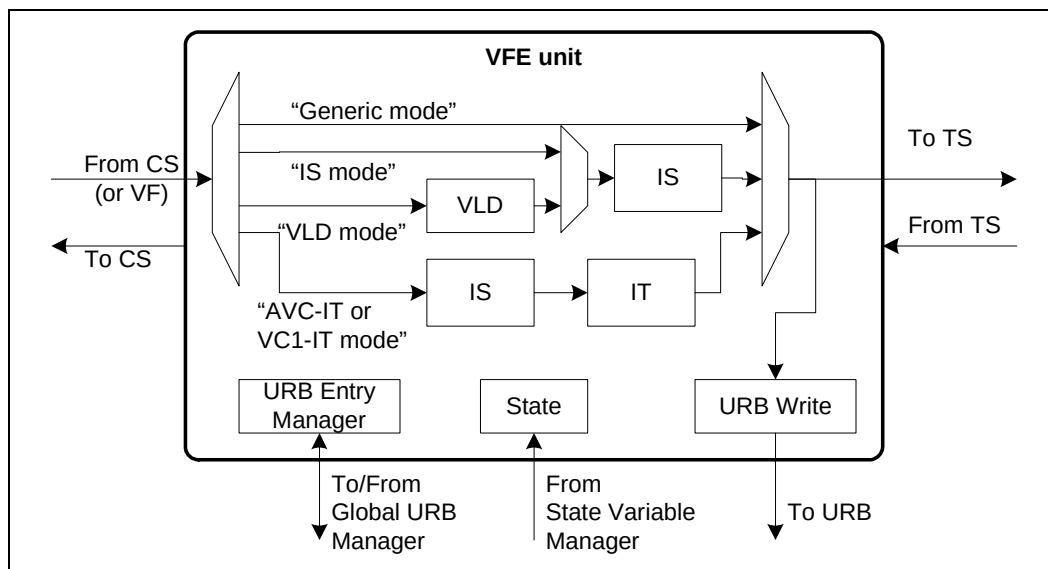
The Video Front End unit is the first fixed function unit in the media pipeline. It processes MEDIA_OBJECT commands to generate root threads by preparing the control (including interface descriptor pointers) and payload (data pushed into the GRF) for the root threads.

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 State 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.

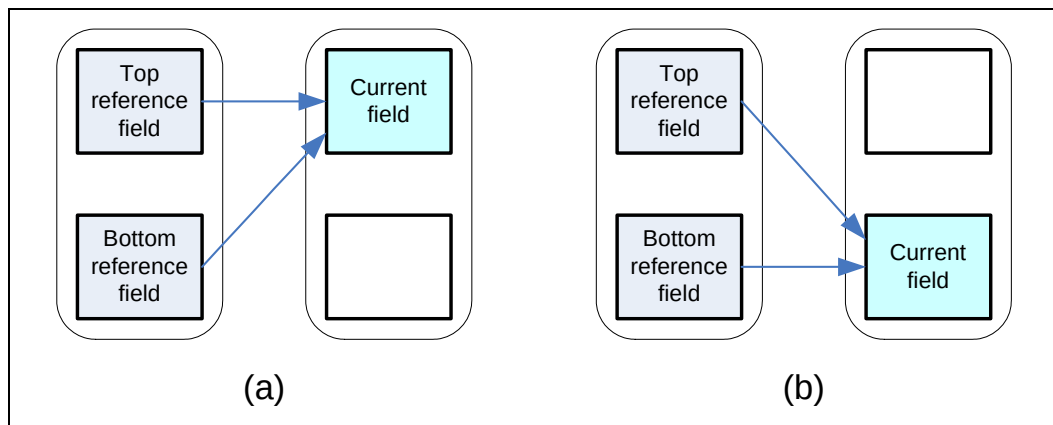
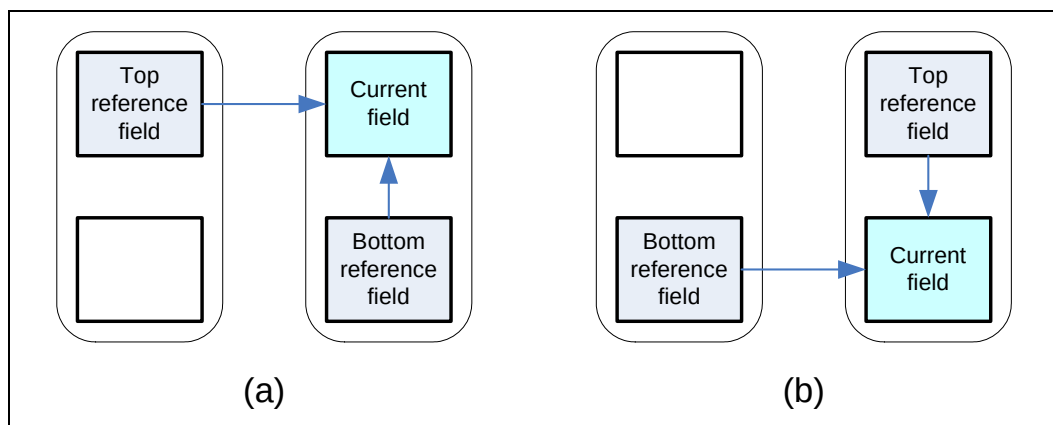


Figure 10-4. Prediction for a P field picture that is a second field, which is (a) a top field, or (b) a bottom field





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 -									
?	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]=1	Dst (M1) if MVFS[0][0]=0 Fwd (M0) if MVFS[0][0]=1	whole field, forward	
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
		MV[1][0]		Fwd	Fwd	x	x	lower 16x8 field, forward		
		v'[0][1][1:0]		Back	Back	x	x	upper 16x8 field, backward		
		v'[1][1][1:0]		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]=1	Dst (M1) if MVFS[0][0]=0 Fwd (M0) if MVFS[0][0]=1	upper 16x8 field, forward	
		MV[1][0]		Fwd (M0)	Fwd (M0)	Fwd (M0) if MVFS[1][0]=0 Dst (M1) if MVFS[1][0]=1	Dst (M1) if MVFS[1][0]=0 Fwd (M0) if MVFS[1][0]=1	lower 16x8 field, forward		
				v'[1][0][1:0]						
1	0	v'[0][1][1:0]	4	MV[0][1]	Back	Back	x	x	upper 16x8 field, backward	B-Pict only
		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])
		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 Destination 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.

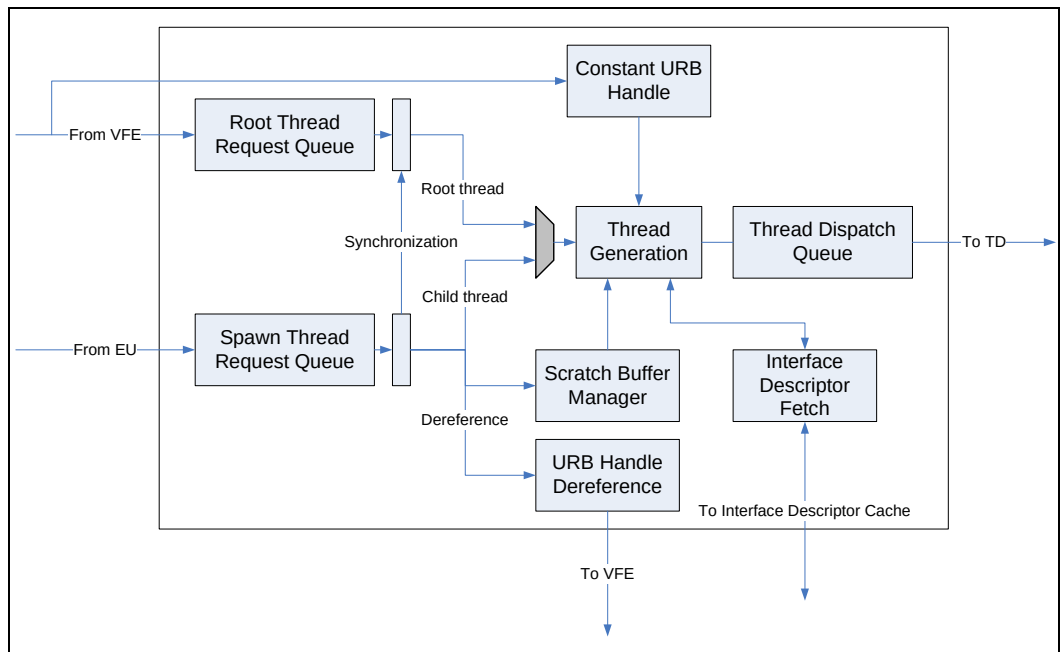
The Debug Counter is not saved/restored during context switch. Therefore, when the media pipe is used by multiple contexts, debug counter may only be used by a single context. In order to not interfere the context being debugged, the debug counter should be left unchanged by software in other contexts.



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 be a macroblock thread created by VFE as in VLD mode, or may be a general-purpose thread assembled by VFE according to full description provided by host software in Generic mode.

Thread requests are stored in the Root Thread Queue. TS keeps everything needed to get the root threads ready for dispatch and then tracks dispatched threads until their retirement.

TS arbitrates between root thread and child thread. The root thread request queue is in the arbitration only if the number of outstanding threads does not exceed the maximum root thread state variable. Otherwise, the root thread request queue is stalled until some other root threads retire/terminate.



Once a root thread is selected to be dispatched, its lifecycle can be described by the following steps:

1. TS forwards the interface descriptor pointer to the L1 interface descriptor cache (a small fully associated cache containing up to 4 interface descriptors). The interface descriptor is either found in the cache or a corresponding request is forwarded to the L2 cache. Interface descriptors return back to TS in requesting order.
 - Once TS receives the interface descriptor, it checks whether maximum concurrent root thread number has reached to determine whether to make a thread dispatch request or to stall the request until some other root threads retire. If the thread requests the use of scratch memory, it also generates a pointer into the scratch space.
2. TS then builds the transparent header and the R0 header.
3. Finally, TS makes a thread request to the Thread Dispatcher.
4. TS keeps track of dispatched thread, and monitors messages from the thread (resource dereference and/or thread termination). When it receives a root thread termination message, it can recover the scratch space and thread slot allocated to it. The URB handle may also be dereferenced for a terminated root thread for future reuse. It should be noted that URB handle dereference may occur before a root thread terminates. See detailed description in the Media Message section.
 - It is the root thread's responsibility (software) to guarantee that all its children have retired before the root thread can retire.

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 dereference at the time of dispatch. TS signals URB handle dereference 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

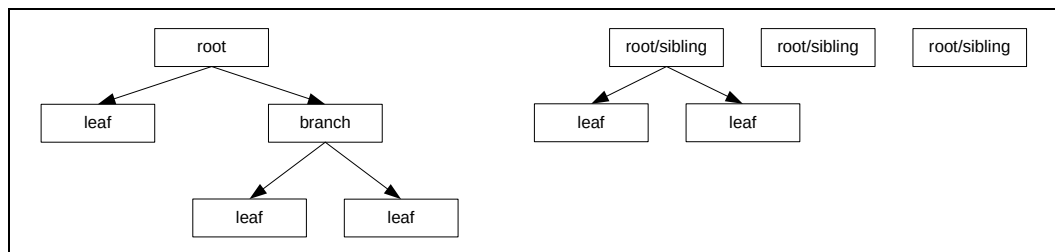
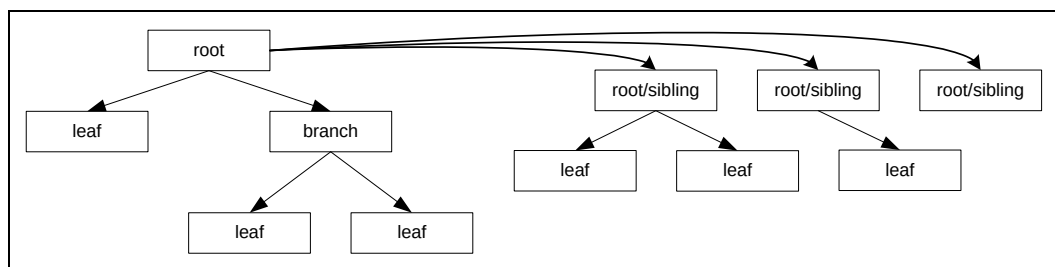


Figure 10-7. An example of thread relationship with root sibling dependency





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.



$$\text{Max\_Outstanding\_Child\_Threads} = (\text{Thread Slot Number} - 1) + (\text{TS Child Queue Depth} - 1)$$

Adding other root threads (synchronized and/or non-synchronized) to the above example, the situation is more complicated. A conservative measure may have to use to prevent deadlock. For example, the root thread spawning child threads may have to exclude the max number of root threads as in the following equation to compute the maximum number of outstanding child threads to be dispatched.

$$\text{Max\_Outstanding\_Child\_Threads} = (\text{Thread Slot Number} - 1) + (\text{TS Child Queue Depth} - 1) - (\text{Max Root Threads} - 1)$$

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 dereference 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

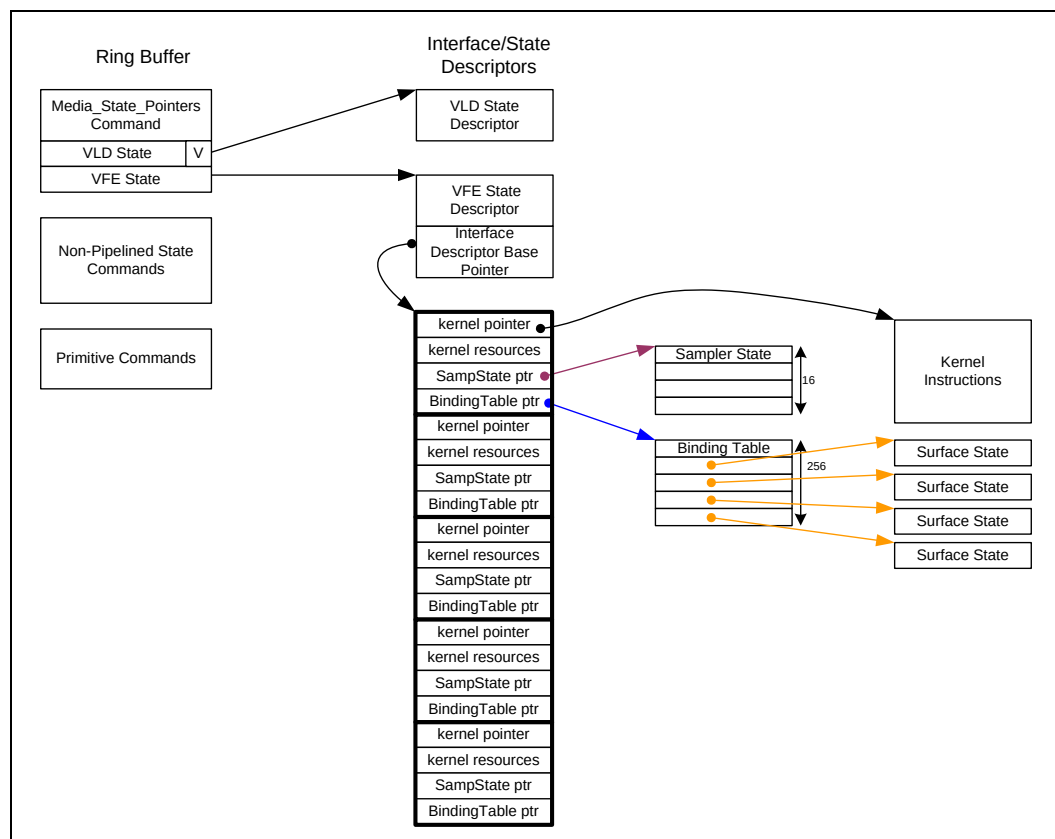
The media state model is based on an indirect state fetching mechanism. State Descriptors provide state information for fixed function units of the media pipeline. Interface Descriptors provide state information for kernels (threads) dispatched from the media pipeline. They are organized in different 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 <i>Graphics Processing Engine</i> 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

Dword	Bit	Description									
0	31:28	f_code[1][1] . Used for backward motion vector prediction. See ISO/IEC 13818-2 §7.6.3.1 for details									
	27:24	f_code[1][0] . Used for backward motion vector prediction. See ISO/IEC 13818-2 §7.6.3.1 for details									
	23:20	f_code[0][1] . Used for forward motion vector prediction. See ISO/IEC 13818-2 §7.6.3.1 for details									
	19:16	f_code[0][0] . Used for forward motion vector prediction. See ISO/IEC 13818-2 §7.6.3.1 for details									
	15:14	Intra DC Precision . See ISO/IEC 13818-2 §6.3.10 for details.									
	13:12	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 <i>ISO/IEC 13818-2</i> §6.3.10 for details. Format = MPEG_PICTURE_STRUCTURE 00 = Reserved 01 = MPEG_TOP_FIELD 10 = MPEG_BOTTOM_FIELD 11 = MPEG_FRAME									
	11	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><td></td><td>Picture Structure = top field</td><td>Picture Structure = bottom field</td></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>		Picture Structure = top field	Picture Structure = bottom field	Second Field = 0	TFF = 1	TFF = 0	Second Field = 1	TFF = 0	TFF = 1
		Picture Structure = top field	Picture Structure = bottom field								
	Second Field = 0	TFF = 1	TFF = 0								
	Second Field = 1	TFF = 0	TFF = 1								
10	Frame Prediction Frame DCT . This field provides constraints on the DCT type and prediction type. It affects the syntax of the bitstream.										
9	Concealment Motion Vector Flag . This field indicates if the concealment motion vectors are coded in intra macroblocks. It affects the syntax of the bitstream.										
8	Quantizer Scale Type . This field specifies the quantizer scaling type. Format = MPEG_Q_SCALE_TYPE 0 = MPEG_QSCALE_LINEAR 1 = MPEG_QSCALE_NONLINEAR										
7	Intra VLC Format . This field is used by VLD.										



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 <i>ISO/IEC 13818-2</i> §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 (SIMDn _{xm} with m = 1) or multiple program flows (SIMDn _{xm} 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 <i>Exceptions</i> and <i>ISA Execution Environment</i> . Format: Enable
	12	Reserved: MBZ
	11	MaskStack Exception Enable. This bit gets loaded into EU CR0.1[11]. See <i>Exceptions</i> and <i>ISA Execution Environment</i> . Format: Enable
	10:8	Reserved: MBZ
	7	Software Exception Enable. This bit gets loaded into EU CR0.1[13] (note the bit # difference). See <i>Exceptions</i> and <i>ISA Execution Environment</i> . 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] <i>This field is ignored for child threads.</i>
	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 <i>This field is ignored for child threads.</i> <i>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.</i>
	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] <i>This field is ignored for child threads.</i>



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] <i>This field is ignored for child threads.</i> <i>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.</i>

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. <i>In VLD mode, this field is ignored by hardware.</i> 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. <i>In VLD mode, this field must be programmed as 0, because the Children Present field in VFE_STATE must be 0 in this mode.</i> 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 <i>ISO/IEC 13818-2</i>, 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 be 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

DWord	Bit	Description																				
4+0	31:28	Motion Vertical Field Select. A bit-wise representation of a long [2][2] array as defined in §6.3.17.2 of the <i>ISO/IEC 13818-2</i> (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 <u>top</u> reference field. 1 = The prediction is taken from the <u>bottom</u> reference field.	Bit	MVector [r]	MVector [s]	MotionVerticalFieldSelect Index	28	0	0	0	29	0	1	1	30	1	0	2	31	1	1	3
Bit	MVector [r]	MVector [s]	MotionVerticalFieldSelect Index																			
28	0	0	0																			
29	0	1	1																			
30	1	0	2																			
31	1	1	3																			



DWord	Bit	Description															
	27	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 <i>ISO/IEC 13818-2</i>. 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.															
	26	Reserved. (HWMC mode)															
	25:24	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 <i>ISO/IEC 13818-2</i> §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 border="1"> <thead> <tr> <th>Value</th><th>Destination = Frame Picture_Structure = 11</th><th>Destination = Field Picture_Structure != 11</th></tr> </thead> <tbody> <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> </tbody> </table>	Value	Destination = Frame Picture_Structure = 11	Destination = Field Picture_Structure != 11	'00'	Reserved	Reserved	'01'	Field	Field	'10'	Frame	16x8	'11'	Dual-Prime	Dual-Prime
Value	Destination = Frame Picture_Structure = 11	Destination = Field Picture_Structure != 11															
'00'	Reserved	Reserved															
'01'	Field	Field															
'10'	Frame	16x8															
'11'	Dual-Prime	Dual-Prime															
	23:22	Reserved. (Scan method)															
	21	DCT Type. This field specifies the DCT type of the current macroblock. The kernel should ignore this field when processing Cb/Cr data. See <i>ISO/IEC 13818-2</i> §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).															
	20	Overlap Transform (H261 Loop Filter). This field, when set, indicates that overlap smoothing filter is performed after motion compensation and before in-loop deblocking.															
	19	4MV Mode: (H263) This field indicates if the current macroblock is coded with 4 motion vectors, one for each 8x8 block.															
	18	Macroblock Motion Backward. This field specifies if the backward motion vector is active. See <i>ISO/IEC 13818-2</i> 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 <i>ISO/IEC 13818-2</i> Tables B-2 through B-4. 0 = No forward motion vector. 1 = Use forward motion vector(s).															



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 <i>ISO/IEC 13818-2</i> 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 <i>ISO/IEC 13818-2</i> 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 <i>ISO/IEC 13818-2</i> §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 <i>ISO/IEC 13818-2</i> §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

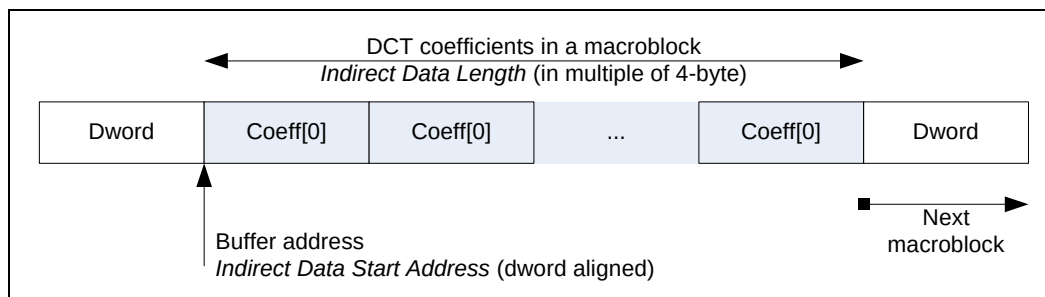
The block data output from the inverse scan will follow immediately after the inline data in the thread's payload, again, aligning to GRF register. Block data output by nature is 8-DW aligned. The actual size depends on the coded block pattern. As each block contains 8x8 16-bit DCT coefficients, if the 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 $2 \cdot 8 + 3 = 19$. 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.



Dispatch Messages: **R**p.d
SEND Instruction Messages: **M**p.d

10.8.1 Thread Payload Messages

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 <i>this thread's</i> 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 <i>this thread's</i> 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.



DWord	Bit	Description																				
	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																				
	5:0	Reserved.																				
R1.1	31:24	Reserved. (Skip Macroblocks)																				
	23:0	Reserved. (Offset into error data)																				
R1.0	31:28	Motion Vertical Field Select. A bit-wise representation of a long [2][2] array as defined in §6.3.17.2 of the <i>ISO/IEC 13818-2</i> (see also §7.6.4). <table border="1"><thead><tr><th>Bit</th><th>MVector [r]</th><th>MVector [s]</th><th>MotionVerticalFieldSelect Index</th></tr></thead><tbody><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></tbody></table> Format = MC_MotionVerticalFieldSelect. 0 = The prediction is taken from the <u>top</u> reference field. 1 = The prediction is taken from the <u>bottom</u> reference field.	Bit	MVector [r]	MVector [s]	MotionVerticalFieldSelect Index	28	0	0	0	29	0	1	1	30	1	0	2	31	1	1	3
	Bit	MVector [r]	MVector [s]	MotionVerticalFieldSelect Index																		
	28	0	0	0																		
29	0	1	1																			
30	1	0	2																			
31	1	1	3																			
27	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 <i>ISO/IEC 13818-2</i> . 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.																					
26	Reserved. (HWMC mode)																					

DWord	Bit	Description															
	25:24	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 <i>ISO/IEC 13818-2</i> §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 border="1"> <thead> <tr> <th>Value</th><th>Destination = Frame Picture_Structure = 11</th><th>Destination = Field Picture_Structure != 11</th></tr> </thead> <tbody> <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> </tbody> </table>	Value	Destination = Frame Picture_Structure = 11	Destination = Field Picture_Structure != 11	'00'	Reserved	Reserved	'01'	Field	Field	'10'	Frame	16x8	'11'	Dual-Prime	Dual-Prime
Value	Destination = Frame Picture_Structure = 11	Destination = Field Picture_Structure != 11															
'00'	Reserved	Reserved															
'01'	Field	Field															
'10'	Frame	16x8															
'11'	Dual-Prime	Dual-Prime															
	23:22	Reserved. (Scan method)															
	21	DCT Type. This field specifies the DCT type of the current macroblock. The kernel should ignore this field when processing Cb/Cr data. See <i>ISO/IEC 13818-2</i> §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).															
	20	Reserved. (H261 Loop Filter)															
	19	Reserved. (H263)															
	18	Macroblock Motion Backward. This field specifies if the backward motion vector is active. See <i>ISO/IEC 13818-2</i> 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 <i>ISO/IEC 13818-2</i> 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 <i>ISO/IEC 13818-2</i> Tables B-2 through B-4. 0 = Non-intra macroblock. 1 = Intra macroblock.															
	15:0	Reserved.															
R2.7	31:0	Reserved.															
R2.6	31:0	Reserved.															
R2.5	31:0	Reserved.															
R2.4	31:0	Reserved.															
R2.3	31:0	Reserved.															
R2.2	31:0	Reserved.															



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 <i>ISO/IEC 13818-2</i> §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 <i>ISO/IEC 13818-2</i> §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 <i>this thread's</i> 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 <i>ISO/IEC 13818-2</i> §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 <i>ISO/IEC 13818-2</i> §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 <i>ISO/IEC 13818-2</i> §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 <i>ISO/IEC 13818-2</i> §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 <i>ISO/IEC 13818-2</i> Table 7-6
R1.1	31:24	Reserved. (Skip Macroblocks)
	23:0	Reserved. (Offset into error data)



DWord	Bit	Description																				
R1.0	31:28	Motion Vertical Field Select. A bit-wise representation of a long [2][2] array as defined in §6.3.17.2 of the <i>ISO/IEC 13818-2</i> (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 <u>top</u> reference field. 1 = The prediction is taken from the <u>bottom</u> reference field.	Bit	MVector [r]	MVector [s]	MotionVerticalFieldSelect Index	28	0	0	0	29	0	1	1	30	1	0	2	31	1	1	3
	Bit	MVector [r]	MVector [s]	MotionVerticalFieldSelect Index																		
	28	0	0	0																		
	29	0	1	1																		
	30	1	0	2																		
	31	1	1	3																		
	27	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 <i>ISO/IEC 13818-2</i>. 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.																				
	26	Reserved. (HWMC mode)																				
	25:24	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 <i>ISO/IEC 13818-2</i> §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>	Value	Destination = Frame Picture_Structure = 11	Destination = Field Picture_Structure != 11	'00'	Reserved	Reserved	'01'	Field	Field	'10'	Frame	16x8	'11'	Dual-Prime	Dual-Prime					
	Value	Destination = Frame Picture_Structure = 11	Destination = Field Picture_Structure != 11																			
'00'	Reserved	Reserved																				
'01'	Field	Field																				
'10'	Frame	16x8																				
'11'	Dual-Prime	Dual-Prime																				
23:22	Reserved. (Scan method)																					
21	DCT Type. This field specifies the DCT type of the current macroblock. The kernel should ignore this field when processing Cb/Cr data. See <i>ISO/IEC 13818-2</i> §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).																					
20	Reserved. (H261 Loop Filter)																					
19	Reserved. (H263)																					

DWord	Bit	Description
	18	Macroblock Motion Backward. This field specifies if the backward motion vector is active. See <i>ISO/IEC 13818-2</i> 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 <i>ISO/IEC 13818-2</i> 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 <i>ISO/IEC 13818-2</i> 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 thread 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 <i>child</i> thread 1 = spawn a <i>root</i> 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 <i>should</i> 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 <i>the child thread's</i> 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

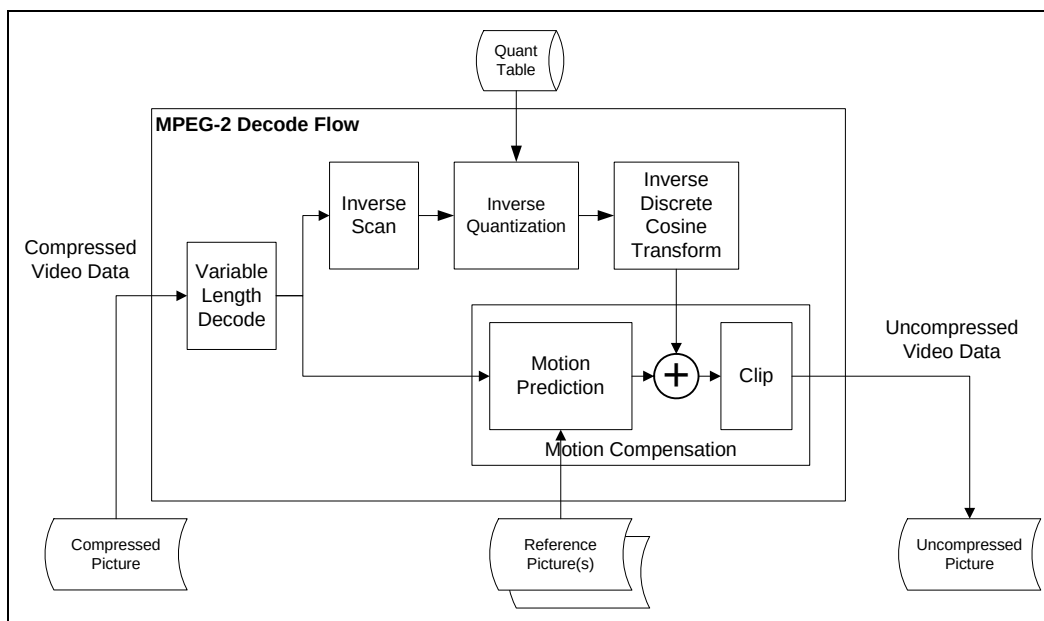
10.9.1 Full MPEG-2 Decode

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-predicted 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:

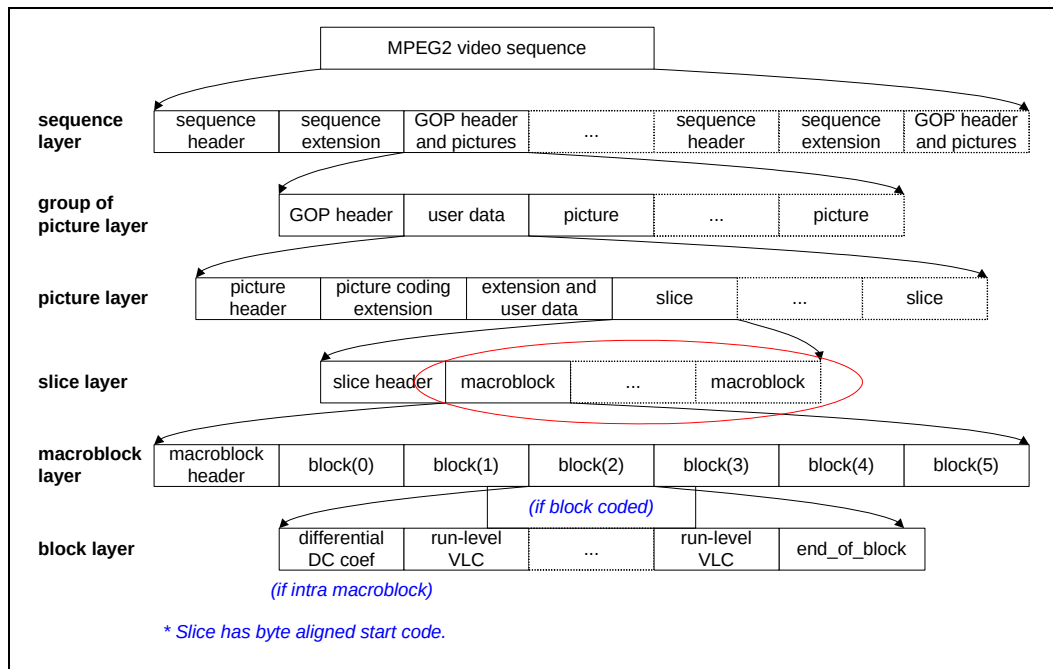
- All layers above slice layer start with unique byte aligned header start-code. The start codes can be easily found within the bitstream without decoding all the symbols in between.
- Majority of bits spent in the macroblock and block layer and there is no start code for these two layers. This indicates 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

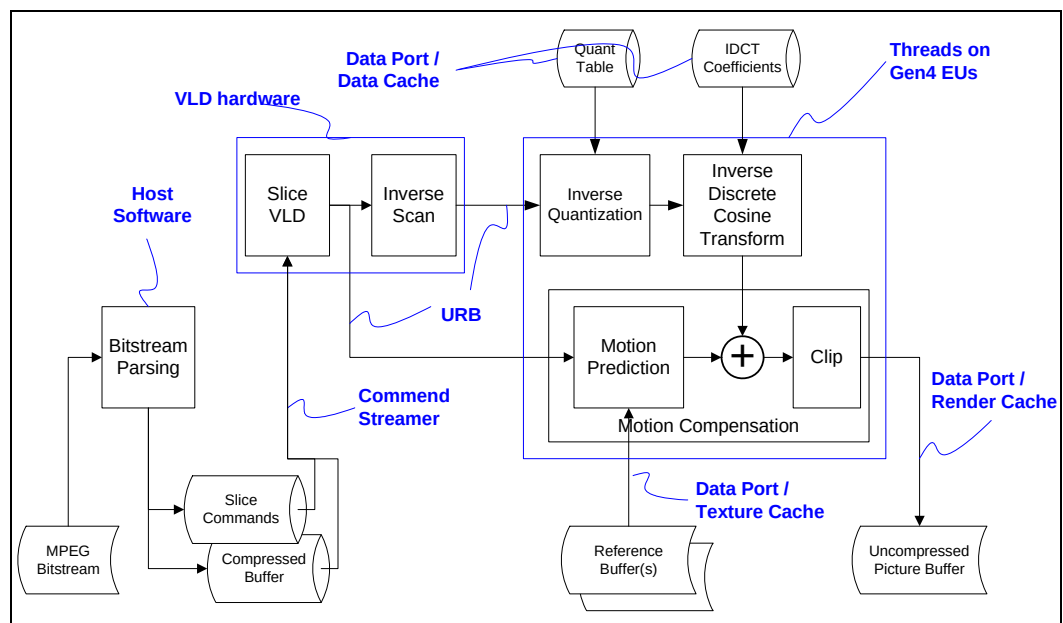


Table 10-14 shows the usage of GEN4 shared resource in this mode of operation.

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

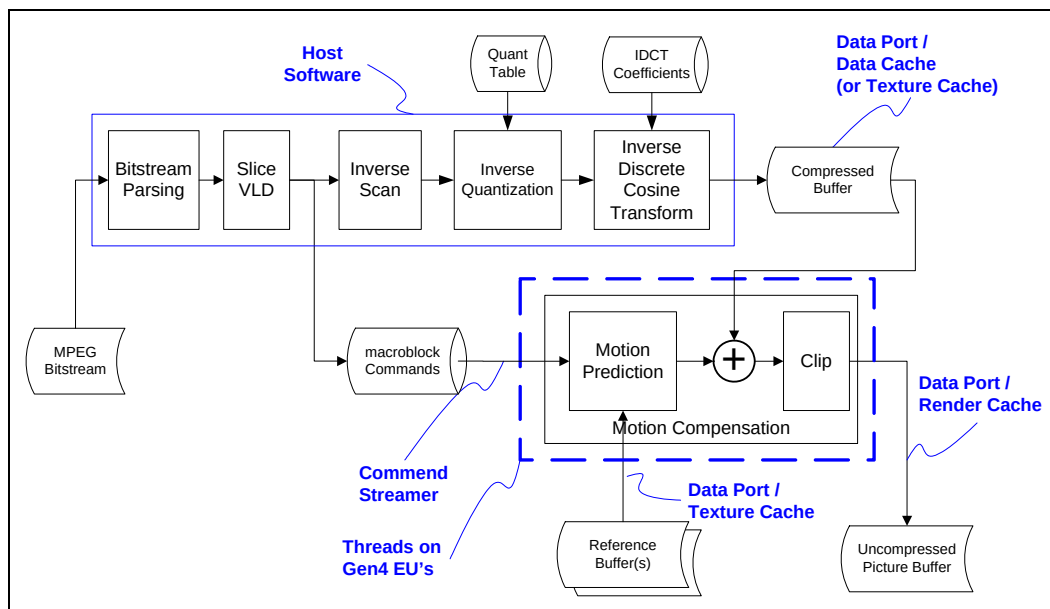


Table 10-15 shows the usage of GEN4 shared resource in this mode of operation.

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

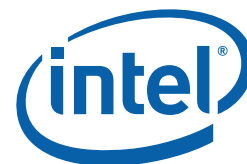
Computers commonly use a non-interlaced video display format, also called a progressive scan format, where an entire frame is scanned line-by-line. In contrast, many sources of consumer video such as NTSC/PAL television signals use an interlaced display format. Interlaced systems interleave two fields 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.



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

Programmer's Reference Manual (PRM)

Volume 3: Display Registers

January 2008

Revision 1.0c

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.

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, or life sustaining applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

The Intel® 965 Express Chipset family and Intel® G35 Express Chipset may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

I2C is a two-wire communications bus/protocol developed by Philips. SMBus is a subset of the I2C bus/protocol and was developed by Intel. Implementations of the I2C bus/protocol may require licenses from various entities, including Philips Electronics N.V. and North American Philips Corporation.

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

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

Copyright © 2008, Intel Corporation. All rights reserved.



Contents

1	Introduction	14
1.1	Notations and Conventions	16
1.1.1	Reserved Bits and Software Compatibility	16
1.2	Terminology	16
2	Display Registers	28
2.1	Introduction and Register Summary	28
2.1.1	Panel Control Register Summary ([DevCL] Only)	29
2.1.2	Display Pipe and Plane Control Registers.....	29
2.1.3	Terminology	30
2.1.4	Register Protection for Panel Protection	31
2.2	Display Mode Set Sequence	31
2.2.1	Pipe register double-buffering	31
2.2.2	Mode Switch Programming Sequences.....	32
2.3	Display Power Down/Up Register Access Sequence.....	34
2.3.1	Power Up/Down for CRT-like Display Devices.....	34
2.3.2	Power Up/Down for Integrated and External Panel Scaler Driven Display Devices.....	34
2.4	GMBUS and I/O Control Registers (05000h–05FFFh).....	35
2.4.1	GPIO Pin Usage (By Functions).....	35
2.4.2	GPIO Pin Usage (By Device).....	35
2.4.3	GPIO Pin Glue Logic.....	37
2.4.3.1	D-Connector	37
2.4.4	GPIO Control Registers.....	38
2.4.5	GMBUS Controller Programming Interface.....	40
2.4.6	GMBUS0—GMBUS Clock/Port Select	41
2.4.7	GMBUS1—GMBUS Command/Status	41
2.4.8	GMBUS2—GMBUS Status Register	44
2.4.9	GMBUS3—GMBUS Data Buffer.....	46
2.4.10	GMBUS4—GMBUS Interrupt Mask	46
2.4.11	GMBUS5—2 Byte Index Register.....	47
2.5	Display Clock Control Registers (06000h–06FFFh).....	47
2.5.1	VGA0—VGA 0 Divisor Register (100.8 MHz dot clk, 25.175 MHz pixel rate)	50
2.5.2	VGA1 —VGA 1 Divisor Register (113.280 MHz dot clk, 28.322 MHz pixel rate)	50
2.5.3	VGA_PD—VGA Post Divisor Values	51
2.5.4	DPLLA_CTRL—DPLL A Control Register	52
2.5.5	DPLLB_CTRL—DPLL B Control Registers.....	54
2.5.6	DPLLAMD—DPLL A SDVO/HDMI Multiplier/Divisor Register	56
2.5.7	DPLLBMD—DPLL B SDVO/HDMI Multiplier/Divisor Register	57
2.5.8	FPA0—DPLL A Divisor Register 0.....	58
2.5.9	FPA1—DPLL A Divisor Register 1.....	58
2.5.10	FPB0—DPLL B Divisor Register	59
2.5.11	FPB1—DPLL B Divisor Register 1.....	59
2.5.12	DPLL_TEST—DPLLA and DPLLB Test Register.....	60
2.5.13	D_STATE—D State Function Control Register.....	62



2.5.14	DSPCLK_GATE_D—Clock Gating Disable for Display Register	63
2.5.15	RENCLK_GATE_D1—Clock Gating Disable for Render Register	66
2.5.16	RENDCLK_GATE_D2—Clock Gating Disable for Render Register 2	69
2.5.17	RAMCLK_GATE_D—GFX RAM Clock Gating Disable Register ([DevCL] Only).....	71
2.5.18	DEUC—Dynamic EU Control ([DevCL] only)	74
2.6	Display Palette Registers (0A000h–AFFFh)	75
2.6.1	DPALETTE_A—Pipe A Display Palette	75
2.6.2	DPALETTE_B—Pipe B Display Palette	77
2.7	Display Pipeline / Port Registers (60000h–6FFFFh)	78
2.7.1	Display Pipeline A.....	78
2.7.1.1	HTOTAL_A—Pipe A Horizontal Total Register	78
2.7.1.2	HBLANK_A—Pipe A Horizontal Blank Register	79
2.7.1.3	HSYNC_A—Pipe A Horizontal Sync Register	80
2.7.1.4	VTOTAL_A—Pipe A Vertical Total Register	80
2.7.1.5	VLANK_A—Pipe A Vertical Blank Register.....	81
2.7.1.6	VSYNC_A—Pipe A Vertical Sync Register	82
2.7.1.7	PIPEASRC—Pipe A Source Image Size	83
2.7.1.8	BCLRPAT_A— Pipe A Border Color Pattern Register	83
2.7.1.9	VSYNCSHIFT_A— Vertical Sync Shift Register	84
2.7.1.10	CRCCtrlColorA—Pipe A CRC Color Channel Control Register (Red)	84
2.7.1.11	CRCCtrlColorA—Pipe A CRC Color Channel Control Register (Green, Blue, Residual)	86
2.7.1.12	CRCResColorA—Pipe A CRC Color Channel Result Register (Red, Green, Blue, Residual1).....	86
2.7.2	Display Pipeline B.....	87
2.7.2.1	HTOTAL_B—Pipe B Horizontal Total Register	87
2.7.2.2	HBLANK_B—Pipe B Horizontal Blank Register.....	87
2.7.2.3	HSYNC_B—Pipe B Horizontal Sync Register	88
2.7.2.4	VTOTAL_B—Pipe B Vertical Total Register	88
2.7.2.5	VLANK_B—Pipe B Vertical Blank Register.....	88
2.7.2.6	VSYNC_B—Pipe B Vertical Sync Register	89
2.7.2.7	PIPEBSRC—Pipe B Source Image Size	89
2.7.2.8	BCLRPAT_B—Pipe B Border Color Pattern Register	89
2.7.2.9	VSYNCSHIFT_B— Vertical Sync Shift Register	90
2.7.2.10	CRCCtrlColorB—Pipe B CRC Color Control Register (Red) 91	
2.7.2.11	CRCCtrlColorB—Pipe B CRC Color Control Register (Green, Blue, Residual)	92
2.7.2.12	CRCResColorB—Pipe B CRC Result Register	92
2.8	Display Port Control.....	93
2.8.1	ADPA—Analog Display Port Register	93
2.8.2	PORT_HOTPLUG_EN	94
2.8.3	PORT_HOTPLUG_STAT	95
2.8.4	sDVO/HDMI B—Digital Display Port B Control Register	96
2.8.5	sDVO/DP—Digital Display Port DFT Register	100
2.8.6	sDVO/DP—Digital Display Port DFT Register 2	102
2.8.7	sDVO/HDMI C—Digital Display Port C Register	103
2.8.8	VIDEO_DIP_CTL—Video DIP Control Register	106
2.8.9	VIDEO_DIP_DATA—Video Data Island Packet Data	108
2.8.10	LVDS—Digital Display Port Control ([DevCL])	109
2.9	Panel Registers	112
2.9.1	Panel Power Sequencing Registers ([DevCL])	112
2.9.1.1	PP_STATUS—Panel Power Status Register ([DevCL])....	112



2.9.1.2	PP_CONTROL—Panel Power Control Register ([DevCL])	114
2.9.1.3	PP_ON_DELAYS—Panel Power on Sequencing Delays ([DevCL])	115
2.9.1.4	PP_OFF_DELAYS—Panel Power off Sequencing Delays ([DevCL])	116
2.9.1.5	PP_DIVISOR—Panel Power Cycle Delay and Reference Divisor ([DevCL])	117
2.9.2	Panel Fitting Registers	118
2.9.2.1	PFIT_CONTROL—Panel Fitting Controls	118
2.9.2.2	PFIT_PGM_RATIOS—Programmed Panel Fitting Ratios	119
2.9.2.3	Reserved (Used to be Auto Scaling Ratios Readback)	119
2.9.2.4	Reserved (Used to be Scaling Initial Phase)	120
2.9.3	Backlight Control and Modulation Histogram Registers ([DevCL])	120
2.9.3.1	BLC_PWM_CLT2—Backlight PWM Control Register 2 ([DevCL])	120
2.9.3.2	BLC_PWM_CTL—Backlight PWM Control Register ([DevCL])	121
2.9.3.3	BLM_HIST_CTL—Image Enhancement Histogram Control Register ([DevCL])	122
2.9.3.4	Image Enhancement Bin Data Register ([DevCL])	123
2.9.3.5	Histogram Threshold Guardband Register ([DevCL])	124
2.10	Display and Cursor Registers (70000h–7FFFFh)	125
2.10.1	Display Pipeline A	125
2.10.1.1	PIPEA_DSL—Pipe A Display Scan Line	125
2.10.1.2	PIPEA_SLC—Pipe A Display Scan Line Count Range Compare	126
2.10.1.3	PIPEACONF—Pipe A Configuration Register	127
2.10.1.4	PIPEAGCMAXRED—Pipe A Gamma Correction Max Red	129
2.10.1.5	PIPEAGCMAXGREEN—Pipe A Gamma Correction Max Green	129
2.10.1.6	PIPEAGCMAXBLUE—Pipe A Gamma Correction Max Blue	129
2.10.1.7	PIPEASTAT—Pipe A Display Status	130
2.10.1.8	DSPARB—Display Arbitration Control	134
2.10.1.9	FW1—Display FIFO Watermark Control 1	135
2.10.1.10	FW2—Display FIFO Watermark Control 2	136
2.10.1.11	FW3—Display FIFO Watermark Control 3	137
2.10.1.12	PipeAFrameHigh— Pipe A Frame Count High	138
2.10.1.13	PipeAFramePixel— Pipe A Frame Count Low and Pixel Count	139
2.10.2	Cursor A Plane Control Registers	139
2.10.2.1	CURACNTR—Cursor A Control Register	140
2.10.2.2	CURABASE—Cursor A Base Address Register	142
2.10.2.3	CURAPOS—Cursor A Position Register	143
2.10.2.4	CURARESV—Cursor A (Reserved)	143
2.10.2.5	CURAPALET[0:3]—Cursor A Palette registers (4 Registers)	144
2.10.3	Cursor B Plane Control Registers	145
2.10.3.1	CURBCNTR—Cursor B Control Register	145
2.10.3.2	CURBBASE—Cursor B Base Address Register	147
2.10.3.3	CURBPOS—Cursor B Position Register	148
2.10.3.4	CURBRESV—Cursor B Reserved	148
2.10.3.5	CURB PALET[0:3]—Cursor B Palette Registers (4 Registers)	149
2.10.4	Display Pipeline B	150
2.10.4.1	PIPEB_DSL—Display Scan Line	150
2.10.4.2	PIPEB_SLC—Pipe B Display Scan Line Range Compare Register	151
2.10.4.3	PIPEBCONF—Pipe B Configuration Register	152
2.10.4.4	PIPEBGCMAXRED—Pipe B Gamma Correction Max Red	154



2.10.4.5	PIPEBGCMAXGREEN—Pipe B Gamma Correction Max Green	154
2.10.4.6	PIPEBGCMAXBLUE—Pipe B Gamma Correction Max Blue	154
2.10.4.7	PIPEBSTAT—Pipe B Status	155
2.10.4.8	PipeBFrameHigh— Pipe B Frame Count High	158
2.10.4.9	PipeBFramePixel — Pipe B Frame Count Low and Pixel Count	158
2.10.5	Display A (Primary) Plane Control	159
2.10.5.1	DSPACNTR—Display A Plane Control Register	159
2.10.5.2	DSPALINOFF—Display A Linear Offset Register	161
2.10.5.3	DSPASTRIDE—Display A Stride Register	162
2.10.5.4	DSPARESV—Display A Reserved	162
2.10.5.5	DSPARESV—Display A Reserved	162
2.10.5.6	DSPAKEYVAL—Sprite Color Key Value Register	163
2.10.5.7	DSPAKEYMSK—Sprite Color Key Mask Register	163
2.10.5.8	DSPASURF—Display A Surface Base Address Register	164
2.10.5.9	DSPATILEOFF—Display A Tiled Offset Register	165
2.10.5.10	DSPAFLPQSTAT—Flip Queue Status Register	165
2.10.6	Display B (Second Primary or Sprite) Control	166
2.10.6.1	DSPBCNTR—Display B/Sprite Plane Control Register	166
2.10.6.2	DSPBLINOFFSET —Display B/Sprite Linear Offset Register	169
2.10.6.3	DSPBSTRIDE—Display B/Sprite Stride Register	169
2.10.6.4	DSPBKEYVAL—Sprite Color Key Value Register	170
2.10.6.5	DSPBKEYMSK—Sprite Color Key Mask Register	170
2.10.6.6	DSPBSURF—Display B Surface Address Register	171
2.10.6.7	DSPBTILEOFF—Display B Tiled Offset Register	172
2.10.6.8	DSPBFLPQSTAT—Flip Queue Status Register	172
2.10.7	Video BIOS Registers (71400h- 714FFh)	173
2.10.7.1	VGACNTRL—VGA Display Plane Control Register	173
2.10.7.2	SWFxx—Software Flag Registers	176
2.10.8	Display C (Sprite or Second Overlay) Control	177
2.10.8.1	DSPCCNTR—Display C Sprite Control Register	177
2.10.8.2	DSPCLINOFF —Display C/Sprite Linear Offset Register	180
2.10.8.3	DSPCSTRIDE—Display C/Sprite Stride Register	180
2.10.8.4	DSPCPOS—Sprite Position Register	181
2.10.8.5	DSPCSIZE—Sprite Height and Width Register	181
2.10.8.6	DSPCKEYMINVAL—Sprite Color Key Min Value Register	182
2.10.8.7	DSPCKEYMSK—Sprite Color Key Mask Register	182
2.10.8.8	DSPCSURF—Display C Surface Address Register	183
2.10.8.9	DSPCKEYMAXVAL—Sprite Color Key Max Value Register	183
2.10.8.10	DSPCTILEOFF—Display C Tiled Offset Register	184
2.10.8.11	DSPCFLPQSTAT—Flip Queue Status Register	184
2.10.9	Display C Color Adjustment	185
2.10.9.1	DCLRC0—Display C Color Correction 0 Register	185
2.10.9.2	DCLRC1—Display C Color Correction 1 Register	186
2.10.9.3	GAMC[5:0]— Display C Gamma Correction Registers	187
2.10.10	Performance Counters	189
2.10.10.1	PCSRC—Performance Counter Source Register	189
2.10.10.2	PCSTAT—Performance Counter Status Register	191
3	Overlay Registers	192
3.1	Introduction and Register Summary	192
3.1.1	Terminology	192
3.2	Overlay Register Definition	193
3.2.1	Introduction to Overlay Registers	193
3.2.1.1	Updating Overlay Registers	196



3.2.1.2	Read Path for Overlay Registers	196
3.2.2	OVADD—Overlay Register Update Address Register	197
3.2.3	OTEST—Overlay Test Register	197
3.2.4	DOVSTA—Display/Overlay Status Register	198
3.2.5	DOVSTAEX—Display/Overlay Extended Status Register	199
3.2.6	OGAMC[5:0]—Overlay Dedicated Gamma Correction Registers	200
3.2.7	Overlay Sync Lock Registers	201
3.2.7.1	OVRSYNCPH[0-3]— Overlay Flip Sync Lock Phase Registers	202
3.2.8	Overlay Memory Image Offset Registers	202
3.2.8.1	Overlay Buffer Start Address Pointer Registers	204
3.2.8.2	OBUF_0Y—Overlay Buffer 0 Y Linear Offset Register	205
3.2.8.3	OBUF_1Y—Overlay Buffer 1 Y Linear Offset Register	205
3.2.8.4	OBUF_0U—Overlay Buffer 0 U Linear Offset Register	206
3.2.8.5	OBUF_0V—Overlay Buffer 0 V Linear Offset Register	206
3.2.8.6	OBUF_1U—Overlay Buffer 1 U Linear Offset Register	207
3.2.8.7	OBUF_1V—Overlay Buffer 1 V Linear Offset Register	207
3.2.8.8	OSTRIDE—Overlay Stride Register	208
3.2.9	Overlay Initial Phase Registers	209
3.2.9.1	YRGB_VPH—Y/RGB Vertical Phase Register	210
3.2.9.2	UV_VPH—UV Vertical Phase Register	211
3.2.9.3	HORZ_PH—Horizontal Phase Register	212
3.2.9.4	INIT_PHS—Initial Phase Shift Register	212
3.2.10	Overlay Destination Window Position/Size Registers	213
3.2.10.1	DWINPOS—Destination Window Position Register	213
3.2.10.2	DWINSZ—Destination Window Size Register	213
3.2.11	Overlay Source Size Registers	214
3.2.11.1	SWIDTH – Source Width Register	214
3.2.11.2	SWIDTHSW—Source SWORD Width Register	215
3.2.11.3	SHEIGHT—Source Height Register	216
3.2.12	Overlay Scale Factor Registers	217
3.2.12.1	YRGBSCALE – Y/RGB Scale Factor Register	217
3.2.12.2	UVSCALE—U V Scale Factor Register	219
3.2.13	Overlay Color Correction Registers	220
3.2.13.1	OCLRC0—Overlay Color Correction 0 Register	220
3.2.13.2	OCLRC1—Overlay Color Correction 1 Register	221
3.2.14	Overlay Destination Color Key Registers	222
3.2.14.1	DCLRKV—Destination Color Key Value Register	222
3.2.14.2	DCLRKM—Destination Color Key Mask Register	223
3.2.15	Overlay Source Chroma Key Registers	224
3.2.15.1	SCHRKVH—Source Chroma Key Value High Register	225
3.2.15.2	SCHRKVL—Source Chroma Key Value Low Register	226
3.2.15.3	SCHRKEN—Source Chroma Key Enable Register	227
3.2.15.4	OCONFIG—Overlay Configuration Register	228
3.2.15.5	OCOMD—Overlay Command Register	230
3.2.15.6	OSTART_0Y— Overlay Surface Y 0 Base Address Register	233
3.2.15.7	OSTART_1Y— Overlay Surface Y 1 Base Address Register	233
3.2.15.8	OSTART_0U— Overlay Surface U 0 Base Address Register	233
3.2.15.9	OSTART_0V— Overlay Surface V 0 Base Address Register	234
3.2.15.10	OSTART_1U— Overlay Surface U 1 Base Address Register	234
3.2.15.11	OSTART_1V— Overlay Surface V 1 Base Address Register	234
3.2.15.12	OTILEOFF_0Y— Overlay Surface Y 0 Tiled Offset Register	235
3.2.15.13	OTILEOFF_1Y— Overlay Surface Y 1 Tiled Offset Register	235
3.2.15.14	OTILEOFF_0U— Overlay Surface U 0 Tiled Offset Register	236
3.2.15.15	OTILEOFF_0V— Overlay Surface V 0 Tiled Offset Register	236



	3.2.15.16 OTILEOFF_1U— Overlay Surface U 1 Tiled Offset Register	237
	3.2.15.17 OTILEOFF_1V— Overlay Surface V 1 Tiled Offset Register	237
3.2.16	Overlay Scaling Registers	238
	3.2.16.1 Reserved FASTHSCALE—Fast Horizontal Downscale Register	238
	3.2.16.2 UVSCALEV—UV Vertical Downscale Integer Register	238
3.2.17	Overlay Polyphase Filter Coefficient Registers	239
	3.2.17.1 Y_VCOEFS—Overlay Y Vertical Filter Coefficient Registers	240
	3.2.17.2 Y_HCOEFS—Overlay Y Horizontal Filter Coefficient Registers	240
	3.2.17.3 UV_VCOEFS—Overlay UV Vertical Filter Coefficient Registers	240
	3.2.17.4 UV_HCOEFS—Overlay UV Horizontal Filter Coefficient Registers	240
4	TV-Out	243
4.1	TV-Out Control Registers	245
	4.1.1 TV_CTL—TV-Out Control	245
	4.1.2 TV_DAC—TV DAC Control / Status	248
4.2	Color Conversion and Control Registers	250
	4.2.1 TV_CSC_Y—Color Space Convert Y	251
	4.2.2 TV_CSC_Y2—Color Space Convert Y2	251
	4.2.3 TV_CSC_U—Color Space Convert U	251
	4.2.4 TV_CSC_U2—Color Space Convert U2	252
	4.2.5 TV_CSC_V—Color Space Convert V	252
	4.2.6 TV_CSC_V2—Color Space Convert V2	252
	4.2.7 TV_CLR_KNOBS—Color Knobs	253
	4.2.8 TV_CLR_LEVEL—Color Level Control	253
4.3	Timing Registers	254
	4.3.1 TV_H_CTL_1—H Control 1	254
	4.3.2 TV_H_CTL_2—H Control 2	254
	4.3.3 TV_H_CTL_3—H Control 3	255
	4.3.4 TV_V_CTL_1—V Control 1	255
	4.3.5 TV_V_CTL_2—V Control 2	256
	4.3.6 TV_V_CTL_3—V Control 3	256
	4.3.7 TV_V_CTL_4—V Control 4	257
	4.3.8 TV_V_CTL_5—V Control 5	257
	4.3.9 TV_V_CTL_6—V Control 6	258
	4.3.10 TV_V_CTL_7—V Control 7	258
4.4	Subcarrier Control Registers	259
	4.4.1 TV_SC_CTL_1—SC Control 1	259
	4.4.2 TV_SC_CTL_2—SC Control 2	259
	4.4.3 TV_SC_CTL_3—SC Control 3	260
4.5	Window Control Registers	260
	4.5.1 TV_WIN_POS—Window Position Register	260
	4.5.2 TV_WIN_SIZE—Window Size Register	261
4.6	Filter Control Registers	261
	4.6.1 TV_FILTER_CTL_1—Filter Control Register 1	261
	4.6.2 TV_FILTER_CTL_2—Filter Control Register 2	262
	4.6.3 TV_FILTER_CTL_3—Filter Control Register 3	263
	4.6.4 SIN_ROM—Sin ROM	263
4.7	Closed Captioning Registers	264
	4.7.1 TV_CC_CTL—CC Control Register	264
	4.7.2 TV_CC_DATA1—CC Data Register Field 1	265



	4.7.3	TV_CC_DATA2—CC Data Register Field 2	265
4.8		PAL Wide Screen Signaling (WSS) Registers	266
	4.8.1	TV_WSS_CTL—WSS Control Register	266
	4.8.2	TV_WSS_DATA—WSS Data Register Field 1	266
4.9		Filter Coefficient Registers	267
	4.9.1	TV_H_LUMA—H Filter Luma Coeffs	268
	4.9.2	TV_H_CHROMA—H Filter Chroma Coeffs	268
	4.9.3	TV_V_LUMA—V Filter Luma Coeffs	268
	4.9.4	TV_V_CHROMA—V Filter Chroma Coeffs	268
5		TV-Out Programming Guide	269
5.1		TV-Out Register Descriptions	269
5.2		TV-Out Programming	271
	5.2.1	Television Standards	271
		5.2.1.1 Timing tables	271
		5.2.1.2 Subcarrier Programming	274
		5.2.1.3 Color Space Conversion/Blank and Black Level Programming	275
	5.2.2	Underscanning and Pixel Aspect Ratios	277
	5.2.3	Programming Filter Coefficients	277
	5.2.4	Setting the Mode	277
		5.2.4.1 Hi-Res Mode Table	278
		5.2.4.2 Hi-Res Modes	278
		5.2.4.3 3 Tap Mode	279
		5.2.4.4 VGA Modes	279
		5.2.4.5 Test Modes	279
	5.2.5	Detection and Determination of the Load	281
		5.2.5.1 Polling Based Load Connect Detection	281
		5.2.5.2 Interrupt Based Load Connect Detection (with hotplug timer) (Not on [DevCL])	281
		5.2.5.3 Interrupt Based Load Connect Detection (without hotplug timer)	282
		5.2.5.4 Load Type Determination	282
		5.2.5.5 Disconnect Detection	282
		5.2.5.6 Alternate methods	282
5.3		PCB Connector and Dongle Design	283
6		VGA and Extended VGA Registers (00000h–00FFFh)	285
6.1		General Control and Status Registers	287
	6.1.1	ST00—Input Status 0	288
	6.1.2	ST01—Input Status 1	289
	6.1.3	FCR—Feature Control	290
	6.1.4	MSR—Miscellaneous Output	291
6.2		Sequencer Registers	293
	6.2.1	SRX—Sequencer Index	293
	6.2.2	SR00—Sequencer Reset	293
	6.2.3	SR01—Clocking Mode	294
	6.2.4	SR02—Plane/Map Mask	295
	6.2.5	SR03—Character Font	296
	6.2.6	SR04—Memory Mode Register	297
	6.2.7	SR07—Horizontal Character Counter Reset	298
6.3		Graphics Controller Registers	298
	6.3.1	GRX—GRX Graphics Controller Index Register	298
	6.3.2	GR00—Set/Reset Register	299



6.3.3	GR01—Enable Set/Reset Register	299
6.3.4	GR02—Color Compare Register	300
6.3.5	GR03—Data Rotate Register	300
6.3.6	GR04—Read Plane Select Register	301
6.3.7	GR05—Graphics Mode Register	301
6.3.8	GR06—Miscellaneous Register	304
6.3.9	GR07—Color Don't Care Register	305
6.3.10	GR08—Bit Mask Register	305
6.3.11	GR10—Address Mapping	306
6.3.12	GR11—Page Selector	308
6.3.13	GR18—Software Flags	308
6.4	Attribute Controller Registers	308
6.4.1	ARX—Attribute Controller Index Register	309
6.4.2	AR[00:0F]—Palette Registers [0:F]	310
6.4.3	AR10—Mode Control Register	310
6.4.4	AR11—Overscan Color Register	312
6.4.5	AR12—Memory Plane Enable Register	312
6.4.6	AR13—Horizontal Pixel Panning Register	313
6.4.7	AR14—Color Select Register	314
6.5	VGA Color Palette Registers	314
6.5.1	DACMASK—Pixel Data Mask Register	315
6.5.2	DACSTATE—DAC State Register	316
6.5.3	DACRX—Palette Read Index Register	316
6.5.4	DACWX—Palette Write Index Register	317
6.5.5	DACDATA—Palette Data Register	317
6.6	CRT Controller Register	318
6.6.1	CRX—CRT Controller Index Register	319
6.6.2	CR00—Horizontal Total Register	320
6.6.3	CR01—Horizontal Display Enable End Register	320
6.6.4	CR02—Horizontal Blanking Start Register	320
6.6.5	CR03—Horizontal Blanking End Register	321
6.6.6	CR04—Horizontal Sync Start Register	321
6.6.7	CR05—Horizontal Sync End Register	322
6.6.8	CR06—Vertical Total Register	322
6.6.9	CR07—Overflow Register (Vertical)	323
6.6.10	CR08—Preset Row Scan Register	325
6.6.11	CR09—Maximum Scan Line Register	326
6.6.12	CR0A—Text Cursor Start Register	327
6.6.13	CR0B—Text Cursor End Register	328
6.6.14	CR0C—Start Address High Register	328
6.6.15	CR0D—Start Address Low Register	329
6.6.16	CR0E—Text Cursor Location High Register	329
6.6.17	CR0F—Text Cursor Location Low Register	329
6.6.18	CR10—Vertical Sync Start Register	330
6.6.19	CR11—Vertical Sync End Register	330
6.6.20	CR12—Vertical Display Enable End Register	331
6.6.21	CR13—Offset Register	331
6.6.22	CR14—Underline Location Register	332
6.6.23	CR15—Vertical Blanking Start Register	333
6.6.24	CR16—Vertical Blanking End Register	333
6.6.25	CR17—CRT Mode Control	334
6.6.26	CR18—Line Compare Register	337
6.6.27	CR22—Memory Read Latch Data Register	337
6.6.28	CR24—Test Register for Toggle State of Attribute Controller Register	338





Figures

Figure 2-1. Programming of the Piecewise-linear Estimation of Gamma Correction Curve.....	188
Figure 3-1. Overlay Register Memory Map	194
Figure 3-2. Programming of the Piecewise-linear Estimation of Gamma Correction Curve.....	201

Tables

Table 1-1. Supported Chipsets	14
Table 2-1. Mode Switch Sequences.....	32
Table 2-2. Simultaneous Display Capabilities on a Single Display Pipe.....	48
Table 2-3. 8-Bit Mode	75
Table 2-4. 10-bit Mode (Even DWord)	77
Table 2-5. 10-bit Mode (Odd DWord)	77
Table 2-6. Cursor Mode Select	141
Table 2-7. Cursor Mode Select	146
Table 3-1. Overlay Register Categories.....	194
Table 3-2. Overlay Memory Offset Registers	203
Table 3-3. Overlay Buffer Start Address Alignment Restriction	204
Table 3-4. Vertical Initial Phase (for 1:1 scaling, upper left sync)	209
Table 3-5. Horizontal Initial Phase (for 1:1 scaling, upper left sync)	210
Table 3-6. Destination Color Key Value Programming in Each Supported Graphics Mode222	
Table 3-7. Destination Color Key Mask Programming in Each Supported Graphics Mode224	
Table 6-1. Analog CRT Display Sync Polarities	292
Table 6-2. Memory Address Counter Address Bits [15:0]	336
Table 6-3. Frame Buffer Address Decoder	336



Revision History

Document Number	Revision Number	Description	Revision Date
3	1.0c	Initial release.	January 2008

§§



1 Introduction

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

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

Table 1-1. Supported Chipsets

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

NOTES:

1. Unless otherwise specified, the information in this document applies to all of the devices mentioned in Table 1-1. For Information that does not apply to all devices, the Device Tag is used.
2. Throughout the PRM, references to "All" in a project field refers 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 covered in this volume.

- **PRM, Volume 4: Subsystem and Cores**

Volume 4 describes the GMCH programmable cores, or EUs, and the “shared functions”, which are shared by more than one EU and perform functions such as I/O and complex math functions.

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

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

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



1.1 Notations and Conventions

1.1.1 Reserved Bits and Software Compatibility

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

- Do not depend on the states of any reserved bits when testing values of registers that contain such bits. Mask out the reserved bits before testing. Do not depend on the states of any reserved bits when storing to instruction or to a register.
- When loading a register or formatting an instruction, always load the reserved bits with the values indicated in the documentation, if any, or reload them with the values previously read from the register.

1.2 Terminology

Term	Abbr.	Definition
3D Pipeline	—	One of the two pipelines supported in the GPE. The 3D pipeline is a set of fixed-function units arranged in a pipelined fashion, which process 3D-related commands by spawning EU threads. Typically this processing includes rendering primitives. See <i>3D Pipeline</i> .
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 “perspective 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.
Destination Width	—	The size of each of (possibly) many elements of the destination of a Gen4 SIMD instruction.



Term	Abbr.	Definition
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 <i>Strips and Fans</i> .
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.
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.



Term	Abbr.	Definition
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 <i>Graphics Processing Engine</i> .
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 <i>Geometry Shader</i> .
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 bits 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.
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.



Term	Abbr.	Definition
In-loop Deblocking Filter	ILDB	The deblocking filter operation in the decoding loop. It is a stage after MC in the video decoding pipe.
Instance	—	In the context of the VF unit, an instance is one of a sequence of sets of similar primitive data. Each set has identical vertex data but may have unique instance data that differentiates it from other sets in the sequence.
Instruction	—	Data in memory directing an EU operation. Instructions are fetched from memory, stored in a cache and executed on one or more Gen4 cores. Not to be confused with commands which are fetched and parsed by the command streamer and dispatched down the 3D or Media pipeline.
Instruction Pointer	IP	The address (really an offset) of the instruction currently being fetched by an EU. Each EU has its own IP.
Instruction Set Architecture	ISA	The GEN4 ISA describes the instructions supported by a GEN4 EU.
Instruction State Cache	ISC	On-chip memory that holds recently-used instructions and state variable values.
Interface Descriptor	—	Media analog of a State Descriptor.
Intermediate Z	IZ	Completion of the Z (depth) test at the front end of the Windower/Masker unit when certain conditions are met (no alpha, no pixel-shader computed Z values, etc.)
Inverse Discrete Cosine Transform	IDCT	The stage in the video decoding pipe between IQ and MC
Inverse Quantization	IQ	A stage in the video decoding pipe between IS and IDCT.
Inverse Scan	IS	A stage in the video decoding pipe between VLD and IQ. In this stage, a sequence of none-zero DCT coefficients are converted into a block (e.g. an 8x8 block) of coefficients. VFE unit has fixed functions to support IS for MPEG-2.
Jitter	—	Just-in-time compiler.
Kernel	—	A sequence of Gen4 instructions that is logically part of the driver or generated by the jitter. Differentiated from a Shader which is an application supplied program that is translated by the jitter to Gen4 instructions.
Least Significant Bit	LSB	Least Significant Bit
MathBox	—	See Extended Math Unit
Media	—	Term for operations such as video decode and encode that are normally performed by the Media pipeline.
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 discontinuous register data to be fetched and used as a single operand.
Render Cache	RC	Cache in which pixel color and depth information is written prior to being written to memory, and where prior pixel destination attributes are read in preparation for blending and Z test.
Render Target	RT	A destination surface in memory where render results are written.
Render Target Array Index	—	Selector of which of several render targets the current operation is targeting.
Root Thread	—	A root-node thread. A thread corresponds to a root-node in a thread generation hierarchy. It is a kind of thread associated with the media fixed function pipeline. A root thread is originated from the VFE unit and forwarded to the Thread Dispatcher by the TS unit. A root thread may or may not have child threads. A root thread may have scratch memory managed by TS. A root thread with children has its URB resource managed by the VFE.
Sampler	—	Shared function that samples textures and reads data from buffers on behalf of EU programs.
Scratch Space	—	Memory allocated to the subsystem that is used by EU threads for data storage that exceeds their register allocation, persistent storage, storage of mask stack entries beyond the first 16, etc.
Shader	—	A Gen4 program that is supplied by the application in an high level shader language, and translated to Gen4 instructions by the jitter.
Shared Function	SF	Function unit that is shared by EUs. EUs send messages to shared functions; they consume the data and may return a result. The Sampler, Data Port and Extended Math unit are all shared functions.
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, <i>r2</i> , is 256-bit wide, 256-bit aligned register. A sub-register, <i>r2.3:d</i> , is the fourth dword of GRF register <i>r2</i> .
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
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.



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



Term	Abbr.	Definition
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 *Display Registers*

2.1 Introduction and Register Summary

This chapter contains the register descriptions for the display portion of a family of integrated graphics devices. These registers do vary by devices within the family of devices so special attention needs to be paid to which devices use which registers and register fields.

Different devices within the family may add, modify, or delete registers or register fields relative to another device in the same family based on the supported functions of that device. This document covers both desktop and mobile products.

The following table contains the sections break down where the register information is contained within this chapter:

Address Range	Description
30000h–3FFFFh	Overlay Registers
05000h–05FFFh	GMBUS and I/O Control
06000h–06FFFh	Display Clocks and Clock Gating
0A000h–0AFFFh	Display Palette A/B Registers
60000h–600FFh	Display Pipeline A
61000h–610FFh	Display Pipeline B
61100h–611FFh	Display Port Control Registers
61200h–612FFh	Panel Fitting/Power Sequencing/LVDS Control
62000h–62FFFh	Reserved
64000h–64FFFh	DisplayPort Registers
70000h–7FFFFh	Display Pipeline, Display Planes, Cursor Planes, and VGA Control Registers



2.1.1 Panel Control Register Summary ([DevCL] Only)

Address	Function	Name	Normal Access	Panel Write Protect
61180h	Port Control	LVDS Port Control	RW	Yes
61200h	Panel Power Sequence	Panel power status	RO	NA
61204h		Panel Power Control	RW	No
61208h		Panel power on sequencing delay	RW	Yes
6120Ch		Panel power off sequencing delay	RW	Yes
61210h		Panel power cycle Delay and Reference	RW	Yes
61254h	Backlight Control	Backlight PWM Control	RW	No
61260h	BLM	Image BLM Control	RW	No
61270h – 61284h		BLM Thresholds	RW	No
61290h–612A4h		BLM Accumulators	RO	NA

2.1.2 Display Pipe and Plane Control Registers

Address Range	Description
70000h – 70027h	Display Pipeline A Control
70028h – 7007Fh	Reserved
70080h – 7009Fh	Display Cursor Plane Registers A
700A0h – 700BFh	Reserved
700C0h – 70FFFh	Reserved
700C0h – 700DFh	Display Cursor Plane Registers B
700E0h – 70FFFh	Reserved
71000h – 7100Ch	Display Pipeline B Control
71010h – 7107Fh	Reserved
70180h – 7018Ch	Display A Plane Control
71090h – 7117Fh	Reserved
71180h – 7119Bh	Display B/Sprite Plane Control
711D0h – 713FFh	Reserved
71400h – 7140Fh	Video BIOS Registers
72010h – 720FFh	Reserved
72100h – 7217Fh	Reserved
72180h – 7219Fh	Display C/Sprite Plane Control
721D0h – 721F7h	Display C Color Adjustment
72000h – 72FFFh	Reserved



2.1.3 Terminology

Description	Software Use	Should be implemented as
Reserved write as zero.	Software must always write a zero to these bits. This allows new features to be added using these bits that will be disabled when using old software and as the default case.	These are read-only bits that always read as zeros or r/w bits that are default to zero.
Reserved write as one.	Software must always write a one to these bits. This allows new features to be added using these bits that will be disabled when using old software and as the default case.	
Reserved for BIOS Do not change	Driver access to these bits must read these bits that have been set through an initialization operation before writing this register so that the bits can remain unchanged.	According to each specific bit
Reserved for Video BIOS	These register bits will be used only by video BIOS and drivers should not change them.	These are read/write bits that have no hardware function. They are intended for use by the video BIOS for storage.
Reserved for Compatibility	For functions that are no longer needed these bits had old use, but now does nothing. New software should use the new method.	Read/write bits that have no functions.
Use for compatibility only	Under specific conditions, these bits functions as in the old part, new software should use the new method.	According to each specific bit
Read-Only	This bit is read-only. The read value is determined by hardware. Writes to this bit have no effect.	According to each specific bit. The bit value is determined by hardware and not affected by register writes to the actual bit.
Reserved read-only	Don't assume a value for these bits. Writes have no effect.	These bits should read as zero.
Reserved read-only write as zero	Don't assume a value for these bits, always write a zero.	These bits should read as zero.
Read/Clear	This bit can be read and writes to it with a one cause the bit to clear.	Hardware events cause the bit to be set and the bit will be cleared on a write operation where the corresponding bit has a one for a value.
Read/Write	This bit can be read or written.	
Double Buffered	Write when desired	Takes effect only after a particular event such as a VBLANK.



2.1.4 Register Protection for Panel Protection

2.2 Display Mode Set Sequence

2.2.1 Pipe register double-buffering

Pipe config registers (0x70008 0x71008):

Contain pipe enable, double wide, gamma mode, interlaced mode, plane enable overrides, frame start delay, force border.

Pipe timing registers (60000/4/8/C/10/14/1C 61000/4/8/C/10/14/1C):

Contain horizontal and vertical total, active, blank start/end, sync start/end, source size.

	Pipe Register Double-Buffering (double-buffering to start of vertical blank)
Pipe config register	Updates at Vblank, or updates if pipe completely off, or updates if VGA native and enabled.
Pipe timing registers	Updates at Vblank after pipe config enable written 0 to 1, or updates if DPLL off or pipe completely off and pipe config register written.

Care must be taken when disabling the pipe. The pipe will not completely turn off until the start of vertical blank after the pipe enable was written to 0. If there is no vertical blank the pipe will not be able to completely turn off. Double-buffering is bypassed if VGA native display is enabled (0x71400 bits 31,25,24 set to 0), allowing pipe to turn off immediately.

Enabling pipe always occurs instantly.



2.2.2 Mode Switch Programming Sequences

General rules to follow:

- DPLL must be enabled and warmed up before pipe or ports are enabled.
- DPLL must be kept enabled until ports are disabled and pipe is completely off.
- DPLL frequency must not be changed until ports are disabled and pipe is completely off, except when in native VGA where SR01 25/28 MHz select can be changed.
- Planes must be disabled before pipe is disabled or pipe timings changed.
- Panelfitter must be enabled or disabled only when pipe is completely off.
- Set port multiply when programming the DPLL.
- The internal TV and CRT ports can be left on during a mode switch if DPLL is not touched.
- Ports can be freely enabled or disabled on a running pipe, except when port multiply needs to be changed.

Wait for pipe off status (using pipe config register bit 30) is the best way to find when pipe is completely off.

Table 2-1. Mode Switch Sequences

Enable sequence
Pipe A must be completely off at this point Write PIPEACONF bits[19:18] = 00 Write DSPACNTR bit[31] = 1 Write DSPASURF = 0x00000000 Write DSPACNTR bit[31] = 0 Write DSPASURF = 0x00000000 Restore PIPEACONF bits[19:18] to original value Program DPLL Enable DPLL Wait for DPLL warmup 150us (Wait ensures clock is running smoothly before enabling pipe) Program pipe timings (Can be done before DPLL programming) Enable panelfitter as needed (Can be done before DPLL and/or pipe timings programming) Enable pipe Enable planes (VGA or hires) Enable ports



Disable sequence
Disable sDVO ports' stall input by clearing 0x61140 and 0x61160 bit 29 to 0 (port enable not changed) Program sDVO ADD device Disable ports Disable planes (VGA or hires) Disable pipe Disable VGA display in 0x71400 bit 31 (Disable VGA display done after disable pipe to allow pipe to turn off when no vblank is available in native VGA mode) If Gen4 { Wait for pipe off status } (Wait ensures planes and pipe have completely turned off prior to disabling panelfitter then DPLL) Disable panelfitter Disable DPLL
Pipe timings change or change between VGA native or VGA center / upperleft or Hires
Use complete disable sequence followed by complete enable sequence with new mode programmings.
VGA native timings change
Program VGA and SR01 25/28 MHz select registers as needed.



2.3 Display Power Down / Up Register Access Sequence

2.3.1 Power Up / Down for CRT-like Display Devices

For products driving a display or transcoder that require changing display timings for a mode change, the programming sequence outlined below should be used during a power up or power down sequence. Examples of this type of display are a CRT device, or one that uses an external scaler but not fixed timings (such as an external TV encoder). Only registers that are double buffered should be updated while the display pipe is enabled in order to avoid screen glitches.

Power down sequence:

1. Turn off the port(s) assigned to that pipe.
2. Turn off display planes that are assigned to the display pipe by writing the enable bit with a zero.
3. Turn off the display pipe (will disable at the next VBLANK).

Power up sequence:

4. Program the pipe timing, source size registers and DPLL register values to the desired mode values.
5. Configure and turn on the planes that are to be assigned to that pipe.
6. Turn on the port(s) assigned to that pipe.

2.3.2 Power Up / Down for Integrated and External Panel Scaler Driven Display Devices

For products driving an integrated panel or a device that operates through a scaler function using fixed timings, the sequence outlined below should be used to power down or up. The screen does not have to be powered down during this mode change operation.

Power down sequence:

1. Turn off the port(s) assigned to that pipe.
2. Turn off display planes that are assigned to the target display pipe (VGA display disabled if VGA mode).
3. Turn off the display pipe (will disable at the next VBLANK).

Power up sequence:

4. Program the pipe timing, source size registers and DPLL register values to the desired mode values.
5. Configure and turn on the plane engines that are to be assigned to that pipe (VGA display enabled if going to a VGA mode, centering, pixel doubling or panel fitting appropriately enabled).
6. Turn on the port(s) assigned to that pipe.



2.4 GMBUS and I/O Control Registers (05000h–05FFFh)

2.4.1 GPIO Pin Usage (By Functions)

GPIO pins allow the support of simple query and control functions such as DDC and I²C interface protocols. GPIO pins exist in pairs (for the most part) and provide a mechanism to control external devices through a register programming interface. GPIO pins can be set to a level or the value of the pin can be read. This allows for a “bit banging” version of an I2C interface to be implemented. An additional function of using the GMBUS engine to run the I2C protocols is also allowed. Refer to the EDS for GPIO signal descriptions. Refer to the *Philips I2C-BUS SPECIFICATION version 2.1* for a description of the I2C bus and protocol.

The number and names of the GPIO pins vary from device type to device type. Some of the GPIO pins will be muxed with other functions and are only available when the other function is not being used. The following subsections describe the GPIO pin to register mapping for the various devices. OEMs have the ability to remap these functions onto other pins as long as the hardware limitations are observed.

2.4.2 GPIO Pin Usage (By Device)

Port	Pin Use (Name)	GMBUS Use	sDVO Use	Internal Pullup	I ² C	Device	Description
7	TVDCONSEL1	Yes	No	No	Yes	All with integrated TV	Used for interface to glue logic generating D-connector HV levels. Default value is tri-stated.
	TVDCONSEL0				Yes		
6	Reserved	No	No	No			
	Reserved						
5	Reserved	No	No	No	No		Reserved
	Reserved				No		Reserved
4	SDVO/HDMIB CTLDATA	Yes	Yes	No – weak pulldown on reset	Yes	All	Used for programming SDVO device via GMBUS protocol. The SDVO device provides extension of this to the EEROM, DDC.
	SDVO/HDMIB CTLCLK				Yes		
3	Reserved	No	No	No	No		Reserved
	Reserved				No		Reserved



Port	Pin Use (Name)	GMBUS Use	sDVO Use	Internal Pullup	I ² C	Device	Description
2	LVDS/DPD DDC Data (DDCPDATA)	Yes	No	No – weak pulldown on reset	Yes	All	DDC for Digital Display connection via the integrated LVDS or DP port D.
	LVDS/DPD DDC Clock (DDCPCLK)				Yes		
1	I2C Data (LCLKCTRLB)	Yes	No	No	Yes	All	For control of SSC clock generator devices on motherboard. Support can be optionally I2C or control level.
	I2C Clock (LCLKCTRLA)				Yes		
0	DAC DDC Data (DDCADATA)	Yes	No	No	Yes	All	DDC for Analog monitor (VGA) connection. This cannot be shared with other DDC or I2C pairs due to legacy monitor issues.
	DAC DDC Clock (DDCACLK)				Yes		



2.4.3 GPIO Pin Glue Logic

The GPIO output can be converted to alternate levels through use of external logic on the board. This section describes conversion logic that will be implemented on the board.

2.4.3.1 D-Connector

The D-connector requires multiple voltage levels to select modes on a display device. The following table describes the translation as done by the board glue logic from the GPIO pins to the video modes. When the GPIO pins are in their default, tri-stated state, the D-connector default value will drive zeros on all three lines, indicating 480i at 4:3.

GPIO Pins			D-Connector Lines			Video Format
TVDCONSEL0	TVDCONSEL1	TVDCONSEL2	Line 1 Voltage (V)	Line 2 Voltage (V)	Line 3 Voltage (V)	
0	0	0	5	0	5	1080i, 16:9
0	0	1	2.2	5	5	720p, 16:9
0	1	0	0	0	5	480i, 16:9
0	1	1	0	5	5	480p, 16:9
1	0	0	0	0	2.2	480i, 4:3L
1	0	1	0	5	2.2	480p, 4:3L
1	1	0	0	0	0	480i, 4:3
1	1	1	0	5	0	480p, 4:3



2.4.4 GPIO Control Registers

The number of registers and their usage may change with each product.

Address offset:	05000h–05003h	Reserved
	05004h–05007h	Reserved
	05008h–0500Ch	Reserved
	05010h–05013h	GPIOCTL_0
	05014h–05017h	GPIOCTL_1
	05018h–0501Bh,	GPIOCTL_2
	0501Ch–0501Fh,	GPIOCTL_3
	05020h–05023h,	GPIOCTL_4
	05024h–05027h	GPIOCTL_5
	05028h–0502Bh	GPIOCTL_6
	0502Ch–0502Fh	GPIOCTL_7
	Default value:	00h, 00h, 000U1000b, 000U1000b
	Normal Access:	Read / Write
Size:		32 bit

These registers define the control of the sets of the so called “general purpose” I/O pins. Each register controls a pair of pins that while can be used for general purpose control, most are designated for specific functions according to the requirements of the device and the system that the device finds itself in. Each pin of the two-pin pair is designated as a clock or data for descriptive purposes. See the table at the beginning of this section to determine for each product which pins/registers are supported and their intended functions. **Board design variations are possible and would affect the usage of these pins.**

GMBUS port 5 is reserved for LT visual status indication and is controlled by hardware only. It is not accessible or programmable through its associated GPIO register which should be considered reserved.

For devices with a PCI Express bus and in the case of the pins that are multiplexed with PCI Express signals, the registers that control those pins should only be utilized if the Digital Port B detected bit in the SDVO/HDMI control register is set to 1.

Bit	Description
31:13	Reserved
12	GPIO_Data In—RO: This is the value that is sampled on the GPIO_Data pin as an input. This input is synchronized to the Core Clock domain. Because the default setting is this buffer is an input, this bit is undefined at reset.
11	GPIO Data Value—R/W: This is the value that should be place on the GPIO Data pin as an output. This value is only written into the register if GPIO DATA MASK is also asserted. The value will appear on the pin if this data value is actually written to this register and the GPIO Data DIRECTION VALUE contains a value that will configure the pin as an output. Default = 1. The GPIO default clock data value is programmed to ‘1’ in hardware. The hardware drives a default of ‘1’ since the I2C interface defaults to a ‘1’. (this mimics the I2C external pull-ups on the bus)



Bit	Description
10	GPIO Data Mask—WO: This is a mask bit to determine whether the GPIO DATA VALUE bit should be written into the register. This value is not stored and when read returns 0. 0 = Do NOT write GPIO Data Value bit (default). 1 = Write GPIO Data Value bit.
9	GPIO Data Direction Value—R/W: This is the value that should be used to define the output enable of the GPIO Data pin. This value is only written into the register if GPIO Data DIRECTION MASK is also asserted. The value that will appear on the pin is defined by what is in the register for the GPIO DATA VALUE bit. 0 = Pin is configured as an input (default) 1 = Pin is configured as an output.
8	GPIO Data Direction Mask—WO: This is a mask bit to determine whether the GPIO DIRECTION VALUE bit should be written into the register. This value is not stored and when read always returns 0. 0 = Do NOT write GPIO Data Direction Value bit (default). 1 = Write GPIO Data Direction Value bit.
7:5	Reserved : MBZ
4	GPIO Clock Data In—RO: This is the value that is sampled on the GPIO Clock pin as an input. This input is synchronized to the Core Clock domain. Because the default setting is this buffer is an input, this bit is undefined at reset.
3	GPIO Clock Data Value—R/W: This is the value that should be place on the GPIO Clk pin as an output. This value is only written into the register if GPIO Clock DATA MASK is also asserted. The value will appear on the pin if this data value is actually written to this register and the GPIO Clock DIRECTION VALUE contains a value that will configure the pin as an output. Default = 1. The GPIO default clock data value is programmed to '1' in hardware. The hardware drives a default of '1' since the I2C interface defaults to a '1'. (this mimics the I2C external pull-ups on the bus)
2	GPIO Clock Data Mask—WO: This is a mask bit to determine whether the GPIO Clock DATA VALUE bit should be written into the register. This value is not stored and when read always returns 0. 0 = Do NOT write GPIO Clock Data Value bit (default). 1 = Write GPIO Clock Data Value bit.
1	GPIO Clock Direction Value—R/W: This is the value that should be used to define the output enable of the GPIO Clock pin. This value is only written into the register if GPIO Clock DIRECTION MASK is also asserted. The value that will appear on the pin is defined by what is in the register for the GPIO Clock DATA VALUE bit. 0 = Pin is configured as an input and the output driver is set to tri-state (default) 1 = Pin is configured as an output.
0	GPIO Clock Direction Mask—WO: This is a mask bit to determine whether the GPIO Clock DIRECTION VALUE bit should be written into the register. This value is not stored and when read returns 0. 0 = Do NOT update the GPIO Clock Direction Value bit on a write (default). 1 = Update the GPIO Clock Direction Value bit. on a write operation to this register.



2.4.5 GMBUS Controller Programming Interface

The GMBUS (Graphic Management Bus) can be used to indirectly access/control devices connected to a GMBUS bus as an alternate to bit-wise programming via software.

The GMBUS interface is I²C compatible. The basic features are listed as follow:

1. Works as the master of a single master bus.
2. The bus clock frequency is selectable by software to be 50KHz, 100KHz, 400KHz , and 1MHz
3. The GMBUS controller can be attached to the selected GPIO pin pairs.
4. 7 or 10-Bit Slave Address and 8- or 16-bit index.
5. Hardware byte counter to track the data transmissions/reception
6. Timing source from 250MHz ungated PCI-Express clock.
7. There is a double buffered data register and a 9 bit counter to support 0 byte to 256 byte transfers.
8. The slave device can cause a stall by pulling down the clock line (Slave Stall), or delay the slave acknowledge response.
9. The master controller detects and reports time out conditions for a stall from a slave device or delayed or missing slave acknowledge.
10. Interrupt may optionally be generated from a GMBUS Timeout error.

The byte counter register is a read/write register, and in receiving mode, is used to track the data bytes received. There is a status register to indicate the error condition, data buffer busy, time out, and data complete acknowledgement.

Note: There is no support for ring buffer based operation of GMBUS. The GMBUS is controlled by a set of memory mapped IO registers. Status is reported through the GMBUS status register.



2.4.6 GMBUS0—GMBUS Clock / Port Select

Address Offset: 5100h
 Default Value: 0000 0000h
 Normal Access: Read/Write
 Size: 32 bits
 Double Buffered: No

The GMBUS0 register will set the clock rate of the serial bus and the device the controller is connected to. The clock rate options are 50 KHz, 100 KHz, 400 KHz, and 1MHz. This register should be set before the first data valid bit is set, because it will be read only at the very first data valid bit, and not read during the period of the transmission until stop is issued and next first data valid bit is set.

Bit	Description
31:11	Reserved
10:8	GMBUS Rate Select: These two bits select the rate that the GMBUS will run at. It also defines the AC timing parameters used. It should only be changed when between transfers when the GMBUS is idle. 1xx = Reserved. 000 = 100 KHz 001 = 50 KHz 010 = 400 KHz 011 = 1 MHz for SDVO
7	Hold Time extension: This bit selects the hold time on the data line driven from the GMCH. 0 = Hold time of 0ns 1 = Hold time of 300 ns
6:3	Reserved
2:0	Pin Pair Select: This field selects an GMBUS pin pair for use in the GMBUS communication. Use the table above to determine which pin pairs are available for a particular device and the intended function of that pin pair. Note that it is not a straight forward mapping of port numbers to pair select numbers. 000 = None (disabled) 001 = Dedicated Control/GMBUS Pins [DevCL] – LCTRLCKA, LCTRLCLKB SSC Clock Device 010 = Dedicated Analog Monitor DDC Pins (DDC1DATA, DDC1CLK) 011 = [DevCL] - Integrated Digital Panel DDC Pins, LVDS or DP D 100 = Reserved 101 = sDVO/HDMI Use 110 = Reserved 111 = D connector control signals

2.4.7 GMBUS1—GMBUS Command / Status

Address Offset: 5104h



Default Value: 0000 0000h
 Normal Access: Read/Write (Write Protect except bit 31)
 Size: 32 bits
 Double Buffered: no

This register lets the software indicate to the GMBUS controller the slave device address, register index, and indicate when the data write is complete.

When the **SW_CLR_INT** bit is asserted, all writes to the GMBUS2, GMBUS3, and GMBUS4 registers are discarded. The GMBUS1 register writes to any other bit except the **SW_CLR_INT** are also lost. Reads to these registers always work normally regardless of the state of the **SW_CLR_INT** bit.

Bit	Description
31	Software Clear Interrupt (SW_CLR_INT): This bit must be clear for normal operation. Setting the bit , then clearing it acts as local reset to the GMBUS controller. This bit is commonly used by software to clear a BUS_ERROR when a slave device delivers a NACK. 0 = If this bit is written as a zero when its current state is a one, will clear the HW_RDY bit and allows register writes to be accepted to the GMBUS registers (Write Protect Off). This bit is cleared to zero when an event causes the HW_RDY bit transition to occur. 1 = Asserted by software after servicing the GMBUS interrupt. Setting this bit causes the INT status bit to be cleared. Setting (1) this bit also asserts the HW_RDY bit (until this bit is written with a 0). When this bit is set, no writes to GMBUS registers will cause the contents to change with the exception of this bit which can be written.
30	Software Ready (SW_RDY): Data handshake bit used in conjunction with HW_RDY bit. 0 = De-asserted via the assertion event for HW_RDY bit 1 = When asserted by software, results in de-assertion of HW_RDY bit
29	Enable Timeout (ENT) Enables timeout for slave response. When this bit is enabled and the slave device response has exceeded the timeout period, the GMBUS Slave Stall Timeout Error interrupt bit is set. 0 = disable timeout counter 1 = enable timeout counter
28	Reserved



Bit	Description
27:25	Bus Cycle Select 000 = No GMBUS cycle is generated. 001 = GMBUS cycle is generated without an INDEX, with no STOP, and ends with a WAIT 010 = Reserved 011 = GMBUS cycle is generated with an INDEX, with no STOP, and ends with a WAIT 100 = Generates a STOP if currently in a WAIT or after the completion of the current byte if active. 101 = GMBUS cycle is generated without an INDEX and with a STOP 110 = Reserved 111 = GMBUS cycle is generated with an INDEX and with a STOP GMBUS cycle will always consist of a START followed by Slave Address, followed by an optional read or write data phase. A read cycle with an index will consist of a START followed by a Slave Address a WRITE indication and the INDEX and then a RESTART with a Slave Address and an optional read data phase. The GMBUS cycle will terminate either with a STOP or by entering a wait state. The WAIT state is exited by generating a STOP or by starting another GMBUS cycle. This can only cause a STOP to be generated if a GMBUS cycle is generated, the GMBUS is currently in a data phase, or it is in a WAIT phase: Note that the three bits can be decoded as follows: 27 = STOP generated 26 = INDEX used 25 = cycle ends in a WAIT
24:16	Total Byte Count (9-bits). This determines the total number of bytes to be transferred during the DATA phase of a GMBUS cycle. The DATA phase can be prematurely terminated by generating a STOP while in the DATA phase (see Bus Cycle Select). Do not change the value of this field during GMBUS cycles transactions.
15:8	8-bit GMBUS Slave Register Index (INDEX): This field specifies the 8-bits of index to be used for the generated bus write transaction or the index used for the WRITE portion of the WRITE/READ pair. It only has an effect if the enable Index bit is set. Do not change this field during a GMBUS transaction.
7:1	7-bit GMBUS Slave Address (SADDR): When a GMBUS cycle is to be generated using the Bus Cycle Select field, this field specifies the value of the slave address that is to be sent out. For use with 10-bit slave address devices, set this value to 11110XXb (where the last two bits (xx) are the two MSBs of the 10-bit address) and the slave direction bit to a write. This is followed by the first data byte being the 8 LSBs of the 10-bit slave address. Special Slave Addresses 0000 000R = General Call Address 0000 000W = Start byte 0000 001x = CBUS Address 0000 010x = Reserved 0000 011x = Reserved 0000 1xxx = Reserved 1111 1xxx = Reserved 1111 0xxx = 10-Bit addressing
0	Slave Direction Bit: When a GMBUS cycle is to be generated based on the Bus Cycle Select, this bit determines if the operation will be a read or a write. A read operation with the index enabled will perform a write with just the index followed by a re-start and a read. 1 = Indicates that a Read from the slave device operation is to be performed. 0 = Indicates that a Write to slave device operation is to be performed.



2.4.8 GMBUS2—GMBUS Status Register

Address Offset:	05108h
Default Value:	0000 0800h
Normal Access:	Read/Write (Write Protect)
Size:	32 bits
Double Buffered:	No

Bit	Description
31:16	Reserved
15	INUSE 0 = read operation that contains a zero in this bit position indicates that the GMBUS engine is now acquired and the subsequent reads of this register will now have this bit set. Writing a 0 to this bit has no effect. 1 = read operation that contains a one for this bit indicates that the GMBUS is currently allocated to someone else and "In use". Once set, a write of a 1 to this bit indicates that the software has relinquished the GMBUS resource and will reset the value of this bit to a 0. Software wishing to arbitrate for the GMBUS resource can poll this bit until it reads a zero and will then own usage of the GMBUS controller. This bit has no effect on the hardware, and is only used as semaphore among various independent software threads that don't know how to synchronize their use of this resource that may need to use the GMBUS logic. Writing a one to this bit is software's indication that the software use of this resource is now terminated and it is available for other clients.
14	Hardware Wait Phase (HW_WAIT_PHASE): Read-Only 0 = The GMBUS engine is not in a wait phase. 1 = Set when GMBUS engine is in wait phase. Wait phase is entered at the end of the current transaction when that transaction is selected not to terminate with a STOP. Once in a WAIT_PHASE, the software can now choose to generate a STOP cycle or a repeated start (RESTART) cycle followed by another GMBUS transaction on the GMBUS.
13	Slave Stall Timeout Error Read-Only. This bit indicates that a slave stall timeout has occurred. It is tied to the Enable Timeout (ENT) bit. 0 = No slave timeout has occurred. 1 = A slave acknowledge timeout has occurred
12	GMBUS Interrupt Status Read-Only. This bit indicates that an event that causes a GMBUS interrupt has occurred. 0 = The conditions that could cause a GMBUS interrupt have not occurred or this bit has been cleared by software assertion of the SW_CLR_INT bit. 1 = GMBUS interrupt event occurred. This interrupt must have been one of the types enabled in the GMBUS4 register.



Bit	Description
11	Hardware Ready (HW_RDY): Read-Only. This provides a method of detecting when the current software client routine can proceed with the next step in a sequence of GMBUS operations. This data handshake bit is used in conjunction with the SW_RDY bit. When this bit is changed to asserted by the GMBUS controller, it results in the de-assertion of the SW_RDY bit. 0 = Condition required for assertion has not occurred or when this bit was a one and: 0 SW_RDY bit has been asserted. 1 During a GMBUS read transaction, after the each read of the data register. 2 During a GMBUS write transaction, after each write of the data register. 3 SW_CLR_INT bit has been cleared. 1 = This bit is asserted under the following conditions: a) After a reset or when the transaction is aborted by the setting of the SW_CLR_INT bit. b) When an active GMBUS cycle has terminated with a STOP. c) When during a GMBUS write transaction, the data register needs and can accept another four bytes of data. d) During a GMBUS read transaction, this bit is asserted when the data register has four bytes of new data or the read transaction DATA phase is complete and the data register contains the last few bytes of the read data. This bit resumes to normal operation when the SW_CLR_INT bit is written to a 0.
10	NAK Indicator (was previously called Slave Acknowledge Timeout Error SATOER): Read-Only. 0 = No bus error has been detected or SW_CLR_INT has been written as a zero since the last bus error. 1 = Set by hardware if any expected device acknowledge is not received from the slave within the timeout.
9	GMBUS Active (GA): Read-Only. This is a status bit that indicates whether the GMBUS controller is in an IDLE state or not. 0 = The GMBUS controller is currently IDLE. 1 = This indicates that the bus is in START, ADDRESS, INDEX, DATA, WAIT, or STOP Phase. Set when GMBUS hardware is not IDLE.
8:0	Current Byte Count: Read-Only. Can be used to determine the number of bytes currently transmitted/received by the GMBUS controller hardware. Set to zero at the start of a GMBUS transaction data transfer and incremented after the completion of each byte of the data phase. Note that because reads have internal storage, the byte count on a read operation may be ahead of the data that has been accepted from the data register.



2.4.9 GMBUS3—GMBUS Data Buffer

Address Offset: 0510Ch
 Default Value: 0000 0000h
 Normal Access: Read/Write (Write Protect)
 Size: 32 bits

This is data read/write register. This register is double buffered. Bit 0 is the first bit sent or read, bit 7 is the 8th bit sent or read, all the way through bit 31 being the 32nd bit sent or read. For GMBUS write operations with a non-zero byte count, this register should be written with the data before the GMBUS cycle is initiated. For byte counts that are greater than four bytes, this register will be written with subsequent data only after the HW_RDY status bit is set indicating that the register is now ready for additional data. For GMBUS read operations, software should wait until the HW_RDY bit indicates that the register contains the next set of valid read data before reading this register.

Bit	Description
31:24	Data Byte 3:
23:16	Data Byte 2:
15:8	Data Byte 1:
7:0	Data Byte 0:

2.4.10 GMBUS4—GMBUS Interrupt Mask

Address Offset: 05110h
 Default Value: 0000 0000h
 Normal Access: Read/Write
 Size: 32 bits

Bit	Description
31:5	Reserved
4:0	Interrupt Mask: This field specifies which GMBUS interrupts events may contribute to the setting of gmbus interrupt status bit in second level interrupt status register PIPEASTAT. Bit 4: GMBUS Slave stall timeout Bit 3: GMBUS NAK Bit 2: GMBUS Idle Bit 1: Hardware wait (GMBUS cycle without a stop has completed) Bit 0: Hardware ready (Data has been transferred) 0 = Disable this type of GMBUS interrupt 1 = Enable this type of GMBUS interrupt



2.4.11 GMBUS5—2 Byte Index Register

Address Offset:	05120h
Default Value:	0000h
Normal Access:	Read/Write
Size:	32 bits
Double Buffered:	no

This register provides a method for the software indicate to the GMBUS controller the 2 byte device index.

Bit	Description
31	2 Byte Index Enable: When this bit is asserted (1), then bits 15:00 are used as the index. Bits 15:8 are used in the first byte which is the most significant index bits. The slave index in the GMBUS1<15:8> are ignored. Bits 7:0 are used in the second byte which is the least significant index bits.
30:16	Reserved
15:00	2 Byte Slave Index: This is the 2 byte index used in all GMBUS accesses when bit 31 is asserted (1).

2.5 Display Clock Control Registers (06000h–06FFFh)

The registers described in this section are used across products. However, slight changes may be present in some registers (i.e., for features added or removed), or some registers may be removed entirely.

The following list of registers may contain one or more differences between products. This table is provided for convenience only, please check each instruction separately to determine its exact impact on a specific product implementation.

[DevCL] When one or more display pipes are enabled, the FW3 bit 31 Enable HPLL off during Self Refresh should be disabled before accessing the 6XXXh MMIO register address space. Software must follow these steps:

1. Disable FW3 bit 31 Enable HPLL off during Self Refresh (if enabled and one display pipe is enabled)
2. Wait for next vblank (switch from hrawclk back to cdclk will occur)
3. Access the 6XXXh address space as needed
4. Re-enable FW3 Enable HPLL off during Self Refresh bit 31

Note that the wait on next vblank step requires an enabled display pipe.



Table 2-2. Simultaneous Display Capabilities on a Single Display Pipe

	Integrated LVDS*	SDVO B (non-TV)	SDVO C (non-TV)	SDVO TV	SDVO LVDS	HDMI *	DP	Integrated TV*	CRT
Integrated LVDS*		No (2, 3)	No (2, 3)	No (1, 2, 3)	No (2,)	No (2, 3)	No (2)	No (1)	No (3)
SDVO B (non-TV)	No (2, 3)		Yes	No (1)	No (2)	No (4,5,6)	No (4,5,6)	No (1)	Yes
SDVO C (non-TV)	No (2, 3)	Yes		No (1)	No (2)	No (4,5,6)	No (4,5,6)	No (1)	Yes
SDVO TV	No (1, 2, 3)	No (1)	No (1)		No (2)	No (1)	No (1)	No (1)	No (1)
SDVO LVDS	No (2)	No (2)	No (2)	No (2)		No (2)	No (2)	No (1)	Yes
HDMI *	No (2,3)	No (4,5,6)	No (4,5,6)	No (1)	No (2)		No(6)	No (1)	No (4)
DP	No (2)	No (4,5,6)	No (4,5,6)	No (1)	No (2)	No(6)		No (1)	No (4)
Integrated TV*	No (1)	No (1)	No (1)	No (1)	No (1)	No (1)	No (1)		No (1)
CRT	No (3)	Yes	Yes	No (1)	Yes	No (4)	No (4)	No (1)	

NOTES:

1. TV Timings don't match
 2. No internal LVDS and sDVO or DP on same pipe
 3. No SSC on CRT, DVI, HDMI or TV. DP optionally has SSC.
 4. Only works if DP is in 24bpp mode
 5. DP is multiplexed with SDVO, only works if the SDVO and DP ports are different (one is B and one is C)
 6. Down config only
- * Not available on UMA [DevBW]

Orange = sDVO/HDMI, Green = LVDS ([DevCL] only)

Pixel Data Rate	Dot Clock	Dual Channel?	External Clock	Data Clock Rate	Multiplier
20-25MHz	100-125MHz	NO	100-125MHz	1.0-1.25GHz	5x
25-50MHz	100-200MHz	NO	100-200MHz	1.0-2.0GHz	4x
50-100MHz	100-200MHz	NO	100-200MHz	1.0-2.0GHz	2x
100-270MHz	100-270MHz	NO	100-270MHz	1.0-2.7GHz	1x
25-112MHz	25-112MHz	NO	25-112MHz	175-784MHz	1x
80-224MHz	80-224MHz	NO	80-224MHz	280-784MHz	1x



Display Modes	Display Clock Frequency Range (MHz)
CRT DAC	20-400
Serial DVO (Single Channel)	100-270
LVDS (Single Channel)	20-112
LVDS (Dual Channel)	80-224
TV-Out on Serial DVO	100-200

The PLL frequency selection must be done such that the internal VCO frequency is within its limits. The PLL Frequency is based on the selected register and the following formula. The post divider register value limits are different for Serial-DVO and LVDS modes.

Reference Frequency: 96MHz for SDVO CRT, HDMI,
96MHz or 100MHz for LVDS.

$$\text{DotClk\_Frequency} = (\text{ReferenceFrequency} * (5 * (M1+2) + (M2+2)) / (N+2)) / (P1 * P2)$$

Item	Units	Range	Notes
Dot Clock	Frequency	20-400	MHz (Combining ALL modes)
VCO	Frequency	1400-2800	MHz
N – Counter	Value	3-8	
M – Counter	Value	70-120	$M = 5 * (M1 + 2) + (M2 + 2)$
M1 and M2		$M1 > M2$	
M1	Value	10-20	
M2	Value	5-9	
P-Div	Value	5-80	Combined P1 and P2 for sDVO/DAC mode
P-Div	Value	7-98]	Combined P1 and P2 for LVDS mode
P1-Div	Value	1-8	All modes



2.5.1 VGA0—VGA 0 Divisor Register (100.8 MHz dot clk, 25.175 MHz pixel rate)

Address Offset: 06000h–06003h
Default Value: 00031108h
Normal Attribute: R/W
Size: 32 bits

This register defines one of the two standard VGA frequencies that can be selected from the VGA register clock control bits for use in VGA Native modes. These values default to the proper VGA standard native frequency using the default 96MHz reference clock.

Bit	Description
31:22	Reserved: Write as zero.
21:16	VGA0 N-Divisor: N-Divisor value calculated for the desired output frequency. The register value is two less than the actual divisor.
15:14	Reserved: Write as zero.
13:8	VGA0 M1-Divisor: M-Divisor value calculated for the desired output frequency. The register value is two less than the actual divisor.
7:6	Reserved: MBZ
5:0	VGA0 M2-Divisor: M-Divisor value calculated for the desired output frequency. The register value is two less than the actual divisor.

2.5.2 VGA1 —VGA 1 Divisor Register (113.280 MHz dot clk, 28.322 MHz pixel rate)

Address Offset: 06004h–06007h
Default Value: 00031406h
Normal Attribute: R/W
Size: 32 bits

This register defines one of the two standard VGA frequencies that can be selected from the VGA register clock control bits for use in VGA Native modes. These values default to the proper VGA native frequency using the default 96MHz reference clock. These registers are only used for native mode VGA and not for centered or panel fitted VGA.

Bit	Description
31:22	Reserved: Write as zero.
21:16	VGA 1 N-Divisor: N-Divisor value calculated for the desired output frequency. The register value is two less than the actual divisor.
15:14	Reserved: Write as zero.
13:8	VGA0 M1-Divisor: M-Divisor value calculated for the desired output frequency. The register value is two less than the actual divisor.
7:6	Reserved: Write as zero.
5:0	VGA 1 M2-Divisor: M-Divisor value calculated for the desired output frequency. The register value is two less than the actual divisor.



2.5.3 VGA_PD—VGA Post Divisor Values

Address Offset: 06010h–06013h
 Default Value: 00020002h
 Normal Attributes: R/W
 Size: 32 bits

Bit	Description
31:26	Reserved: Write as zero.
25:24	VGA1 P2 Clock Divide: 00 = Divide by 10. This is used when Dot Clock $\leq$ 270MHz in sDVO or DAC modes 01 = Divide by 5. This is used when Dot Clock $>$ 270MHz in sDVO or DAC modes 10 = Reserved 11 = Reserved
23:16	VGA1P1 Post Divisor: Writes to this byte cause the staging register contents to be written into the active register when in the VGA mode of operation. This will also occur when the VGA MSR register is written. 00000001b = Divide by one 00000010b = Divide by two 00000100b = Divide by three 00001000b = Divide by four 00010000b = Divide by five 00100000b = Divide by six 01000000b = Divide by seven 10000000b = Divide by Eight All other values are illegal and should not be used
15_10	Reserved: Write as zero.
9:8	VGA0 P2 Clock Divide: 00 = Divide by 10. This is used when Dot Clock $\leq$ 270MHz in sDVO or DAC modes 01 = Divide by 5. This is used when Dot Clock $>$ 270MHz in sDVO or DAC modes 10 = Reserved 11 = Reserved
7:0	VGA0 P1 Post Divisor: Writes to this byte cause the staging register contents to be written into the active register when in the VGA mode of operation. This will also occur when the VGA MSR register is written. 00000001b = Divide by one 00000010b = Divide by two 00000100b = Divide by three 00001000b = Divide by four 00010000b = Divide by five 00100000b = Divide by six 01000000b = Divide by seven 10000000b = Divide by Eight All other values are illegal and should not be used



2.5.4 DPLLA_CTRL—DPLL A Control Register

Address Offset: 6014h–06017h
 Default Value: 04020C00h
 Normal Attribute: R/W
 Size: 32 bits

Bit	Description
31	DPLL VCO Enable: This bit will enable or disable the PLL VCO. Pipe A palette accesses require that the DPLL for that pipe be running. Disabling the PLL will cause the display clock to stop and the display pipe to be disabled. 0 = DPLL is disabled in its lowest power state (default) 1 = DPLL is enabled and operational
30	DPLLA Serial DVO High Speed IO clock Enable 0 = High Speed IO Clock Disabled (default) 1 = High Speed IO Clock Enabled (must be set in Serial DVO and HDMI modes)
29	Reserved : Write as zero.
28	VGA Mode Disable: When in native VGA modes, writes to the VGA MSR register causes the value in the selected (by MSR bits) VGA clock control register to be loaded into the active register. This allows the VGA clock select to select the pixel frequency between the two standard VGA pixel frequencies. 0 = VGA MSR<3:2> Clock Control bits select DPLL A Frequency 1 = Disable VGA Control
27:26	DPLLA Mode Select : Configure the DPLLA for various supported Display Modes 00 = Reserved 01 = DPLLA in DAC/Serial DVO/Integrated TV mode 10 = DPLLA in LVDS mode (Mobile devices ONLY) otherwise RESERVED 11 = Reserved
25:24	FPA0/FPA1 P2 Clock Divide: 00 = Divide by 10. This is used when Dot Clock ≤ 270MHz in sDVO, HDMI, or DAC modes 01 = Divide by 5. This is used when Dot Clock >270MHz 10 = Reserved 11 = Reserved For DPLLA in LVDS mode, BITS(27:26)=10 00 = Divide by 14. This is used in Single-Channel LVDS 01 = Divide by 7. This is used in Dual-Channel LVDS 10 = Reserved 11 = Reserved



Bit	Description
23:16	FPA0/ FPA1 P1 Post Divisor: Writes to this byte cause the staging register contents to be written into the active register when in the VGA mode of operation. This will also occur when the VGA MSR register is written. 00000001b = Divide by one 00000010b = Divide by two 00000100b = Divide by three 00001000b = Divide by four 00010000b = Divide by five 00100000b = Divide by six 01000000b = Divide by seven 10000000b = Divide by Eight All other values are illegal and should not be used
15	Reserved: Write as zero
14:13	PLL Reference Input Select: The PLL reference should be selected based on the display device that is being driven. The standard reference clock is used for CRT modes using the analog display port or LCD panels for both the sDVO connected transmitter or the integrated LVDS. TV Clock in should be selected when driving an sDVO connected TV encoder. 00 = DREFCLK (default is 96 MHz) 01 = Reserved 10 = SDVO TVCLKIN 11 = Spread spectrum input clock
12:9	Parallel to Serial Load Pulse phase selection: Programmable select bits to choose the relative phase of the high speed (10X) DPLL clock used for generating the parallel to serial load pulse for digital display port on PCIe. <u>The relative phase is the number of flop delays</u> (phase 0 represents 1 flop delay) of the 1X parallel data synchronization signal in the 10X clock domain. The earliest selectable clock phase is 4. A phase selection of 10 or greater simply extends the flop delay count to sample delayed data. 0100 = use clock phase-4 0101 = use clock phase-5 0110 = use clock phase-6 (Default value) 0111 = use clock phase-7 1000 = use clock phase-8 1001 = use clock phase-9 1010 = use clock phase-10 1011 = use clock phase-11 1100 = use clock phase-12 1101 = use clock phase-13 Phases 0 through 3 are not available for Load Pulse selection. [DevCL] The following programming is recommended for DevCL based on PV timing analysis: 1101 – use clock phase-13
8:0	Reserved



2.5.5 DPLLB_CTRL—DPLL B Control Registers

Address Offset: 06018h–0601Bh
 Default Value: 04020C00h
 Normal Attribute: R/W
 Size: 32 bits

Bit	Description
31	DPLLB VCO Enable: This bit will enable or disable the PLL VCO. If the PLL is disabled, the display clock will stop and the display pipe will be disabled. 0 = DPLL is disabled in its lowest power state (default) 1 = DPLL is enabled and operational
30	DPLLB Serial DVO High Speed IO clock Enable 0 = High Speed IO Clock Disabled (default) 1 = High Speed IO Clock Enabled (must be set in Serial DVO and HDMI modes)
29	Reserved : Write as zero.
28	VGA Native Mode Disable: VGA native mode uses two frequencies selected by a VGA register field. The two frequencies are programmed in the VGA0 and VGA1 registers and selected by the VGA register bits. Writes to the VGA MSR register causes the value in the selected (by MSR bits) VGA clock control register to be loaded into the active register and thereby select the proper VGA frequency. Setting this bit to a zero only makes sense when the VGA disable bit is not set. 0 = VGA MSR<3:2> Clock Control bits select DPLL B Frequency 1 = Disable Native mode VGA Control (High Res or VGA centered)
27:26	DPLLB Mode Select : Configure the DPLLB for various supported Display Modes 00 = Reserved 01 = DPLLB in DAC/Serial DVO/Integrated TV mode 10 = DPLLB in LVDS mode (Mobile devices ONLY) otherwise RESERVED 11 = Reserved
25:24	FPB0/FPB1 P2 Clock Divide: For DPLLB in Serial DVO or DAC mode , BITS(27:26)=01 00 = Divide by 10. This is used when Dot Clock ≤ 270MHz in sDVO, HDMI, or DAC modes 01 = Divide by 5. This is used when Dot Clock >270MHz 10 = Reserved 11 = Reserved For DPLLB in LVDS mode, BITS(27:26)=10 00 = Divide by 14. This is used in Single-Channel LVDS 01 = Divide by 7. This is used in Dual-Channel LVDS 10 = Reserved 11 = Reserved



Bit	Description
23:16	FPB0 / FPB1 P1 Post Divisor: Writes to this byte cause the staging register contents to be written into the active register when in the VGA mode of operation. This will also occur when the VGA MSR register is written. 00000001b = Divide by one 00000010b = Divide by two 00000100b = Divide by three 00001000b = Divide by four 00010000b = Divide by five 00100000b = Divide by six 01000000b = Divide by seven 10000000b = Divide by Eight All other values are illegal and should not be used
15	Reserved : Write as zero.
14:13	PLL Reference Input Select: The PLL reference should be selected based on the display device that is being driven. The standard reference clock is used for CRT modes using the analog display port or LCD panels for both the sDVO connected transmitter or the integrated LVDS. Spread spectrum should only be selected when driving the internal LVDS. One of the TV Clock in references should be selected when driving a DVO connected TV encoder. 00 = DREFCLK (default 96 MHz) for DAC/Serial-DVO 01 = Reserved 10 = SDVO TVCLKIN 11 = Spread spectrum input clock (mobile devices only)
12:9	Parallel to Serial Load Pulse phase selection: Programmable select bits to choose the relative phase of the high speed (10X) DPLL clock used for generating the parallel to serial load pulse for digital display port on PCIe. <u>The relative phase is the number of flop delays</u> (phase 0 represents 1 flop delay) of the 1X parallel data synchronization signal in the 10X clock domain. The earliest selectable clock phase is 4. A phase selection of 10 or greater simply extends the flop delay count to sample delayed data. 0100 = use clock phase-4 0101 = use clock phase-5 0110 = use clock phase-6 (Default value) 0111 = use clock phase-7 1000 = use clock phase-8 1001 = use clock phase-9 1010 = use clock phase-10 1011 = use clock phase-11 1100 = use clock phase-12 1101 = use clock phase-13 Phases 0 through 3 are not available for Load Pulse selection. [DevCL] The following programming is recommended for DevCL based on PV timing analysis: 0101 – use clock phase-5
8:0	Reserved



2.5.6 DPLLAMD—DPLL A SDVO / HDMI Multiplier / Divisor Register

Address Offset: 0601Ch–0601Fh
 Default Value: 00000003h
 Normal Attribute: R/W
 Size: 32 bits

Bit	Description
31:30	Reserved
29:24	DPLL A HDMI Divider: Hi-Res: When the source is high resolution, this field determines the number of pixels to be included in the multiplied packet defined by the SDVO/HDMI multiplier. For SDVO and CRT, the only valid setting is 1x. SDVO example: If the pixel clock is set to 35 MHz and a 4x multiplier is selected, the HDMI divider must be set to 1 and the SDVO/HDMI multiplier must be set to 4. This ensures that 1 pixel and 4 fill codes are sent over SDVO, and that 1 pixel is repeated 4 times (at 4x35MHz) on the CRT. HDMI example: If the pixel clock on the display should be 180MHz and the display PLL is set to 270MHz, two pixels and one fill code must be sent over HDMI (fixed frequency mode only). Therefore, the HDMI divider should be set to 2 and the SDVO/HDMI multiplier should be set to 3, since 180 MHz (pixel clock) = 2/3*270MHz (link character clock) This divider must be set to 1x for any mode except HDMI fixed frequency mode. Value in this register = number of pixels per packet – 1 Default: 0000 – 1 pixel per packet (Default value, must be set to 1x for any mode except HDMI fixed frequency mode) Range: 0-63 (1 pixel per packet – 64 pixels per packet)
23:22	Reserved
21:16	DPLL A HDMI Divider: VGA: When the source is VGA, these bits specify the HDMI divider. The format of this field is the same as that of the hi-res divider.
15:14	Reserved
13:8	DPLL A SDVO/HDMI multiplier: Hi-Res: This field determines the data multiplier for sDVO and is also applied to CRT. In order to keep the clock rate to a more narrow range of rates, the multiplier is set and the Display PLL programmed to a multiple of the display mode's actual clock rate. This is unrelated to the pixel multiply that is selectable per plane. 6x and higher multipliers can only be used for HDMI mode. Value in this register = multiplication factor - 1 Default: 000000 (1X) Range: 0 – 63 (1X – 64X)
7:6	Reserved
5:0	DPLL A SDVO/HDMI multiplier: VGA: When the source is VGA, these bits specify the HDMI multiplier. The format of this field is the same as that of the hi-res multiplier. 6x and higher multipliers can only be used for HDMI mode. Value in this register = multiplication factor - 1 Default: 000011 (4X) Range: 0 – 63 (1X – 64X)



2.5.7 DPLL BMD—DPLL B SDVO / HDMI Multiplier / Divisor Register

Address Offset: 06020h–06023h
 Default Value: 00000003h
 Normal Attribute: R/W
 Size: 32 bits

Bit	Description
31:30	Reserved
29:24	DPLL B HDMI Divider: Hi-Res: When the source is high resolution, this field determines the number of pixels to be included in the multiplied packet defined by the SDVO/HDMI multiplier. For SDVO and CRT, the only valid setting is 1x. SDVO example: If the pixel clock is set to 35 MHz and a 4x multiplier is selected, the HDMI divider must be set to 1 and the SDVO/HDMI multiplier must be set to 4. This ensures that 1 pixel and 4 fill codes are sent over SDVO, and that 1 pixel is repeated 4 times (at 4x35MHz) on the CRT. HDMI example: If the pixel clock on the display should be 180MHz and the display PLL is set to 270MHz, two pixels and one fill code must be sent over HDMI (fixed frequency mode only). Therefore, the HDMI divider should be set to 2 and the SDVO/HDMI multiplier should be set to 3, since 180 MHz (pixel clock) = 2/3*270MHz (link character clock) This divider must be set to 1x for any mode except HDMI fixed frequency mode. Value in this register = number of pixels per packet – 1 Default: 0000 – 1 pixel per packet (Default value, must be set to 1x for any mode except HDMI fixed frequency mode) Range: 0-63 (1 pixel per packet – 64 pixels per packet)
23:22	Reserved
21:16	DPLL B HDMI Divider: VGA: When the source is VGA, these bits specify the HDMI divider. The format of this field is the same as that of the hi-res divider.
15:14	Reserved
13:8	DPLL B SDVO/HDMI multiplier: Hi-Res: This field determines the data multiplier for sDVO and is also applied to CRT. In order to keep the clock rate to a more narrow range of rates, the multiplier is set and the Display PLL programmed to a multiple of the display mode's actual clock rate. This is unrelated to the pixel multiply that is selectable per plane. 6x and higher multipliers can only be used for HDMI mode. Value in this register = multiplication factor - 1 Default: 000011 (4X) Range: 0 – 63 (1X – 64X)
7:6	Reserved
5:0	DPLL B SDVO/HDMI multiplier: VGA: When the source is VGA, these bits specify the HDMI multiplier. The format of this field is the same as that of the hi-res multiplier. 6x and higher multipliers can only be used for HDMI mode. Value in this register = multiplication factor - 1 Default: 000000 (1X) Range: 0 – 63 (1X – 64X)



2.5.8 FPA0—DPLL A Divisor Register 0

Address Offset: 06040h–06043h
Default Value: 00031108h
Normal Attribute: R/W
Size: 32 bits

Bit	Description
31:22	Reserved: Write as zero.
21:16	FPA0 N-Divisor: N-Divisor value calculated for the desired output frequency. The register value is programmed two less than the actual divisor.
15:14	Reserved: Write as zero.
13:8	FPA0 M1-Divisor: M-Divisor value calculated for the desired output frequency. The register value is programmed to two less than the actual divisor.
7:6	Reserved: MBZ
5:0	FPA0 M2-Divisor: M-Divisor value calculated for the desired output frequency. The register value is programmed two less than the actual divisor.

2.5.9 FPA1—DPLL A Divisor Register 1

Address Offset: 06044h–06047h
Default Value: 00031108h
Normal Attribute: R/W
Size: 32 bits

Bit	Description
31:22	Reserved: Write as zero.
21:16	FPA1 N-Divisor: N-Divisor value calculated for the desired output frequency. The register value is two less than the actual divisor.
15:14	Reserved: Write as zero.
13:8	FPA1 M1-Divisor: M-Divisor value calculated for the desired output frequency. The register value is two less than the actual divisor.
7:6	Reserved: MBZ
5:0	FPA1 M2-Divisor: M-Divisor value calculated for the desired output frequency. The register value is two less than the actual divisor.



2.5.10 FPB0—DPLL B Divisor Register

Address Offset: 06048h–0604Bh
Default Value: 00031108h
Normal Attribute: R/W
Size: 32 bits

Bit	Description
31:22	Reserved: Write as zero.
21:16	FPB0 N-Divisor: N-Divisor value calculated for the desired output frequency. The register value is two less than the actual divisor.
15:14	Reserved: Write as zero.
13:8	FPB0 M1-Divisor: M-Divisor value calculated for the desired output frequency. The register value is two less than the actual divisor.
7:6	Reserved: MBZ
5:0	FPB0 M2-Divisor: M-Divisor value calculated for the desired output frequency. The register value is two less than the actual divisor.

2.5.11 FPB1—DPLL B Divisor Register 1

Address Offset: 0604Ch–0604Fh
Default Value: 00031108h
Normal Attribute: R/W
Size: 32 bits

Bit	Description
31:22	Reserved: Write as zero.
21:16	FPB1 N-Divisor: N-Divisor value calculated for the desired output frequency. The register value is two less than the actual divisor.
15:14	Reserved: Write as zero.
13:8	FPB1 M1-Divisor: M-Divisor value calculated for the desired output frequency. The register value is two less than the actual divisor.
7:6	Reserved: MBZ
5:0	FPB1 M2-Divisor: M-Divisor value calculated for the desired output frequency. The register value is two less than the actual divisor.



2.5.12 DPLL_TEST—DPLLA and DPLLB Test Register

Address Offset: 0606Ch-0606Fh
 Default Value: 00010001h
 Normal Attribute: R/W
 Size: 32 bits

Bit	Description
31:24	Reserved: Write as zero.
23:22	DPLLB Pre-Divider select for 2GHz SDVO clock: DPLL pins OCPDPLL2GDIVSEL[1:0] 00 = Divide by 1 (Selects all non-2GHz CTM modes) 01 = Divide by 2 (Selects 200/2GHz CTM) 10 = Divide by 4 (Selects 400/2GHz CTM) 11 = Reserved
21:20	DPLLB Post Divider P2 select for Feedback core clock (DPLLLKCLKP): Valid ONLY when DPLL pin OCPDPLLFDBKSEL is set to 1. DPLL pins OCPDFDBKP2DVSEL[1:0] In Serial DVO or DAC mode X0 = Clockout is forced to Gnd X1 = Divide by 10. All others = Reserved In LVDS mode X0 = Clockout is forced to Gnd X1 = Divide by 14. All others = Reserved
19	DPLLB Input N-divider Bypass select: DPLL pin OCPDPLLNDVBYPASS 0 = DPLL uses output of N-divider (Default) 1 = Output of N-Divider is By-passed
18	DPLLB M-divider Bypass select: DPLL pin OCPDPLLMDVBYPASS 0 = DPLL uses output of M-divider (Default) 1 = Output of M-Divider is By-passed
17	DPLLB Feedback clock select: DPLL pin OCPDPLLFDBKSEL 0 = DPLL locks on internal Feedback (default) 1 = DPLL locks on feedback from Core (DPLL pin DLKFBCLK)
16	Enable DPLLB Input Clock Buffer: DPLL pin OCPDPLLBUFFEN 0 = Disables the Differential Input clock buffer 1 = Enables the Differential Input clock buffer (Default)
15:6	Reserved: Write as zero.



Bit	Description
5:4	DPLLA Post Divider P2 select for Feedback core clock (DPLLLKCLKP): Valid ONLY when DPLL pin OCPDPLLFBKSEL is set to 1. DPLL pins OCPDFDBKP2DVSEL[1:0] In Serial DVO or DAC mode X0 = Clockout is forced to Gnd X1 = Divide by 10. All others = Reserved In LVDS mode X0 = Clockout is forced to Gnd X1 = Divide by 14. All others = Reserved
3	DPLLA Input N-divider Bypass select. DPLL pin OCPDPLLNDVBYPASS 0 = DPLL uses output of N-divider (Default) 1 = Output of N-Divider is By-passed
2	DPLLA M-divider Bypass select: DPLL pin OCPDPLLMDVBYPASS 0 = DPLL uses output of M-divider (Default) 1 = Output of M-Divider is By-passed
1	DPLLA Feedback clock select: DPLL pin OCPDPLLFBKSEL 0 = DPLL locks on internal Feedback (default) 1 = DPLL locks on feedback from Core (DPLL pin DLKFBCLK)
0	Enable DPLLA Input Clock Buffer: DPLL pin OCPDPLLBUFFEN 0 = Disables the Differential Input clock buffer 1 = Enables the Differential Input clock buffer (Default)



2.5.13 D_STATE—D State Function Control Register

Address Offset: 06104–06107h
 Default Value: 00000000h
 Normal Attribute: R/W
 Size: 32 bits

Bit	Description
31:4	Reserved: MBZ
3	Dot Clock PLL Power Down in D3: This bit determines whether the PCI Power State Powers down the Dot Clock PLLs when in D3. A 0 on this bit does not power down the DPLLs, requiring software to gate them if necessary. When this bit is a 1, the dot PLLs are powered down when in D3. The PCI power state is determined by bits 1:0 of the PCI Power Management Control/Status register.
2	Reserved
1	Graphics Core Clock Gating: This bit determines whether the PCI Power State gates the Graphics Core clocks when in the D3 state. A 0 on this bit does not gate the clocks, requiring software to gate them if necessary. When this bit is a 1, the graphics core clocks are gated at the outputs of the PLLs when in D3. The PCI power state is determined by bits 1:0 of the PCI Power Management Control/Status register. This register field has no use in current products.
0	Dot Clock Gating: This bit determines whether the PCI Power State gates the Dot clocks when in the D3 state. A 0 on this bit does not gate the clocks, requiring software to gate them if necessary. When this bit is a 1, the dot clocks are gated at the outputs of the PLLs when in D3. The PCI power state is determined by bits 1:0 of the PCI Power Management Control/Status register.



2.5.14 DSPCLK_GATE_D—Clock Gating Disable for Display Register

Address Offset: 06200h–06203h
 Default Value: 10000000h (00000000h for [DevBW])
 Normal Access: Read/Write
 Size: 32 bits

Bit	Description
31	Reserved. MBZ. This bit is not connected on [DevBW].
30	Dpunit (PipeB) Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
29	vsunit Clock Gating Disable (this bit used to be in PCI space in Calistoga) 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
28	vrhunit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function Clock gating should not be enabled for this unit (this bit should always be set to 1.) [DevBW]: Reserved. MBZ. This bit is not connected on [DevBW].
27	vrduit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
26	audunit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function [DevBW]: Reserved. MBZ. This bit is not connected on [DevBW].
25	Dpunit (PipeA) Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
24	dpcunit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
23	Tvrunit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function [DevBW]: Reserved. MBZ. This bit is not connected on [DevBW].



Bit	Description
22	Tvcunit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function [DevBW]: Reserved. MBZ. This bit is not connected on [DevBW].
21	Tvfunit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function [DevBW]: Reserved. MBZ. This bit is not connected on [DevBW].
20	Tveunit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function [DevBW] Reserved. MBZ. This bit is not connected on [DevBW].
19	Dvsunit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
18	Ddbunit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
17	gmbusunit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
16	Dprunit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
15	Dpfunit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
14	DPLRunit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function [DevBW] Reserved. MBZ. This bit is not connected on [DevBW].
13	Dplsunit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function [DevBW] Reserved. MBZ. This bit is not connected on [DevBW].
12	Dplunit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function [DevBW]: Reserved. MBZ. This bit is not connected on [DevBW].



Bit	Description
11	Dpounit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
10	Dpbunit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
9	Dcunit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
8	dpgcunit (pipe B) Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
7	dpgcunit (pipe A) Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
6	DPIOunit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
5	Ovfunit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
4	Ovbunit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
3	Ovrunit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
2	Ovcunit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
1	Ovuunit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
0	Ovlunit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function



2.5.15 RENCLK_GATE_D1—Clock Gating Disable for Render Register

Address Offset: 06204h–06207h
 Default Value: 00000000h
 Normal Access: Read/Write
 Size: 32 bits

Bit	Description
31	Reserved
30	rcz unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function [DevBW-A,B] Erratum BWT018: This bit must always be set (always disabling clock gating) for proper RC operation.
29	rcc unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function [DevBW], [DevCL] Erratum GEN4019: This bit must always be set (always disabling clock gating) for proper RC operation.
28	rcpb unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function [DevBW-A,B] Erratum BWT018: This bit must always be set (always disabling clock gating) for proper RC operation.
27	dap unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
26	roc unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
25	gw unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
24	td unit Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
23	isc unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function [DevBW]: this bit must be set to '1'



Bit	Description
22	ic unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
21	eu unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
20:18	Reserved
17	so unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
16	[DevCL] fbcunit Clock Gating Disable: (this bit used to be in PCI space in Calistoga) 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function [DevBW]: Reserved. MBZ. This bit is not connected on [DevBW].
15	mari unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
14	masf unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function [DevBW-A] Errata BWT009: This bit must be set at all times (masf clock gating must <i>not</i> be enabled.)
13	mawb unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
12	em unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
11	uc unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
10:7	Reserved
6	si unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
5	mt unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function



Bit	Description
4	pl unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
3	dg unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
2	qc unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
1	ft unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
0	dm unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function



2.5.16 RENDCLK_GATE_D2—Clock Gating Disable for Render Register 2

Address Offset: 06208h–0620Bh
Default Value: 00000000h
Normal Access: Read/Write
Size: 32 bits

Bit	Description
31:19	Reserved
18	sc unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
17	fl unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
16	svg unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
15	svsm unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
14	svdr unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
13	svdw unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
12	svrr unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
11	svrw unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
10	svts unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
9	vf unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function



Bit	Description
8	vs0 unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
7	gs unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
6	cl unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
5	sf unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
4	wiz unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
3	bf, bd unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
2	ts unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
1	urb unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function
0	vfe unit Clock Gating Disable: 0 = Clock gating controlled by unit enabling logic 1 = Disable clock gating function [DevBW-A0] Erratum: This field must always be set to 1.



2.5.17 RAMCLK_GATE_D—GFX RAM Clock Gating Disable Register ([DevCL] Only)

Address Offset: 06210h–06213h
Default Value: 00000000h
Normal Access: Read/Write
Size: 32 bits

Bit	Description
31	TVOUT RAM Clock Gating Disable: 0 = Enable RAM bank clock gating function (default) 1 = Disable RAM bank clock gating function
30	Panel Fitter RAM Clock Gating Disable: 0 = Enable RAM bank clock gating function (default) 1 = Disable RAM bank clock gating function
29	Cursor Data Buffer RAM Clock Gating Disable: 0 = Enable RAM bank clock gating function (default) 1 = Disable RAM bank clock gating function
28	WIZ Z coeff readback return FIFO Clock Gating Disable: 0 = Enable RAM bank clock gating function (default) 1 = Disable RAM bank clock gating function
27	Display Data Buffer2 (Overlay) Gating Disable: 0 = Enable RAM bank clock gating function (default) 1 = Disable RAM bank clock gating function
26	Display Data Buffer1 RAM Clock Gating Disable: 0 = Enable RAM bank clock gating function (default) 1 = Disable RAM bank clock gating function
25	ME RAM Clock Gating Disable: 0 = Enable RAM bank clock gating function (default) 1 = Disable RAM bank clock gating function
24	WIZ polygon FIFO RAM Clock Gating Disable: 0 = Enable RAM bank clock gating function (default) 1 = Disable RAM bank clock gating function
23	VF RAM Clock Gating Disable: 0 = Enable RAM bank clock gating function (default) 1 = Disable RAM bank clock gating function
22	SF RAMClock Gating Disable: 0 = Enable RAM bank clock gating function (default) 1 = Disable RAM bank clock gating function



Bit	Description
21	WMIZ Latency FIFO Clock Gating Disable: 0 = Enable RAM bank clock gating function (default) 1 = Disable RAM bank clock gating function
20	TC FIFO Clock Gating Disable: 0 = Enable RAM bank clock gating function (default) 1 = Disable RAM bank clock gating function
19	SV FIFO Clock Gating Disable: 0 = Enable RAM bank clock gating function (default) 1 = Disable RAM bank clock gating function
18	Latency FIFO Clock Gating Disable: 0 = Enable RAM bank clock gating function (default) 1 = Disable RAM bank clock gating function
17	URB Clock Gating Disable: 0 = Enable RAM bank clock gating function (default) 1 = Disable RAM bank clock gating function
16	L2 Instruction Tag RAM Clock Gating Disable: 0 = Enable RAM bank clock gating function (default) 1 = Disable RAM bank clock gating function
15	Data RAM Clock Gating Disable: 0 = Enable RAM bank clock gating function (default) 1 = Disable RAM bank clock gating function
14	TAG RAM Clock Gating Disable: 0 = Enable RAM bank clock gating function (default) 1 = Disable RAM bank clock gating function
13	L2 Instruction Cache Clock Gating Disable: 0 = Enable RAM bank clock gating function (default) 1 = Disable RAM bank clock gating function
12	MRFRAM Clock Gating Disable: 0 = Enable RAM bank clock gating function (default) 1 = Disable RAM bank clock gating function
11	GRF RAM Clock Gating Disable: 0 = Enable RAM bank clock gating function (default) 1 = Disable RAM bank clock gating function
10	Data Cache CAM Clock Gating Disable: 0 = Enable RAM bank clock gating function (default) 1 = Disable RAM bank clock gating function
9	Data Cache Gating Disable: 0 = Enable RAM bank clock gating function (default) 1 = Disable RAM bank clock gating function



Bit	Description
8	Render Cache Latency FIFO Clock Gating Disable: 0 = Enable RAM bank clock gating function (default) 1 = Disable RAM bank clock gating function
7	Render PA Tag RAM (Z) Clock Gating Disable: 0 = Enable RAM bank clock gating function (default) 1 = Disable RAM bank clock gating function
6	Render PA Tag RAM (Color) Clock Gating Disable: 0 = Enable RAM bank clock gating function (default) 1 = Disable RAM bank clock gating function
5	Render Cache Write Back FIFO Clock Gating Disable: 0 = Enable RAM bank clock gating function (default) 1 = Disable RAM bank clock gating function
4	Render Cache (Z) Clock Gating Disable: 0 = Enable RAM bank clock gating function (default) 1 = Disable RAM bank clock gating function
3	Render Cache (color) Clock Gating Disable: 0 = Enable RAM bank clock gating function (default) 1 = Disable RAM bank clock gating function
2	L2 Mapping Cache CAM Clock Gating Disable: 0 = Enable RAM bank clock gating function (default) 1 = Disable RAM bank clock gating function
1	L2 Mapping Tag RAM Clock Gating Disable: 0 = Enable RAM bank clock gating function (default) 1 = Disable RAM bank clock gating function
0	L2 Mapping Cache Clock Gating Disable: 0 = Enable RAM bank clock gating function (default) 1 = Disable RAM bank clock gating function



2.5.18 DEUC—Dynamic EU Control ([DevCL] only)

Address Offset: 6214–6215h
 Default Value: 00h
 Access: R/W/L;
 Size: 16 bits

This register generally provides control on whether 0, 1, or 2 eus should be gated dynamically. Lock control is provided by TCO[7].

Bit	Access	Default Value	Description
15:13	RW		EU disable on ME clock throttling: Specifies as a percentage the minimum throttling level of ep_2xclk, ep_clk, arc_clk that must occur to allow all EU's to be enabled if EU can be disabled based on ME activity. 000 = 0% - 12.4% 001 = 12.5% - 24.9% 010 = 25% - 37.4% 011 = 37.5% - 49.9% 100 = 50% - 62.4% 101 = 62.5% - 74.9% 110 = 75% - 87.4% 111 = 87.5% - 100%
12:6	RO	0h	Reserved
5	RO		Dynamic EU Control Status 0 = selected EUs are not idle 1 = selected EUs are completely idle
4	RW	0h	SW eu Control Enable 0 = SW does not affect DEUC mode actions 1 = SW assertion of this bit does affect DEUC mode actions
3	RWL	0h	Thermal sensor trip eu Control enable 0 = Thermal sensor trip does not affect DEUC mode actions 1 = Thermal sensor trip does affect DEUC mode actions
2	RWL	0h	ME on eu control enable 0 = ME on does not affect DEUC mode actions 1 = ME on does affect DEUC mode actions
1:0	RW	0h	DEUC Mode (Dynamic EU Control Mode) 00 = no EUs are affected is on 01 = No more than 1 EU is affected (if one EU is already disabled, this setting not cause another to be disabled). If software control has been enabled with this mode, then one EU will be affected. 10 = At least 1 EU is affected (if one EU is already disabled, this setting will cause another to be disabled. If the reason for one already being disabled goes away, this setting will leave one disabled) 11 = 2 EUS are affected



2.6 Display Palette Registers (0A000h–AFFFh)

Display palettes provide a method for converting index data values to color values for VGA and 8-bpp indexed display modes. It also provides methods to gamma correct the true color data that was derived from either 16- or 32-bpp display modes. Accesses to the palette entries require that the core display clock is running at the time of the update. All devices support a Display Palette associated with a particular Display Pipe.

The Display Palettes can be accessed through two methods and operate in one of two modes. There are 256 8-bit entries per color channel in the palette in the 8-bit mode. In the 8-bit mode, each DWord write will load each of the three channels with a single byte. These entries are used for graphics color translation in indexed modes and gamma correction in true and high color modes.

The palette is also accessible via the VGA palette register I/O addresses and method when enabled through the VGA control bits. For VGA palette accesses through the VGA palette register I/O addresses, the palette can look as though there are only 6 bits per color component. When using the palette for VGA, it should be set to the 8-bit mode. The palette entries are always accessed as 24 bits within a DWord. Byte or word writes are not allowed, they must always be DWord. Accesses to the palette must be done with the DPLL for that pipe and display core enabled.

2.6.1 DPALETTE_A—Pipe A Display Palette

Address Offset: 0A000h–0A3FFh
Default: UUh
Normal Attributes: R/W (DWORD only)

Table 2-3. 8-Bit Mode

31	24	23	16	15	8	7	0
Reserved		Red Palette Entry		Green Palette Entry		Blue Palette Entry	

Bit	Description
31:24	Reserved: Read-Only.
23:16	Pipe A Red Palette Entry:
15:8	Pipe A Green Palette Entry:
7:0	Pipe A Blue Palette Entry:



10-bit Programming Notes:

The 10-bit gamma correction curve is represented by specifying a set of reference points spaced equally along the curve. Red, Green, and Blue each have 129 reference points. The first 128 reference points are stored in the palette RAM, and the final value is stored in the GCMAX register. The first 128 reference points are 16 bits represented in a 10.6 format with 10 integer and 6 fractional bits. The final reference points are 17 bits represented in a 11.6 format with 11 integer and 6 fractional bits.

The appropriate reference point pairs (adjacent) are selected for each color, and the output is interpolated between these two reference point values.

To program the gamma correction reference points calculate the desired gamma curve for inputs from 0 to 1024.

Every 8th point on the curve (0,8,16...1016,1024) becomes a reference point. Convert the gamma value to the 10.6 format. The first 128 reference points are saved to the palette RAM, where the odd DWords contain the lower 8 bits of the reference point value, and the even DWords contain the upper 8 bits of the reference point value. The final 129th reference point is saved in the GCMAX register in 11.6 format..

Example equation for gamma curve of 2.2:

For (X = 0..1024) { gamma = [(X / 1024) ^ 2.2] * 1024 }

The curve is assumed to be flat or increasing, never decreasing.

The start and end reference points are not fixed values.

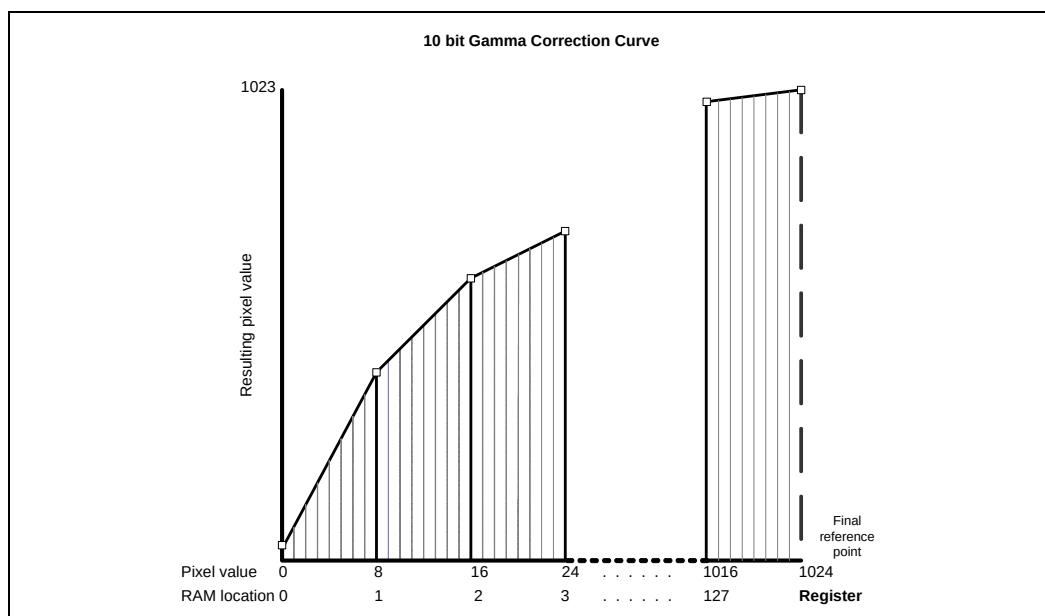




Table 2-4. 10-bit Mode (Even DWord)

31							24	23				22	21			16
Reserved								Red Base[1:0]				Red Fractional Address				
15							8	7								0
Green Base[1:0]				Green Fractional Address				Blue Base[1:0]				Blue Fractional Address				

Table 2-5. 10-bit Mode (Odd DWord)

31							24	23								16
Reserved								Red Base [9:2]								
15							8	7								0
Green Base[9:2]								Blue Base [9:2]								

2.6.2 DPALETTE_B—Pipe B Display Palette

Address Offset: 0A800h–0ABFFh
 Default: UUh
 Normal Attributes: R/W (DWORD only)

8-Bit Mode

Bit	Description
31:24	Reserved: Read-Only.
23:16	Pipe B Red Palette Entry:
15:8	Pipe B Green Palette Entry:
7:0	Pipe B Blue Palette Entry:

See DPALETTE_A for 10-bit Mode definitions.



2.7 Display Pipeline / Port Registers (60000h–6FFFFh)

2.7.1 Display Pipeline A

2.7.1.1 HTOTAL_A—Pipe A Horizontal Total Register

Address Offset: 60000h–60003h
 Default Value: 00000000h
 Normal Access: R/W

Bit	Description
31:29	Reserved: Write as zero.
28:16	Pipe A Horizontal Total Display Clocks: This 13-bit field provides Horizontal Total up to 8192 pixels encompassing the Horizontal Active Display period, front/back border and retrace period. Any pending event (HSYNC, ACTIVE, HBLANK) is reset at HTOTAL and the programmed sequence begins again. This field is programmed to the number of clocks desired minus one. This number of clocks needs to be a multiple of two when driving data out the digital port out the LVDS port in two channel mode. This value should always be equal or greater to the sum of the horizontal active and the horizontal blank, and border region sizes.
15: 12	Reserved: Write as zero.
11: 0	Pipe A Horizontal Active Display Pixels: This 12-bit field provides Horizontal Active Display resolutions up to 4096 pixels. Note that the first horizontal active display pixel is considered pixel number 0. The value programmed should be the (active pixels/line – 1). The number of active pixels will be limited to multiples of two pixels when driving the integrated LVDS port in two channel mode. For proper results during VGA centering mode this value needs to be large enough to fit the largest VGA mode supported, this should be at least 720/1440 pixels for standard VGA type modes or 640/1280 pixels if the nine-dot disable bit in the VGA control register is set. When using the internal panel fitting logic, the minimum horizontal size allowed will be three pixels.



2.7.1.2 HBLANK_A—Pipe A Horizontal Blank Register

Address Offset: 60004h–60007h
Default Value: 00000000h
Normal Access: R/W

Bit	Description
31:29	Reserved: Read-Only.
28:16	Pipe A Horizontal Blank End: This 13-bit field specifies the position of Horizontal Blank End expressed in terms of the absolute pixel number relative to the horizontal active display start. The value programmed should be the HBLANK End pixel position, where the first active pixel is considered position 0; the second active pixel is considered position 1, etc. Horizontal blank ending at the same point as the horizontal total indicates that there is no left hand border area. HBLANK size has a minimum value of 32 clocks. The number of clocks within blank needs to be a multiple of two when driving data out LVDS in two channel mode. The value loaded in the register would be equal to $\text{RightBorder} + \text{Active} + \text{HBlank} - 1$. If this pipe is connected to the TVout port or Panel Fitter 2 the border must be zero. In that case this register is programmed to the same value as the HTOTAL register.
15: 13	Reserved: Read-Only.
12:0	Pipe A Horizontal Blank Start: This 13-bit field specifies the Horizontal Blank Start position expressed in terms of the absolute pixel number relative to the horizontal active display start. The value programmed should be the HBLANK Start pixel position, where the first active pixel is considered position 0; the second active pixel is considered position 1, etc. The number of clocks for both left and right borders need to be a multiple of two when driving data out the LVDS port in two channel mode. Horizontal blank should only start after the end of the horizontal active region. The value loaded in the register would be equal to $\text{RightBorder} + \text{Active} - 1$. If this pipe is connected to the TVout port or Panel Fitter 2 the border must be zero. In that case this register is programmed to the same value as the HACTIVE register.



2.7.1.3 HSYNC_A—Pipe A Horizontal Sync Register

Address Offset: 60008h–6000Bh
 Default Value: 00000000h
 Normal Access: R/W

Bit	Description
31:29	Reserved: Write as zero.
28:16	Pipe A Horizontal Sync End: This 13-bit field specifies the horizontal Sync End position expressed in terms of the absolute pixel number relative to the horizontal active display start. The value programmed should be the HSYNC End pixel position, where the first active pixel is considered position 0; the second active pixel is considered position 1, etc. The number of clocks in the sync period needs to be a multiple of two when driving data out the LVDS port in two channel mode. This value should be greater than the horizontal sync start position and would be loaded with the Active+RightBorder+FrontPorch+Sync-1.
15: 13	Reserved: Read-Only.
12:0	Pipe A Horizontal Sync Start: This 13-bit field specifies the horizontal Sync Start position expressed in terms of the absolute pixel number relative to the horizontal active display start. The value programmed should be the HSYNC Start pixel position, where the first active pixel is considered position 0; the second active pixel is considered position 1, etc. Note that when HSYNC Start is programmed equal to HBLANK Start, both HSYNC and HBLANK will be asserted on the same pixel clock. It should never be programmed to less than HBLANK start. The number of cycles from the beginning of the line needs to be a multiple of two when driving data out the LVDS port in two channel mode. This register should not be less than the horizontal active end. This register should be loaded with the Active+RightBorder+FrontPorch-1.

2.7.1.4 VTOTAL_A—Pipe A Vertical Total Register

Address Offset: 6000Ch–6000Fh
 Default Value: 00000000h
 Normal Access: R/W

Bit	Description
31:29	Reserved: Read-Only.
28:16	Pipe A Vertical Total Display Lines: This 13-bit field provides Vertical Total up to 8192 lines encompassing the Vertical Active Display Lines, top/bottom border and retrace period. The value programmed should be the number of lines required minus one. Vertical total needs to be large enough to be greater than the sum of the vertical active, vertical border, and the vertical blank regions. The vertical counter is incremented on the leading edge of the horizontal sync. For interlaced display modes, this indicates the total number of lines in both fields. In interlaced modes, hardware automatically divides this number by 2 to get the number of lines in each field.
15:12	Reserved: Read-Only.
11:0	Pipe A Vertical Active Display Lines: This 12-bit field provides vertical active display resolutions up to 4096 lines. It should be programmed with the desired number of lines minus one. When using the internal panel fitting logic, the minimum vertical active area must be three lines. For interlaced display modes, this indicates the total number of lines in both fields. In interlaced modes, hardware automatically divides this number by 2 to get the number of lines in each field.



2.7.1.5 VBLANK_A—Pipe A Vertical Blank Register

Address Offset: 60010h–60013h
 Default Value: 00000000h
 Normal Access: R/W

Bit	Description
31:29	Reserved: Read-Only.
28:16	Pipe A Vertical Blank End: This 13-bit field specifies the Vertical Blank End position expressed in terms of the absolute Line number relative to the vertical active display start. The value programmed should be the VBLANK End line position, where the first active line is considered line 0, the second active line is considered line 1, etc. The end of vertical blank should be after the start of vertical blank and before or equal to the vertical total. This register should be loaded with the $V_{active} + \text{BottomBorder} + V_{blank} - 1$. For interlaced display modes, hardware automatically divides this number by 2 to get the vertical blank end in each field. It does not count the two half lines that get added when operating in modes with half lines. If this pipe is connected to the TVout port or Panel Fitter 2 the border must be zero. In that case this register is programmed to the same value as the VTOTAL register.
15:13	Reserved: Read-Only.
12: 0	Pipe A Vertical Blank Start: This 13-bit field specifies the Vertical Blank Start expressed in terms of the absolute line number relative to the vertical active display start. The value programmed should be the VBLANK Start line position, where the first active line is considered line 0, the second active line is considered line 1, etc. Minimum vertical blank size is required to be at least three lines. Blank should start after the end of active. This register is loaded with the $V_{active} + \text{BottomBorder} - 1$. For interlaced display modes, hardware automatically divides this number by 2 to get the vertical blank start in each field. It does not count the two half lines that get added when operating in modes with half lines. If this pipe is connected to the TVout port or Panel Fitter 2 the border must be zero. In that case this register is programmed to the same value as the VACTIVE register.



2.7.1.6 VSYNC_A—Pipe A Vertical Sync Register

Address Offset: 60014h–60017h
Default Value: 00000000h
Normal Access: R/W

Bit	Description
31:29	Reserved: Read-Only.
28:16	Pipe A Vertical Sync End: This 13-bit field specifies the Vertical Sync End position expressed in terms of the absolute Line number relative to the vertical active display start. The value programmed should be the VSYNC End line position, where the first active line is considered line 0, the second active line is considered line 1, etc. This register should be loaded with $V_{active} + BottomBorder + FrontPorch + Sync - 1$. For interlaced display modes, hardware automatically divides this number by 2 to get the vertical sync end in each field. It does not count the two half lines that get added when operating in modes with half lines.
15: 13	Reserved: Read-Only.
12:0	Pipe A Vertical Sync Start: This 13-bit field specifies the Vertical Sync Start position expressed in terms of the absolute line number relative to the vertical active display start. The value programmed should be the VSYNC Start line position, where the first active line is considered line 0, the second active line is considered line 1, etc. This register would be loaded with $V_{active} + BottomBorder + FrontPorch - 1$. For interlaced display modes, hardware automatically divides this number by 2 to get the vertical sync start in each field. It does not count the two half lines that get added when operating in modes with half lines.



2.7.1.7 PIPEASRC—Pipe A Source Image Size

Address Offset: 6001Ch–6001Fh
 Default Value: 00000000h
 Normal Access: Read/Write

Bit	Description
31:28	Reserved: Write as zero
27:16	Pipe A Horizontal Source Image Size: This 12-bit field specifies Horizontal source image size up to 4096. This determines the size of the image created by the display planes sent to the blender. The value programmed should be the source image size minus one. The actual source size must be two times the programmed value in the “pixel multiply mode.” It must represent a size that is a multiple of two (even numbers) when driving the LVDS port in two channel mode. This implies that for this mode, the value programmed will always be an odd number. Except in the case of panel fitting internal or in an external device, this register field would be programmed to a value identical to the horizontal active. This is the only register of the timing registers that is allowed to be programmed while the pipe is enabled.
15: 12	Reserved: Write as zero
11: 0	Pipe A Vertical Source Image Size: This 12-bit field specifies the vertical source image size up to 4096 lines. This determines the size of the image created by the display planes sent to the blender. The value programmed should be the source image size minus one. Note that the actual number of lines needs to be at least twice the planes programmed value when in the pixel multiply mode. Except in the case of panel fitting internal or in an external device, this register field would be programmed to a value identical to the vertical active. For interlaced display modes, hardware automatically divides this number by 2 to get the vertical source image size in each field.

2.7.1.8 BCLRPAT_A— Pipe A Border Color Pattern Register

Address Offset: 60020h–60023h
 Default Value: 00000000h
 Normal Access: Read/Write
 Size: 32 bits

This register value determines what color should be sent to the display in the border region, the space between the end of active and the beginning of blank and the end of blank and the beginning of active.

Bit	Description
31:24	Reserved
23:16	Pipe A Border Red Channel Value:
15:8	Pipe A Border Green Channel Value:
7:0	Pipe A border Blue Channel Value:



2.7.1.9 VSYNCSHIFT_A— Vertical Sync Shift Register

Address Offset: 60028h–6002Bh
Default Value: 00000000h
Normal Access: Read/Write
Size: 32 bits

Bit	Description
31:13	Reserved: Write as zero.
12:0	Pipe A Second Field Vertical Sync Shift: This value specifies the vertical sync alignment for the start of the interlaced second field expressed in terms of the absolute pixel number relative to the horizontal active display start. This value will only be used if the PIPEACONF is programmed to an interlaced mode using vsync shift. Otherwise a legacy value of $\text{floor}[\text{htotal} / 2]$ will be used. Typically, the interlaced second field vertical sync should start one pixel after the point halfway between successive horizontal syncs, so the value of this register should be programmed to: $(\text{horizontal sync start} - \text{floor}[\text{horizontal total} / 2])$ (use the actual horizontal sync start and horizontal total values and not the minus one values programmed into registers). This vertical sync shift only occurs during the interlaced second field. In all other cases the vertical sync start position is aligned with horizontal sync start.

2.7.1.10 CRCtrlColorA—Pipe A CRC Color Channel Control Register (Red)

Address Offset: 60050h–60053h (Red)
Default Value: 00000000h
Normal Access: Read/Write

The value read is obtained from a double-buffer of this register that is updated on VSync except when a CRC error is detected.

For any changes to the CRC controls, you need to wait for two VBLANK events for a valid CRC result. After that, a CRC will be generated each frame.

Border area is always included in the CRC calculation.

There are four CRC calculators, (Red, Green, Blue, Residual1), each with an 8-bit data input and 23-bit CRC result.

For Pipe, SDVO, or TV CRC, the 30-bit pixel data is spread across the inputs of four of the CRC calculators. The fifth is unused and will be ignored for expected CRC comparison and error generation.



Pipe CRC should not be run when Display Port or TV is enabled on this pipe.

Pipe CRC inputs:

Red	Pipe R[9:2]
Green	Pipe G[9:2]
Blue	Pipe B[9:2]
Residual1	Pipe R[1:0] & G[1:0] & B[1:0] & "00"

SDVOB or SDVOC CRC inputs:

Red	SDVO R[9:2]
Green	SDVO G[9:2]
Blue	SDVO B[9:2]
Residual1	SDVO R[1:0] & G[1:0] & B[1:0] &

TV Encoder CRC inputs:

Red	TV DAC A[9:2]
Green	TV DAC B[9:2]
Blue	TV DAC C[9:2]
Residual1	TV DAC A[1:0] & B[1:0] & C[1:0] &

TV Filter CRC inputs:

Red	TV Filter Y[9:2]
Green	TV Filter U[9:2]
Blue	TV Filter V[9:2]
Residual1	TV Filter Y[1:0] & U[1:0] & V[1:0] &

Bit	Description
31	Enable Color CRC: Enables the CRC calculations. After being enabled for the first time, you need to wait for two VBLANK events for a valid CRC result. After that, a CRC will be generated each frame. 0 = CRC Calculations are disabled 1 = CRC Calculations are enabled
30:28	CRC Source Select: These bits select the source of the data to put into the CRC logic. 000 = Pipe A (Not available when DisplayPort or TV is enabled on this pipe) 001 = sDVOB (30-bit format) 010 = sDVOC (30-bit format) 011 = Reserved 100 = TV Encoder outputs (30-bit format) 101 = TV filter outputs (30-bit format) 110 = Reserved 111 = Reserved
27:23	Reserved: Write as zero
22:0	Expected CRC Value: Expected CRC Value for Color Channel. This is the value used to generate the CRC error status and interrupt. Resultant CRC values are compared to this register after the completion of a CRC calculation. Status indications are in the PIPEASTAT register.



2.7.1.11 CRCtrlColorA—Pipe A CRC Color Channel Control Register (Green, Blue, Residual)

Address Offset: 60054h–60057h (Green)
 Address Offset: 60058h–6005Bh (Blue)
 Address Offset: 6005Ch–6005Fh (Residual1)
 Default Value: 00000000h
 Normal Access: Read/Write

Calculation is enabled in the CRCtrlColorA(Res) register.

The value read is obtained from a double-buffer of this register that is updated on VSync except when a CRC error is detected.

Border area is always included in the CRC calculation.

The value in the residual registers will not necessarily be zero during 8bpc CRC calculations.

Bit	Description
31:23	Reserved: Write as zero
22:0	Expected CRC Value: Expected CRC Value for Color Channel. This is the value used to generate the CRC error status and interrupt. Resultant CRC values are compared to this register after the completion of a CRC calculation. Status indications are in the PIPEASTAT register.

2.7.1.12 CRCResColorA—Pipe A CRC Color Channel Result Register (Red, Green, Blue, Residual1)

Address Offset: 60060h–60063h (Red)
 Address Offset: 60064h–60067h (Green)
 Address Offset: 60068h–6006Bh (Blue)
 Address Offset: 6006Ch–6006Fh (Residual1)
 Default Value: UUUUUUUUh
 Normal Access: Read-Only
 Size: 32 bits

The value in the residual registers will not necessarily be zero during 8bpc CRC calculations.

Bit	Description
31:23	Reserved: Read-only
22:0	Color Channel CRC Result Value: This field contains the resultant CRC value for the particular Color Channel at the end of a frame. A status bit can be used as an indication that the data is the valid result of a CRC calculation.



2.7.2 Display Pipeline B

2.7.2.1 HTOTAL_B—Pipe B Horizontal Total Register

Address Offset: 61000h–61003h
Default Value: 00000000h
Normal Access: R/W (Write Protect by Panel Power Sequencer on [DevCL])

Bit	Description
31:29	Reserved: Write as zero.
28:16	Pipe B Horizontal Total Display: See pipe A description.
15:12	Reserved: Write as zero.
11:0	Pipe B Horizontal Active Display: See pipe A description

2.7.2.2 HBLANK_B—Pipe B Horizontal Blank Register

Address Offset: 61004h–61007h
Default Value: 00000000h
Normal Access: R/W (Write Protect by Panel Power Sequencer on [DevCL])

Bit	Description
31:29	Reserved. Write as zero.
28:16	Pipe B Horizontal Blank End: See pipe A description
15:12	Reserved: Write as zero.
12:0	Pipe B Horizontal Blank Start: See pipe A description.



2.7.2.3 HSYNC_B—Pipe B Horizontal Sync Register

Address Offset: 61008h–6100Bh
Default Value: 00000000h
Normal Access: R/W (Write Protect by Panel Power Sequencer on [DevCL])

Bit	Description
31:29	Reserved: Write as zero.
28: 16	Pipe B Horizontal Sync End: See pipe A description.
15:12	Reserved: Write as zero.
12:0	Pipe B Horizontal Sync Start: See pipe A description

2.7.2.4 VTOTAL_B—Pipe B Vertical Total Register

Address Offset: 6100Ch–6100Fh
Default Value: 00000000h
Normal Access: R/W (Write Protect by Panel Power Sequencer on [DevCL])

Bit	Description
31:29	Reserved: Write as zero.
28:16	Pipe B Vertical Total Display: See pipe A description.
15:11	Reserved: Write as zero.
11:0	Pipe B Vertical Active Display: See pipe A description.

2.7.2.5 VBLANK_B—Pipe B Vertical Blank Register

Address Offset: 61010h–61013h
Default Value: 00000000h
Normal Access: R/W (Write Protect by Panel Power Sequencer on [DevCL])

Bit	Description
31:29	Reserved: Write as zero.
28:16	Pipe B Vertical Blank End: See pipe A description.
15:12	Reserved: Write as zero.
12:0	Pipe B Vertical Blank Start: See pipe A description.



2.7.2.6 VSYNC_B—Pipe B Vertical Sync Register

Address Offset: 61014h–61017h
 Default Value: 00000000h
 Normal Access: R/W (Write Protect by Panel Power Sequencer on [DevCL])

Bit	Description
31:29	Reserved: Write as zero.
28:16	Pipe B Vertical Sync End: See pipe A description.
15:12	Reserved: Write as zero.
12:0	Pipe B Vertical Sync Start: See pipe A description.

2.7.2.7 PIPEBSRC—Pipe B Source Image Size

Address Offset: 6101Ch–6101Fh
 Default Value: 00000000h
 Normal Access: Read/Write

Bit	Description
31:28	Reserved: Write as zero
27:16	Pipe B Horizontal Source Image Size: See pipe A description.
15:11	Reserved: Write as zero
11:0	Pipe B Vertical Source Image Size: See pipe A description.

2.7.2.8 BCLRPAT_B—Pipe B Border Color Pattern Register

Address Offset: 61020h–61023h
 Default Value: 00000000h
 Normal Access: Read/Write

This register determines the color sent during the border region, the periods between the end of blank and the start of active and the end of active and the start of blank. Also same color will be sent during pseudo border period. VGA border color is determined by the VGA border “overscan” color register.

Bit	Description
31:24	Reserved
23:16	Pipe B Red channel color value:
15:8	Pipe B Green channel color value:
7:0	Pipe B Blue channel color value:



2.7.2.9 VSYNCSHIFT_B— Vertical Sync Shift Register

Address Offset: 61028h–6102Bh
Default Value: 00000000h
Normal Access: Read/Write

Bit	Description
31:13	Reserved: Write as zero.
12:0	Pipe B Second Field Vertical Sync Shift: This value specifies the vertical sync alignment for the start of the interlaced second field expressed in terms of the absolute pixel number relative to the horizontal active display start. This value will only be used if the PIPEBCONF is programmed to an interlaced mode using vsync shift. Otherwise a legacy value of $\text{floor}[\text{htotal} / 2]$ will be used. Typically, the interlaced second field vertical sync should start one pixel after the point halfway between successive horizontal syncs, so the value of this register should be programmed to: $(\text{horizontal sync start} - \text{floor}[\text{horizontal total} / 2])$ (use the actual horizontal sync start and horizontal total values and not the minus one values programmed into registers). This vertical sync shift only occurs during the interlaced second field. In all other cases the vertical sync start position is aligned with horizontal sync start.



2.7.2.10 CRCtrlColorB—Pipe B CRC Color Control Register (Red)

Address Offset: 61050h–61053h (Red)
 Default Value: 00000000h
 Normal Access: Read/Write

The value read is obtained from a double-buffer of this register that is updated on VSync except when a CRC error is detected.

For any changes to the CRC controls, you need to wait for two VBLANK events for a valid CRC result. After that, a CRC will be generated each frame.

Border area is always included in the CRC calculation.

See description of CRCtrlColorA for more details

Bit	Description
31	Enable Color Channel CRC: After being enabled for the first time, you need to wait for two VBLANK events for a valid CRC result. After that, a CRC will be generated each frame. 0 = CRC Calculations are disabled 1 = CRC Calculations are enabled
30:28	CRC Source Select: These bits select the source of the data to put into the CRC logic. 000 = Pipe B (Not available when DisplayPort or TV is enabled on this pipe) 001 = sDVOB (30-bit format) 010 = sDVOC (30-bit format) 011 = Reserved 100 = TV Encoder outputs (30-bit format) 101 = TV Filter outputs (30-bit format) 110 = Reserved 111 = Reserved
27:23	Reserved: Write as zero
22:0	Expected CRC Value: Expected CRC Value for the Color Channel. This is the value used to generate the CRC error status and interrupt. Resultant CRC values are compared to this register after the completion of a CRC calculation. The status bit is in the PIPEBSTAT register.



2.7.2.11 CRCtrlColorB—Pipe B CRC Color Control Register (Green, Blue, Residual)

Address Offset: 61054h–61057h (Green)
 Address Offset: 61058h–6105Bh (Blue)
 Address Offset: 6105Ch–6105Fh (Residual1)
 Default Value: 00000000h
 Normal Access: Read/Write

Calculation is enabled in the CRCtrlColorA(Red) register.

The value read is obtained from a double-buffer of this register that is updated on VSync except when a CRC error is detected.

Border area is always included in the CRC calculation.

The value in the residual registers will not necessarily be zero during 8bpc CRC calculations.

Bit	Description
31:23	Reserved: Write as zero
22:0	Expected CRC Value: Expected CRC Value for the Color Channel. This is the value used to generate the CRC error status and interrupt. Resultant CRC values are compared to this register after the completion of a CRC calculation. The status bit is in the PIPEBSTAT register.

2.7.2.12 CRCResColorB—Pipe B CRC Result Register

Address Offset: 61060h–61063h (Red)
 Address Offset: 61064h–61067h (Green)
 Address Offset: 61068h–6106Bh (Blue)
 Address Offset: 6106Ch–6106Fh (Residual1)
 Default Value: UUUUUUUUh
 Normal Access: Read-Only
 Size: 32 bits

The value in the residual registers will not necessarily be zero during 8bpc CRC calculations.

Bit	Description
31:23	Reserved: Read-only
22:0	Color Channel CRC Result Value: This field contains the resultant CRC value for the Color Channel at the end of a frame. A status bit can be used as an indication that the data is the valid result of a CRC calculation. The result of a CRC on an empty frame will be 7FFFFh.



2.8 Display Port Control

2.8.1 ADPA—Analog Display Port Register

Address Offset: 61100h–61103h
 Default Value: 00000000h
 Normal Access: Read/Write

Bit	Description
31	ADPA (Analog Display Port A) Enable: This bit enables or disables the DAC output. It has no effect on the horizontal or vertical sync outputs. 1 = Enable. This bit enables analog port DAC 0 = Disable the DAC and go to a low power state. [DevBW] When port multiply is not programmed to 1X and CRT is the only port enabled on a pipe, it should be <u>enabled</u> as follows: 1) Set bit #31 to '1' to enable the DAC and bits #[11:10] to '11' to disable hsync/vsync CRT output. 2) Wait for Vsync. 3) Clear bits #[11:10] to '00'
30	Pipe Select: Determines which display pipe will feed this DAC port. This only applies to dual display pipe devices. It is reserved in all other devices. 0 = Pipe A 1 = Pipe B
29:16	Reserved: Software must preserve the contents of these bits.
15	ADPA HSYNC and VSYNC Polarity Select: VGA VSYNC and HSYNC polarity bits are ignored on this port when this bit is clear. This should only be set if this port is used for VGA display in the VGA native mode. 1 = Use VGA registers to select VSYNC and HSYNC Polarities. (VGA Native Mode) 0 = Use bits 4 and 3 of this register (ADPA) to select VSYNC and HSYNC Polarities.
14	Reserved: Software must preserve the contents of this field.
13:12	Reserved
11:10	Monitor DPMS: (for CRT port) When the graphics device is in D0, these bits force the HSYNC/VSYNC output signal to the state specified in the polarity selection or allow the standard timing generated syncs to continue. These bits enable proper handling of DPMS monitors that support the D1 or D2 states during display device power management by toggling sync signals required. This field should always be loaded with the display device D state during display power management operations. 00 = Monitor in D0. Monitor On. (will not affect sync pulses) 01 = Monitor in D2. Monitor Suspend (HSYNC pulses, VSYNC does not.) 10 = Monitor in D1. Monitor Standby (VSYNC pulses, HSYNC does not) 11 = Monitor in D3. Monitor Off (Neither HSYNC nor VSYNC pulses)
9:5	Reserved : Software must preserve the contents of these bits.
4	VSYNC Polarity Control: The output VSYNC polarity is controlled either by the VGA control bits or this bit when in VGA modes. This is used to implement display modes that require inverted polarity syncs and to set the disabled state of the VSYNC signal. 1 = Active high. 0 = Active low.



Bit	Description
3	HSYNC Polarity Control: According to the ADPA polarity select bit the HSYNC polarity is controlled by either the VGA sync polarity register bit or this bit. 1 = Active high. 0 = Active low.
2:0	Reserved: Software must preserve the contents of these bits.

2.8.2 PORT_HOTPLUG_EN

Memory Offset Address: 61110 – 61113h
 Default: 00000020h
 Normal Access: Read/Write

Note: For correct operation of display port hot plug detection, the device 2 configuration register GMBUSFREQ at offset 0xCC-0xCD must be programmed correctly.

Bit	Description
31:27	Reserved : MBZ
26	SDVOB Hot Plug Interrupt Detect Enable: This will enable the consideration of the hot plug interrupt status bit in the Port Hotplug Status register, offset 61114h. This bit enables detection on the SDVOB interrupt input pin pair. 0 = SDVOB Hot Plug Detect Disabled (Default) 1 = SDVOB Hot Plug Detect Enabled
25	SDVOC Hot Plug Interrupt Detect Enable: This will enable the consideration of the hot plug interrupt status bit in the Port Hotplug Status register, offset 61114h. This bit enables detection on the SDVOC interrupt input pin pair. 0 = SDVOC Hot Plug Detect Disabled (Default) 1 = SDVOC Hot Plug Detect Enabled
24	Reserved: MBZ
23:19	Reserved : MBZ
18	TV Hot Plug Detect Interrupt Enable This will enable the consideration of the TV hot plug interrupt status bit. 0 = TV Hot Plug Detect Disabled (bit 10 of the port hotplug status register no longer detects interrupts, Default) 1 = TV Hot Plug Detect Enabled
17:10	Reserved : MBZ
9	CRT Hot plug Interrupt Enable: Hotplug detection is used to cause an interrupt or status bit based on the connection or disconnection of a CRT to the analog video connection. 0 = No hot plug interrupt is enabled (Default) 1 = Hot plug detection is enabled



Bit	Description
8	CRT Hot plug Circuit Activation Period: This bit sets the activation period for the CRT hot plug circuit detection 0 = 32 cdclk periods (Default) 1 = 64 cdclk periods
7	CRT DAC on time Value: Powerup time for 0 = CRT DAC requires 2M cdclks for warmup (Default) 1 = CRT DAC requires 4M cdclks for warmup
6:5	CRT Hot plug Voltage Compare Value: Compare value for CRT hotplug detect Vref to determine whether the analog port is connected to a CRT. The voltage is forced at the beginning of the active region of the screen every 2 seconds. 00 = 40, 01 = 50, (Default) 10 = 60 11 = 70
4	CRT Hot Plug Detect Delay: This bit determines the length of time between polling periods when the DAC/pipe are disabled 0 = 1G cdclks (default) 1 = 2G cdclks
3	Force CRT detect trigger: Triggers a CRT hotplug/unplug detection cycle independent of the interrupt enable bit. Bits 5:8 of this register must be correctly programmed when forcing a trigger. This bit is automatically cleared after the detection is completed. The result of this trigger is reflected in bits 9:8 of the port hotplug interrupt status register. The CRT interrupt status bit #11 in the hot plug status register (61114) will get set the first time Force CRC detect trigger is used after reset. Software must reset status after a force CRT detect trigger. 0 = No trigger (Default) 1 = Trigger
2:0	Reserved : MBZ

2.8.3 PORT_HOTPLUG_STAT

Memory Offset Address: 61114 – 61117h
Default: 00000000h
Normal Access: Read/Write

This register is the second level of a two level interrupt and status scheme. Status bits in this register are 'sticky'; once set they can be cleared by writing a one to that bit. A write of a zero does not affect the corresponding Interrupt status bit. The corresponding enable bits determine if the interrupt status bit should be propagated to the first line interrupt status register. When an interrupt occurs, the first line interrupt register indicates the second line source of the interrupt. Reading the second line register will determine the precise source for the interrupt.

Before clearing a Port-sourced interrupt (e.g., CRT hotplug) in the IIR, the corresponding interrupt (source) status in the PORT_HOTPLUG_STAT must be cleared by writing a '1' to the appropriate bit. In the case where fields are larger than 1 bit



wide, all bits in the field must be cleared by writing a '1' to them.

Bit	Descriptions
31:12	Reserved : MBZ
11	CRT Hot Plug Interrupt Status: This bit is set when a CRT hot plug or unplug event has been detected. A hot plug or unplug event is defined as the change in connection state of the CRT as determined by the hardware CRT detect sequence which is enabled through bit #9 (CRT hot plug interrupt enable) or bit #3 (Force CRT detect trigger) in the Port_HotPlug_En register 0x61110. After reset, the CRT is considered "unconnected" even if physically connected until the first detect sequence occurs. Physically plugging or unplugging the CRT device will also be detected as a change of connection state. Writing a 1 to this bit clears it. 0 =CRT Interrupt has not occurred 1 = CRT Interrupt has occurred
10	TV Hot Plug Interrupt Status: This bit is set when a TV hot plug or unplug event has been detected. Reflects the state of bit 31 of the TV DAC state register, offset 68004-68007h. Software must write a one to these bits to clear the status. 0 =TV Interrupt has not occurred 1 = TV Interrupt has occurred
9:8	CRT Hot Plug Detection Status (read-only): These bits are set when a CRT hot plug or unplug event has been detected. 00 = No channels attached (default) 01 = Blue channel only is attached 10 = Green channel only is attached 11 = Both blue and green channel attached
7:2	Reserved : MBZ
5:4	SDVO C Hot Plug Interrupt Detect Status : This reflects hot plug interrupt status on SDVO port C. Graphics software must write a one to these bits to clear the status. This bit is used for either monitor hotplug/unplug. This bit feeds into the first line interrupt status register when bit 25 of the hotplug enable status register is set. 00 = SDVO C Hot Plug event not detected x1 = SDVO C long pulse (>50mS) Hot Plug event detected 1x = SDVO C short pulse(<50mS) Hot Plug event detected
3:2	SDVO B Hot Plug Interrupt Detect Status: This reflects hot plug interrupt status on SDVO port B. Graphics software must write a one to these bits to clear the status. This bit is used for either monitor hotplug/unplug. This bit feeds into the first line interrupt status register when bit 26 of the hotplug enable status register is set. 00 = SDVO B Hot Plug event not detected x1 = SDVO B long pulse (>50mS) Hot Plug event detected 1x = SDVO B short pulse (50mS) Hot Plug event detected
1:0	Reserved : MBZ

2.8.4 sDVO /HDMI B—Digital Display Port B Control Register

Address Offset: 61140h–61143h
Default Value: 00000018h
Normal Access: Read/Write
Size: 32 bits



Note: This Digital Display Port defaults to sDVO port functionality when it is not programmed as a HDMI port. The operating mode of the port is determined by the setting of the encoding register field (bits 11:10).

Bit	Description
31	sDVO/HDMI B Enable (Digital Display Port B Enable): Disabling this port will put it in its lowest power state. Port enable takes place on the Vblank after being written. This port must not be enabled simultaneously with DisplayPort B. 1 = Enable. This bit enables the Digital Display Port B interface for HDMI or SDVO modes. 0 = Disable and tristates the Digital Display Port B interface for HDMI or SDVO modes.
30	Pipe Select: This bit determines from which display pipe the source data will originate. This only applies to devices with dual display pipes. Pipe selection takes place on the Vblank after being written 0 = Pipe A 1 = Pipe B
29	Stall Select: This bit selects stall for external scaling functionality only on SDVO. Programming notes: It is only valid to have a single stall indication to a particular pipe. In cases where two ports are being driven from a single pipe, one of the ports must set this bit to 0. Only sDVOB or sDVOC can select the stall function, as only a single stall input is available between the two interfaces. Set the stall input to unused before programming the external device creating the stall. 0 = Stall input signal is unused on this port 1 = Stall input signal is used to stall the pipe attached to this port
28:26	Color Format: This field selects the number of bits per color sent to a receiver device connected to this port. Color format takes place on the Vblank after being written. Color format change must be done as a part of mode set since different color depths require different pixel clock settings. Selecting a pixel color depth higher or lower than the pixel color depth of the frame buffer results in dithering the output stream. 000 = 8 bits per color (Default, x3 mode) 001 = RESERVED for 10 bits per color 010 = RESERVED for 6 bits per color 011 = RESERVED 1xx = RESERVED
25:19	Reserved
18	sDVO/HDMI B Clock Output Inversion (TEST MODE): Please note that this applies to all modes and is instantly updated. 1 = sDVO/HDMI B Clock output is inverted 0 = sDVO/HDMI B Clock output is NOT inverted (DEFAULT)
17:16	Reserved: MBZ



Bit	Description
15	Port Lane Reversal: This bit reverses the order of the 4 lanes within the port. Port lane reversal takes place on the Vblank after being written. It is an OEM configurable feature. 0 = (Default) Not reversed 1 = Reversed
14	Reserved
13	Clock Output Disable: This bit disables the clock output on the digital output port. For 8b/10b modes the clock output should be disabled. 0 = (Default) Clock output enabled 1 = Clock output disabled ([DevCL] only)
12	Reserved: MBZ
11:10	Encoding These bits select among encoding types. It is set as part of the display detection process. Control codes for ANSI 8b/10b and TMDS encoding must be programmed using these bits. Please note that ANSI 8b/10b and TMDS encoding can only be enabled on one port at a time, as only one HPD pin is available for use between ports B and C. 00 –SDVO encoding. (Default) Refer to the SDVO EDS for control code descriptions. In this mode, the SDVOB hotplug input pin pair is used to generate hotplug. 01 = RESERVED 10 = TMDS encoding ([DevCL] external link and HDMI only) See the HDMI specification for control codes. In this mode, the external HPD pin is used to generate hotplug. In fixed frequency mode, start of fill and end of fill values for TMDS must be programmed using register 6114C. 11 = Reserved
9	Null packets enabled during Vsync: This bit enables a null packet (32 bytes of a value of 0) to be sent when Vsync=1 on this port, required for HDMI operation. It also enables preambles and guardbands prior to the null packets, in accordance with the HDMI specification. It is only valid for modes that use TMDS encoding. 0 = Disable null infoframe packets when Vsync=1 on this port. (Default) 1 = Enable null infoframe packets when Vsync=1 on this port.
8	Color Range Select: This bit is used to select the color range of RGB outputs in HDMI mode. It is only valid when using TMDS encoding and 8-bit per color mode. 0 = Apply full 0-255 color range to the output (Default) 1 = Apply 16-235 color range to the output ([DevCL] only)
7	sDVOB Border Enable: This bit determines if the border data from native VGA or the timing generator is to be considered valid pixel data at the external component. 1 = Border to the sDVOB encoder is enabled. Blank# is used to generate the DE output (used in all cases except when the external scaler is used in a DVI panel, over SDVO) . 0 = Border to the sDVOB encoder is disabled. DE (Display Enable) is used
6	Reserved: MBZ
5	Reserved : MBZ



Bit	Description
4:3	Sync Polarity: Please note that sync polarity does not apply to ANSI coding. Indicates the polarity of Hsync and Vsync. Inverted polarity is transmitted as SYNC-BLANK-SYNC and standard polarity is transmitted as BLANK-SYNC-BLANK. For example, if Vsync is not inverted and Hsync is inverted, an Hsync period transmitted during Vsync would be transmitted as BLANK+VS+HS – BLANK+VS – BLANK+VS+HS. Please note that in native VGA modes, these bits have no effect. In native VGA modes, sync polarity is determined by VRshr3c2d76b[7:6], the VGA polarity bits in VGA control. 00 = VS and HS are active low (inverted) 01 = VS is active low (inverted), HS is active high 10 = VS is active high, HS is active low (inverted) 11 = (Default) VS and HS are active high
2	Digital Port B Detected: Read-only bit indicating whether a digital port B was detected during initialization. It signifies the level of the GMBUS port 4 (sDVO B/C) data line at boot. This bit is valid regardless of whether the port is enabled. 0 = Digital Port B not detected during initialization 1 = Digital Port B detected during initialization
1:0	Reserved : MBZ



2.8.5 sDVO / DP—Digital Display Port DFT Register

Address Offset: 61150h–61153h
 Default Value: 00000000h
 Normal Access: Read/Write
 Size: 32 bits

Bit	Description
31	sDVO / DPB Test Mode Enable (Test Mode Only): Enables Test mode for sDVO/DPB logic in the PCI-Express AFE. Enables mode for the sDVO/DPB port only. 1 = Test mode Enabled. sDVO/DPB Data will be internally looped back into the receive logic and registers. 0 = Test mode Disabled. (Default)
30	sDVO / DPB Test Mode Check Enable 1 = Enable checking of Test mode data 0 = Disable Check
29	sDVO / DPB Pattern Select 1 = "1010101010" 0 = "0101010101"
28	sDVO / DPC Test Mode Enable (Test Mode Only): Enables Test mode for sDVO/DP logic in the PCI-Express AFE. Enables mode for sDVO/DPC only. 1 = Test mode Enabled. sDVO/DPC Data will be internally looped back into the receive logic and registers. Lane reversal must be activated when testing sDVO/DPC, since the internal loopback hardware is shared between sDVO/DPB and sDVO/DPC. 0 = Test mode Disabled. (Default)
27	sDVO / DPC Test Mode Check Enable 1 = Enable checking of Test mode data 0 = Disable Check
26	sDVO / DPC Pattern Select 1 = "1010101010" 0 = "0101010101"
25	sDVO / DP DC Balance Set 1 = DC Balance circuitry will be reset on every frame, with Vsynch 0 = DC Balance circuitry does not get reset on every frame.
24:8	Reserved : MBZ
7	sDVO / DPC Clock Test Result 1 = "Fail" 0 = "Pass"



Bit	Description
6	sDVO / DPC Blue Test Result 1 = "Fail" 0 = "Pass"
5	sDVO / DPC Green Test Result 1 = "Fail" 0 = "Pass"
4	sDVO / DPC Red Test Result 1 = "Fail" 0 = "Pass"
3	sDVO / DPB Clock Test Result 1 = "Fail" 0 = "Pass"
2	sDVO / DPB BlueTest Result 1 = "Fail" 0 = "Pass"
1	sDVO / DPB Green Test Result 1 = "Fail" 0 = "Pass"
0	sDVO / DPB Red Test Result 1 = "Fail" 0 = "Pass"



2.8.6 sDVO / DP—Digital Display Port DFT Register 2

Address Offset: 61154h–61158h
 Default Value: 00000000h
 Normal Access: Read/Write
 Size: 32 bits

Bit	Description
31:8	Reserved : MBZ
7	Test pattern 8-bit programmed input on pipe B: 0 = disable the test pattern 1 = enable the test pattern
6	Test pattern 8-bit programmed input on pipe A: 0 = disable the test pattern 1 = enable the test pattern
5	Scrambled 0's on pipe B: 0 = disable the scramble 0's 1 = enable the scramble 0's
4	Scrambled 0's on pipe A: 0 = disable the scramble 0's 1 = enable the scramble 0's
3	PRBS7 test pattern on pipe B: 0 = disable the test pattern 1 = enable the test pattern
2	PRBS7 test pattern on pipe A: 0 = disable the test pattern 1 = enable the test pattern
1	Test mode : Scramble 1's a frame on pipe B: 0 = disable the scramble 1's a frame 1 = enable the scramble 1's a frame
0	Test mode : Scramble 1's a frame on pipe A: 0 = disable the scramble 1's a frame 1 = enable the scramble 1's a frame



2.8.7 sDVO /HDMIC—Digital Display Port C Register

Address Offset: 61160h–61163h
 Default Value: 00000018h
 Normal Access: Read/Write

Note: This Digital Display Port defaults to sDVO port functionality when it is not programmed as a HDMI port. The operating mode of the port is determined by the setting of the encoding register field (bits 11:10).

Bit	Description
31	sDVO/HDMIC Enable (Digital Display Port C Enable): Disabling this port will put it in its lowest power state. Port enable takes place on the Vblank after being written. This port must not be enabled simultaneously with DisplayPort C. 1 = Enable. This bit enables the Digital Display Port C interface for HDMI or SDVO modes. 0 = Disable and tristates the Digital Display Port C interface for HDMI or SDVO modes.
30	Pipe Select: This bit determines from which display pipe the source data will originate. This only applies to devices with dual display pipes. Pipe selection takes place on the Vblank after being written 0 = Pipe A 1 = Pipe B
29	Stall Select: This bit selects stall for external scaling functionality only on SDVO. Programming notes: It is only valid to have a single stall indication to a particular pipe. In cases where two ports are being driven from a single pipe, one of the ports must set this bit to 0. Only sDVOB or sDVOC can select the stall function, as only a single stall input is available between the two interfaces. Set the stall input to unused before programming the external device creating the stall. 0 = Stall input signal is unused on this port 1 = Stall input signal is used to stall the pipe attached to this port
28:26	Color Format: This field selects the number of bits per color sent to a receiver device connected to this port. Color format takes place on the Vblank after being written. Color format change must be done as a part of mode set since different color depths require different pixel clock settings. Selecting a pixel color depth higher or lower than the pixel color depth of the frame buffer results in dithering the output stream. 000 = 8 bits per color (Default) 001 = RESERVED for 10 bits per color 010 = RESERVED for 6 bits per color 011 = RESERVED 1xx = RESERVED
25:19	Reserved
18	sDVO/HDMIC Clock Output Inversion (TEST MODE): Please note that this applies to all modes and is instantly updated. 1 = sDVO/HDMIB Clock output is inverted 0 = sDVO/HDMIB Clock output is NOT inverted (DEFAULT)



Bit	Description
17:16	Symbol Clock Duty Cycle: These bits control the output clock duty cycle to enable EMI mitigation on the external HDMI link. 10/90 cycle has been measured to have ~13dB EMI improvement over a 50/50 duty cycle. 00 = (Default) 50/50 duty cycle: Clock output is 0000011111 01 = 10/90 duty cycle: Clock output is 0111111111 followed by 0000000001 ([DevCL] HDMI only) 10 = 20/80 duty cycle: Clock output is 0011111111 followed by 0000000011 ([DevCL] HDMI only) 11 = Reserved
15	Port Lane Reversal: This bit reverses the order of the 4 lanes within the port. Port lane reversal takes place on the Vblank after being written. It is an OEM configurable feature. 0 = (Default) Not reversed 1 = Reversed
14	Reserved
13	Clock Output Disable: This bit disables the clock output on the digital output port. For 8b/10b modes the clock output should be disabled. 0 = (Default) Clock output enabled 1 = Clock output disabled ([DevCL] only)
12	Reserved: MBZ
11:10	Encoding: These bits select among encoding types. It is set as part of the display detection process. Control codes for ANSI 8b/10b and TMDS encoding must be programmed using these bits. Please note that ANSI 8b/10b and TMDS encoding can only be enabled on one port at a time, as only one HPD pin is available for use between ports B and C. 00 –SDVO encoding. (Default) Refer to the SDVO EDS for control code descriptions. In this mode, the SDVOB hotplug input pin pair is used to generate hotplug. 01 = RESERVED 10 = TMDS encoding ([DevCL] external link and HDMI only) See the HDMI specification for control codes. In this mode, the external HPD pin is used to generate hotplug. In fixed frequency mode, start of fill and end of fill values for TMDS must be programmed using register 6114C. 11 = Reserved
9	Null packets enabled during Vsync This bit enables a null packet (32 bytes of a value of 0) to be sent when Vsync=1 on this port, required for HDMI operation. It also enables preambles and guardbands prior to the null packets, in accordance with the HDMI specification. It is only valid for modes that use TMDS encoding. 0 = Disable null infotrame packets when Vsync=1 on this port. (Default) 1 = Enable null infotrame packets when Vsync=1 on this port.
8	Color Range Select: This bit is used to select the color range of RGB outputs in HDMI mode. It is only valid when using TMDS encoding and 8-bit per color mode. 0 = Apply full 0-255 color range to the output (Default) 1 = Apply 16-235 color range to the output ([DevCL] only)
7	sDVOC Border Enable: This bit determines if the border data from native VGA or the timing generator is to be considered valid pixel data at the external component. 1 = Border to the sDVOC encoder is enabled. Blank# is used to generate the DE output (used in all cases except when the external scaler is used in a DVI panel, over SDVO) . 0 = Border to the sDVOC encoder is disabled. DE (Display Enable) is used



Bit	Description
6	Reserved: MBZ
5	Reserved : MBZ
4:3	Sync Polarity: Please note that sync polarity does not apply to ANSI coding. Indicates the polarity of Hsync and Vsync. Inverted polarity is transmitted as SYNC-BLANK-SYNC and standard polarity is transmitted as BLANK-SYNC-BLANK. For example, if Vsync is not inverted and Hsync is inverted, an Hsync period transmitted during Vsync would be transmitted as BLANK+VS+HS – BLANK+VS – BLANK+VS+HS. Please note that in native VGA modes, these bits have no effect. In native VGA modes, sync polarity is determined by VRshr3c2d76b[7:6], the VGA polarity bits in VGA control. 00 = VS and HS are active low (inverted) 01 = VS is active low (inverted), HS is active high 10 = VS is active high, HS is active low (inverted) 11 = (Default) VS and HS are active high
2	Reserved
1:0	Reserved : MBZ



2.8.8 VIDEO_DIP_CTL—Video DIP Control Register

Address Offset: 61170–61173h
 After Reset: 00000000h
 Access: Read/Write

Please note that writes to this register take effect immediately. Therefore, it is critical for software to follow the write and read sequences as described in the bit 31 text.

Bit	Description
31	Enable Graphics Data Island Packet (r/w): Data Island Packet (DIP) is a mechanism that allows up to 36 bytes to be sent over digital port during VBLANK, according to the HDMI specification. This includes header, payload, checksum and ECC information. Each type of DIP can be sent once per vsync, once every other vsync, or once. This data can be transmitted on either port (digital port B or digital port C), but not both simultaneously. Programming notes: • Partial DIPs are never sent out while the port is enabled. Disabling the DIP at the same time it is being transferred will result in the DIP being completed before the function is disabled. • Shutting off the port on which DIP is being transmitted will result in partial transfer of DIP data. There is no need to switch off the DIP enable bit if the port transmitting DIP is disabled. • When disabling both the DIP port and DIP transmission, first disable the port and then disable DIP. • Enabling an DIP function at the same time that the DIP would have been sent out (had it already been enabled) will result in the DIP being sent on the following frame. • Enabling should only be done after the buffer contents have been written. • If DIP is enabled but DIP types are all disabled, no DIP is sent. However, a single Null DIP will be sent at the same point in the stream that DIP packets would have been sent. This is done to keep the port in HDMI mode, otherwise it would revert to DVI mode. The "Null packets enabled during vsync" mode (bit #9 of port control register) overrides this behavior. Write sequence: 1) Wait for 2 VSynCs to ensure completion of any pending DIP transmissions. 2) Wait for 3 HSynCs to ensure the DIP being written will transmit on the next frame. 3) Disable the DIP type (bits 24:21) and set the DIP buffer index (bits 20:19) for the DIP being written. 4) Set the DIP access address (bits 3:0) to 0, or to the desired DWORD to be written. 5) Write DIP data 1 DWORD at a time. The IF access address autoincrements with each DWORD write, wrapping around to address 0 when the max buffer address size of 0xF has been reached. Please note that software must write an entire DWORD at a time. 6) Enable the DIP type and transmission frequency. Reading sequence: 1) Set the DIP buffer index (bits 20:19) for the DIP being read. 2) Set the DIP access address to 0, or to the desired DWORD to be read. 3) Read DIP data 1 DWORD at a time. The DIP access address autoincrements with each DWORD read, wrapping around to address 0 when the max buffer address size of 0xF has been reached. 0 = Video DIP is disabled 1 = Video DIP is enabled



Bit	Description
30:29	Port Select (r/w) This selects which port is to transmit the data island. This field must not be changed while data island transmission is enabled. Reserved settings are ignored. 00 = Reserved (Default) 01 = Digital port B 10 = Digital port C 11 = Reserved
28	DIP buffer transmission active indicator (read-only): This bit indicates whether the DIP buffer referred to by the DIP buffer index (bits 20:19 of this register) is currently being transmitted. 0 = Buffer transmission inactive 1 = Buffer transmission active
27:25	Reserved
24:21	Data Island Packet type enable (r/w): These bits enable the output of a given data island packet (DIP) type. It can be updated while the port is enabled and is immediately updated (not double-buffered). Within 2 vblank periods, the DIP is guaranteed to have been transmitted. XXX1 = Enable AVI DIP (Default = enabled) XX1X = Enable Vendor-specific DIP (Default = disabled) X1XX = Reserved 1XXX = Enable Source Product Description DIP (Default = disabled)
20:19	DIP buffer index (r/w): This field is used during programming of different DIPs. These bits are used as an index to their respective DIP buffers. The transmission frequency must also be written when programming the buffer. 00 = (Default) AVI DIP (31 bytes of space available) 01 = Vendor-specific DIP 10 = Reserved 11 = Source Product Description DIP
18	Reserved
17:16	Video DIP transmission frequency (r/w): These bits dictate the frequency of Video DIP transmission for the DIP buffer index designated in bits 20:19. When writing Video DIP data, this value is also latched when the first DW of the Video DIP is written. When read, this value reflects the Video DIP transmission frequency for the Video DIP buffer designated in bits 20:19. 00 = (Default) Send once 01 = Send every vsync (Default for AVI) 10 = Send at least every other vsync 11 = Reserved
15:12	Reserved: Read as zeroes
11:8	Video DIP buffer size (read-only): This reflects the buffer size in dwords available for the type of Video DIP being indexed by bits 20:19 of this register, including the header. The maximum size of a Video DIP is 36 bytes. Please note that this count includes ECC bytes, which are not writable by software. The minimum, including the checksum, header byte and ECC is ten bytes. These bits are immediately valid after write of the DIP index.



Bit	Description
7:4	Reserved: Read as zeroes
3:0	Video DIP RAM access address (r/w): Selects the DWORD address for access to the Video DIP buffers. This value is automatically incremented after each read or write of the Video DIP Data Register. The value wraps back to zero when it autoincrements past the max address value of 0xF. This field change takes effect immediately after being written. The read value indicates the current access address.

2.8.9 VIDEO_DIP_DATA–Video Data Island Packet Data

Address Offset: 61178–6117Bh
 After Reset: 00000000h
 Access: Read/write

Bit	Description
31:0	Video DIP DATA: When read, this returns the current value at the location specified in the Video DIP buffer index select and Video DIP RAM access address fields. The index used to address the RAM is incremented after each read or write of this register. DIP data can be read at any time. Data should be loaded into the RAM before enabling the transmission through the DIP type enable bit. Accesses to this register are on a per-DWORD basis.

Construction of DIP write:

MSB		LSB		
DW0	ECC for header (read-only, calculated by HW)	Header byte 2	Header byte 1	Header byte 0
DW1	Data byte 3	Data byte 2	Data byte 1: start of payload	Data byte 0: Checksum for payload
...				
DW8 (read-only, calculated by HW)	ECC		ECC for data bytes 7-13	ECC for data bytes 0-6



2.8.10 LVDS—Digital Display Port Control ([DevCL])

Address offset: 61180h–61183h
Default: 4000 0000h
Normal Access: Read/Write (Write Protect by Panel Power Sequencer on [DevCL])

Bit	Description
31	LVDS Port Enable: When disabled the LVDS port is inactive and in its low power state. Enabling the LVDS port changes the way that the PLL for this pipe is programmed. This bit must be set before the display PLL is enabled and the port is power sequenced on using the panel power sequencing logic. 0 = The port is disabled and all LVDS pairs are powered down. 1 = The port is enabled (port must be enabled before powering up a connected panel)
30	LVDS Port Pipe Assign 0 = The port gets data from pipe A 1 = The port gets data from pipe B (default)
29:26	Reserved
25	Dither Enable: This bit enables or disables (bypassing) 8-6-bit color dithering function. The usage of this bit would be on for 18-bpp panels and off for 24-bpp panels. 0 = disabled 1 = enabled
24	Data Format Select: Combined with the other control bits it selects the LVDS data format. Other control bits in this register determine if two channel is enabled and 18 or 24-bit color is enabled. 0 = 1x18.0, 2x18.0, 1x24.0 or 2x24.0 1 = 1x24.1 or 2x24.1
23	LE Control Enable: This bit is used when the second channel control signal field indicates that we are using the LE instead of HS and the two channel mode is enabled. In single channel mode, this bit has no effect. 0 = Send 0 on second channel HS (B2<2>) 1 = Send 1 on second channel HS
22	LF Control Enable: This bit is used when the second channel control signal field indicates that we are using the LF instead of VS and two channel mode is enabled. In single channel mode, this bit has no effect. 0 = Send 0 on second channel VS (B2<3>) 1 = Send 1 on second channel VS
21	VSYNC Polarity: This controls the polarity of the VSYNC indicator that is sent over the LVDS connection. Panels may require one or the other polarity or work with either polarity. 0 = No inversion (1=active) 1 = Invert the sense (0=active)



Bit	Description
20	HSYNC Polarity (LP Invert): This controls the polarity of the HSYNC indicator that is sent over the LVDS connection. Panels may require one or the other polarity or work with either polarity. 0 = No inversion (1=active) 1 = Invert the sense (0=active)
19	DE invert: This controls the polarity of the DE indicator that is sent over the LVDS connection. 0 = No inversion of DE (1=active) 1 = Invert the sense of DE (0=active)
18:17	Second Channel Control Signals: This bit only applies to the two channel modes of operation it has no effect in single channel modes. 00 = Send DE, HS, VS on second channel if enabled 01 = Reserved 10 = Do not send DE, HS, VS on second channel use zero instead 11 = Use DE=0, HS=LE, VS=LF on second channel
16	Channel Reserved Bits 0 = Send 0 for the channel reserved bits 1 = Send duplicate data bit for reserved bits
15	LVDS Border Enable: This selects whether the border data should be included in the active display data sent to the panel. Border should be used when in VGA centered (un-scaled) mode or when scaling a 4:3 source image to a wide screen panel (typical 16:9). 0 = Border to the LVDS transmitter is disabled. DE (Display Enable) is used. 1 = Border to the LVDS transmitter is enabled. Blank# is used as DE for the panel.
14:11	Reserved
10	Buffer Power Down State: This bit selects the state of the LVDS buffers during a powered down state caused by the power sequence logic power down. This selection will be made based on the connected panel requirements. 0 = Zero Volts (Driven on both lines of the pairs) 1 = Tri-State (High impedance state)
9:8	ClkA, A0, A1, A2 Control: This field controls the A0-A2 data pairs and CLKA. It sets the highest level of activity that is allowed on these lines when the panel is powered on. Power sequencing for LVDS connected panels overrides the control. When the power sequencer is in the power down mode all signals are in the power down state. 00 = Power Down all A channel signals including A3 (0V) 01 = Power up – A0, A1, A2 Data bits forced to 0, Timing active, Clock Active 10 = Reserved 11 = Power up – Data lines and clock active



Bit	Description
7:6	Eight bit color channel A3, (B3) Control: This field can control both the A3 and B3 data pairs. Enabling those pairs indicates the selection of 8-bit per color channel mode. It sets the highest level of activity that is allowed on these lines when the panel is powered on. The A3 pair will only be powered up if both this field and the A0, A1, A2, CLKA field indicates that the pair should be powered up and will only be active if both indicate that it should be active. The B3 pair will only be powered up if both this field and the B0, B1, B2, (B3) field indicates that the pair should be powered up and will only be active if both indicate that it should be active. Power sequencing for LVDS connected panels overrides the control. When the power sequencer is in the power down mode all signals are in the power down state. 00 = Power Down all signals A3, B3 (common mode) 01 = Power up – A3, (B3) Data (pixel data not control) lines forced to 0 output 10 = Reserved 11 = Power up – A3, (B3) Data lines active
5:4	Two channel mode ClkB Control: When in two channel mode, this field controls the CLKB pair. It sets the highest level of activity that is allowed on these lines when the panel is powered on. The CLKB pair should only be powered up if the B0, B1, B2, (B3) field indicates that the second channel should be powered up and will only be active if both indicate that it should be active. Power sequencing for LVDS connected panels overrides the control. 00 = Power Down CLKB (common mode) 01 = Power up – CLKB Forced to 0 10 = Reserved 11 = Power up – Clock B active
3:2	Two channel mode B0, B1, B2 Control: This field controls both the set B0-B2 data pairs. It sets the highest level of activity that is allowed on these lines when the panel is powered on. Power sequencing for LVDS connected panels overrides the control. During single channel operation (1x18.0), these bits need to be both zero. Two channel operation is selected by setting them to ones. Note that the second clock can be optionally enabled or disabled by the two channel mode ClkB control field. 00 = Power Down all signals including B3 and CLKB 01 = Power up – B0, B1, B2, Data lines forced to 0, timing is active 10 = Reserved 11 = Power up – Data lines active (color and timing)
1:0	Reserved



2.9 Panel Registers

2.9.1 Panel Power Sequencing Registers ([DevCL])

2.9.1.1 PP_STATUS—Panel Power Status Register ([DevCL])

Address offset: 61200-61203h
 After Reset: 08000000h
 Normal Access: Read-Only

Bit	Description
31	Panel Power On Status 0 = Indicates that the panel power down sequencing has completed. A power cycle delay may be currently active. It is safe and allowed to program pipe timing and DPLL registers. If this bit is not a zero, it activates the register write protect and writes to those registers will be ignored unless the write protect key value is set in the panel sequencing control register. 1 = In conjunction with bits Power Sequence Progress field and Power Cycle Delay Active, this bit set to a one indicates that the panel is currently powered up or is currently in the power down sequence and it is unsafe to change the pipe timing and DPLL registers for the pipe that is assigned to the LVDS output. If the LVDS port is selected as the target for the panel control, Software is responsible for enabling the LCD display by writing a "1" to the port enable bit only after all pipe timing, DPLL registers are properly programmed, and the PLL has locked to the reference signal. This bit is cleared (set to "0") only after the panel power down sequencing is completed.
30	Require Asset Status: This bit indicates the status of programming of the display PLL and the selected display port. This a power on cycle will not be allowed unless this status indicates that the required assets are programmed and ready for use. 0 = All required assets are not properly programmed. 1 = All required assets are ready for the driving of a panel. The following conditions determine that the assets are ready: 1) Display Pipe PLL Enabled and frequency locked (bit-31 of DPLL Control Register for the pipe attached to the LVDS port). 2) Display Pipe Enabled (bit-31 of PIPECONF—Pipe Configuration Register. For the pipe attached to the LVDS port) 3) LVDS Port is Programmed Enabled
29:28	Power Sequence Progress 00 = Indicates that the panel is not in a power sequence 01 = Indicates that the panel is in a power up sequence (may include power cycle delay) 10 = Indicates that the panel is in a power down sequence 11 = Reserved



Bit	Description
27	Power Cycle Delay Active: Power cycle delays occur after a panel power down sequence or after a hardware reset. On reset, a power cycle delay will occur using the default value for the timing. 0 = A power cycle delay is not currently active 1 = A power cycle delay (T4) is currently active
26:4	Reserved
3:0	Internal Sequence State (For test / debug) 0000 = Power Off Idle (S0.0) 0001 = Power Off, Wait for cycle delay (S0.1) 0010 = Power Off (S0.2) 0011 = Power Off (S0.3) 0100 = Reserved 0101 = Reserved 0110 = Reserved 0111 = Reserved 1000 = Power On Idle (S1.0) 1001 = Power On, (S1.1) 1010 = Power On, (S1.2) 1011 = Power On, Wait for cycle delay (S1.3) 1100 = Reserved 1101 = Reserved 1110 = Reserved 1111 = – Reset



2.9.1.2 PP_CONTROL—Panel Power Control Register ([DevCL])

Address offset: 61204-61207h
 Default: 000000000h
 Normal Access: Read/Write

Bit	Description
31:16	Write Protect Key ABCD – Write protect off When this field is programmed to anything except the write protect off setting and the panel is either powered up or in the process of a power up sequence, a set of registers involved in generation of panel timing or control become write-protected. Any write cycles to those write-protected registers, while they will complete as normal, will not change the value of the register when write-protected. When this register field contains the write protect off key value, write protect will be unconditionally disabled. In situations where the LVDS port is unused, the port should remain powered down and the write protect will be inactive. This field in normal operation should be left to all zeros and never programmed with the key value. It exists only to allow testing and workarounds. List of Write-protected registers: (LVDS and Panel sequencing Registers): LVDS – Digital Display Port Control – Address: 61180h–61183h Panel power on sequencing delays - Address: 61208-6120Bh Panel power off sequencing delays – Address: 6120Ch – 6120Fh Panel power cycle delay and Reference Divisor – Address: 61210h – 61213 (DPLL registers): DPLL Control Registers FPB0—DPLL Divisor Register FPB1—DPLL Divisor Register 1 (Display Pipe timing registers except source size) HTOTAL—Horizontal Total Register HBLANK—Horizontal Blank Register HSYNC_—Horizontal Sync Register VTOTAL_—Vertical Total Register VBLANK_—Vertical Blank Register VSYNC_—Vertical Sync Register
15:2	Reserved
1	Power Down on Reset: Enabling this bit causes the panel to power down when a reset warning comes to the GMCH from the ICH. When system reset is initiated, the LVDS automatically begins the panel power down sequence. If the panel is not on during a reset event, this bit is ignored. 0 = Do not run panel power down sequence when reset is detected 1 = Run panel power down sequence when system is reset



Bit	Description
0	Power State Target: Writing this bit can occur any time, it will only be used at the completion of any current power cycle. 0 = The panel power state target is off, if the panel is either on or in a power on sequence, a power off sequence is started as soon as the panel reaches the power on state. This may include a power cycle delay. If the panel is currently off, there is no change of the power state or sequencing done. 1 = The panel power state target is on, if the panel is in either the off state or a power off sequence, if all pre-conditions are met, a power on sequence is started as soon as the panel reaches the power off state. This may include a power cycle delay. If the panel is currently off, there is no change of the power state or sequencing done. While the panel is on or in a power on sequence, the register write lock will be enabled.

2.9.1.3 PP_ON_DELAYS—Panel Power on Sequencing Delays ([DevCL])

Address offset: 61208-6120Bh
Default: 00000000h
Normal Access: Read/Write (Write Protect by Panel Power Sequencer on [DevCL])

Bit	Description
31:30	Panel Control port select: These bits define to which port the LCD panel is connected. This is used for automatic control of the panel power. If the selected port is disabled or if the port is not on pipe-B, then, the power sequence will not allow a panel power up. 00 = Panel is connected to the LVDS display port 01 = Reserved 10 = Reserved 11 = Reserved The selection of non-existent ports is not allowed (such as DVO A). This programming will disable panel power sequencing logic.
29	Reserved
28:16	Power up delay: Programmable value of panel power sequencing delay during panel power up. This provides the time delay for the T1+T2 time sequence. The time unit used is the 100us timer.
15:13	Reserved
12:0	Power on to Backlight enable delay: Programmable value of panel power sequencing delay during panel power up. This provides the time delay for the T5 time sequence. The time unit used is the 100us timer.



2.9.1.4 PP_OFF_DELAYS—Panel Power off Sequencing Delays ([DevCL])

Address offset: 6120Ch – 6120Fh
Default: 00000000h
Normal Access: Read/Write (Write Protect by Panel Power Sequencer on Mobile products)

Bit	Description
31:29	Reserved
28:16	Power Down delay: Programmable value of panel power sequencing delay during power up. This provides the time delay for the T3 time sequence. The time unit used is the 100us timer.
15:13	Reserved
12:0	Power Backlight off to power down delay: Programmable value of panel power sequencing delay during power down. This provides the time delay for the Tx time sequence. The time unit used is the 100us timer.



2.9.1.5 PP_DIVISOR—Panel Power Cycle Delay and Reference Divisor ([DevCL])

Address offset: 61210h – 61213
 Default: 00270F04h
 Normal Access: Read/Write (Write Protect by Panel Power Sequencer on Mobile Products)

This register selects the reference divisor and controls how long the panel must remain in a power off condition once powered down. This has a default value that allows a timer to initiate directly after device reset. If the panel limits how fast we may sequence from up to down to up again. Typically this is .5-1.5 sec. but limited to 400ms in the SPWG specification. This register forces the panel to stay off for a programmed duration. Special care is needed around reset and D3 cold situations to conform to power cycle delay specifications.

Bit	Description								
31:8	Reference divider: This field provides the value of the divider used for the creation of the panel timer reference clock. The output of the divider is used as the fastest of the three time bases (100us) for all other timers. The other time bases are divided from this frequency. The value of zero should not be used. When it is desired to divide by N, the actual value to be programmed is (N/2)-1. The value should be (100*RefinMHz/2)-1. The default value assumes the default value for the display core clock that is for [DevCL] a 200MHz reference value. The following are examples for other memory speeds. <table> <tr> <td>Display Core Frequency</td><td>Value of Field</td></tr> <tr> <td>233MHz</td><td>2D81h</td></tr> <tr> <td>200MHz</td><td>270Fh</td></tr> <tr> <td>133MHz</td><td>19F9h</td></tr> </table>	Display Core Frequency	Value of Field	233MHz	2D81h	200MHz	270Fh	133MHz	19F9h
Display Core Frequency	Value of Field								
233MHz	2D81h								
200MHz	270Fh								
133MHz	19F9h								
7:5	Reserved								
4:0	Power Cycle Delay: Programmable value of time panel must remain in a powered down state after powering down. For devices coming out of reset, the default values will define how much time must pass before a power on sequence can be started. This field uses the .1 S time base unit from the divider. If the panel power on sequence is attempted during this delay, the power on sequence will commence once the power cycle delay is complete. Writing a value of 0 selects no delay or is used to abort the delay if it is active. During the initial power up reset, a D3 cold power cycle, or a user instigated system reset, the timer will be set to the default value and the count down will begin after the de-assertion of reset. Writing this field to a zero while the count is active will abort this portion of the sequence. This corresponds to the T4 of the SPWG specification. Note: Even if the panel is not enabled, the T4 count happens after reset. This register needs to be programmed to a "+1" value. For instance for meeting the SPWG specification of 400mS, program 5 to achieve at least 400mS delay prior to powerup.								



2.9.2 Panel Fitting Registers

2.9.2.1 PFIT_CONTROL—Panel Fitting Controls

Address offset: 61230h
 Default: 20000000h
 Normal Access: Read/Write

Bit	Description
31	Panel Fitting Enabled: Disables the panel fitting function by forcing pixels to bypass. Panel fitting must be disabled when running VGA native modes or interlaced modes on the same pipe. Panel fitting should be enabled or disabled before the pipe is enabled. 0 = Bypass the panel fitting (1:1 ratio) 1 = Enable panel fitting (Ratios include 1:1)
30:29	Pipe Select: Indicates the pipe attached to the panel fitter 00 = Panel fitter is attached to Display Pipe A. 01 = Panel fitter is attached to Display Pipe B. This is the default after reset. 10 = Reserved for pipe C 11 = Reserved for pipe D
28:26	Scaling Mode 000 = Auto-scale (source and destination should have the same aspect ratios) 001 = Programmed scaling: Values in register 61234h will be used for horizontal and vertical scaling factors 010 = Pillarbox (example: 4:3 to 16:9 auto conversion) use only when destination has wider aspect ratio than source 011 = Letterbox (example: 16:9 to 4:3 auto conversion) use only when destination has taller aspect ratio than source 1xx = Reserved
25:24	Filter Coefficient Select: Selects the set of predefined filter coefficients to use for panel fitting 00 = Fuzzy filtering 01 = Crisp edge enhancing filtering 10 = Median between fuzzy and crisp filtering 11 = Reserved
23	Debug: Force Two Line Mode
22	Debug: Force Three Pixel Mode when in Two Line Mode
21:19	Debug: Create extra stalls in VGA mode 000 = No stall 001 = 33% stall 010 = 50% stall 011 = 66% stall 100 = 75% stall 101 = 80% stall 110 = 90% stall 111 = Reserved
18:5	Reserved:
4	Reserved : write as zero
3:0	Reserved



2.9.2.2 PFIT_PGM_RATIOS—Programmed Panel Fitting Ratios

Address offset: 61234h
 Default: 00000000h
 Normal Access: Read/Write

When programmed scaling mode (Panel Fitting Controls 28:26 = “001”) is selected, this determines the vertical and horizontal ratios used for panel fitting scaling. The values should be based on the source sizes and active sizes programmed into the pipe timing registers. The values written into the register should be rounded to the proper number of bits for the best precision. The value programmed should be $[(\text{source size register value} + 1) / (\text{active size register value} + 1)]$

When programmed scaling mode is not selected, read back of this register gives the auto-generated vertical and horizontal scaling ratios used for panel fitting scaling. Register writes will be ignored. The ratios are calculated each VBLANK. When in HiRes modes, the values are based on the source sizes and active sizes programmed into the pipe timing registers. When in VGA modes it is determined by the VGA source sizes calculated by the VGA and active sizes from the pipe timing registers. VGA source sizes may have invalid values due to mode change transitions. These will eventually be correct when the mode change is complete. The value read is internally generated $[(\text{source size register value} + 1) / (\text{active size register value} + 1)]$

For each register field the MSB is the 1 bit integer value and the lower 12 bits are the fractional value. A value of 1.0 will indicate 1:1 scaling. A value greater than 1.0 will indicate downscaling. A value less than 1.0 will indicate upscaling. The vertical and horizontal ratios are usually identical, except for when source and active aspect ratios differ.

Bit	Description
31:29	Reserved : Reads as zeros
28:16	Panel fitting vertical ratio: Vertical scaling ratio for panel fitting.
15:13	Reserved – Reads as zeros
12:0	Panel fitting horizontal ratio: Horizontal scaling ratio for panel fitting.

2.9.2.3 Reserved (Used to be Auto Scaling Ratios Readback)

Address offset: 61238h
 Default: 00000000h
 Normal Access: Read-Only

Bit	Description
31:0	Reserved



2.9.2.4 Reserved (Used to be Scaling Initial Phase)

Address offset: 6123Ch
 Default: 01000100h
 Normal Access: Read/Write

Bit	Description
31:0	Reserved

2.9.3 Backlight Control and Modulation Histogram Registers ([DevCL])

2.9.3.1 BLC_PWM_CLT2—Backlight PWM Control Register 2 ([DevCL])

Address offset: 61250h
 Default: 00000000h
 Normal Access: Read/Write
 Double Buffered: No

Bit	Description
31	PWM Enable: This bit enables the PWM counter logic 0 = PWM disabled (drives 0 always) 1 = PWM enabled
30	BLM Legacy Mode 0 = PWM Duty Cycle is derived from the Backlight Duty Cycle only 1 = PWM Duty Cycle is a combination of Backlight Duty Cycle and Legacy Backlight Control (LBPC) Note: '1' implies the Duty Cycle = $\text{if}(\text{BPC}[7:0] < \text{xFF}) \text{ then } \text{BPCR}[15:0] * \text{BPC}[7:0] \text{ Else } \text{BPCR}[15:0]$
29	PWM Pipe assignment: This bit assigns PWM to a pipe. The PWM counter will run off of this pipe's PLL. The PWM function must be disabled in order to change the value of this field. 0 = Pipe A 1 = Pipe B
28	Backlight Polarity. This field controls the polarity of the PWM signal. 0 = Active High 1 = Active Low
27	Reserved



Bit	Description
26	Phase-In Interrupt Status: This bit will be set by hardware when a Phase-In interrupt has occurred. Software will clear this bit by writing a '1', which will reset the interrupt generation. [DevCL-A,B] Reserved
25	Phase In Enable: Setting this bit enables a PWM phase in based on the programming of the Phase In registers below. This bit clears itself when the phase in is completed.
24	Phase In Interrupt Enable: Setting this bit enables an interrupt to be generated when the PWM phase in is completed.
23:16	Phase In time base: This field determines the number of VBLANK events that pass before one increment occurs. 0 = invalid 1 = 1 vblank 2 = 2 vblanks etc.
15:8	Phase In Count: This field determines the number of increment events in this phase in. Writes to this register should only occur when hardware-phase-ins are disabled. Reads to this register can occur any time, where the value in this field indicates the number of increment events remaining to fully apply a phase-in request as hardware automatically decrements this value. A value of 0 is invalid.
7:0	Phase In Increment: This field indicates the amount to adjust the PWM duty cycle register on each increment event. This is a two's complement number.

2.9.3.2 BLC_PWM_CTL—Backlight PWM Control Register ([DevCL])

Address offset: 61254h
Default: 00000000h
Normal Access: Read/Write

Bit	Description
31:16	Backlight Modulation Frequency. This field determines the number of time base events in total for a complete cycle of modulated backlight control. This field is normally set once during initialization based on the frequency of the clock that is being used and the desired PWM frequency. This value represents the period of the PWM stream in display core clocks multiplied by 128.
15:0	Backlight Duty Cycle. This field determines the number of time base events for the active portion of the PWM backlight control. This should never be larger than the frequency field. A value of zero will turn the backlight off. A value equal to the backlight modulation frequency field will be full on. This field gets updated when it is desired to change the intensity of the backlight, it will take effect at the end of the current PWM cycle. This value represents the active time of the PWM stream in display core clock periods multiplied by 128.



2.9.3.3 BLM_HIST_CTL—Image Enhancement Histogram Control Register ([DevCL])

Address offset: 61260h
 Default: 00000000h
 Normal Access: Read/Write
 Double buffer: No

Bit	Description
31	Image Enhancement Histogram Enabled: This bit enables the Image Enhancement histogram logic to collect data. 0 = Image histogram is disabled 1 = The Image histogram is enabled. When this bit is changed from a zero to a one, histogram calculations will begin after the next VBLANK of the assigned pipe.
30	Image Enhancement Modification Table Enabled: This bit enables the Image Enhancement modification table. 0 = disabled 1 = enabled. When this bit is changed from a zero to a one, modifications begin after the next VBLANK of the assigned pipe.
29	Image Enhancement Pipe assignment: This bit assigns the IE function to a pipe. IE events will be synchronized to the VBLANK of the selected pipe. The IE function must be disabled in order to change the value of this field. 0 = Pipe A 1 = Pipe B
28:25	Reserved : Always write as 0s.
24	Histogram Mode Select: 0 = YUV Luma Mode 1 = HSV Intensity Mode - Reserved on [DevCL]
23:16	Sync to Phase In Count: This field indicates the phase in count number on which the Image Enhancement table will be loaded if the Sync to Phase in is enabled.
15	Reserved : Always write as 0.
14:13	Enhancement mode: 00 = Direct look up mode 01 = Additive mode 10 = Multiplicative mode - Reserved on [DevCL] 11 = Reserved
12	Sync to Phase In: Setting this bit enables the double buffered registers to be loaded on the phase in count value specified instead of the next vblank.
11	Bin Register Function Select: This field indicates what data is being written to or read from the bin data register. 0 = Bin Threshold Count. A read from the bin data register returns that bin's threshold value from the most recent vblank load event (guardband threshold trip). Valid range for the Bin Index is 0 to 31. 1 = Bin Image Enhancement Value. Valid range for the Bin Index is 0 to 32



Bit	Description
10:7	Reserved : Always write as 0's.
6:0	Bin Register Index: This field indicates the bin number whose data can be accessed through the bin data register. This value is automatically incremented by a read or a write to the bin data register if the busy bit is not set.

2.9.3.4 Image Enhancement Bin Data Register ([DevCL])

Address offset: 61264h
Default: 00000000h
Normal Access: Depends on function)
Double buffer: Depends on function

Writes to this address are steered to the correct register by programming the Bin Register Function Select and the Bin Register Index.

Function 0 usage (Threshold Count) this Function is Read-Only

Bit	Description
31	Busy Bit: If set , the engine is busy, the rest of the register is undefined. If clear, the register contains valid data.
30:22	Reserved
21:0	Bin Count: The total number of pixels in this bin, value is updated at the start of each vblank.

Function 1 usage (Image Enhancement) this Function is Read/Write

Bit	Description
31:10	Reserved
9:0	Image Bin Correction Factor The correction value for this bin, writes to this register are double buffered on the next vblank if in normal mode, or on the phase in Sync event frame if it is enabled. The value written here is the 10bit corrected channel value for the lowest point of the bin.

For Additive mode the data format is a two's complement number. For multiply mode the correction factor is a 1.9 fixed point number is the range 0.0 to 1.9999. (ex, 1.000 = 200h, 1.5 = 300h, 0.5 = 100h)



2.9.3.5 Histogram Threshold Guardband Register ([DevCL])

Address offset: 61268h
Default: 00000000h
Normal Access: Read/Write
Double buffer: Yes

Bit	Description
31	Histogram Interrupt enable: 0 = Disabled 1 = Enabled. This generates a histogram interrupt once a Histogram event occurs.
30	Histogram Event status: When a Histogram event has occurred, this will get set by the hardware. For any more Histogram events to occur, the software needs to clear this bit by writing a '1'. The default state for this bit is '0'. 0 = Histogram event has not occurred. 1 = Histogram event has occurred. [DevCL-A,B]: Read-Only
29:22	Guardband Interrupt Delay: An interrupt is generated after this many consecutive frames of the guardband threshold being surpassed. This value is double buffered on start of vblank. A value of 0 is invalid.
21:0	Threshold Guardband: This value is used to determine the guardband for the threshold interrupt generation. This single value is used for all the segments. This value is double buffered on start of vblank



2.10 Display and Cursor Registers (70000h–7FFFFh)

These registers are memory mapped and accessible through normal 32 bit, 16 bit, or 8-bit accesses.

2.10.1 Display Pipeline A

2.10.1.1 PIPEA_DSL—Pipe A Display Scan Line

Memory Offset Address: 70000h
Default: 00000000h
Normal Access: Read-Only

This register enables the read back of the display pipe vertical “line counter”. The display line value is from the display pipe A timing generator and is reset to zero at the beginning of a scan. The value increments at the leading edge of HSYNC and can be safely read any time. For normal operation, scan line zero is the first active line of the display. When in VGA centering mode, the scan line 0 is the 1st active scan line of the pseudo border not the centered active VGA image. In interlaced display timings, the scan line counter provides the current line in the field. One field will have a total number of lines that is one greater than the other field.

Programming Note: In order to cause the scan line logic to report the correct Line Counter value, the corresponding Display Pipeline timing registers must be programmed to valid, non-zero (e.g., 640x480 @ 60Hz) values before enabling the Pipe or programming VGA timing and enabling native VGA.

Bit	Descriptions
31	Reserved: Read-only.
30:13	Reserved: Read-only.
12:0	Line Counter for Display [12:0]: Provides read back of the display pipe A vertical line counter. This is an indication of the current display scan line to be used by software to synchronize with the display.



2.10.1.2 PIPEA_SLC—Pipe A Display Scan Line Count Range Compare

Memory Offset Address: 70004h
Default: 00000000h
Normal Access: Read-Only

This register can be written via the command stream processor using the MI_LOAD_SCAN_LINES_INCL or MI_LOAD_SCAN_LINES_EXCL commands. They can safely be accessed at any time.

The Top and Bottom Line Count Compare registers are compared with the display line values from display A timing generator. The Top compare register operator is a less than or equal, while the Bottom compare register operator is a greater than or equal. The results of these 2 comparisons are communicated to the command stream controller for generating interrupts, status, and command stream flow control (“wait for within range” and “wait for not within range”). For range check, the value programmed should be the **(desired value – 1)**, so for line 0, the value programmed is VTOTAL, and for line 1, the value programmed is 0.

Bit	Descriptions
31	Inclusive / Exclusive: 1 = Inclusive: within the range. 0 = Exclusive: outside of the range.
30:29	Reserved: Read-only.
28:16	End Scan Line Number: This field specifies the ending scan line number of the Scan Line Window. Format = U16 in scan lines, where scan line 0 is the first line of the display frame. Range = [0, Display Buffer height in lines-1].
15:13	Reserved: Read-only.
12:0	Start Scan Line Number: This field specifies the starting scan line number of the Scan Line Window. Format = U16 in scan lines, where scan line 0 is the first line of the display frame. Range = [0, Display Buffer height in lines-1].



2.10.1.3 PIPEACONF—Pipe A Configuration Register

Memory Offset Address: 70008h
 Default: 00000000h
 Normal Access: Read/Write double buffered

Bit	Descriptions
31	Pipe A Enable: Setting this bit to the value of one, turns on pipe A. This must be done before any planes are enabled on this pipe. Changing it to a zero should only be done when all planes that are assigned to this pipe have been disabled. Turning the pipe enable bit off disables the timing generator in this pipe. Plane disable occurs after the next VBLANK event after the plane is disabled. Synchronization pulses to the display are not maintained if the timing generator is disabled. Power consumption will be at its lowest state when disabled. A separate bit controls the DPLL enable for this pipe. Pipe timing registers should contain valid values before this bit is enabled. 0 = Disable 1 = Enable
30	Pipe State: This bit indicates the actual state of the pipe. Since there can be some delay between disabling the pipe and the pipe actually shutting off, this bit indicates the true current state of the pipe. 0 = Disabled 1 = Enabled
29	Reserved
28:27	Frame Start Delay: (TEST MODE) Used to delay the frame start signal that is sent to the display planes. Normal operation uses the default 00 value and test modes can use the delayed frame start to shorten the test time. Care must be taken to insure that there are enough lines during VBLANK to support this setting. 00 = Frame Start occurs on the first HBLANK after the start of VBLANK 01 = Frame Start occurs on the second HBLANK after the start of VBLANK 10 = Frame Start occurs on the third HBLANK after the start of VBLANK 11 = Frame Start occurs on the forth HBLANK after the start of VBLANK
26	Reserved
25	FORCE_BORDER: (TEST MODE) 0 = Normal Operation 1 = Color information is ignored and border color is substituted during active region
24	Pipe A Gamma Unit Mode: This bit selects which mode the pipe gamma correction logic works in. In the palette mode, it behaves as a 3X256x8 RAM lookup. VGA and indexed mode operation should use the palette in 8-bit mode. In the 10-bit gamma mode, it will act as a piecewise linear interpolation. Other gamma units such as in the overlay or sprite are unaffected by this bit. 0 = 8-bit Palette Mode 1 = 10-bit Gamma Mode



Bit	Descriptions
23:21	Interlaced Mode These bits are used for software control of interlaced behavior. They are updated immediately if the pipe is off, or in the vertical blank after programming if pipe is enabled. 0xx = Progressive 100 = Interlaced embedded panel using programmable vertical sync shift. 101 = Interlaced using vertical sync shift. Backup option to 110 setting. 110 = Interlaced with VSYNC/HSYNC Field Indication using legacy vertical sync shift. Used for SDVO. 111 = Interlaced with Field 0 Only using legacy vertical sync shift. Not used Note: VGA display modes, sDVO line stall, and Panel fitting do not work while in interlaced modes Setting the Interlaced embedded panel mode causes hardware to automatically modify the output to match the specifications of panels that support interlaced mode.
20	Reserved
19	Display / Overlay Planes Off: This bit when set will cause all enabled Display and overlay planes that are assigned to this pipe to be disabled by overriding the current setting of the plane enable bit, at the next VBLANK. Timing signals continue as they were but the screen becomes blank. Setting the bit back to a zero will then allow the display and overlay planes to resume on the following VBLANK. 0 = Normal Operation 1 = Planes assigned to this pipe are disabled.
18	Cursor Planes Off: This bit when set will cause all enabled cursor planes that are assigned to this pipe to be disabled by overriding the current setting of the plane enable bit, at the next VBLANK. Timing signals continue as they were but the cursor(s) no longer appear on the screen. Setting the bit back to a zero will then allow the cursor planes to resume on the following VBLANK. 0 = Normal Operation 1 = Planes assigned to this pipe are disabled.
17:16	Refresh Rate CxSR Mode Association These bits select how refresh rates are tied to big FIFO mode on pipe A. When they are set to anything other than 00, bits 23:21 of this register must be programmed to 0xx. Switching between 01 and 10 settings directly is not allowed. Software must program this field to 00 before switching. Software is responsible for enabling this mode only for integrated display panels that support corresponding mode. 00 – Default – no dynamic refresh rate change enabled. Software control only. 01 – Progressive-to-progressive refresh rate change enabled and tied to big FIFO mode. Pixel clock values set in FPA0/FPA1 settings in the DPLL control register and FPA0/FPA1 divider registers. FPA0 is tied to non-big-FIFO mode 10 – Progressive-to-interlaced refresh rate change enabled and tied to big FIFO mode. Pixel clock value does not change in this case. Scaling must be disabled in this mode. Uses programmable VS shift 11 – Reserved
15:10	Reserved: Write as zero
9:1	Reserved : MBZ
0	Reserved: Write as zero



2.10.1.4 PIPEAGCMAXRED—Pipe A Gamma Correction Max Red

Memory Offset Address: 70010h
Default: 00010000h
Normal Access: Read/Write

Bit	Descriptions
31:17	Reserved
16:0	Max Red Gamma Correction Point: 129 th reference point for red channel of the pipe piecewise linear gamma correction. The value should always be programmed to be less than or equal to 1024.0. Format: 11.6 Default: 0x10000

2.10.1.5 PIPEAGCMAXGREEN—Pipe A Gamma Correction Max Green

Memory Offset Address: 70014h
Default: 00010000h
Normal Access: Read/Write

Bit	Descriptions
31:17	Reserved
16:0	Max Green Gamma Correction Point: 129 th reference point for green channel of the pipe piecewise linear gamma correction. The value should always be programmed to be less than or equal to 1024.0. Format: 11.6 Default: 0x10000

2.10.1.6 PIPEAGCMAXBLUE—Pipe A Gamma Correction Max Blue

Memory Offset Address: 70018h
Default: 00010000h
Normal Access: Read/Write

Bit	Descriptions
31:17	Reserved
16:0	Max Blue Gamma Correction Point: 129 th reference point for blue channel of the pipe piecewise linear gamma correction. The value should always be programmed to be less than or equal to 1024.0. Format: 11.6 Default: 0x10000



2.10.1.7 PIPEASTAT—Pipe A Display Status

Memory Offset Address: 70024h
 Default: 00000000h
 Normal Access: Read/Write

This register is the second level of a two level interrupt and status scheme. A single bit in the first line interrupt status register represents the state of this register which is equal to the AND of a status bits with their corresponding enable bits OR'ed together. First line interrupt status bits can cause interrupts or writes of the status register to cacheable memory. Bits in this register indicate the status of the display pipe A and can cause interrupt status bit changes in the first level interrupt and status register. Status bits in this register as 'sticky' and once they are set will be cleared by writing a one to that bit. A write of a zero will not have an effect on the corresponding Interrupt status bit. The corresponding enable bits will determine if the interrupt status bit should be used in the first line interrupt status register. When an interrupt occurs, the first line interrupt register indicates the second line source of the interrupt. Reading the second line register will determine the precise source for the interrupt.

Programming:

1. Prior to clearing a Display Pipe-sourced interrupt (e.g., Display Pipe A VBLANK) in the IIR, the corresponding interrupt (source) status in the PIPEASTAT or PIPEBSTAT register (e.g., Pipe A VBLANK Interrupt Status bit of PIPEASTAT) must first be cleared. Note that clearing these status bits requires writing a '1' to the appropriate bit position.

Bit	Descriptions
31	FIFO A Under-run Status: Set when a pipe A FIFO under-run occurs, cleared by a write of a 1. An underrun has occurred on an attempt to pop an empty FIFO. This does not feed into the first line interrupt status register. This will occur naturally during mode changes, to be useful, it should be cleared after a mode change has occurred. This bit is only valid after Pipe A has been completely configured. 1 = FIFO A Underflow occurred 0 = FIFO A Underflow did not occur
30	Reserved:
29	CRC Error Enable: This will enable the consideration of the CRC error status bit in the first line interrupt/status logic. 0 = CRC Error Detect Disabled 1 = CRC Error Detect Enabled
28	CRC Done Enable: This will enable the consideration of the CRC error status bit in the first line interrupt/status logic. 0 = CRC Done Detect Disabled 1 = CRC Done Detect Enabled
27	GMBUS Event Enable: This will enable the use of the GMBUS interrupt status bit in the first line interrupt/status logic. 0 = No GMBUS event enabled 1 = GMBUS event enabled



Bit	Descriptions
26	Reserved: Write as zero
25	Vertical Sync Interrupt Enable: This will enable the consideration of the vertical sync interrupt status bit in the first line interrupt logic. 0 = Vertical Sync Interrupt/Status Disabled 1 = Vertical Sync Interrupt/Status Enabled
24	Display Line Compare Enable: This will enable the consideration of the line compare interrupt status bit in the first line interrupt/status logic. 0 = Display Line Compare Interrupt/Status Disabled 1 = Display Line Compare Interrupt/Status Enabled
23	[DevCL]: DPST Event Enable: This interrupt is generated by the DPST logic. 0 = No DPST event enabled 1 = DPST event enabled
22	[DevCL]: Legacy BLC Event Enable: This will enable writes to the PCI Backlight Control Register to cause and the display A event status to be set and an Interrupt if Display A Event interrupt is enabled. 0 = No BLC Event enabled 1 = BLC Event enabled
21	Odd Field Interrupt Event Enable: This bit should only be used when this pipe is in an interlaced display timing. 0 = Odd Field Event disable 1 = Odd Field Event enable
20	Even Field Interrupt Event Enable: This bit should only be used when this pipe is in an interlaced display timing. 0 = Even field Event disable 1 = Even field Event enable
19	Reserved: Write as zero.
18	Start of Vertical Blank Interrupt Enable: This will enable the consideration of the start of vertical blank interrupt status bit in the first line interrupt/status logic. 0 = Start of Vertical Blank Interrupt/Status Disabled 1 = Start of Vertical Blank Interrupt/Status Enabled
17	Vertical Blank Interrupt Enable: This will enable the consideration of the vertical blank interrupt status bit in the first line interrupt/status logic. 0 = Vertical Blank Interrupt/Status Disabled 1 = Vertical Blank Interrupt/Status Enabled
16	Overlay Registers Updated Enable: 0 = Overlay Registers have been updated during Vertical Blank Status Disabled 1 = Overlay Registers have been updated during Vertical Blank Status Enabled
15	Reserved: MBZ



Bit	Descriptions
14	Reserved
13	CRC Error Interrupt Status: This sticky status bit is set when a Pipe A CRC error is detected. It is cleared by a write of a one. For this bit to be meaningful, the pipe and pixel clock should be enabled and running. 0 = No CRC error has occurred 1 = CRC Error Detected
12	CRC Done Interrupt Status: This sticky status bit is set when Pipe A CRC calculation and compare are complete. It is cleared by a write of a one. For this bit to be meaningful, the pipe and pixel clock should be enabled and running. 0 = CRC Not Done 1 = CRC Done
11	GMBUS Interrupt Status: This status bit will be set on a GMBUS event. To use this bit in a polling manner, clear the bit by writing a one to it followed by the polling loop waiting for it to become set. 0 = GMBUS event has not occurred 1 = GMBUS event has occurred
10	Reserved
9	Vertical Sync Interrupt Status: This bit provides a sticky status that is set when a pipe A vertical sync occurs, cleared by a write of a 1. For interlaced timing modes, this occurs once per field, when in progressive, it occurs once per frame. For this bit to be meaningful, the pipe and pixel clock should be enabled and running. 0 = Vertical Sync has not occurred 1 = Vertical Sync has occurred
8	Display Line Compare Interrupt Status: Set when a pipe A compare match occurs, cleared by a write of a 1. 0 = Display Line Compare has not been satisfied 1 = Display Line Compare has been satisfied
7	[DevCL]: DPST Event Status: This bit is cleared when a write to this register occurs with this bit as a one. Writes with this bit as a zero has no effect on the value of the bit. Multiple DPST events (Histogram or Phase In) can cause this bit to be asserted, determination of which event occurred is done in the DPST registers. 0 = DPST Interrupt has not occurred on pipe A 1 = DPST Interrupt has occurred on pipe A
6	[DevCL]: Legacy BLC Event Status: This status bit indicates that a write to the PCI Backlight Control Register (LBPC) has occurred. Software must clear this bit in order to detect subsequent writes, for example while servicing the Event Interrupt. This bit is cleared when a write to this register occurs with this bit as a one. Writes with this bit as a zero has no effect on the value of the bit. 0 = No BLC write detected 1 = A BLC write was detected



Bit	Descriptions
5	Odd Field Interrupt Status: This status bit will be set on a Odd field VBLANK event. This bit should only be used when this pipe is in an interlaced display timing. For synchronization with register updates, the actual event will occur one line after the start of VBLANK. To use this bit in a polling manner, clear the bit by writing a one to it followed by the polling loop waiting for it to become set. Note: This bit will not be set when pipe is in "Interlaced with Field 0 Only using legacy vertical sync shift" mode. 0 = Odd Field Vertical Blank has not occurred 1 = Odd Field Vertical Blank has occurred
4	Even Field Interrupt Status: This status bit will be set on a even field VBLANK event. This bit should only be used when this pipe is in an interlaced display timing. For synchronization with register updates, the actual event will occur one line after the start of VBLANK. To use this bit in a polling manner, clear the bit by writing a one to it followed by the polling loop waiting for it to become set. Note: This bit will not be set when pipe is in "Interlaced with Field 0 Only using legacy vertical sync shift" mode. 0 = Even Field Vertical Blank has not occurred 1 = Even Field Vertical Blank has occurred
3	Reserved
2	Start of Vertical Blank Interrupt Status: This status bit will be set at the beginning of a VBLANK event. At this point, the double buffered display registers flip, taking their new values. To use this bit in a polling manner, clear the bit by writing a one to it followed by the polling loop waiting for it to become set. 0 = Start of Vertical Blank has not occurred 1 = Start of Vertical Blank has occurred
1	Vertical Blank Interrupt Status: This status bit will be set on a VBLANK event, when the frame start occurs. The display registers are updated at the start of vertical blank, but the new register data is not utilized by the display pipeline until the point in the vertical blank period when the frame start occurs, which is the event that triggers this bit. To use this bit in a polling manner, clear the bit by writing a one to it followed by the polling loop waiting for it to become set. 0 = Vertical Blank has not occurred 1 = Vertical Blank has occurred
0	Overlay Registers Updated Interrupt Status: This is not a pipe A event. It exists in this register for compatibility reasons only. The bit is set when an overlay register update completes, cleared by a write of a 1. 0 = Overlay Registers update has not occurred 1 = Overlay Registers update has occurred.



2.10.1.8 DSPARB—Display Arbitration Control

Memory Offset Address: 70030h
 Default: 00001D9Ch (FIFO Sizes A=28, B=31, C=37)
 Normal Access: Read/Write

Notes:

Each active display plane A, B, or C requires a FIFO to cover for memory latency. The FIFOs all come from a single RAM that is divided into areas for each display plane. The amount of the RAM used by each display plane is defined by this register. The two fields in the register split the display RAM into three portions, allocated between display planes A, B, and C. This register is double buffered and updated on the leading edge of Vertical Blank of the pipe that the planes are assigned to. This register should only be changed when a single pipe is enabled or if all of the Display A, B, C planes are disabled. It takes effect on the next VBLANK for whichever pipe is currently active. Each display plane needs a minimum FIFO size that is at least $\text{MaxLatencyForPlane} * \text{PixelRate} * \text{PixelSize} + 512$. All values should be rounded up to the next unit of 64B.

A special C3 mode can occur when a single display (of Display A and Display B) is active and the overlay and Display C are disabled. In that mode, when C3 is entered, the values in the BSTART and CSTART fields are ignored and the entire RAM is allocated to the single active display plane.

[DevBW] The control granularity of FIFO size is 64-bytes and the total size of the RAM is 384*16 bytes making TOTALSIZE equal to 96. The range of values for CSTART and BSTART is 0-95.

[DevCL] The control granularity of FIFO size is 64-bytes and the total size of the RAM is 512*16 bytes making TOTALSIZE equal to 128. The range of values for CSTART and BSTART is 0-127.

The display dot clock frequency or pixel rate must not exceed 90% of the core display clock. When a primary plane is enabled with 64bpp format and sprite is also enabled on the same pipe, the dot clock frequency or pixel rate must be less than 80% of the core display clock.

Bit	Descriptions
31:14	Reserved: Write as zero.
13:7	CSTART: This field selects the end of the ram used for display B and the start of the RAM for display C. If display B is unused, this field can be set to the same value as BSTART. If display C is unused, this field can be set to TOTALSIZE-1. It must be programmed to a number greater than or equal to the value in BSTART and less than the total size of the RAM (TOTALSIZE). The size of the display B FIFO will be $(\text{CSTART}-\text{BSTART}) * 64$. The size of the display C FIFO will be $(\text{TOTALSIZE}-\text{CSTART}-1) * 64$ bytes.
6:0	BSTART: This field selects the end of the ram used for display A and the start of the RAM for display B. If display A is unused, this field can be set to zero. The value should never exceed the size of the RAM (TOTALSIZE). The size of the display A FIFO will be $(\text{BSTART}) * 64$ bytes.



2.10.1.9 FW1—Display FIFO Watermark Control 1

Memory Offset Address: 70034h
 Default: (3F8F0F0Fh for [DevCL])
 (1F8F0F0Fh for [DevBW])
 Normal Access: Read/Write

These control values only apply to high-resolution (non-VGA) modes of operation. The hardware depends on these registers being set properly since it is possible to set the watermarks to states causing starvation of the sync FIFO.

Bit	Description
31:23	Display FIFO Self Refresh Watermark. This register defines the value of the watermark used by the Display streamer in case the CPU is in C2/C3/C4 and the memory has entered self refresh. Number in 64Bs of space in FIFO above which the Display Stream will generate requests to Memory (Value should be as recommended in the high priority bandwidth analysis spreadsheet). Programming Note [DevCL]: When calculating watermark values for 15/16bpp display formats, assume 32bpp for purposes of calculation using the high priority bandwidth analysis spreadsheet.
22	Reserved: MBZ
21:16	Cursor B FIFO Watermark. Number in 64Bs of space in the Cursor B FIFO above which the Cursor B Stream will generate requests to Memory (Value should be as recommended in the high priority bandwidth analysis spreadsheet). Always program to 8.
15	Reserved: MBZ
14:8	Display Plane B FIFO Watermark: Number in 64Bs of space in FIFO above which the Display B Stream will generate requests to Memory (Value should be as recommended in the high priority bandwidth analysis spreadsheet). Always program to 8.
7	Reserved: MBZ
6:0	Display Plane A FIFO Watermark: Number in 64Bs of space in FIFO above which the Display A Stream will generate requests to Memory (Value should be as recommended in the high priority bandwidth analysis spreadsheet). Always program to 8.



2.10.1.10 FW2—Display FIFO Watermark Control 2

Memory Offset Address: 70038h
 Default: 00 00 0F 0Fh
 Normal Access: Read/Write

These control values only apply to high-resolution (non-VGA) modes of operation. The hardware depends on these registers being set properly since it is possible to set the watermarks to states causing starvation of the sync FIFO.

Bit	Description
31:14	Reserved: MBZ
13:8	Cursor A FIFO Watermark. Number in 64Bs of space in the Cursor A FIFO above which the Cursor A Stream will generate requests to Memory (Value should be as recommended in the high priority bandwidth analysis spreadsheet). Always program to 8.
7	Reserved: MBZ
6:0	Display Plane C FIFO Watermark. Number in 64Bs of space in FIFO above which the Display C Stream will generate requests to Memory (Value should be as recommended in the high priority bandwidth analysis spreadsheet). Always program to 8.



2.10.1.11 FW3—Display FIFO Watermark Control 3

Memory Offset Address: 7003Ch
 Default: 00000000h
 Normal Access: Read/Write

These control values only apply to high-resolution (non-VGA) modes of operation. The hardware depends on these registers being set properly since it is possible to set the watermarks to states causing starvation of the sync FIFO.

Bit	Description
31	Enable HPLL off during Self Refresh. 0 = Disabled 1 = Enabled [DevCL] This bit may be enabled <u>only</u> if the BLC_PWM_CTL duty cycle register offset (0x61254) is programmed to 100% <u>and</u> non-legacy backlight is enabled. This restriction does not apply when I2C is used for back light modulation. [DevCL] When one or more display pipes are enabled, this bit should be disabled <u>before</u> accessing the 6XXXh MMIO register address space. Software must follow these steps: • disable this bit (if enabled <u>and</u> one display pipe is enabled) • wait for next vblank (switch from hrawclk back to cdclk will occur) • access the 6XXXh address space as needed • re-enable this bit Note that the wait on next vblank step requires an enabled display pipe.
30	Reserved: MBZ
29:24	Cursor FIFO Self Refresh Watermark. Number in 64Bs of space in the Cursor FIFO above which the Cursor Stream will generate requests to Memory during self -refresh. (Value should be as recommended in the high priority bandwidth analysis spreadsheet).
23:22	Reserved: MBZ
21:16	HPLL Self Refresh Cursor Watermark. Number in 64Bs of space in the Cursor FIFO above which the Cursor Stream will generate requests to Memory during HPLL self -refresh. (Value should be as recommended in the high priority bandwidth analysis spreadsheet).
15:9	Reserved: MBZ
8:0	HPLL Self Refresh Display Watermark. Number in 64Bs of space in the FIFO above which the Display Stream will generate requests to Memory during HPLL self -refresh. (Value should be as recommended in the high priority bandwidth analysis spreadsheet).



2.10.1.12 PipeAFrameHigh— Pipe A Frame Count High

Memory Offset Address: 70040h
 Default: 00000000h
 Normal Access: Read-Only (Requires that this pipe's PLL is running)

Bit	Descriptions
31:16	Reserved:
15:0	Pipe A Frame Count High. This field provides the most significant 16-bits of the free running frame counter for this display pipe. When combined with the low 8-bits of the frame counter, it allows the frame count to wrap only once per 16M frames. The counter is reset to zero when the device is reset or when the pipe transitions from off to on. It is incremented when the low frame counter rolls over to zero. This counter wraps, when the maximum count is reached, the next count value will be zero. When combined with the least significant bits of the frame counter forms a 24-bit value that indicates the number of frames since the pipe was enabled. This register should only be read if the display PLL for this display pipe is running. The hardware does not attempt to synchronize this value with the read of the least significant bits. Software must take the appropriate actions when it is desired to form a full frame count value by combining the two portions of the frame counter. The following example is a possible method: Read the Frame count high → High1 Read the Frame count low → Low 1, Pixel1 Read the Frame count high → High2 Read the Frame count low → Low 2, Pixel2 Read the Frame count high → High3 If (Both versions of the frame count high are equal (High1 = High2)) then: Frame# ← High1 Low1 Line# ← int(Pixel1/Htotal) Pixel# ← Pixel1 – (Line#*Htotal) Elseif (High2 = High3) then: Frame# ← High2 Low2 Line# ← int(Pixel2/Htotal) Pixel# ← Pixel2 – (Line#*Htotal) Else: Error, Unable to acquire frame number – Indicates that the above register read sequence takes more than 256 display frames. EndIf



2.10.1.13 PipeAFramePixel— Pipe A Frame Count Low and Pixel Count

Memory Offset Address: 70044h
 Default: 00000000h
 Normal Access: Read-Only (Requires that this pipe's PLL be running)

Bit	Descriptions
31:24	Pipe A Frame Count Low. This field provides the least significant 8-bits of the frame counter for this display pipe. It is reset when the device is reset or when the pipe transitions from off to on. It is incremented at the beginning of horizontal active of the first line of vertical active. This counter wraps, when the maximum count is reached, the next count value will be zero.
23:0	Pipe A Pixel Count. This field provides a pixel counter for the display pipe. It is reset at the beginning of horizontal active of the first line of vertical active and is incremented every pipe pixel clock. It will increment to a maximum of the number of clocks in a frame minus one ($H_{total} * V_{total} - 1$). To determine the line that the pixel count represents, the value is divided by the horizontal total for this pipe. The pixel count and the frame count low operate together, if a display frame has for example 64,000 clocks and we are on frame 5, this register value would read Frame=5 Pixel=63,999 on the last pixel of frame five, one display clock later it would read Frame=6 Pixel = 0. Pixel Count tracks pixels at the output. Even in the most basic cases, the actual fetch of the source data for that pixel may occur a significant amount of time before the pixel makes it to the output. In addition, there are cases where there is a difference between pixels in the source data and the output. These cases include: - Panel fit images - Integrated TV using overscan scaler - Modes that use line stall (i.e. External LVDS/TMDS scalers) - HDMI game modes - Display Plane 2X modes - Various methods of display rotation This field is only to be used for display modes with 16M or less clocks per frame. This is not to be used in VGA display operation.

2.10.2 Cursor A Plane Control Registers

The hardware cursor registers are memory mapped and accessible through 32 bit, 16 bit, or 8-bit accesses. They are all double-buffered, including the palette registers. Writes to cursor registers are performed to a holding register. The actual register update will occur based on the associated pipe's Vertical Blank signal only after a write cycle to the base address register (setting the trigger) has occurred. Writes to any register other than the trigger register will disable an active trigger if that occurs before the vertical blank.



2.10.2.1 CURACNTR—Cursor A Control Register

This register, and all other cursor registers will remain in their holding register (readable) after a write. The holding registers are transferred into the active registers on the asserting edge of Vertical Blank only after a write cycle to the base address register has completed.

Memory Offset Address: 70080h
 Default: 00000000h
 Normal Access: Read/Write

Bit	Description
31:30	Reserved: Write as zero.
29:28	Pipe Select: A state machine handles the synchronization of the switch to both vertical blank signals. So as far as the software is concerned, when both display pipes are being used, it can be switched at any time; the hardware will synchronize the switch. 00 = HW cursor is attached to Display Pipe A. This is the default after reset. 01 = HW cursor is attached to Display Pipe B. 10 = Reserved for pipe C 11 = Reserved for pipe D
27	Popup Cursor Enabled. This bit should be turned on when using Cursor A as a popup cursor. When in popup mode, hardware interprets the cursor base address as a <u>physical</u> address instead of a graphics address. 0 = Cursor A is hi-res 1 = Cursor A is popup
26	Cursor Gamma Enable: This bit only has an effect when using the cursor in a non-VGA mode. In VGA pop-up operation, the cursor data will always bypass the gamma (palette) unit. 0 = Cursor pixel data bypasses gamma correction or palette (default). 1 = Cursor pixel data is gamma to be corrected in the pipe.
25:16	Reserved: Write as zero
15	180° Rotation: This mode causes the cursor to be rotated 180°. In addition to setting this bit, software must also set the base address to the lower right corner of the unrotated image. Only 32 bits per pixel cursors can be rotated. This field must be zero when the cursor format is 2 bits per pixel. 0 = No rotation 1 = 180° Rotation of 32-bit per pixel cursors
14:6	Reserved
5	Cursor Mode Select: See following table.
4:3	Reserved
2:0	Cursor Mode Select: These three bits together with bit 5 select the mode for cursor as shown in the following table.

Table 2-6. Cursor Mode Select

Bit 5	Bits 2:0	Mode
0	000	Cursor is disabled. This is the default after reset. When the cursor register value changes from enabled to disabled, the cursor will stop fetching data at the following VBLANK event. The cursor enable can be overridden by the pipe cursor disable bit. The value of these bits do not change when disabled by the pipe cursor disable bit.
0	001	Reserved
0	010	128x128 32bpp XRGB (not available for VGA use)
0	011	256x256 32bpp XRGB (not available for VGA use)
0	100	64 x 64 2bpp 3-color and transparency mode
0	101	64 x 64 2bpp AND/XOR 2-plane mode
0	110	64 x 64 2bpp 4-color mode
0	111	64 x 64 32bpp AND/XOR (not available for VGA use) For each pixel: Least Significant Three Bytes Provides cursor RGB 888 color information Most Significant Byte: All Ones: Opaque show the cursor color All Zeros: Transparent (color must also equal zero) - Other: Invert the underlying display pixel data (ignore the color)
1	000	Reserved
1	001	Reserved
1	010	128x128 32bpp ARGB (not available for VGA use)
1	011	256x256 32bpp ARGB (not available for VGA use)
1	100	Reserved
1	101	Reserved
1	110	Reserved
1	111	64 x 64 32bpp ARGB (not available for VGA use)



2.10.2.2 CURABASE—Cursor A Base Address Register

Memory Offset Address: 70084h
 Default: 00000000h
 Normal Access: Read/Write

This register specifies the graphics memory address at which the cursor image data is located. Writes to this register acts like a trigger that enables atomic updates of the cursor registers. When updating the cursor registers, this register should be written last in the sequence. This register should be written even if the actual contents did not change to allow the holding registers to move to the active registers on the next VBLANK.

For legacy cursor modes, this register is sufficient to specify the address of the entire cursor. For ARGB modes, this register specifies the address of the first page of the cursor data.

Bit	Description
31:4	Cursor Base Address. This field specifies bits 31:4 of the <u>graphics</u> address of the base of the cursor. If the cursor is a popup, this field specifies bits 31:4 of the <u>physical</u> address of the base of the cursor, and bits 35:32 of the address are specified in the LSBs of this register. Popup cursor mode is selected within the CURACNTR register. The cursor surface address must be 4K byte aligned. The cursor must be in linear memory, it cannot be tiled. When performing 180° rotation, this offset must be the difference between the last pixel of the last line of the cursor data in its unrotated orientation and the cursor surface address. A write to this register also acts as a trigger event to force the update of active registers from the staging registers on the next display event. Each cursor register is double-buffered. The CPU writes to a set of holding registers. The active registers are updated from the holding registers following the leading edge of the vertical blank pulse. The update is postponed until the next vblank if a write cycle is active to any of the cursor registers at the time of the vblank. The update is also postponed if a write sequence is in progress. It is assumed that if the cursor mode is changed, the cursor image will also be changed. To prevent the cursor from appearing when it is only partially programmed, the active registers will not be updated until both the cursor control and base address registers have been programmed. If the cursor control register is written, the cursor base address must also be written before the change will be effective. However, the base address register may be changed (e.g., to change the shape of the cursor) without also writing to the control register. If both are to be written, the control register must be written first.
3:0	Popup Cursor Base Address MSBs. This field specifies bits 35:32 of the popup cursor physical address. If popup mode is not selected, this field is ignored.



2.10.2.3 CURAPOS—Cursor A Position Register

Memory Offset Address: 70088h
 Default: 00000000h
 Normal Access: Read/Write

This register specifies the screen position of the cursor. The origin of the cursor position is always the upper left corner of the active image for the display pipe that the cursor is assigned. This register can be loaded atomically (requires that the base address be written) and is double buffered.

Bit	Description
31	Cursor Y-Position Sign Bit: This bit provides the sign bit of a signed 13-bit value that specifies the horizontal position of cursor. (Default is 0.) For normal high resolution display modes, the cursor must have at least a single pixel positioned over the active screen. For use as a VGA Popup, the entire cursor must be positioned over the active area of the VGA image.
30:28	Reserved: Write as zero.
27:16	Cursor Y-Position Magnitude Bits 11:0: This register provides the magnitude bits of a signed 12-bit value that specifies the vertical position of cursor. The sign bit of this value is provided by bit 31 of this register. (Default is 0.) For use as a VGA Popup, the entire cursor must be positioned over the active area of the VGA image. Enabling the border in VGA (VGA Border Enable bit in the VGA Config register) includes the border in what is considered the “active area”. For HDMI modes where the vertical zoom is greater than 1x, the position is specified using the zoomed grid. When performing 180° rotation, this field specifies the vertical position of the lower right corner relative to the end of the active video area in the unrotated orientation.
15	Cursor X-Position Sign Bit: This bit provides the sign bit of a signed 13-bit value that specifies the horizontal position of cursor. (Default is 0.) For normal high resolution display modes, the cursor must have at least a single pixel positioned over the active screen. For use as a VGA Popup, the entire cursor must be positioned over the active area of the VGA image. Enabling the border in VGA (VGA Border Enable bit in the VGA Config register) includes the border in what is considered the “active area”.
14:12	Reserved: Write as zero.
11:0	Cursor X-Position Magnitude Bits 11:0: These 12 bits provide the signed 13-bit value that specifies the horizontal position of cursor. The sign bit is provided by bit 15 of this register. (Default is 0.) For HDMI modes where the horizontal zoom is greater than 1x, the position is specified using the zoomed grid. When performing 180° rotation, this field specifies the horizontal position of the lower right corner relative to the end of the active video area in the unrotated orientation.

2.10.2.4 CURARESV—Cursor A (Reserved)

Memory Offset Address: 7008Ch
 Default: 00000000h
 Normal Access: Read-Only



2.10.2.5 CURAPALET[0:3]—Cursor A Palette registers (4 Registers)

Memory Offset Address: 70090–7009Fh
CURAPALET0: 70090–70093h
CURAPALET1: 70094–70097h
CURAPALET2: 70098–7009Bh
CURAPALET3: 7009C–7009Fh
Default: 00000000h
Normal Access: Read/Write

These palette registers can be accessed through this MMIO interface register locations combined with an enable bit. This is the preferred method. The cursor palette provides color information when using one of the indexed modes. The two-bit index selects one of the four colors or two of the colors when in the AND/XOR cursor mode. The cursor palette provides color information when using one of the indexed modes. The two-bit index selects one of the four colors or two of the colors when in the AND/XOR cursor mode.

The table below describes the palette usage for different cursor modes and indexes.

Index	2 color	3color	4color
00	palette 0	palette 0	palette 0
01	palette 1	palette 1	palette 1
10	transparent	transparent	palette 2
11	invert destination	palette 3	palette 3

Bit	Descriptions
31:24	Reserved: Write as zero.
23:16	Red or Y Value: These registers specify the cursor palette. RGB data is full range unsigned numbers. YUV data will be unsigned for the Y and excess 128 notation for the UV values. The data can be pre-gamma corrected and bypass the gamma correction logic or passed through the gamma corrector.
15:8	Green or U Value:
7:0	Blue or V Value:



2.10.3 Cursor B Plane Control Registers

2.10.3.1 CURBCNTR—Cursor B Control Register

Memory Offset Address: 700C0h
Default: 00000000h
Normal Access: Read/Write

The hardware cursor registers are memory mapped and accessible through 32 bit, 16 bit, or 8-bit accesses. They are all, including the palette registers double buffered. Writes to cursor registers are done to a holding register. The actual register update will occur based on the assigned pipes VBLANK. It is recommended that the base register be accessed through a 32-bit write only. To update all cursor registers atomically, a sequence that ends with a base address register write should be used.

Bit	Descriptions
31:30	Reserved: Write as zero.
29:28	Pipe Select: A state machine handles the synchronization of the switch to both vertical blank signals. So as far as the software is concerned, when both display pipes are being used, it can be switched at any time; the hardware will synchronize the switch. 00 = HW cursor is attached to Display Pipe A. This is the default after reset. 01 = HW cursor is attached to Display Pipe B. 10 = Reserved for to Display Pipe C. 11 = Reserved for to Display Pipe D.
27	Reserved: Write as zero.
26	Cursor Gamma Enable: 0 = Cursor pixel data bypasses gamma correction (default). 1 = Cursor pixel data is gamma to be corrected.
25:16	Reserved
15	180° Rotation: This mode causes the cursor to be rotated 180°. In addition to setting this bit, software must also set the base address to the lower right corner of the unrotated image. Only 32 bits per pixel cursors can be rotated. This field must be zero when the cursor format is 2 bits per pixel. 0 = No rotation 1 = 180° Rotation of 32-bit per pixel cursors
14:6	Reserved: Write as zero
5	Cursor Mode Select: See following table.
4:3	Reserved
2:0	Cursor Mode Select: These three bits together with bit 5 select the mode for cursor as shown in the following table.



Table 2-7. Cursor Mode Select

Bit 5	Bits 2:0	Mode
0	000	Cursor is disabled: This is the default after reset. When the cursor register value changes from enabled to disabled, the cursor will stop fetching data at the following VBLANK event. The cursor enable can be overridden by the pipe cursor disable bit. The value of these bits does not change when disabled by the pipe cursor disable bit.
0	001	Reserved
0	010	128x128 32bpp XRGB (not available for VGA use)
0	011	256x256 32bpp XRGB (not available for VGA use)
0	100	64 x 64 2bpp 3-color and transparency mode
0	101	64 x 64 2bpp AND/XOR 2-plane mode
0	110	64 x 64 2bpp 4-color mode
0	111	64 x 64 32bpp AND/XOR (not available for VGA use) For each cursor pixel: Least Significant Three Bytes Provides cursor RGB 888 color information Most Significant Byte: All Ones: Opaque show the cursor color All Zeros: Transparent (color must also equal zero) - Other: Invert the underlying display pixel data (ignore the color)
1	000	Reserved
1	001	Reserved
1	010	128x128 32bpp ARGB (not available for VGA use)
1	011	256x256 32bpp ARGB (not available for VGA use)
1	100	Reserved
1	101	Reserved
1	110	Reserved
1	111	64 x 64 32bpp ARGB



2.10.3.2 CURBBASE—Cursor B Base Address Register

Memory Offset Address: 700C4h
Default: 00000000h
Normal Access: Read/Write

This register specifies the memory address at which the cursor data is located. Writes to this register should be done with 32-bit accesses and acts as a trigger to atomically update the cursor register set. For legacy cursor modes, this register is sufficient to specify the address of the entire cursor. The address is the graphics address. For ARGB modes, this register specifies the address of the first page of the cursor data.

Bit	Description
31:0	Cursor Base Address: This register specifies the graphics address of the entire cursor. It also acts as a trigger event to force the update of active registers on the next display event. The cursor surface address must be 4K byte aligned. The cursor must be in linear memory, it cannot be tiled. When performing 180° rotation, this offset must be the difference between the last pixel of the last line of the cursor data in its unrotated orientation and the cursor surface address. A write to this register also acts as a trigger event to force the update of active registers from the staging registers on the next display event. Each cursor register is double-buffered. The CPU writes to a set of holding registers. The active registers are updated from the holding registers following the leading edge of the vertical blank pulse. The update is postponed until the next vblank if a write cycle is active to any of the cursor registers at the time of the vblank. The update is also postponed if a write sequence is in progress. It is assumed that if the cursor mode is changed, the cursor image will also be changed. To prevent the cursor from appearing when it is only partially programmed, the active registers will not be updated until both the cursor control and base address registers have been programmed. If the cursor control register is written, the cursor base address must also be written before the change will be effective. However, the base address register may be changed (e.g., to change the shape of the cursor) without also writing to the control register. If both are to be written, the control register must be written first.



2.10.3.3 CURBPOS—Cursor B Position Register

Memory Offset Address: 700C8h
 Default: 00000000h
 Normal Access: Read/Write

This register specifies the screen position of the cursor. The origin of the cursor position is always the upper left corner of the active image for the display pipe that the cursor is assigned. This register can be loaded atomically and is double buffered. The load register is transferred into the active register on the leading edge of Vertical Blank of the pipe cursor is currently assigned after the trigger has been set.

Bit	Descriptions
31	Cursor Y-Position Sign Bit: This bit provides the sign bit of a signed 13-bit value that specifies the horizontal position of cursor. (default is 0).). For normal high resolution display modes, the cursor must have at least a single pixel positioned over the active screen.
30:28	Reserved: Write as zero.
27:16	Cursor Y-Position Magnitude Bits 11:0: This register provides the magnitude bits of a signed 13-bit value that specifies the vertical position of cursor. The sign bit of this value is provided by bit 31 of this register. (default is 0) When performing 180° rotation, this field specifies the vertical position of the lower right corner relative to the end of the active video area in the unrotated orientation.
15	Cursor X-Position Sign Bit: This bit provides the sign bit of a signed 13-bit value that specifies the horizontal position of cursor. (default is 0).). For normal high resolution display modes, the cursor must have at least a single pixel positioned over the active screen. For HDMI modes where the vertical zoom is greater than 1x, the position is specified using the zoomed grid.
14:12	Reserved: Write as zero.
11:0	Cursor X-Position Magnitude Bits 11:0: These 12 bits provide the signed 13-bit value that specifies the horizontal position of cursor. The sign bit is provided by bit 15 of this register. (default is 0) For HDMI modes where the horizontal zoom is greater than 1x, the position is specified using the zoomed grid. When performing 180° rotation, this field specifies the vertical position of the lower right corner relative to the end of the active video area in the unrotated orientation.

2.10.3.4 CURBRESV—Cursor B Reserved

Memory Offset Address: 700CCh
 Default: 00000000h
 Normal Access: Read



2.10.3.5 CURB PALET[0:3]—Cursor B Palette Registers (4 Registers)

Memory Offset Address: 700D0–700DCh
CURBPALET0: 700D0–700D3h
CURBPALET1: 700D4–700D7h
CURBPALET2: 700D8–700DBh
CURBPALET3: 700DC–700DFh
Default: 00000000h
Normal Access: Read/Write

These palette registers can be accessed through this MMIO interface or through a legacy mode using the VGA palette register locations combined with an enable bit. This is the preferred method. The cursor palette provides color information when using one of the indexed modes. In the two-bit AND/XOR cursor modes, the two-bit index selects one of the four colors or two of the colors when in the mode. RGB data is full range unsigned numbers. YUV data will be unsigned for the Y and excess 128 notation for the UV values. The data can be pre-gamma corrected and bypass the gamma correction logic or passed through the gamma corrector.

Bit	Descriptions
31:30	Reserved: Write as zero.
23:16	Red or Y:
15:8	Green or U Value:
7:0	Blue or V Value:



2.10.4 Display Pipeline B

2.10.4.1 PIPEB_DSL—Display Scan Line

Memory Address Offset: 71000h
Default: 00h
Normal Access: Read-Only

This register enables the read back of the display pipe vertical “line counter”. The display line value is from the display pipe B timing generator and is reset to zero at the beginning of a scan. The value increments at the leading edge of HSYNC and can be safely read any time. For normal operation, scan line zero is the first active line of the display. When in VGA centering mode, the scan line 0 is the 1st active scan line of the pseudo border not the centered active VGA image display area. In interlaced display timings, the scan line counter provides the current line in the field. One field will have a total number of lines that is one greater than the other field.

Programming Note: In order to cause the scan line logic to report the correct Line Counter value, the corresponding Display Pipeline timing registers must be programmed to valid, non-zero (e.g., 640x480 @ 60Hz) values before enabling the Pipe or programming VGA timing and enabling native VGA.

Bit	Description
31:13	Reserved: Read-only.
12: 0	Pipe B Display Line Counter: This register enables the read back of the display vertical “line counter”. The display line values are from the pipe B timing generator. They change at the leading edge of HSYNC, and can be safely read at any time.



2.10.4.2 PIPEB_SLC—Pipe B Display Scan Line Range Compare Register

Memory Address Offset: 71004h
Default: 00h
Normal Access: Read-Only

The Start and End Line Count Compare registers are compared with the display line values from the timing generator. They change at the leading edge of HSYNC. They can safely be accessed at any time. The End compare register operator is a less than or equal, while the Start compare register operator is a greater than or equal. The results of these 2 comparisons are communicated to the command stream controller for generating interrupts, status, and command stream flow control (“wait for within range” and “wait for not within range”).

For range check, the value programmed should be the desired value - 1. So for line 0, the value programmed is VTOTAL and for line 1, the value programmed is 0.

This register can be written via the command stream processor using the MI_LOAD_SCAN_LINES_INCL or MI_LOAD_SCAN_LINES_EXCL commands.

Bit	Description
31	Inclusive / Exclusive: 1 = Inclusive Within Range, 0 = Exclusive Out of Range
30:29	Reserved: Read-only.
28:16	End Scan Line Number: This field specifies the ending scan line number of the Scan Line Window. Format = U16 in scan lines, where scan line 0 is the first line of the display frame. Range = [0, Display Buffer height in lines-1].
15:13	Reserved: Read-only.
12:0	Start Scan Line Number: This field specifies the starting scan line number of the Scan Line Window. Format = U16 in scan lines, where scan line 0 is the first line of the display frame. Range = [0, Display Buffer height in lines-1].



2.10.4.3 PIPEBCONF—Pipe B Configuration Register

Memory Offset Address: 71008h
 Default: 00000000h
 Normal Access: Read/Write double buffered

Bit	Descriptions
31	Pipe B Enable: Setting this bit to the value of one, turns on pipe B. This must be done before any planes are enabled on this pipe. Changing it to a zero should only be done when all planes that are assigned to this pipe have been disabled. Turning the pipe enable bit off disables the timing generator in this pipe. Plane disable occurs after the next VBLANK event after the plane is disabled. Synchronization pulses to the display are not maintained if the timing generator is disabled. Power consumption will be at its lowest state when disabled. A separate bit controls the DPLL enable for this pipe. Pipe timing registers should contain valid values before this bit is enabled. Disabling the Pipe and changing the timing registers and re-enabling the pipe before the next VBLANK will cause the mode change to occur at the end of the current frame. This requires no wait on the software's part. On the other hand, if this is the disabling of the pipe, that does require a software wait for VBLANK to occur. Synchronization pulses to the display are not maintained if the timing generator is disabled. Power consumption is at its lowest state. 1 = Enable 0 = Disable
30	Pipe State: This bit indicates the actual state of the pipe. Since there can be some delay between disabling the pipe and the pipe actually shutting off, this bit indicates the true current state of the pipe. 0 = Disabled 1 = Enabled
29	Reserved: Write as zero.
28:27	Frame Start Delay: Used to delay the frame start signal that is sent to the display planes. Normal operation uses the default 00 value and test modes can use the delayed frame start to shorten the test time. This would be set to 00 for normal operation. 00 = Frame Start occurs on the first HBLANK after the start of VBLANK 01 = Frame Start occurs on the second HBLANK after the start of VBLANK 10 = Frame Start occurs on the third HBLANK after the start of VBLANK 11 = Frame Start occurs on the forth HBLANK after the start of VBLANK
26	Reserved: Write as zero.
25	FORCE_BORDER: (TEST MODE) 0 = Normal Operation 1 = Color information is ignored and border color is substituted during active region
24	Pipe B Gamma Unit Mode. This bit selects which mode the pipe gamma correction logic works in. In the palette mode, it behaves as a 3X256x8 RAM lookup. VGA and indexed mode operation should use the palette in 8-bit mode. In the 10-bit gamma mode, it will act as a piecewise linear interpolation. Other gamma units such as in the overlay and sprite are unaffected by this bit. 0 = 8-bit Palette Mode 1 = 10-bit Gamma Mode



Bit	Descriptions
23:21	Interlaced Mode These bits are used for software control of interlaced behavior. They are updated immediately if the pipe is off, or in the vertical blank after programming if pipe is enabled. 0xx = Progressive 100 = Interlaced embedded panel using programmable vertical sync shift 101 = Interlaced using vertical sync shift. Backup option to setting 110. 110 = Interlaced with VSYNC/HSYNC Field Indication using legacy vertical sync shift. Used for SDVO. 111 = Interlaced with Field 0 Only using legacy vertical sync shift. Not used. Note: VGA display modes, sDVO line stall, and Panel fitting do not work while in interlaced modes Setting the Interlaced embedded panel mode causes hardware to automatically modify the output to match the specifications of panels that support interlaced mode.
20	Reserved: Write as zero
19	Display / Overlay Planes Off. This bit when set will cause all enabled Display and overlay planes that are assigned to this pipe to be disabled by overriding the current setting of the plane enable bit, at the next VBLANK. Timing signals continue as they were but the screen becomes blank. Setting the bit back to a zero will then allow the display and overlay planes to resume on the following VBLANK. 0 = Normal Operation 1 = Planes assigned to this pipe are disabled.
18	Cursor Planes Off. This bit when set will cause all enabled cursor planes that are assigned to this pipe to be disabled by overriding the current setting of the plane enable bit, at the next VBLANK. Timing signals continue as they were but the screen becomes blank. Setting the bit back to a zero will then allow the cursor planes to resume on the following VBLANK. 0 = Normal Operation 1 = Planes assigned to this pipe are disabled.
17:16	Refresh Rate CxSR Mode Association These bits select how refresh rates are tied to CxSR on pipe B. When they are set to anything other than 00, bits 23:21 of this register must be programmed to 0xx. Switching between 01 and 10 settings directly is not allowed. Software must program this field to 00 before switching. Software is responsible for enabling this mode only for integrated display panels that support corresponding mode. 00 – Default – no dynamic refresh rate change enabled. Software control only. 01 – Progressive-to-progressive refresh rate change enabled and tied to CxSR. Pixel clock values set in FPB0/FPB1 settings in the DPLL control register and FPB0/FPB1 divider registers. 10 – Progressive-to-interlaced refresh rate change enabled and tied to CxSR. Pixel clock value does not change in this case. Scaling must be disabled in this mode. Uses programmable VS shift 11 – Reserved
15:10	Reserved: Write as zero
9:1	Reserved : MBZ
0	Reserved: Write as zero



2.10.4.4 PIPEBGCMAXRED—Pipe B Gamma Correction Max Red

Memory Offset Address: 71010h
Default: 00010000h
Normal Access: Read/Write

Bit	Descriptions
31:17	Reserved
16:0	Max Red Gamma Correction Point. 129th reference point for red channel of the pipe piecewise linear gamma correction. The value should always be programmed to be less than or equal to 1024.0. Format: 11.6 Default: 0x10000

2.10.4.5 PIPEBGCMAXGREEN—Pipe B Gamma Correction Max Green

Memory Offset Address: 71014h
Default: 00010000h
Normal Access: Read/Write

Bit	Descriptions
31:17	Reserved
16:0	Max Green Gamma Correction Point. 129th reference point for green channel of the pipe piecewise linear gamma correction. The value should always be programmed to be less than or equal to 1024.0. Format: 11.6 Default: 0x10000

2.10.4.6 PIPEBGCMAXBLUE—Pipe B Gamma Correction Max Blue

Memory Offset Address: 71018h
Default: 00010000h
Normal Access: Read/Write

Bit	Descriptions
31:17	Reserved
16:0	Max Blue Gamma Correction Point. 129th reference point for blue channel of the pipe piecewise linear gamma correction. The value should always be programmed to be less than or equal to 1024.0. Format: 11.6 Default: 0x10000



2.10.4.7 PIPEBSTAT—Pipe B Status

Memory Offset Address: 71024h
 Default: 00000000h
 Normal Access: Read/Write

Programming:

Prior to clearing a Display Pipe-sourced interrupt (e.g., Display Pipe A VBLANK) in the IIR, the corresponding interrupt (source) status in the PIPEASTAT or PIPEBSTAT register (e.g., Pipe A VBLANK Interrupt Status bit of PIPEASTAT) must first be cleared. Note that clearing these status bits requires writing a '1' to the appropriate bit position.

Bit	Descriptions
31	Pipe B Underflow Status: This bit is set when an underflow occurs at the display pipe B. It is cleared by writing a one to this bit. This event will occur naturally during mode changes, to be effective, it should be cleared after a mode change. This bit is only valid after Pipe B has been completely configured. 1 = FIFO B Underflow occurred 0 = FIFO B Underflow did not occur
30	Reserved: Write as zero.
29	CRC Error Enable: This will enable the consideration of the CRC error status bit in the first line interrupt/status logic. 0 = CRC Error Detect Disabled 1 = CRC Error Detect Enabled
28	CRC Done Enable: This will enable the consideration of the CRC done status bit in the first line interrupt/status logic. 0 = CRC Done Detect Disabled 1 = CRC Done Detect Enabled
27:26	Reserved: Write as zero.
25	Vertical Sync Interrupt Enable: 0 = Vertical Sync Interrupt/Status Disabled 1 = Vertical Sync Interrupt/Status Enabled
24	Display Line Compare Enable: 0 = Pipe B Display Line Compare Status Report Disabled 1 = Pipe B Display Line Compare Status report Enabled
23	[DevCL]: BLM Event Enable. This interrupt is generated by the image brightness segment comparators. Which segment cause an interrupt are controlled by the BLM Histogram control register. 0 = No BLM event enabled 1 = BLM event enabled



Bit	Descriptions
22	[DevCL]: Legacy BLC Event Enable. This will enable writes to the PCI Backlight Control Register to cause the display B event status to be set and an Interrupt if Display B Event interrupt is enabled. 0 = No BLC Event enabled 1 = BLC Event enabled
21	Odd Field Interrupt Event Enable. This bit should only be used when this pipe is in an interlaced display timing. 0 = Odd Field Event disable 1 = Odd Field Event enable
20	Even Field Interrupt Event Enable. This bit should only be used when this pipe is in an interlaced display timing. 0 = Even field Event disable 1 = Even field Event enable
19	Reserved: Read-only as zero
18	Start of Vertical Blank Interrupt Enable: This will enable the consideration of the start of vertical blank interrupt status bit in the first line interrupt/status logic. 0 = Start of Vertical Blank Interrupt/Status Disabled 1 = Start of Vertical Blank Interrupt/Status Enabled
17	Vertical Blank Interrupt Enable: This will enable the consideration of the vertical blank interrupt status bit in the first line interrupt/status logic. 0 = Vertical Blank Interrupt/Status Disabled 1 = Vertical Blank Interrupt/Status Enabled
16	Reserved: Write as zero.
15:14	Reserved: Read-only.
13	CRC Error Status: This bit is set when a Pipe B CRC error is detected. It is cleared by a write of a one. 0 = No CRC Error 1 = CRC Error detected
12	CRC Done Interrupt Status: This bit is set when Pipe B CRC calculation and compare are complete. It is cleared by a write of a one. 0 = CRC Not Done 1 = CRC Done
11:10	Reserved: Read-only, write as zero
9	Pipe B Vertical Sync Status: 0 = Vertical Sync not asserted 1 = Vertical Sync asserted
8	Pipe B Display Line Compare Status: This bit is cleared when a write to this register occurs with this bit as a one. Writes with this bit as a zero has no effect on the value of the bit. 0 = Display Line Compare Status not asserted 1 = Display Line Compare Status asserted



Bit	Descriptions
7	[DevCL]: BLM Image Brightness Status. This bit is cleared when a write to this register occurs with this bit as a one. Writes with this bit as a zero has no effect on the value of the bit. 0 = DPST Interrupt has not occurred on pipe B 1 = DPST Interrupt has occurred on pipe B
6	[DevCL]: Legacy BLC Event Status. This status bit indicates that a write to the PCI Backlight Control Register (LBPC) has occurred. Software must clear this bit in order to detect subsequent writes, for example while servicing the Event Interrupt. This bit is cleared when a write to this register occurs with this bit as a one. Writes with this bit as a zero has no effect on the value of the bit. 0 = No BLC write detected 1 = A BLC write was detected
5	Odd Field Interrupt Status. This status bit will be set on a Odd field VBLANK event. This bit should only be used when this pipe is in an interlaced display timing. For synchronization with register updates, the actual event will occur one line after the start of VBLANK. To use this bit in a polling manner, clear the bit by writing a one to it followed by the polling loop waiting for it to become set. Note: This bit will not be set when pipe is in "Interlaced with Field 0 Only using legacy vertical sync shift" mode. 0 = Odd Field Vertical Blank has not occurred 1 = Odd Field Vertical Blank has occurred
4	Even Field Interrupt Status. This status bit will be set on a even filed VBLANK event. This bit should only be used when this pipe is in an interlaced display timing. For synchronization with register updates, the actual event will occur one line after the start of VBLANK. To use this bit in a polling manner, clear the bit by writing a one to it followed by the polling loop waiting for it to become set. Note: This bit will not be set when pipe is in "Interlaced with Field 0 Only using legacy vertical sync shift" mode. 0 = Even Field Vertical Blank has not occurred 1 = Even Field Vertical Blank has occurred
3	Reserved: Read-only, write as zero
2	Start of Vertical Blank Interrupt Status: This status bit will be set at the beginning of a VBLANK event. At this point, the double buffered display registers flip, taking their new values. To use this bit in a polling manner, clear the bit by writing a one to it followed by the polling loop waiting for it to become set. 0 = Start of Vertical Blank has not occurred 1 = Start of Vertical Blank has occurred
1	Vertical Blank Interrupt Status: This status bit will be set on a VBLANK event, when the frame start occurs. The display registers are updated at the start of vertical blank, but the new register data is not utilized by the display pipeline until the point in the vertical blank period when the frame start occurs, which is the event that triggers this bit. To use this bit in a polling manner, clear the bit by writing a one to it followed by the polling loop waiting for it to become set. 0 = Vertical Blank has not occurred 1 = Vertical Blank has occurred
0	Reserved: Read-only, write as zero



2.10.4.8 PipeBFrameHigh— Pipe B Frame Count High

Memory Offset Address: 71040h
Default: 00000000h
Normal Access: Read-Only

Bit	Descriptions
31:16	Reserved
15:0	Pipe B Frame Count High See PipeAFrameHigh description.

2.10.4.9 PipeBFramePixel — Pipe B Frame Count Low and Pixel Count

Memory Offset Address: 71044h
Default: 00000000h
Normal Access: Read-Only

Bit	Descriptions
31:24	Frame Count Low. See PipeAFramePixel description.
23:0	Pixel Count. See PipeAFramePixel description.



2.10.5 Display A (Primary) Plane Control

2.10.5.1 DSPACNTR—Display A Plane Control Register

Memory Offset Address: 70180h
 Default: 00000000h
 Normal Access: Read/Write Double buffered

Note: The active set of registers will be updated on the VBlank (of the currently selected pipe) after the “trigger” register (the Start Address register or the Control register when plane enable bit transitioning from a zero to a one) is written – thus providing an atomic update of all display controls. If the currently selected pipe is disabled, the update is immediate.

Bit	Descriptions
31	Display Plane A (Primary A) Enable: When this bit is set, the primary plane will generate pixels for display. When set to zero, display plane A memory fetches cease and display is blanked (from this plane) at the next VBLANK event from the pipe that display A is assigned. The display pipe must be enabled to enable this plane. There is an override for the enable of this plane in the Pipe Configuration register. 1 = Enable 0 = Disable
30	Display A Gamma Enable: This bit should only be changed after the plane has been disabled. It controls the bypassing of the display pipe gamma unit for this display plane's pixel data only. For 8-bit indexed display data, this bit should be set to a one. 0 = Display A pixel data bypasses the display pipe gamma correction logic (default). 1 = Display A pixel data is gamma corrected in the display pipe gamma correction logic.
29:26	Display A Source Pixel Format: These bits should only be changed after the plane has been disabled. Pixel formats with an alpha channel (8:8:8:8) should not use source keying. Pixel format of 8-bit indexed uses the palette. Before entering the blender, each source format is converted to 10 bits per pixel (details are described in the intermediate precision for the blender section of the Display Functions chapter). 000x = Reserved. 0010 = 8-bpp Indexed. 0011 = Reserved. 0100 = Reserved. 0101 = 16-bit BGRX (5:6:5:0) pixel format (XGA compatible). 0110 = 32-bit BGRX (8:8:8:8) pixel format. Ignore alpha. 0111 = Reserved. 1000 = 32-bit RGBX (10:10:10:2) pixel format. Ignore alpha. 1001 = Reserved. 1010 = BGRX 10:10:10:2 1011 = Reserved. 1100 = 64-bit RGBX (16:16:16:16) 16-bit floating point pixel format. Ignore alpha. 1101 = Reserved. 1110 = 32-bit RGBX (8:8:8:8) pixel format. Ignore alpha. 1111 = Reserved.



Bit	Descriptions
25:24	Display A Pipe Select: This is read-only and selects the display pipe that this plane is assigned to. It is hardwired to pipe A. 00 = Select Pipe A (default: cannot be changed) 01 = Select Pipe B 10 = Reserved for pipe C 11 = Reserved for pipe D
23	Key Window Enable. This bit applies only to devices with a display plane C. This bit is set to one when the color key is used as a destination key for display C. Display plane C must be enabled on the same pipe and its Z-order should be programmed to be behind display A for this to be set to a one. 0 = Source Key applies to entire display plane A 1 = Source Key applies to only pixels within the intersection between Display A and Display C
22	Key Enable. This bit enables source keying for display A. Source keying allows a plane that is behind (below) this plane to show through where the display A key matches the display A data. This function is overloaded to provide display C destination keying when combined with the key window enable bit. Setting this bit is not allowed when the display pixel format includes an alpha channel. 0 = Source key is disabled 1 = Source key is enabled
21:20	Pixel Multiply: This cause the display plane to duplicate lines and pixels sent to the assigned pipe. In the pixel multiply mode, the horizontal pixels are doubled and lines are sent twice. Asynchronous flips are not used in this mode. Programming Notes: Asynchronous flips are not permitted when pixel multiply is enabled. 00 = No duplication 01 = Line/pixel Doubling 10 = Reserved 11 = Pixel Doubling only
19	Reserved: Software must preserve the contents of this bit.
18	Reserved: Write as zero
17:16	Reserved: Software must preserve the contents of this bit.
15	180° Display Rotation: This mode causes the display plane to be rotated 180°. In addition to setting this bit, software must also set the base address to the lower right corner of the unrotated image. [DevCL] Do not enable 180° rotation together with Frame Buffer Compression 0 = No rotation 1 = 180° rotation
14:11	Reserved
10	Tiled Surface. This bit indicates that the display A surface data is in tiled memory. The tile pitch is specified in bytes in the DSPASTRIDE register. Only X tiling is supported for display surfaces. When this bit is set, it affects the hardware interpretation of the DSPATILEOFF, DSPALINOFF, and DSPASURF registers. 0 = Display A surface uses linear memory 1 = Display A surface uses X-tiled memory
9:0	Reserved: Write as zero



2.10.5.2 DSPALINOFF—Display A Linear Offset Register

Memory Offset Address: 70184h
Default: 00000000h
Normal Access: Read/Write Double buffered

This register specifies the panning for the display surface. The surface base address is specified in the DSPASURF register, and this register is used to describe an offset from that base address. Bit 10 of DSPACNTR specifies whether the display A surface is in linear or tiled memory. When the surface is in linear memory, this field contains the byte offset of the plane data in graphics memory. When the surface is tiled, the contents of this register are ignored.

This register can be written directly through software or by load register immediate command packets in the command stream.

This register is double buffered by VSYNC only. A change to this register will take effect on the next vsync following the write.

Bit	Descriptions
31:0	Display A Offset: This register provides the panning offset into the display A plane. This value is added to the surface address to get the graphics address of the first pixel to be displayed. This offset must be at least pixel aligned. This offset is the difference between the address of the upper left pixel to be displayed and the display surface address. When performing 180° rotation, this offset must be the difference between the last pixel of the last line of the display data in its unrotated orientation and the display surface address.



2.10.5.3 DSPASTRIDE—Display A Stride Register

Memory Offset Address: 70188h
 Default: 00000000h
 Normal Access: Read/Write Double Buffered

Bit	Descriptions
31:0	Display A Stride: This is the stride for display A in bytes. When using linear memory, this must be 64 byte aligned. When using tiled memory, this must be 512 byte aligned. This value is used to determine the line to line increment for the display. This register is updated either through a command packet passed through the command stream or writes to this register. When it is desired to update both this and the start register, the stride register must be written first because the write to the start register is the trigger that causes the update of both registers on the next VBLANK event. When using tiled memory, the actual memory buffer stride is limited to a maximum of 16K bytes. The display stride must be power of 2 when doing Asynch Flips. The display stride must be 8KB or greater when doing Asynch Flips together with 180 rotation. The value in this register is updated through the command streamer during a synchronous flip.

2.10.5.4 DSPARESV—Display A Reserved

Memory Offset Address: 7018Ch–7018Fh
 Default: 00000000h
 Normal Access: Read-Only

2.10.5.5 DSPARESV—Display A Reserved

Memory Offset Address: 70190h
 Default: 00000000h
 Normal Access: Read/Write Double Buffered



2.10.5.6 DSPAKEYVAL—Sprite Color Key Value Register

Memory Offset Address: 70194h
Default: 00000000h
Normal Access: Read/Write Double Buffered

This register specifies the key color to be used with the mask bits to determine if the display source data matches the key. This register will only have an effect when the display color key is enabled. The overlay destination key value is used for overlay keying when Display A is being used as a primary display with overlay destination keying enabled. This key can be used as a Display C destination key onto Display A.

Bit	Descriptions
31:24	Reserved: Write as zero
23:16	Red Key Value: Specifies the color key value for the sprite red/Cr channel.
15:8	Green Key Value: Specifies the color key value for the sprite green/Y channel.
7:0	Blue Key Value: Specifies the color key value for the sprite blue/Cb channel.

2.10.5.7 DSPAKEYMSK—Sprite Color Key Mask Register

Memory Offset Address: 70198h
Default: 00000000h
Normal Access: Read/Write Double Buffered

This register specifies the key mask to be used with the color value bits to determine if the display source data matches the key when enabled. A zero bit in the mask indicates that the corresponding bit match failure should be ignored when determining if the pixel matches.

Bit	Descriptions
31:24	Reserved: Write as zero
23:16	Red mask Value: Specifies the color key mask for the sprite red/Cr channel.
15:8	Green mask Value: Specifies the color key mask for the sprite green/Y channel.
7:0	Blue mask Value: Specifies the color key mask for the sprite blue/Cb channel.



2.10.5.8 DSPASURF—Display A Surface Base Address Register

Memory Offset Address: 7019Ch
 Default: 00000000h
 Normal Access: Read/Write Double buffered

Writing to this register triggers the display plane flip. When it is desired to change multiple display A registers, this register should be written last as a write to this register will cause all new register values to take effect.

Bit	Descriptions
31:29	Reserved
28:12	Display A Surface Base Address. This address specifies the surface base address. When the surface is tiled, panning is specified using (x, y) offsets in the DSPATILEOFF register. When the surface is in linear memory, panning is specified using a linear offset in the DSPALINOFF register. This address must be 4K aligned. When performing asynchronous flips and the display surface is in tiled memory, this address must be 256K aligned. This register can be written directly through software or by command packets in the command stream. It represents an offset from the graphics memory aperture base and is mapped to physical pages through the global GTT. This address must be 128K aligned for linear memory.
11:0	Reserved



2.10.5.9 DSPATILEOFF—Display A Tiled Offset Register

Memory Offset Address: 701A4h
Default: 00000000h
Normal Access: Read/Write Double buffered

This register specifies the panning for the display surface. The surface base address is specified in the DSPASURF register, and this register is used to describe an offset from that base address. Bit 10 of DSPACNTR specifies whether the display A surface is in linear or tiled memory. When the surface is in linear memory, the offset is specified in the DSPALINOFF register and the contents of this register are ignored. When the surface is tiled, the start position is specified in this register as an (x, y) offset from the beginning of the surface.

This register can be written directly through software or by load register immediate command packets in the command stream.

This register is double buffered by VBLANK only. A change to this register will take effect on the next vblank following the write.

Bit	Descriptions
31:28	Reserved: Write as zero
27:16	Plane Start Y-Position: These 12 bits specify the vertical position in lines of the beginning of the active display plane relative to the display surface. When performing 180° rotation, this field specifies the vertical position of the lower right corner relative to the start of the active display plane in the unrotated orientation.
15:12	Reserved: Write as zero
11:0	Plane Start X-Position: These 12 bits specify the horizontal offset in pixels of the beginning of the active display plane relative to the display surface. When performing 180° rotation, this field specifies the horizontal position of the lower right corner relative to the start of the active display plane in the unrotated orientation. When display stride is 16KB and doing Asynch Flips, do not program the offset to give pans of 7680 to 8191 bytes.

2.10.5.10 DSPAFLPQSTAT—Flip Queue Status Register

Memory Offset Address: 70200h
Default: 00000800h
Normal Access: Read/Write with Read-Only fields

Bit	Descriptions
31:16	Reserved: Write as zero (RO)
15:8	Queue Free Entry Count (RO). This value indicates the number of free entries in the queue at the time that the register was read. The total number of entries in the queue is the sum of the occupied entry count and the free entry count.
7:0	Queue Occupied Entry Count (RO). This value indicates the number of occupied entries in the queue at the time that the register was read. The total number of entries in the queue is the sum of the occupied entry count and the free entry count.



2.10.6 Display B (Second Primary or Sprite) Control

All Display B/Sprite registers are double buffered. The active set of registers will be updated on the VBlank (of the currently selected pipe) after the “trigger” register (the Start Address register or the Control register when plane enable bit transitioning from a zero to a one) is written – thus providing an atomic update of all display controls. If the currently selected pipe is disabled, the update is immediate.

2.10.6.1 DSPBCNTR—Display B/Sprite Plane Control Register

Memory Offset Address: 71180h
Default: 01000000h
Normal Access: Read/Write Double Buffered

The active set of registers will be updated on the VBlank (of the currently selected pipe) after the “trigger” register (the Start Address register or the Control register when plane enable bit transitioning from a zero to a one) is written – thus providing an atomic update of all display controls. If the currently selected pipe is disabled, the update is immediate.

Bit	Descriptions
31	Display B/Sprite (Primary B) Enable: This bit will enable or disable the display B/sprite. When this bit is set, the plane will generate pixels for display. When set to zero, memory fetches cease and display is blanked (from this plane) at the next VBLANK event from the pipe that this plane is assigned. At least one of the display pipes must be enabled to enable this plane. There is an override for the enable of this plane in the Pipe Configuration register. 1 = Enable 0 = Disable
30	Display B/Sprite Gamma Enable: This bit should only be changed after the plane has been disabled. It controls the bypassing of the display pipe gamma unit for this display plane pixel data only. For 8-bit indexed display data, this bit should be set to a one. 0 = Display B pixel data bypasses the pipe gamma correction logic (default). 1 = Display B pixel data is gamma corrected in the pipe gamma correction logic



Bit	Descriptions
29:26	Display B Source Pixel Format: This field selects the pixel format for the sprite/display B. Pixel formats with an alpha channel (8:8:8:8) should not use source keying. Before entering the blender, each source format is converted to 10 bits per pixel (details are described in the intermediate precision for the blender section of the Display Functions chapter). 000x = Reserved. 0010 = 8-bpp Indexed. 0011 = Reserved. 0100 = Reserved. 0101 = 16-bit BGRX (5:6:5:0) pixel format (XGA compatible). 0110 = 32-bit BGRX (8:8:8:8) pixel format. Ignore alpha. 0111 = Reserved. 1000 = 32-bit RGBX (10:10:10:2) pixel format. Ignore alpha. 1001 = Reserved. 1010 = BGRX 10:10:10:2 1011 = Reserved. 1100 = 64-bit RGBX (16:16:16:16) 16-bit floating point pixel format. Ignore alpha. 1101 = Reserved. 1110 = 32-bit RGBX (8:8:8:8) pixel format. Ignore alpha. 1111 = Reserved.
25:24	Display B/Sprite Pipe Select: This is read-only and selects the display pipe that this plane is assigned to. It is hardwired to pipe B. 00 = Select Pipe A 01 = Select Pipe B (default: cannot be changed) 10 = Reserved for pipe C 11 = Reserved for pipe D
23	Key Window Enable: This applies only to devices with a Display Plane C. It determines what area of the screen the source key compare should be applied. This bit is set to one when the color key is used as a destination key for display C. Display plane C must be enabled on the same pipe and display A should not be enabled on this pipe for this to be used. The function is only effective when display C is enabled and defined by Z-order to be behind display B. 0 = If keying is enabled, it applies to the entire display B plane 1 = If keying is enabled, it applies only to the intersection between display B and display C
22	Source Key Enable: When used as a sprite or a secondary this enables source color keying. Sprite pixel values that match the key will become transparent. Source keying allows a plane that is behind (below) this plane to show through where the display B data matches the display B key. This function is overloaded to provide display C destination keying when combined with the key window enable bit.. Setting this bit is not allowed when the display pixel format includes an alpha channel. 0 = Sprite source key is disabled (default) 1 = Sprite source key is enabled.



Bit	Descriptions
21:20	Pixel Multiply: This cause the display plane to duplicate lines and pixels sent to the assigned pipe. In the line/pixel doubling mode, the horizontal pixels are doubled and lines are sent twice. Asynchronous flips are not used in this mode. Programming Notes: Asynchronous flips are not permitted when pixel multiply is enabled. 00 = No duplication 01 = Line/pixel Doubling 10 = Reserved 11 = Pixel Doubling only
19:16	Reserved: Write as zero
15	180° Display Rotation: This mode causes the display plane to be rotated 180°. In addition to setting this bit, software must also set the base address to the lower right corner of the unrotated image. [DevCL] Do not enable 180° rotation together with Frame Buffer Compression 0 = No rotation 1 = 180° rotation
14:11	Reserved
10	Tiled Surface This bit indicates that the display B surface data is in tiled memory. The tile pitch is specified in bytes in the DSPBSTRIDE register. Only X tiling is supported for display surfaces. When this bit is set, it affects the hardware interpretation of the DSPBLINOFF, DSPBTILEOFF, and DSPBSURF registers. 0 = Display B surface uses linear memory 1 = Display B surface uses X-tiled memory
9:0	Reserved: Write as zero



2.10.6.2 DSPBLINOFFSET—Display B/Sprite Linear Offset Register

Memory Offset Address: 71184h
Default: 00000000h
Normal Access: Read/Write Double Buffered

This register specifies the panning for the display surface. The surface base address is specified in the DSPBSURF register, and this register is used to describe an offset from that base address. Bit 10 of DSPBCNTR specifies whether the display B surface is in linear or tiled memory. When the surface is in linear memory, this field contains the byte offset of the plane data in graphics memory. When the surface is tiled, the contents of this register are ignored.

This register can be written directly through software or by load register immediate command packets in the command stream.

This register is double buffered by VSYNC only. A change to this register will take effect on the next vsync following the write.

Bit	Descriptions
31:0	Display B Offset: This register provides the panning offset into the display B plane. This value is added to the surface address to get the graphics address of the first pixel to be displayed. This offset must be at least pixel aligned. This offset is the difference between the address of the upper left pixel to be displayed and the display surface address. When performing 180° rotation, this offset must be the difference between the last pixel of the last line of the display data in its unrotated orientation and the display surface address.

2.10.6.3 DSPBSTRIDE—Display B/Sprite Stride Register

Memory Offset Address: 71188h
Default: 00000000h
Normal Access: Read/Write Double Buffered

Bit	Descriptions
31:0	Display B/Sprite Stride: This is the stride for display B/Sprite in bytes. When using linear memory, this must be 64 byte aligned. When using tiled memory, this must be 512 byte aligned. The maximum value for this register is fixed. This register is updated through a command packet passed through the command stream or writes to this register. When it is desired to update both this and the start register, the stride register must be written first because the write to the start register is the trigger that causes the update of both registers on the next VBLANK event. When using tiled memory, the actual memory buffer stride is limited to a maximum of 16K bytes. The display stride must be power of 2 when doing Asynch Flips. The display stride must be 8KB or greater when doing Asynch Flips together with 180 rotation. The value in this register is updated through the command streamer during a synchronous flip.



2.10.6.4 DSPBKEYVAL—Sprite Color Key Value Register

Memory Offset Address: 71194h
Default: 00000000h
Normal Access: Read/Write Double Buffered

This register specifies the key color to be used with the mask bits to determine if the sprite source data matches the key. This register will only have an effect when the sprite color key is enabled. The overlay destination key value is used for overlay keying when Display B is being used as a secondary display with overlay destination keying enabled.

Bit	Descriptions
31:24	Reserved: Write as zero
23:16	Red Key Value: Specifies the color key value for the sprite red/Cr channel.
15:8	Green Key Value: Specifies the color key value for the sprite green/Y channel.
7:0	Blue Key Value: Specifies the color key value for the sprite blue/Cb channel.

2.10.6.5 DSPBKEYMSK—Sprite Color Key Mask Register

Memory Offset Address: 71198h
Default: 00000000h
Normal Access: Read/Write Double Buffered

This register specifies the key mask to be used with the color value bits to determine if the sprite source data matches the key when enabled. A zero bit in the mask indicates that the corresponding bit match failure should be ignored when determining if the pixel matches.

Bit	Descriptions
31:24	Reserved: Write as zero
23:16	Red Mask Value: Specifies the color key mask for the sprite red/Cr channel.
15:8	Green Mask Value: Specifies the color key mask for the sprite green/Y channel.
7:0	Blue Mask Value: Specifies the color key mask for the sprite blue/Cb channel.



2.10.6.6 DSPBSURF—Display B Surface Address Register

Memory Offset Address: 7119Ch
Default: 00000000h
Normal Access: Read/Write Double buffered

Writing to this register triggers the display plane flip. When it is desired to change multiple display B registers, this register should be written last as a write to this register will cause all new register values to take effect.

Bit	Descriptions
31:29	Reserved
28:12	Display B Surface Base Address: This address specifies the surface base address. When the surface is tiled, panning is specified using (x, y) offsets in the DSPBTILEOFF register. When the surface is in linear memory, panning is specified using a linear offset in the DSPBLINOFF register. This address must be 4K aligned. When performing asynchronous flips and the display surface is in tiled memory, this address must be 256K aligned. This register can be written directly through software or by command packets in the command stream. It represents an offset from the graphics memory aperture base and is mapped to physical pages through the global GTT. This address must be 128K aligned for linear memory.
11:0	Reserved



2.10.6.7 DSPBTILEOFF—Display B Tiled Offset Register

Memory Offset Address: 711A4h
 Default: 00000000h
 Normal Access: Read/Write Double buffered

This register specifies the panning for the display surface. The surface base address is specified in the DSPBSURF register, and this register is used to describe an offset from that base address. Bit 10 of DSPBCNTR specifies whether the display B surface is in linear or tiled memory. When the surface is in linear memory, the offset is specified in the DSPBLINOFF register and the contents of this register are ignored. When the surface is tiled, the start position is specified in this register as an (x, y) offset from the beginning of the surface.

This register can be written directly through software or by load register immediate command packets in the command stream.

This register is double buffered by VBLANK only. A change to this register will take effect on the next vblank following the write.

Bit	Descriptions
31:28	Reserved: Write as zero
27:16	Plane Start Y-Position: These 12 bits specify the vertical position in lines of the beginning of the active display plane relative to the display surface. When performing 180° rotation, this field specifies the vertical position of the lower right corner relative to the start of the active display plane in the unrotated orientation.
15:12	Reserved: Write as zero
11:0	Plane Start X-Position: These 12 bits specify the horizontal offset in pixels of the beginning of the active display plane relative to the display surface. When performing 180° rotation, this field specifies the horizontal position of the lower right corner relative to the start of the active display plane in the unrotated orientation. When display stride is 16KB and doing Asynch Flips, do not program the offset to give pans of 7680 to 8191 bytes.

2.10.6.8 DSPBFLPQSTAT—Flip Queue Status Register

Memory Offset Address: 71200h
 Default: 00000800h
 Normal Access: Read/Write with Read-Only fields

Bit	Descriptions
31:16	Reserved: Write as zero (RO)
15:8	Queue Free Entry Count (RO): This value indicates the number of free entries in the queue at the time that the register was read. The total number of entries in the queue is the sum of the occupied entry count and the free entry count.
7:0	Queue Occupied Entry Count (RO): This value indicates the number of occupied entries in the queue at the time that the register was read. The total number of entries in the queue is the sum of the occupied entry count and the free entry count.



2.10.7 Video BIOS Registers (71400h- 714FFh)

2.10.7.1 VGACNTRL—VGA Display Plane Control Register

Memory Offset Address: 71400h
 Default: 00000000h
 Normal Access: Read/Write

This register provides support for VGA compatibility modes. This register is used by video BIOS only.

Bit	Descriptions
31	VGA Display Disable: This bit will disable the VGA compatible display mode. It has no effect on VGA register or A0000-BFFFF memory aperture accesses which are controlled by the PCI configuration and VGA register settings. VGA display should only be enabled if all display planes other than VGA are disabled. After enabling the VGA, most display planes need to stay disabled, only the VGA popup (cursor A) can be enabled. VGA 132 Column text mode is not supported. 0 = VGA Display Enabled 1 = VGA Display Disabled
30	VGA /Pop-up 2X Centered Mode Scaling. When this bit is set to a one, the VGA and pop-up data is scaled using pixel doubling in both the horizontal and vertical direction for use on un-scaled flat panel displays. Setting this bit allows the VGA to run at higher dot clock frequencies and creates a larger (4x the size) image for better quality on larger displays. It is intended for use in one of the centering modes when not using the internal panel fitting. Do not use it for native VGA modes or when internal panel fitting is used to scale VGA. In the situations where it is used, for 1280 wide or larger panels this bit should be set. For exactly 1280 wide panels, the Nine-dot disable bit should also be set. This operation is in addition to the VGA functions that double the pixels and lines. 0 = VGA display is normal size 1 = VGA and VGA popup data is doubled in the horizontal and vertical direction.
29	VGA Pipe Select: This bit only applies to devices with dual pipe support. For devices with a single display pipe, this bit will be ignored. For dual pipe devices, this bit determines which pipe is to receive the VGA display data. This must be changed only when the VGA display is in the disabled state via the VGA display disable bit or during the write to enable VGA display. 0 = Selects Assigns the VGA display to Pipe A 1 = Selects Assigns the VGA display to Pipe B
28:27	Reserved: Software must preserve the contents of these bits.
26	VGA Border Enable: This bit determines if the VGA border areas during VGA centering modes are included in the active display area and do or do not appear on integrated TV encoder output and devices that use centering such as on DVO connected flat panel, TV displays, or integrated panels. For use with the internal panel fitting logic, the border if enabled will be scaled along with the pixel data. Setting this bit allows the popup to be positioned overlapping the border area of the image. 0 = VGA Border areas are not included in the image size calculations for centering only active area. 1 = VGA Border areas are enabled and is passed to the display pipe for display and used in the image size calculation for centering modes



Bit	Descriptions
25:24	VGA Centering Enable: VGA centering modes use the pipe timing generators to determine the actual display timings. This would normally correspond to the display panel size and timings. The VGA registers determine the centered VGA image height and width. The VGA border may or may not be considered in the calculation selected by the VGA Border Enable bit. For a proper image, the VGA image size should not exceed the pipe timing generator active rectangle. When using the internal panel fitting logic, the horizontal image size needs to be less than or equal to 2048 pixels to generate a proper image. The VGA image will either be centered within the pipe timing rectangle or appear in the upper left corner. Upper left corner centered mode is generally used for external panel scaling where the DVO stall signal is used and is always used for internal panel fitting operation. When panel fitter is enabled on the same pipe as VGA this register setting is ignored and upper left corner centered mode is always selected. When centering is disabled, the VGA CRTC registers determine the display timing compatible with legacy VGA devices for driving CRT like devices. 00 = VGA centering is disabled, VGA operates in Native VGA mode or when driving integrated TV (even when using integrated TV for hires mode). 01 = Reserved, invalid setting. 10 = VGA centering is enabled, VGA image appears in the upper left corner of the larger rectangle 11 = Reserved, invalid setting.
23	VGA Palette Read Select: This bit only applies to dual display pipe devices and determines which palette VGA palette read accesses will occur from. 0 = VGA palette reads will access Palette A (default). 1 = VGA palette reads will access Palette B VGA palette reads are reads from I/O address 0x3c9.
22	VGA Palette A Write Disable: This determines which palette the VGA palette writes will have as a destination. One or both palettes can be the destination. If both are disabled, writes will not affect the palette RAM contents. 0 = VGA palette writes will update Palette A (default). 1 = VGA palette writes will not update Palette A VGA palette writes are writes to I/O address 0x3C9h.
21	Dual Pipe VGA Palette B Write Disable: This determines which palette the VGA palette writes will have as a destination. One or both palettes can be the destination. If both are disabled, writes will not affect the palette RAM contents. 0 = VGA palette writes will update Palette B (default). 1 = VGA palette writes will not update Palette B VGA palette writes are writes to I/O address 0x3C9h.
20	Legacy VGA 8-Bit Palette Enable: This bit only affects reads and writes to the palette through VGA I/O addresses. In the 6-bit mode, the 8-bits of data are shifted up two bits on the write (upper two bits are lost) and shifted two bits down on the read. It provides backward compatibility for original VGA programs (in its default state) as well as VESA VBE support for 8-bit palette. It does not affect palette accesses through the palette register MMIO path. 0 = 6-bit DAC (default). 1 = 8-bit DAC.
19	Palette Bypass (TEST MODE): 0 = Pass VGA data through the palette for translation (Normal Operation) 1 = Bypass the palette for allowing testing without loading palette both VGA and popup data will bypass the palette in this mode.



Bit	Descriptions
18	Nine Dot Disable Prevents DOS applications from setting the VGA display into a real 9-dot per character operation mode, instead the device emulates that using 8-dots per character. This is intended to provide VGA compatibility on DVI type connectors and integrated panels where there would otherwise not be room for the 720 horizontal pixels or 1440 pixels when horizontally doubled. The VGA register bit SR01<0> functionality is disabled. VGA panning control handles the pseudo 9-dot mode when both this bit is set and SR01<0> is clear. 0 = Enable use of 9-dot enable bit in VGA registers 1 = Ignore the 9-dot per character bit and always use 8
17	Reserved
16:8	Reserved: Software must preserve the contents of these bits.
7:6	Blink Duty Cycle: Controls the VGA text mode blink duty cycle <u>relative to the cursor blink duty cycle</u>. 00 = 100% Duty Cycle, Full Cursor Rate (Default) 01 = 25% Duty Cycle, $\frac{1}{2}$ Cursor Rate 10 = 50% Duty Cycle, $\frac{1}{2}$ Cursor Rate 11 = 75% Duty Cycle, $\frac{1}{2}$ Cursor Rate
5:0	VSYNC Blink Rate: Controls the VGA blink rate in terms of the number of VSYNCs per on/off cycle. These bits are programmed with the (VSYNCs/cycle)/2-1. The proper programming of this register is determined by the VSYNC rate that the display requires when in a VGA display mode.



2.10.7.2 SWFxx—Software Flag Registers

Memory Offset Address:	SWF00 = 70410h
	SWF01 = 70414h
	SWF02 = 70418h
	SWF03 = 7041Ch
	SWF04 = 70420h
	SWF05 = 70424h
	SWF06 = 70428h
	SWF07 = 7042Ch
	SWF08 = 70430h
	SWF09 = 70434h
	SWF0A = 70438h
	SWF0B = 7043Ch
	SWF0C = 70440h
	SWF0D = 70444h
	SWF0E = 70448h
	SWF0F = 7044Ch
	SWF10 = 71410h
	SWF11 = 71414h
	SWF12 = 71418h
	SWF13 = 7141Ch
	SWF14 = 71420h
	SWF15 = 71424h
	SWF16 = 71428h
	SWF17 = 7142Ch
	SWF18 = 71430h
	SWF19 = 71434h
	SWF1A = 71438h
	SWF1B = 7143Ch
	SWF1C = 71440h
	SWF1D = 71444h
	SWF1E = 71448h
	SWF1F = 7144Ch
	SWF30 = 72414h
	SWF31 = 72418h
	SWF32 = 7241Ch

Default:	00000000h
Normal Access:	Read/Write

These 32-bit registers are used as scratch pad data storage space and have no direct effect on hardware operation. The use of these registers is defined by the software architecture.

Bit	Descriptions
31:0	Reserved for Video BIOS and Drivers



2.10.8 Display C (Sprite or Second Overlay) Control

All of the basic control Display C/Sprite registers are double buffered. The active set is updated after the trigger register is written followed by a VBLANK event on the pipe that the Display C/Sprite is assigned. The Display C color adjustment registers are not double buffered and take effect immediately.

2.10.8.1 DSPCCNTR—Display C Sprite Control Register

Memory Offset Address: 72180h
Default: 00000000h
Normal Access: Read/Write Double Buffered

The active set of basic control registers will be updated on the VBlank (of the currently selected pipe) after the “trigger” register (the Start Address register or the Control register when plane enable bit transitioning from a zero to a one) is written – thus providing an atomic update of all display controls with the exception of the Display C color control registers. If the currently selected pipe is disabled, the update is immediate.

Bit	Descriptions
31	Display C/Sprite Enable: This bit will enable or disable the display C/sprite. When this bit is set, the plane will generate pixels for display to be combined by the blender for the target pipe. When set to zero, memory fetches cease and display is blanked (from this plane) at the next VBLANK event from the pipe that this plane is assigned. At least one of the display pipes must be enabled to enable this plane. There is an override for the enable of this plane in the Pipe Configuration register. 0 = Disable 1 = Enable
30	Display C/Sprite Gamma Enable: There are two gamma adjustments possible in the display C data path. This bit controls the gamma correction in the display pipe not the gamma control in this plane. It affects only the pixel data from this display plane. For pixel format of 8-bit indexed, this bit should be set to a one. Gamma correction logic that is contained in the display C logic is disabled by loading the default values into those registers. 0 = Display C pixel data bypasses the display pipe gamma correction logic (default). 1 = Display C pixel data is gamma corrected in the pipe gamma correction logic



Bit	Descriptions
29:26	Display C Source Pixel Format: This field selects the pixel format for the sprite/display C. Pixel formats with an alpha channel should not use source keying. Before entering the blender, each source format is converted to 10 bits per pixel (details are described in the intermediate precision for the blender section of the Display Functions chapter). 0000 = YUV 4:2:2 packed (see byte order below). 0001 = Reserved 0010 = 8-bpp Indexed. 0011 = Reserved. 0100 = Reserved. 0101 = 16-bit BGRX (5:6:5:0) pixel format (XGA compatible). 0110 = 32-bit BGRX (8:8:8:8) pixel format. Ignore alpha. 0111 = 32-bit BGRA (8:8:8:8) pixel format with pre-multiplied alpha channel. 1000 = 32-bit RGBX (10:10:10:2) pixel format. Ignore alpha. 1001 = 32-bit RGBA (10:10:10:2) pixel format 1010 = Reserved. 1011 = Reserved. 1100 = Reserved. 1101 = Reserved. 1110 = 32-bit RGBX (8:8:8:8) pixel format. Ignore alpha. 1111 = 32-bit RGBA (8:8:8:8)
25:24	Display C/Sprite Pipe Select: This selects the display pipe that this plane is assigned to. This bit can change when the sprite is active and causes a flip to the other display pipe. The position and size is still required to fit within the pipe source rectangle. The synchronization is handled in the hardware. 00 = Select Pipe A 01 = Select Pipe B 10 = Reserved for pipe C 11 = Reserved for pipe D
23	Reserved
22	Sprite Source Key Enable: When used as a sprite in the 16/32-bpp modes without alpha this enables source color keying. Sprite pixel values that match (within range) the key will become transparent. Setting this bit is not allowed when the display C pixel format includes an alpha channel. [DevBW] Erratum: This bit must always be set to 0 when display C pixel format is YUV 0 = Sprite source key is disabled (default) 1 = Sprite source key is enabled.
21:20	Pixel Multiply: This cause the display plane to duplicate lines and pixels sent to the assigned pipe. In the line/pixel doubling mode, the horizontal pixels are doubled and lines are sent twice. This is a method of scaling the source image by two (both H and V). 00 = No line/Pixel duplication 01 = Line/Pixel Doubling 10 = Reserved 11 = Pixel Doubling only



Bit	Descriptions
19	Color Conversion Disabled: This bit enables or disables the color conversion logic. Color conversion is intended to be used with the formats that support YUV formats such as the YUV 4:2:2 packed format and x:8:8:8 and 8:8:8:8 formats. Formats such as RGB5:5:5 and 5:6:5 do not have YUV versions. 0 = Pixel data is sent through the conversion logic (only applies to YUV formats) 1 = Pixel data is not sent through the YUV->RGB conversion logic.
18	YUV Format: This bit specifies the source YUV format for the YUV to RGB color conversion operation. This field is ignored when source data is RGB. 0 = ITU-R Recommendation BT.601 1 = ITU-R Recommendation BT.709
17:16	YUV byte Order: This field is used to select the byte order when using YUV 4:2:2 data formats. For other formats, this field is ignored. 00 = YUYV 01 = UYVY 10 = YVYU 11 = VYUY
15	180° Display Rotation: This mode causes the display plane to be rotated 180°. In addition to setting this bit, software must also set the base address to the lower right corner of the unrotated image and calculate the x, y offset as relative to the lower right corner. 0 = No rotation 1 = 180° rotation
14:11	Reserved
10	Tiled Surface: This bit indicates that the display C surface data is in tiled memory. The tile pitch is specified in bytes in the DSPCSTRIDE register. Only X tiling is supported for display surfaces. When this bit is set, it affects the hardware interpretation of the DSPCTILEOFF, DSPCLINOFF, and DSPCSURF registers. 0 = Display C surface uses linear memory 1 = Display C surface uses X-tiled memory
9:3	Reserved: Write as zero
2	Display C Bottom: This bit will force the display C plane to be on the bottom of the Z order. 0 = Display C Z order is determined by the other control bits 1 = Display C is forced to be on the bottom of the Z order.
1:0	Reserved



2.10.8.2 DSPCLINOFF —Display C/Sprite Linear Offset Register

Memory Offset Address: 72184h
Default: 00000000h
Normal Access: Read/Write Double Buffered

This register specifies the panning for the display surface. The surface base address is specified in the DSPCSURF register, and this register is used to describe an offset from that base address. Bit 10 of DSPCCNTR specifies whether the display A surface is in linear or tiled memory. When the surface is in linear memory, this field contains the byte offset of the plane data in graphics memory. When the surface is tiled, the contents of this register are ignored.

This register can be written directly through software or by load register immediate command packets in the command stream.

This register is double buffered by VBLANK only. A change to this register will take effect on the next vblank following the write.

Bit	Descriptions
31:0	Display C Offset: This register provides the panning offset into the display C plane. This value is added to the surface address to get the graphics address of the first pixel to be displayed. This offset must be at least pixel aligned. This offset is the difference between the address of the upper left pixel to be displayed and the display surface address. When performing 180° rotation, this offset must be the difference between the last pixel of the last line of the display data in its unrotated orientation and the display surface address.

2.10.8.3 DSPCSTRIDE—Display C/Sprite Stride Register

Memory Offset Address: 72188h
Default: 00000000h
Normal Access: Read/Write Double Buffered

Bit	Descriptions
31:0	Display C/Sprite Stride: This is the stride for display C/Sprite in bytes. When using linear memory, this must be 64 byte aligned. When using tiled memory, this must be 512 byte aligned. This register is updated through a command packet passed through the command stream or writes to this register. When it is desired to update both this and the start register, the stride register must be written first because the write to the start register is the trigger that causes the update of both registers on the next VBLANK event. When using tiled memory, the actual memory buffer stride is limited to a maximum of 16K bytes.



2.10.8.4 DSPCPOS—Sprite Position Register

Memory Offset Address: 7218Ch
Default: 00000000h
Normal Access: Read/Write Double Buffered

These registers specify the screen position and size of the sprite. This register is double buffered. The load register is transferred into the active register on the asserting edge of Vertical Blank for the pipe that the display is assigned. When using the sprite as a secondary display, this should be set to the entire display rectangle.

Bit	Descriptions
31:28	Reserved: Write as zero
27:16	SpriteY-Position: These 12 bits specify the vertical position in lines of the sprite (upper left corner) relative to the beginning of the active video area. When performing 180° rotation, this field specifies the vertical position of the lower right corner relative to the end of the active video area in the unrotated orientation. The defined sprite rectangle must always be completely contained within the displayable area of the screen image.
15:12	Reserved: Write as zero
11:0	Sprite X-Position: These 12 bits specify the horizontal position in pixels of the sprite (upper left corner) relative the beginning of the active video area. When performing 180° rotation, this field specifies the horizontal position of the original lower right corner relative to the original end of the active video area in the unrotated orientation. The defined sprite rectangle must always be completely contained within the displayable area of the screen image.

2.10.8.5 DSPCSIZE—Sprite Height and Width Register

Memory Offset Address: 72190h
Default: 00000000h
Normal Access: Read/Write Double Buffered

This register specifies the height and width of the sprite in pixels and lines. The rectangle defined by the size and position should never exceed the boundaries of the display rectangle that the sprite is assigned to.

Bit	Descriptions
31:28	Reserved: Write as zero
27:16	Sprite Height: This register field is used to specify the height of the sprite in lines. The value in the register is the height minus one. The defined sprite rectangle must always be completely contained within the displayable area of the screen image.
15:12	Reserved: Write as zero
11:0	Sprite Width: This register field is used to specify the width of the sprite in pixels. This does not have to be the same as the stride but should be less than or equal to the stride (converted to pixels). The value in the register is the width minus one. The defined sprite rectangle must always be completely contained within the displayable area of the screen image. The sprite width is limited to even values when YUV source pixel format is used, or Pixel Multiply is set to Line/Pixel doubling or Pixel doubling only (actual width, not the width minus one value).



2.10.8.6 DSPCKEYMINVAL—Sprite Color Key Min Value Register

Memory Offset Address: 72194h
Default: 00000000h
Normal Access: Read/Write Double Buffered

This register specifies the key color to be used with the mask bits to determine if the sprite source data matches the key. This register will only have an effect when the sprite color key is enabled. The unused bits of the 5:5:5 or 5:6:5 formats must be filled with duplicates of the three or two MSBs of the pixel value.

Bit	Descriptions
31:24	Reserved: Write as zero
23:16	Red Key Min Value: Specifies the color key minimum value for the sprite red/Cr channel.
15:8	Green Key Min Value: Specifies the color key minimum value for the sprite green/Y channel.
7:0	Blue Key Min Value: Specifies the color key minimum value for the sprite blue/Cb channel.

2.10.8.7 DSPCKEYMSK—Sprite Color Key Mask Register

Memory Offset Address: 72198h
Default: 00000000h
Normal Access: Read/Write Double Buffered

Bit	Descriptions
31:3	Reserved: Write as zero
2	Red Channel Enable: Specifies the source color key enable for the red/Cr channel.
1	Green Channel Enable: Specifies the source color key enable for the green/Y channel.
0	Blue Channel Enable: Specifies the source color key enable for the blue/Cb channel



2.10.8.8 DSPCSURF—Display C Surface Address Register

Memory Offset Address: 7219Ch
Default: 00000000h
Normal Access: Read/Write Double buffered

Writing to this register triggers the display plane flip. When it is desired to change multiple display C registers, this register should be written last as a write to this register will cause all new register values to take effect.

Bit	Descriptions
31:29	Reserved
28:12	Display C Surface Base Address: This address specifies the surface base address. When the surface is tiled, panning is specified using (x, y) offsets in the DSPCTILEOFF register. When the surface is in linear memory, panning is specified using a linear offset in the DSPCLINOFF register. This address must be 4K aligned. This register can be written directly through software or by command packets in the command stream. It represents an offset from the graphics memory aperture base and is mapped to physical pages through the global GTT. The value in this register is updated through the command streamer during synchronous flips. This address must be 128K aligned for linear memory.
11:0	Reserved

2.10.8.9 DSPCKEYMAXVAL—Sprite Color Key Max Value Register

Memory Offset Address: 721A0h
Default: 00000000h
Normal Access: Read/Write Double Buffered

This register specifies the key color to be used with the mask bits to determine if the sprite source data matches the key. This register will only have an effect when the sprite color key is enabled.

Bit	Descriptions
31:24	Reserved: Write as zero
23:16	Red Key Max Value: Specifies the color key value for the sprite red/Cr channel.
15:8	Green Key Max Value: Specifies the color key value for the sprite green/Y channel.
7:0	Blue Key Max Value: Specifies the color key value for the sprite blue/Cb channel.



2.10.8.10 DSPCTILEOFF—Display C Tiled Offset Register

Memory Offset Address: 721A4h
Default: 00000000h
Normal Access: Read/Write Double buffered

This register specifies the panning for the display surface. The surface base address is specified in the DSPCSURF register, and this register is used to describe an offset from that base address. Bit 10 of DSPCCNTR specifies whether the display A surface is in linear or tiled memory. When the surface is in linear memory, the offset is specified in the DSPCLINOFF register and the contents of this register are ignored. When the surface is tiled, the start position is specified in this register as an (x, y) offset from the beginning of the surface.

This register can be written directly through software or by load register immediate command packets in the command stream.

This register is double buffered by VBLANK only. A change to this register will take effect on the next vblank following the write.

Bit	Descriptions
31:28	Reserved: Write as zero
27:16	Plane Start Y-Position: These 12 bits specify the vertical position in lines of the beginning of the active display plane relative to the display surface. When performing 180° rotation, this field specifies the vertical position of the lower right corner relative to the start of the active display plane in the unrotated orientation.
15:12	Reserved: Write as zero
11:0	Plane Start X-Position: These 12 bits specify the horizontal offset in pixels of the beginning of the active display plane relative to the display surface. When performing 180° rotation, this field specifies the horizontal position of the lower right corner relative to the start of the active display plane in the unrotated orientation.

2.10.8.11 DSPCFLPQSTAT—Flip Queue Status Register

Memory Offset Address: 72200h
Default: 00000800h
Normal Access: Read/Write with Read-Only fields

Bit	Descriptions
31:16	Reserved: Write as zero (RO)
15:8	Queue Free Entry Count (RO): This value indicates the number of free entries in the queue at the time that the register was read. The total number of entries in the queue is the sum of the occupied entry count and the free entry count.
7:0	Queue Occupied Entry Count (RO): This value indicates the number of occupied entries in the queue at the time that the register was read. The total number of entries in the queue is the sum of the occupied entry count and the free entry count.



2.10.9 Display C Color Adjustment

These functions provide mechanisms for control of image colors generated from Display C sources. These functions are mainly intended for use for YUV color format sources. Adjustments are made before the YUV to RGB conversion. They take effect even when the Conversion Bypass bit is set. For display source input data in RGB format, software must set all the color correction registers to their default values (equivalent to a bypass mode). **These registers are not double buffered, they take effect immediately after loading.**

2.10.9.1 DCLRC0—Display C Color Correction 0 Register

Memory Address: 721D0h
Default Value: 01000000h
Normal Access: RW
Size: 32 bits

Bit	Description
31:27	Reserved
26:18	Contrast: Contrast adjustment applies to YUV data. The Y channel is multiplied by the value contained in the register field. This signed fixed-point number is in 3i.6f format with the first 3 MSBs as the integer value and the last 6 LSBs as the fraction value. The allowed contrast value ranges from 0 to 7.53125 decimal. Bypassing Contrast, for YUV modes and for source data in RGB format, is accomplished by programming this field to a field value that represents 1.0 decimal or 001.000000 binary .
17:8	Reserved
7:0	Brightness: This field provides the brightness adjustment with a 8-bit 2's complement value ranging [-128, +127]. This value is added to the Y value after contrast multiply and before YUV to RGB conversion. A value of zero disables this adjustment affect. This 8-bit signed value provides half of the achievable brightness adjustment dynamic range. A full range brightness value would have a programmable range of [-255, +255]. Bypassing Brightness for YUV formats and for source data in RGB format is accomplished by programming this field to 0.



2.10.9.2 DCLRC1—Display C Color Correction 1 Register

Memory Address: 721D4h
 Default Value: 00000080h
 Normal Access: RW
 Size: 32 bits

The sum of the absolute value of SH_SIN and SH_COS must be limited to less than 8.

$$\text{ABS}(\text{SH\_SIN}) + \text{ABS}(\text{SH\_COS}) < 8$$

Bit	Description
31:27	Reserved
26:16	Saturation and Hue SIN (SH_SIN): This 11-bit signed fixed-point number is in 2's complement (s3i.7f) format with the MSB as the sign, next 3 MSBs as the integer value and the last 7 LSBs as the fraction value. This field can be used in two modes. When full range YUV data is operated on, this field contains the saturation value. When the range-limited YCbCr data is used, software should program this field with the product of the saturation multiplier value multiplied by the CbCr range scale factor (=128/112). Similar to the contrast field, there is no limit for saturation reduction – saturation = 0 means all pixels become the same value. However, increasing contrast can only be increased by a factor less than 8. For example, the largest contrast with value of 0x7.7F can bring input range [0, 32] to a full display color range of [0, 255]. Bypassing Hue, even for source data in RGB format, is accomplished by programming this field to 0.0.
15:10	Reserved
9:0	Saturation and Hue COS (SH_COS): This unsigned fixed-point number is in 3i.7f format with the first 3 MSBs be the integer value and the last 7 LSBs be the fraction value. This field can be used in two modes. When full range YUV data is operated on, this field contains the saturation value. When the range-limited YCbCr data is used, software should program this field with the product of the saturation multiplier value multiplied by the CbCr range scale factor (=128/112). Similar to the contrast field, there is no limit for saturation reduction – saturation = 0 means all pixels become the same value. However, increasing contrast can only be increased by a factor less than 8. For example, the largest contrast with value of 0x7.7F can bring input range [0, 32] to a full display color range of [0, 255]. Bypassing Saturation, even for source data in RGB format, is accomplished by programming this field to 1.0.



2.10.9.3 GAMC[5:0]— Display C Gamma Correction Registers

Memory Address:	721E0h – 721F7h GAMC5: 721E0h–721E3h GAMC4: 721E4h–721E7h GAMC3: 721E8h–721EBh GAMC2: 721ECh–721EFh GAMC1: 721F0h–721F3h GAMC0: 721F4h–721F7h
Default Value:	Linear R/G/B ramp GAMC5 = C0C0C0h GAMC4 = 808080h GAMC3 = 404040h GAMC2 = 202020h GAMC1 = 101010h GAMC0 = 080808h
Normal Access:	R/W
Size:	6 x 32 bits

These registers are used to determine the characteristics of the gamma correction for the display C pixel data pre-blending. Additional gamma correction can be done in the display pipe gamma if desired. The pixels input to the gamma correction are 8-bit per channel pixels, and the output of the gamma correction is 10 bit per channel pixels. The gamma curve is represented by specifying a set of points along the curve. Each register has 32 bits, which are written to and read from together when accessed by the software. They are the six individual breakpoints on a logarithmically spaced color intensity space as shown in the following figure. The 8-bit values in the register are extended to 10 bit values in hardware by concatenating two zeroes onto the LSBs. The two end points (0 and 1023) have fixed values 0 and 1023, respectively. The appropriate Gamma breakpoint pairs (adjacent) are selected for each color component (Red, Green and Blue), and the output is interpolated between these two breakpoint values. The Gamma Correction registers (GAMC0 to GAMC5) are not double-buffered. They should be updated when the sprite is off. Otherwise, screen artifacts may show.

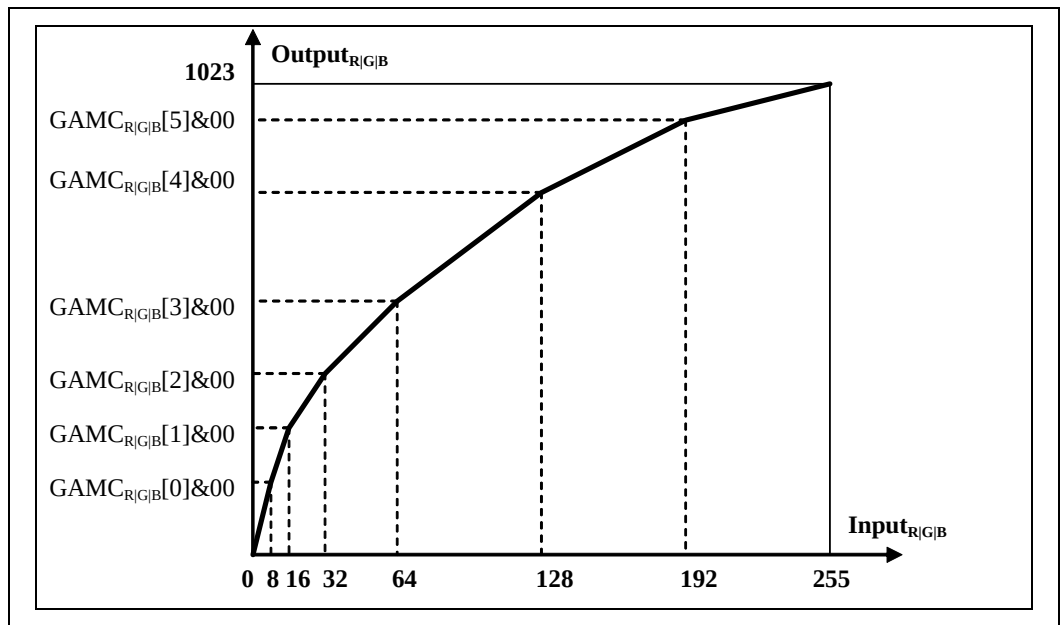
When the output from sprite is set in YUV format by programming CSC bypass, normally software should also bypass this gamma unit. However, since this gamma unit can also be viewed as a nonlinear transformation, it can be used, for whatever reason, in YUV output mode. In this case, the mapping of the three sets of piecewise linear map are as the following:

- Red to Cr (also called V)
- Green to Y
- Blue to Cb (also called U)

Bit	Description
31:24	Reserved
23:16	Red (V/Cr):
15:8	Green (Y):
7:0	Blue (U/Cb):



Figure 2-1. Programming of the Piecewise-linear Estimation of Gamma Correction Curve





2.10.10 Performance Counters

The performance counter hardware provides a method for software to monitor memory latency and hardware FIFO statistics for the purpose of optimizing memory accesses for the display planes.

2.10.10.1 PCSRC—Performance Counter Source Register

Memory Address Offset: 73000h
Default Value: 00000000h
Normal Access: RW
Size: 32 bits

Bit	Description
31	Performance Counter Enable: This bit enables the performance counter. 0 = Performance counter is disabled 1 = Performance counter is enabled.
30	Interrupt Enable: This bit enables/disables an interrupt when the threshold value programmed in the Performance Counter Source register matches the value of the performance counter. 0 = Interrupt disabled 1 = Interrupt enabled
29	Reset Counter: This bit indicates when the counter will be reset. 1 = Reset after each frame, summing all events in the frame 0 = Reset after each event within the frame
28	Max Or Min: This bit tells whether the stored counter value for an event is the maximum or the minimum value. The previous value is used to do the compare. 0 = Stored value is the maximum latency 1 = Stored value is the minimum latency



Bit	Description
27:23	Source For Performance Counter: These bits indicate the source for the performance counter. 00000 = Overlay Register Request Latency 00001 = VGA Font Request Latency 00010 = VGA Character Request Latency 00011 = Display A FIFO Status 00100 = Display B FIFO Status 00101 = Display C FIFO Status 00110 = Cursor A FIFO Status 00111 = Cursor B FIFO Status 01000 = Display Streamer A TLB Latency 01001 = Display Streamer B TLB Latency 01010 = Display Streamer C TLB Latency 01011 = Cursor Streamer A TLB Latency 01100 = Cursor Streamer B TLB Latency 01101 = Overlay Streamer TLB Latency 01110 = Display Streamer A Request Latency 01111 = Display Streamer B Request Latency 10000 = Display Streamer C Request Latency 10001 = Cursor Streamer A Request Latency 10010 = Cursor Streamer B Request Latency 10011 = Overlay Streamer Request Latency 10100 = Display A Command Request Latency 10101 = Display B Command Request Latency 10110 = Display C Command Request Latency 10111 = Cursor A Command Request Latency 11000 = Cursor B Command Request Latency 11001 = Overlay Command Request Latency 11010 = Reserved 11011 = Reserved
22:16	Reserved: Write as zero.
15:0	Performance Counter Threshold Value: This value is used to compare against the performance counter. If the performance counter matches this value, an interrupt is generated if the interrupt bit is enabled. When the source selected is DDB FIFO status, the threshold value is used to program the value needed to monitor in the DDB FIFO. No interrupt is generated in this condition.



2.10.10.2 PCSTAT—Performance Counter Status Register

Memory Address Offset: 73004h
 Default Value: 00000000h
 Normal Access: RO
 Size: 32 bits

Bit	Description
31	Overflow: This bit indicates whether the 16-bit counter overflowed or not. 0 = Counter is valid 1 = Counter is invalid since it overflowed
30	Reset Counter: This bit indicates when the counter will be reset. 1 = Reset after each frame, sum of all event in the frame 0 = Reset after each event within the frame
29	Max Or Min: This bit tells whether the stored counter value for an event is the maximum or the minimum value of the previous event. 0 = Stored value is the maximum latency 1 = Stored value is the minimum latency
28:24	Source For Performance Counter: These bits indicate the source for the performance counter. 00000 = Overlay Register Request Latency 00001 = VGA Font Request Latency 00010 = VGA Character Request Latency 00011 = Display A FIFO Status 00100 = Display B FIFO Status 00101 = Display C FIFO Status 00110 = Cursor A FIFO Status 00111 = Cursor B FIFO Status 01000 = Display Streamer A TLB Latency 01001 = Display Streamer B TLB Latency 01010 = Display Streamer C TLB Latency 01011 = Cursor Streamer A TLB Latency 01100 = Cursor Streamer B TLB Latency 01101 = Overlay Streamer TLB Latency 01110 = Display Streamer A Request Latency 01111 = Display Streamer B Request Latency 10000 = Display Streamer C Request Latency 10001 = Cursor Streamer A Request Latency 10010 = Cursor Streamer B Request Latency 10011 = Overlay Streamer Request Latency 10100 = Display A Command Request Latency 10101 = Display B Command Request Latency 10110 = Display C Command Request Latency 10111 = Cursor A Command Request Latency 11000 = Cursor B Command Request Latency 11001 = Overlay Command Request Latency 11010 = Reserved 11011 = Reserved
23:16	Reserved: Write as zero.
15:0	Performance Counter Value: This is the value of the performance counter for the source indicated in the source field.



3 Overlay Registers

3.1 Introduction and Register Summary

This chapter contains the register descriptions for the overlay portion of a family of integrated graphics devices. These registers do vary by device within the family of devices so special attention needs to be paid to which devices use which registers and register fields.

Different devices within the family may add, modify, or delete registers or register fields relative to another device in the same family based on the supported functions of that device. Additional information on the use and programming of these registers can be found in the display chapter. This document covers both desktop and mobile products.

The following table contains the sections break down where the register information is contained within this chapter:

Address Range	Description
30000h–3FFFFh	Overlay Registers

3.1.1 Terminology

Description	Software Use	Should be implemented as
Reserved write as zero.	Software must always write a zero to these bits. This allows new features to be added using these bits that will be disabled when using old software and as the default case.	These are read-only bits that always read as zeros or r/w bits that are default to zero.
Reserved write as one.	Software must always write a one to these bits. This allows new features to be added using these bits that will be disabled when using old software and as the default case.	
Reserved for BIOS Do not change	Driver access to these bits must read these bits that have been set through an initialization operation before writing this register so that the bits can remain unchanged.	According to each specific bit
Reserved for Video BIOS	These register bits will be used only by video BIOS and drivers should not change them.	These are read/write bits that have no hardware function. They are intended for use by the video BIOS for storage.



Description	Software Use	Should be implemented as
Reserved for Compatibility	For functions that are no longer needed these bits had old use, but now does nothing. New software should use the new method.	Read/write bits that have no functions.
Use for compatibility only	Under specific conditions, these bits functions as in the old part, new software should use the new method.	According to each specific bit
Read-Only	This bit is read-only. The read value is determined by hardware. Writes to this bit have no effect.	According to each specific bit. The bit value is determined by hardware and not affected by register writes to the actual bit.
Reserved read-only	Don't assume a value for these bits. Writes have no effect.	These bits should read as zero.
Reserved read-only write as zero	Don't assume a value for these bits, always write a zero.	These bits should read as zero.
Read/Clear	This bit can be read and writes to it with a one cause the bit to clear.	Hardware events cause the bit to be set and the bit will be cleared on a write operation where the corresponding bit has a one for a value.
Read/Write	This bit can be read or written.	
Double Buffered	Write when desired	Takes effect only after a particular event such as a VBLANK.

3.2 Overlay Register Definition

3.2.1 Introduction to Overlay Registers

The registers detailed in this chapter are used across products. However, slight changes may be present in some registers (i.e., for features added or removed), or, alternatively, some registers may be removed entirely. This section contains the description of the registers that control the Overlay hardware in the graphics controller. The registers include:

- Overlay Control registers
- Overlay Filter Coefficient registers
- Overlay Gamma and color control registers
- Overlay Scaling registers
- Overlay Source and destination sizes
- Overlay keying values

One hardware overlay is implemented in the graphics device. Note that most of the Overlay registers are indirectly written by first setting up a buffer in memory and then instructing the graphics controller to update the on-chip registers from this buffer.



Software can invoke the update process by either writing to the OVADD (with restrictions) register or by issuing an Overlay Flip instruction (MI_OVERLAY_FLIP).

The Overlay registers can be used for overlay gamma correction and color adjustment. They are read/written directly by MMIO (memory mapped I/O) without going through the buffer in memory.

The register categories are listed in the Overlay Register Categories table.

Figure 3-1. Overlay Register Memory Map

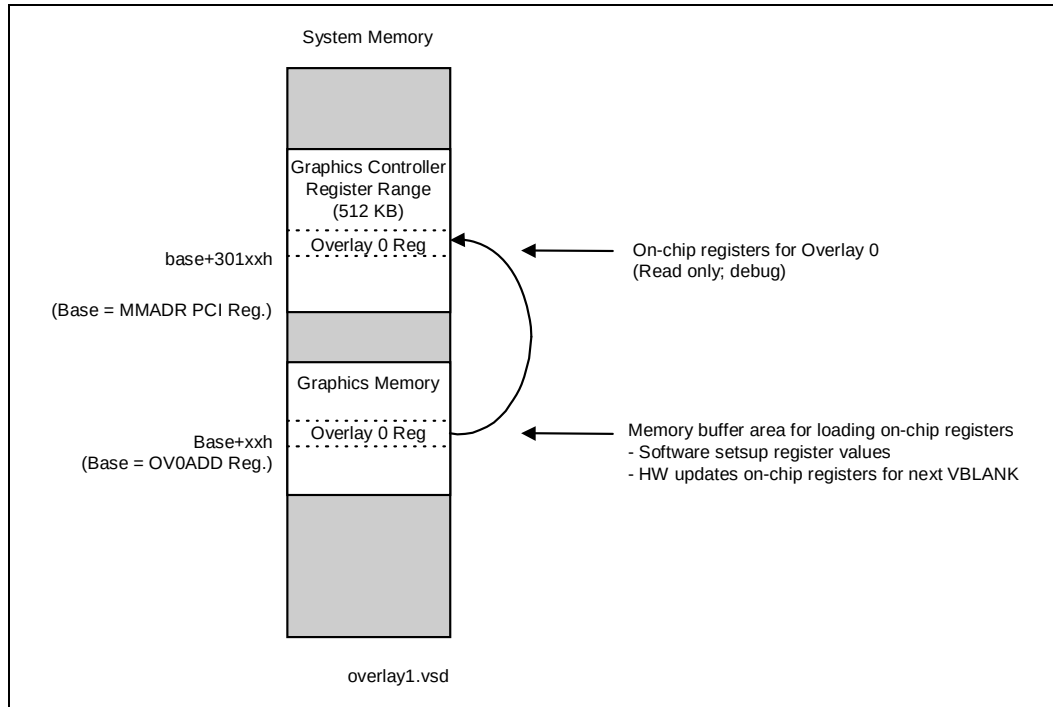


Table 3-1. Overlay Register Categories

Memory Address Offset	Register / Instruction Category	MMI (r/w)	Command Streamer (W)	Load from Memory	Comment
30000h–30003h	Overlay Control Register Update Address (OVADD)	RW	W (See Flip Inst)	No	Provides physical memory address of buffer area used for updating on-chip registers Used to update Overlay Control registers: A write to register OVADD causes hardware to update on-chip registers on next display VBLANK.
30004h–30007h	Overlay Test Register (OTEST)	RW	No	No	Overlay test
30008h–3000Bh	Display/Overlay Status Register (DOVSTA)	RO	No	No	Overlay status bits



Memory Address Offset	Register / Instruction Category	MMI (r/w)	Command Streamer (W)	Load from Memory	Comment
3000Ch–3000Fh	Display/Overlay Extended Status Register (DOVSTAEX)	RO	No	No	Overlay extended status bits
30010h–30027h	Piecewise Linear Gamma Correction Registers (GAMMA[0:5])	RW	No	No	Six breakpoint values in the adjustment curve define a gamma correction function using a piece-wise linear approximation. The end points of the curve are fixed to the lowest and highest possible values. Each color has its own correction curve. All three colors are accessed simultaneously for each breakpoint. Breakpoints on an eight-bit resolution are at 8,16,32,64,128, and 192 respectively with 0 and 255 as the two end points.
30028h–30057h	Reserved	N/A	No	No	
30058h–30067h	Video Sync Lock Phase Registers (SYNCPH0-3)	RO	No	No	Used for software-based display sync lock to a hardware or software overlay flip.
30068h– 300FFh	Reserved	N/A	No	No	
30100h–30177h (on chip RO)	Compatible Overlay Register Sets Buffer Pointers Stride/Source Size Initial Phase Window Position/Size Scale Factor Color Correction Color Key Configuration Command	RO	No	Yes	Memory image registers: On-chip registers are not directly write-able but are loaded from a memory image or via the command stream. Software sets up buffer in memory. Software writes to OVADD Register, which provides memory buffer address location and causes hardware to read memory buffer and update on-chip registers during next appropriate Display VBLANK.
30178h–3019Fh	Reserved	N/A	No	No	
301A0h–301A7h (on chip RO)	Overlay Scaling Registers (FASTHSCALE and UVSCALEV)	RO	No	Yes	Memory buffer registers: Base_Address + (A0h to A4h)
301ACh–302FFh	Reserved	N/A	No	No	
30300h–307FFh	Overlay Filter Coefficients	No	No	Yes	Memory buffer registers: Base_Address + (200h to 600h)



Memory Address Offset	Register / Instruction Category	MMI (r/w)	Command Streamer (W)	Load from Memory	Comment
30424h–30FFFh	Reserved	N/A	No	No	

3.2.1.1 Updating Overlay Registers

Most Overlay Registers are double-buffered (specifically, all those with a “Yes” in the “Load from memory” column in the above table). The write-able copy resides in memory; this memory-resident buffer is called the “Overlay Register Back Buffer”, or the “memory back buffer”. This allows software to initiate an update of Overlay Register values at any time, including during active overlay display. The Overlay Register Back Buffer includes Overlay Control Registers and Overlay Filter Coefficient Registers. If two sequential requests to load registers occur before the actual registers are loaded result in the last request to be honored.

Multiple display pipes are supported, where the pipe assign bit selects which display pipe VBLANK event causes the registers to be loaded except in the case where the currently assigned display pipe is currently disabled. When the pipe assign bit is changing, a two step process automatically disables the overlay from its current pipe on the VBLANK event for that pipe and enables it on the VBLANK event of the new display pipe. This means that the pipe switch can take two VBLANKs to complete.

Two conditions can cause the on-chip Overlay Registers to be updated from the Overlay Register Back Buffer:

1. A write to the OVADD register clears bit 31 of the Overlay Status Register and causes the values currently in the Overlay Register Back Buffer in memory to be loaded into the corresponding on-chip registers on the next Display VBLANK event. (VBLANK event is defined here as the active edge of the Display Vertical Blank period, including border). The Overlay Register Update Status bit is asserted after all registers are updated from memory. This method cannot be used to configure the cache during the initial enabling of the overlay or final disable. The cache must be configured to load registers. In general this method is not to be used!
2. Similarly, an Overlay Flip (or Extended Flip) Instruction, issued via the Command Stream, will cause the exact same register update sequence, upon the next Display VBLANK event.

3.2.1.2 Read Path for Overlay Registers

The on-chip overlay registers, with the exception of filter coefficient registers, are readable for state saving, debug readback, or other software purposes. The address offsets correspond to the memory offsets from the graphics base address used to read the registers.

Memory Address Offset: 30xxxh (xxx = register offset)

In normal operation, software manages the Overlay Register Back Buffer.



3.2.2 OVADD—Overlay Register Update Address Register

Memory Address Offset: 30000h–30003h
Default Value: 00000000h
Normal Access: R/W
Size: 32 bits

This register provides a graphics memory address that will be used on the next Overlay register update. This graphics memory address points to an array of Overlay registers. This register cannot be used to flip the overlay if the cache has not been configured through a command pipe flip packet.

Bit	Description
31:12	Register Update Address: Graphics memory address that will be used on the next Overlay register update. For forward compatibility purpose, software must ensure that the graphics memory address is at least 4K aligned (i.e., the lower 1 bits are assumed to be zero and do not exceed the proper values).
11:1	Reserved: MBZ
0	Coefficient Flag: This bit determines if the coefficients will be loaded in addition to the other overlay registers during an overlay flip. 0 = Load registers only, do not load coefficients 1 = Load registers and coefficients

3.2.3 OTEST—Overlay Test Register

Memory Address Offset: 30004h–30007h
Default Value: XXXXXXXX
Normal Access: R/W
Size: 32 bits

Bit	Description
31:0	Reserved: MBZ
2:0	Reserved: MBZ



3.2.4 DOVSTA—Display / Overlay Status Register

Memory Address Offset: 30008h–3000Bh
 Default Value: XXXXXXXX (values change dynamically)
 Normal Access: RO
 Size: 32 bits

This read-only register indicates status for the overlay.

Since the Overlay pipe can be assigned to either display pipe, references to display are either the pipe A timing generator or the pipe B timing generator depending on which the Overlay logic is currently slaved to.

Bit	Description
31	Overlay Register Update Status (OVR_UPDT): This status bit is only applicable to the case that software writes OVADD (via either MMIO or Command Stream) to trigger hardware to load the on-chip overlay registers from the memory back buffer. This bit is cleared to zero (by hardware) when OVADD is written. It is set to one when the hardware completes loading the on-chip registers. This bit in effect acknowledges that a buffer flip has completed. When flipping the overlay from one active display pipe to the other, this occurs when the VBLANK of the target display pipe following the VBLANK of the current display pipe has occurred. 0 = Overlay Register Update register has been written, however, display VBLANK event has not yet occurred and Overlay Registers have not been loaded from memory. 1 = Overlay Register has not been updated since the last VBLANK event. This is the power-on default value.
30:22	Reserved: MBZ
21:20	Overlay Current Buffer (OC_BUF): This field indicates which overlay buffer is currently being displayed. It is updated at display VBLANK. The update occurs before display VBLANK interrupt. 00 = Buffer 0 01 = Buffer 1 1x = Reserved
19	Overlay Current Field (OC_FIELD): This bit indicates the overlay source field currently displayed. It is updated at display VBLANK, occurring before display VBLANK interrupt. It is only valid in interleaved buffer (or Field) mode. In non-interleaved buffer (or Frame) mode, this bit is always 0. (See bit 5 of the Command Register for more on Field/Frame modes). 0 = Field 0 1 = Field 1
18	Reserved: MBZ
17	Overlay Hardware Error (OV_ERR): This status bit indicates that there has been an error detected during Overlay operation. It is set when the data underrun condition is detected. It's cleared by reading this register. This bit can be used in diagnostic tests to determine if there is enough memory bandwidth for a given resolution.
16	RESERVED (was Target Field (T_FIELD)) MBZ
15	Reserved: MBZ



Bit	Description
14	Not Active Display Pixel (NACT_PEL): This bit indicates the Display Horizontal Blank Active state of the graphics pipe that the overlay is associated with. The Display Horizontal Blank includes the horizontal Border. It is updated in real time, set by the leading edge of Overlay's display HBLANK and cleared by the trailing edge of the HBLANK. 0 = HBLANK inactive 1 = HBLANK active
13	Reserved: MBZ
12	Not Active Display Scan Line (NACT_LINE): This bit indicates the Display Vertical Blank Active state of the graphics pipe that the overlay is associated with. The Display Vertical Blank includes the vertical Border. This field is updated in real time, set by leading edge of display VBLANK and cleared by the trailing edge of VBLANK. 0 = VBLANK inactive 1 = VBLANK active
11:0	Reserved: MBZ

3.2.5 DOVSTAEX—Display / Overlay Extended Status Register

Memory Address Offset: 3000Ch–3000Fh
Default Value: XXXXXXXX (values change dynamically)
Normal Access: RO
Size: 32 bits

This read-only register provides extended status information about the overlay. The format is RESERVED.



3.2.6 OGAMC[5:0]—Overlay Dedicated Gamma Correction Registers

Memory Address Offset:	30010h – 30027h GAMC5: 30010h–30013h GAMC4: 30014h–30017h GAMC3: 30018h–3001Bh GAMC2: 3001Ch–3001Fh GAMC1: 30020h–30023h GAMC0: 30024h–30027h
Default Value:	Linear R/G/B ramp GAMC5 = C0C0C0h GAMC4 = 808080h GAMC3 = 404040h GAMC2 = 202020h GAMC1 = 101010h GAMC0 = 080808h
Normal Access:	R/W
Size:	5 x 32 bits

These registers are used to determine the characteristics of the gamma correction for the overlay data. The gamma correction receives 8-bit per channel pixels input, and sends out 10 bit per channel pixels to the display blender. Each register has 32 bits, which are written to and read from together when accessed by the software. They are the six individual breakpoints on a logarithmically spaced color intensity space as shown in the following figure. The two end points (0 and 255) have fixed values 0 and 255, respectively. The appropriate Gamma breakpoint pairs (adjacent) are selected for each color component (Red, Green and Blue), and the output is interpolated between these two breakpoint values. The difference between any two points should never exceed 7E hex and sequential points must be greater than or equal to the previous point. The Gamma Correction registers (GAMC0 to GAMC5) are not double-buffered. They should be updated when the overlay is off. Otherwise, video anomaly may show.

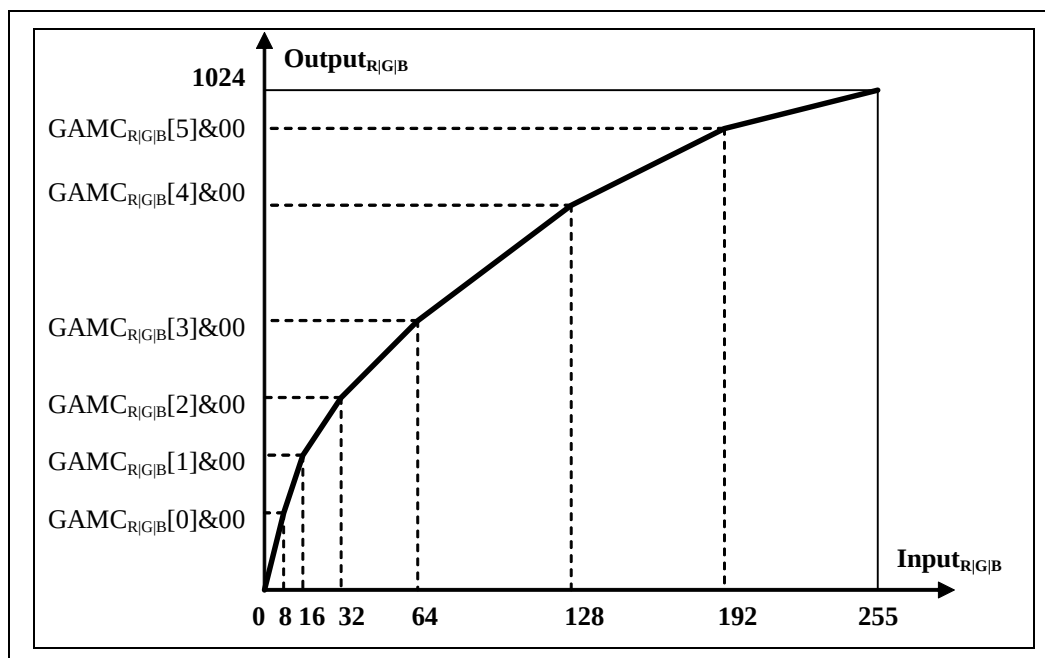
When the output from overlay is set in YUV format by programming CSC bypass, normally software should also bypass this gamma unit. However, since this gamma unit can also be viewed as a nonlinear transformation, it can be used, for whatever reason, in YUV output mode. In this case, the mapping of the three sets of piecewise linear map are as the following:

- Red to Cr (also called V)
- Green to Y
- Blue to Cb (also called U)

Errata: Overlay fails when gamma point 5 is set to 0x80.

Bit	Description
31:24	Reserved: MBZ
23:16	Red (V/Cr)
15:8	Green (Y)
7:0	Blue (U/Cb)

Figure 3-2. Programming of the Piecewise-linear Estimation of Gamma Correction Curve



3.2.7 Overlay Sync Lock Registers

These registers are used to implement a software involved sync lock. The four registers contain the last four flip event phases and are loaded in a round robin fashion. The valid bit indicates that the register contains valid phase information. The frame value is the frame number determines the order that the registers were loaded and the phase is the line that the flip occurred on. Display events are driven from the display pipe that the overlay is assigned to. The registers are reset to zero at power up and when the overlay is disabled.



3.2.7.1 OVRSYNCPH[0-3]— Overlay Flip Sync Lock Phase Registers

Memory Address Offset: 30058h–30067h
 SYNCPH0: 30058h–3005Bh
 SYNCPH1: 3005Ch–3005Fh
 SYNCPH2: 30060h–30063h
 SYNCPH3: 30064h–30067h
 Default Value: 00h
 Normal Access: RO
 Size: 4 x 32 bits

Bit	Descriptions
31	Data Valid: This bit informs software that the phase information in the register is valid. It is set when a flip event occurs on the overlay (capture or software) and is cleared when the register is read. 0 = Data is not valid 1 = Data is valid
30	Overflow: This bit indicates that the register was updated before software had read the register and cleared the valid bit. 0 = No overflow has occurred 1 = Overflow
29:24	Frame: This field contains the value in the frame counter (a free running counter which counts display VBLANK events) when the flip occurred. When the frame counter reaches its maximum value, it just wraps around to zero.
23:12	Reserved: MBZ
11:0	Phase: This field indicates the phase in lines between the display VBLANK event and the overlay flip event.

3.2.8 Overlay Memory Image Offset Registers

Only the Register Update Address register should be written while the overlay is active. There is a debug read path to the on-chip active overlay control registers for testing overlay internal register functionality. The debug read is from the active overlay registers.

Note that the Overlay Enable bit in Overlay Command register must be 0 after power up. Default value for the rest on-chip compatible control registers are meaningless (don't care), since it is required to load them from memory before the Overlay engine is enabled. The newly-added registers have to be defaulted to the compatible values in order for the new hardware to support legacy software, which does not update the additional registers (including control and filter coefficient registers).



Table 3-2. Overlay Memory Offset Registers

Register Mnemonic	Register Name	Memory Address Offset	On-chip Address Offset
Compatible Overlay Control Registers			
OBUF_0Y	Overlay Buffer 0 Y Pointer	00h–03h	30100h–30103h
OBUF_1Y	Overlay Buffer 1 Y Pointer	04h–07h	30104h–30107h
OBUF_0U	Overlay Buffer 0 U Pointer	08h–0Bh	30108h–3010Bh
OBUF_0V	Overlay Buffer 0 V Pointer	0Ch–0Fh	3010Ch–3010Fh
OBUF_1U	Overlay Buffer 1 U Pointer	10h–13h	30110h–30113h
OBUF_1V	Overlay Buffer 1 V Pointer	14h–17h	30114h–30117h
OSTRIDE	Overlay Stride	18h–1Bh	30118h–3011Bh
YRGB_VPH	Y/RGB Vertical Phase 0/1	1Ch–1Fh	3011Ch–3011Fh
UV_VPH	UV Vertical Phase 0/1	20h–23h	30120h–30123h
HORZ_PH	Horizontal Phase	24h–27h	30124h–30127h
INIT_PHS	Initial Phase Shift	28h–2Bh	30128h–3012Bh
DWINPOS	Destination Window Position	2Ch–2Fh	3012Ch–3012Fh
DWINSZ	Destination Window Size	30h–33h	30130h–30133h
SWIDTH	Source Width	34h–37h	30134h–30137h
SWIDTHSW	Source Width In SWORDS	38h–3Bh	30138h–3013Bh
SHEIGHT	Source Height	3Ch–3Fh	3013Ch–3013Fh
YRGBSCALE	Y/RGB Scale Factor	40h–43h	30140h–30143h
UVSCALE	U V Scale Factor	44h–47h	30144h–30147h
OCLRC0	Overlay Color Correction 0	48h–4Bh	30148h–3014Bh
OCLRC1	Overlay Color Correction 1	4Ch–4Fh	3014Ch–3014Fh
DCLRKV	Destination Color Key Value	50h–53h	30150h–30153h
DCLRKM	Destination Color Key Mask	54h–57h	30154h–30157h
SCHRKVH	Source Chroma Key Value High	58h–5Bh	30158h–3015Bh
SCHRKVL	Source Chroma Key Value Low	5Ch–5Fh	3015Ch–3015Fh
SCHRKEN	Source Chroma Key Enable	60h–63h	30160h–30163h
OCONFIG	Overlay Configuration	64h–67h	30164h–30167h
OCMD	Overlay Command	68h–6Bh	30168h–3016Bh
	Reserved	6Ch–6Fh	30168h–3016Fh
OSTART_0Y	Overlay Surface Start 0 Y Pointer	70h–73h	30170h–30173h
OSTART_1Y	Overlay Surface Start 1 Y Pointer	74h–77h	30174h–30177h
OSTART_0U	Overlay Surface Start 0 U Pointer	78h–7Bh	30178h–3017Bh
OSTART_0V	Overlay Surface Start 0 V Pointer	7Ch–7Fh	3017Ch–3017Fh
OSTART_1U	Overlay Surface Start 1 U Pointer	80h–84h	30180h–30184h
OSTART_1V	Overlay Surface Start 1 V Pointer	84h–87h	30184h–30187h
OTILEOFF_0Y	Overlay Surface 0 Y Tiled Offset	88h–8Bh	30188h–3018Bh
OTILEOFF_1Y	Overlay Surface 1 Y Tiled Offset	8Ch–8Fh	3018Ch–3018Fh



Register Mnemonic	Register Name	Memory Address Offset	On-chip Address Offset
OTILEOFF _0U	Overlay Surface 0 U Tiled Offset	90h–93h	30190h–30193h
OTILEOFF _0V	Overlay Surface 0 V Tiled Offset	94h–97h	30194h–30197h
OTILEOFF _1U	Overlay Surface 1 U Tiled Offset	98h–9Bh	30198h–3019Bh
OTILEOFF _1V	Overlay Surface 1 V Tiled Offset	9Ch–9Fh	3019Ch–3019Fh
Additional Overlay Control Registers			
FASTHSCALE	Fast Horizontal Downscale	A0h–A3h	301A0h–301A3h
UVSCALEV	UV Vertical Downscale Integer	A4h–A7h	301A4h–301A7h
	Reserved	A8h–1FFh	301C0h–302FFh
Overlay Filter Coefficient Registers			
Y_VCOEFS	Overlay Y Vertical Filter Coefficient	200h–267h	30300h–30367h
	Reserved	268h–2FFh	30368h–303FFh
Y_HCOEFS	Overlay Y Horizontal Filter Coefficients	300h–3ABh	304000h–304ABh
	Reserved	3ACh–3FFh	304ACh–3034FFh
UV_VCOEFS	Overlay UV Vertical Filter Coefficients	500h–567h	30600h–30667h
	Reserved	568h–5FFh	30668h–306FFh
UV_HCOEFS	Overlay UV Horizontal Filter Coefficients	600h–667h	30700h–30767h
	Reserved	668h–6FFh	30768h–307FFh

3.2.8.1 Overlay Buffer Start Address Pointer Registers

These registers provide overlay buffer start address pointers into graphics memory. While the start address has alignment restrictions based on pixel format, the buffers must be 64B aligned. Pixel panning on a pixel basis (for formats that support per pixel panning) is done using the start address pointer. Overlay buffers need to be 64B aligned and the stride should be a 64B multiple. In addition, buffer start address pointers should always be aligned to the natural boundaries based on the data format, as shown in the following table.

Table 3-3. Overlay Buffer Start Address Alignment Restriction

Pixel Format		Alignment	
Color	Organization	Pixels	Bytes
YUV4:2:2	Packed	2	4
YUV /RGB (Test Only)	Planar	1	1



3.2.8.2 OBUF_0Y—Overlay Buffer 0 Y Linear Offset Register

Memory Buffer Address Offset: 00h–03h (R/W)
On-chip Reg. Mem Addr Offset: 30100h–30103h (RO; debug path)
Default Value: 00h
Access: see address offset above
Size: 32 bits

This value specifies the panning for the overlay surface. Bit 19 of OCOMD specifies whether the overlay surfaces are in linear or tiled memory. When the surface is in linear memory, this field contains the byte offset of the plane data in graphics memory. When the surface is tiled, the value in this register is ignored.

Bit	Description
31:0	Overlay Buffer 0 Y Linear Offset: This is the offset for the Y planar or YUV packed color data. This value is added to the surface address to get the graphics address of the first pixel to be displayed. It must be pixel aligned (e.g., low order bit is zero for 16-bpp packed formats). This offset is the difference between the address of the upper left pixel to be displayed and the overlay surface address. When mirroring horizontally (X backward) this field points to first byte of the last pixel of the line.

3.2.8.3 OBUF_1Y—Overlay Buffer 1 Y Linear Offset Register

Memory Address Offset: 04h–07h (R/W)
On-chip Reg. Mem Addr Offset: 30104h–30107 (RO; debug path)
Default Value: 00h
Access: see address offset above
Size: 32 bits

This value specifies the panning for the overlay surface. Bit 19 of OCOMD specifies whether the overlay surfaces are in linear or tiled memory. When the surface is in linear memory, this field contains the byte offset of the plane data in graphics memory. When the surface is tiled, the value in this register is ignored.

Bit	Description
31:0	Overlay Buffer 1 Y Linear Offset: This is the offset for the Y planar or YUV packed color data. This value is added to the surface address to get the graphics address of the first pixel to be displayed. It must be pixel aligned (e.g., low order bit is zero for 16-bpp packed formats). This offset is the difference between the address of the upper left pixel to be displayed and the overlay surface address. When mirroring horizontally (X backward) this field points to first byte of the last pixel of the line.



3.2.8.4 OBUF_0U—Overlay Buffer 0 U Linear Offset Register

Memory Address Offset: 08h–0Bh (R/W)
On-chip Reg. Mem Addr Offset: 30108h–3010B (RO; debug path)
Default Value: 00h
Access: see address offset above
Size: 32 bits

This value specifies the panning for the overlay surface. Bit 19 of OCOMD specifies whether the overlay surfaces are in linear or tiled memory. When the surface is in linear memory, this field contains the byte offset of the plane data in graphics memory. When the surface is tiled, the value in this register is ignored.

Bit	Description
31:0	Overlay Buffer 0 U Linear Offset: This is the offset for the U planar data. This value is added to the surface address to get the graphics address of the first pixel to be displayed. This offset is the difference between the address of the upper left pixel to be displayed and the overlay surface address. When mirroring horizontally (X backward) this field points to first byte of the last pixel of the line.

3.2.8.5 OBUF_0V—Overlay Buffer 0 V Linear Offset Register

Memory Address Offset: 0Ch–0Fh (R/W)
On-chip Reg. Mem Addr Offset: 3010Ch–3010Fh (RO; debug path)
Default Value: 00h
Access: see address offset above
Size: 32 bits

This value specifies the panning for the overlay surface. Bit 19 of OCOMD specifies whether the overlay surfaces are in linear or tiled memory. When the surface is in linear memory, this field contains the byte offset of the plane data in graphics memory. When the surface is tiled, the value in this register is ignored.

Bit	Description
31:0	Overlay Buffer 0 V Linear Offset: This is the offset for the V planar data. This value is added to the surface address to get the graphics address of the first pixel to be displayed. This offset is the difference between the address of the upper left pixel to be displayed and the overlay surface address. When mirroring horizontally (X backward) this field points to first byte of the last pixel of the line.



3.2.8.6 OBUF_1U—Overlay Buffer 1 U Linear Offset Register

Memory Address Offset: 10h–13h (R/W)
On-chip Reg. Mem Addr Offset: 30110h–30113 (RO; debug path)
Default Value: 00h
Access: see address offset above
Size: 32 bits

This value specifies the panning for the overlay surface. Bit 19 of OCOMD specifies whether the overlay surfaces are in linear or tiled memory. When the surface is in linear memory, this field contains the byte offset of the plane data in graphics memory. When the surface is tiled, the value in this register is ignored.

Bit	Description
31:0	Overlay Buffer 1 U Linear Offset: This is the offset for the U planar data. This value is added to the surface address to get the graphics address of the first pixel to be displayed. This offset is the difference between the address of the upper left pixel to be displayed and the overlay surface address. When mirroring horizontally (X backward) this field points to first byte of the last pixel of the line.

3.2.8.7 OBUF_1V—Overlay Buffer 1 V Linear Offset Register

Memory Address Offset: 14h–17h (R/W)
On-chip Reg. Mem Addr Offset: 30114h–30117h (RO; debug path)
Default Value: 00h
Access: see address offset above
Size: 32 bits

This value specifies the panning for the overlay surface. Bit 19 of OCOMD specifies whether the overlay surfaces are in linear or tiled memory. When the surface is in linear memory, this field contains the byte offset of the plane data in graphics memory. When the surface is tiled, the value in this register is ignored.

Bit	Description
31:0	Overlay Buffer 1 V Linear Offset: This is the offset for the V planar data. This value is added to the surface address to get the graphics address of the first pixel to be displayed. This offset is the difference between the address of the upper left pixel to be displayed and the overlay surface address. When mirroring horizontally (X backward) this field points to first byte of the last pixel of the line.



3.2.8.8 OSTRIDE—Overlay Stride Register

Memory Address Offset: 18h–1Bh (R/W)
On-chip Reg. Mem Addr Offset: 30118h–3011B (RO; debug path)
Default Value: 00h
Access: see address offset above
Size: 32 bits

These values represent the stride of the overlay video data buffers. A stride value determines the line-to-line increment of the buffer, which is independent of the actual width of the overlay video data that gets displayed.

Bit	Description
31:16	Overlay UV Planar Buffer Stride (UV_STRIDE) : Only used for YUV Planar formats and gives the U or V buffer stride in bytes. This is a two's complement number and will be negative when Y mirroring is used with linear memory. Hardware ignores the least significant 6 bits. When using tiled memory, the stride must be positive, 512 byte aligned, and cannot exceed 2kb.
15:0	Overlay Y Planar or YUV422 Packed Buffer Stride (YRGB_STRIDE) . Buffer (Y planar or YUV packed) stride in bytes. This is a two's complement number and will be negative when Y mirroring is used with linear memory. Low order 6 bits must always be zero. The minimum stride is 512 bytes and the maximum is 8kb for packed mode and 4kb for linear modes. When using tiled memory, the stride must be positive.



3.2.9 Overlay Initial Phase Registers

These provide a spatial sub-pixel accurate adjustment in overlay source data coordinate. The values in the following initial phase registers are always fractional positive numbers. With 12-bit accuracy, the value ranges from 0 to (1-1/FFFh). When combined with the subtract one flags in the Initial Phase Shift register, the possible range for an initial phase is limited to:

$$-1 \geq \text{Initial Phase} < 2$$

There are two separate vertical initial phase registers that are used in field based video display operation. YRGB_VPH is for Y/RGB buffers. UV_VPH is for UV buffers in planar formats. An additional field indicates an additional offset to be used to eliminate edge effects when panning.

Initial Phase Shift table

Table 3-4. Vertical Initial Phase (for 1:1 scaling, upper left sync)

Format	Y line #	Yip	Yip Range	Cip	Cip Range	Ratio
422	n	0.00	-.5 to +.5	0.000	-.5 to +.5	1:1
420	1p	0.00	-.5 to +.5	-0.250	-.5 to +.5	1:2
	2p	1.00	+.5 to +1.5	0.250	0.0 to +.5	1:2
420i2i	1f0	0.00	-.5 to +.5	-0.125	-.375 to .125	1:2
	2f0	1.00	.5 to 1.5	0.375	.125 to .625	1:2
	1f1	0.00	-.5 to +.5	-0.375	-.625 to -.125	1:2
	2f1	1.00	.5 to 1.5	0.125	-.125 to .375	1:2
420i2p	1f0	0.25	-.25 to .75	0.000	-.125 to +.125	1:2
	2f0	1.25	.75 to 1.75	0.500	.25 to .75	1:2
	1f1	-0.25	-.75 to +.25	-0.500	-.75 to .25	1:2
	2f1	0.75	.25 to 1.25	0.000	-.125 to .125	1:2
410	1	0.00	-.5 to .5	-0.375	-.5 to -.25	1:4
	2	1.00	.5 to 1.5	-0.125	-.25 to 0.0	1:4
	3	1.00	.5 to 1.5	0.125	0.0 to .25	1:4 adjust Y base address + 1 line
	4	1.00	.5 to 1.5	0.375	.25 to .5	1:4 adjust Y base address + 2 lines



Table 3-5. Horizontal Initial Phase (for 1:1 scaling, upper left sync)

Format	Y pixel #	Yip	Yip Range	Cip	Cip Range	Ratio
422	1	0.00	-.5 to .5	0.000	-.25 to .25	1:2
	2	1.00	.5 to 1.5	0.500	.375 to .625	1:2
420mp2	1	0.00	-.5 to .5	0.000	-.25 to .25	1:2
	2	1.00	.5 to 1.5	0.500	.25 to .75	1:2
420mp1	1	0.00	-.5 to .5	-0.250	-.5 to 0.0	1:2
	2	1.00	.5 to 1.5	0.250	0.0 to .5	1:2
410	1	0.00	-.5 to .5	-0.375	-.5 to -.25	1:4
	2	1.00	.5 to 1.5	-0.125	-.25 to 0.0	1:4
	3	2.00	1.5 to 2.5	0.125	0.0 to .25	1:4
	4	3.00	2.5 to 3.5	0.375	.25 to .5	1:4

3.2.9.1 YRGB_VPH—Y/RGB Vertical Phase Register

Memory Address Offset: 1Ch–1Fh (R/W)
On-chip Reg. Mem Addr Offset: 3011Ch–3011Fh (RO; debug path)
Default Value: 00h
Access: see address offset above
Size: 32 bits

Bit	Description
31:20	Y/RGB Vertical Phase for Field 1 (YRGB_VPHASE1): This fractional value sets the initial vertical phase for field 1. For packed formats, it applies to both YUV and RGB buffers. For planar formats, it only applies to Y buffers. When the overlay buffers are in Interleaved buffer format, this value sets the initial vertical phase for field 1 in all YUV/RGB buffers. When the overlay buffers are in Non-Interleaved format, this value is intended for field 1. The value now applies to a particular YUV/RGB buffer.
19:16	Reserved: (Software should set all the 16 bits for forward compatibility in case more accurate DDA phase is implemented.)
15:4	Y/RGB Vertical Phase for Field 0 (YRGB_VPHASE0): This fractional value sets the initial vertical phase for field 0. For packed formats, it applies to YUV or RGB buffers. For planar formats, it only applies to Y buffers. When the overlay buffers are in Interleaved buffer format, this value sets the initial vertical phase for field 0 in all YUV/RGB buffers. When the overlay buffers are in Non-Interleaved format, this value is intended for field 0. The value applies to a particular YUV/RGB buffer.
3:0	Reserved: (Software should set all the 16 bits for forward compatibility in case more accurate DDA phase is implemented.)



3.2.9.2 UV_VPH—UV Vertical Phase Register

Memory Address Offset: 20h–23h (R/W)
On-chip Reg. Mem Addr Offset: 30120h–30123h (RO; debug path)
Default Value: 00h
Access: see address offset above
Size: 32 bits

Bit	Description
31:20	UV Vertical Phase for Field 1 (UV_VPHASE1): This fractional value is only used in YUV planar formats where the UV plane may have a different vertical initial phase from the Y data. When the overlay buffers are in Interleaved buffer format, this value sets the initial vertical phase for field 1 in all UV data buffers. When the overlay buffers are in Non-Interleaved format, this value is intended for buffers containing field 1 data. The value applies to a particular UV data buffer.
19:16	Reserved: (Software should set all the 16 bits for forward compatibility in case more accurate DDA phase is implemented.)
15:4	UV Vertical Phase for Field 0 (UV_VPHASE0): This fractional value is only used in YUV planar formats where the UV plane may have a different vertical initial phase from the Y data. When the overlay buffers are in Interleave buffer format, this value sets the initial vertical phase for field 0 in all UV data buffers. When the overlay buffers are in Non-Interleaved format, this value is intended for buffers containing field 0 data. The value applies to a particular UV data buffer.
3:0	Reserved: (Software should set all the 16 bits for forward compatibility in case more accurate DDA phase is implemented.)



3.2.9.3 HORZ_PH—Horizontal Phase Register

Memory Address Offset: 24h–27h (R/W)
 On-chip Reg. Mem Addr Offset: 30124h–30127h (RO; debug path)
 Default Value: 00h
 Access: see address offset above
 Size: 32 bits

Bit	Description
31:20	UV Horizontal Phase (UV_HPHASE): Sets the initial horizontal phase for the U and V data. Only used in YUV modes.
19:16	Reserved: MBZ
15:4	Y/RGB Horizontal Phase (YRGB_HPHASE): Sets the initial horizontal phase for both buffers/fields. Unlike the vertical initial phases, this does not change buffer-to-buffer or field-to-field. YUV modes use a separate initial phase for Y and UV data. This value will either be the actual initial phase or the initial phase minus one.
3:0	Reserved: MBZ

3.2.9.4 INIT_PHS—Initial Phase Shift Register

Memory Address Offset: 28h–2Bh (R/W)
 On-chip Reg. Mem Addr Offset: 30128h–3012Bh (RO; debug path)
 Default Value: 00h
 Access: see address offset above
 Size: 32 bits

This register provides a method to create a negative initial phase or one with a positive integer offset. If the corresponding bits are set to all ones, the initial phase is the fractional phase register value minus one. These bits should only be set in cases where the buffer pointer is pointing to the first pixel of the line or column because it will effectively cause the first pixel to be duplicated.

Bit	Description
31:24	Reserved: MBZ
23:20	Y0_VPP Initial Phase for Vertical Panning: 0000 = 0 Positive fractional phase 0001 = fractional phase plus one 0010 = fractional phase plus two 0011–1110 = Reserved 1111 = fractional phase minus one (-1)
19:16	Y1_VPP Initial Phase for Vertical Panning:
15:12	Y_HPP Initial Phase for Horizontal Panning:
11:8	UV0_VPP Initial Phase for Vertical Panning:
7:4	UV1_VPP Initial Phase for Vertical Panning:
3:0	UV_HPP Initial Phase for Horizontal Panning:



3.2.10 Overlay Destination Window Position/Size Registers

These registers allow for the positioning of the overlay data relative to the current graphics pipe that overlay is attached to (see the overlay and display association bits in OVIMMCTL register). It allows pixel accurate positioning.

3.2.10.1 DWINPOS—Destination Window Position Register

Memory Address Offset: 2Ch–2Fh (R/W)
On-chip Reg. Mem Addr Offset: 3012Ch–3012Fh (RO; debug path)
Default Value: 00h
Access: see address offset above
Size: 32 bits

Bit	Description
31:28	Reserved: MBZ
27:16	Destination Vertical Top: The vertical top line position of the overlay destination window in lines. Zero means the first active display line. When performing 180° rotation, this field specifies the vertical bottom line of the overlay destination window in its unrotated orientation. The range is [0, 4095].
15:12	Reserved: MBZ
11:0	Destination Horizontal Left: The horizontal left position of the overlay destination window in pixels. It determines where on the display screen the overlay window will begin. Zero means the first active pixel of the display screen. When performing 180° rotation, this field specifies the horizontal right position of the overlay destination window in its unrotated orientation. The range is [0, 4095].

3.2.10.2 DWINSZ—Destination Window Size Register

Memory Address Offset: 30h–33h (R/W)
On-chip Reg. Mem Addr Offset: 30130h–30133h (RO; debug path)
Default Value: 00h
Access: see address offset above
Size: 32 bits

Bit	Description
31:28	Reserved: MBZ
27:16	Destination Height: Overlay destination window height in lines (never specifies scan lines off the active display). For devices with internal panel fitting, the single line mode requires this value to be based on the panel height instead of the source height.
15:12	Reserved: MBZ
11:0	Destination Width: Overlay destination window width in pixels (never specifies pixels off the active display)



3.2.11 Overlay Source Size Registers

These registers provide information to the overlay engine on what data needs to be fetched from memory. If the overlay destination window is smaller than the result of the scaled (up or down) source, it will be clipped on the right and bottom of the overlay window.

3.2.11.1 SWIDTH – Source Width Register

Memory Address Offset: 34h–37h (R/W)
 On-chip Reg. Mem Addr Offset: 30134h–30137h (RO; debug path)
 Default Value: 00h
 Access: see address offset above
 Size: 32 bits

Bit	Description
31:27	Reserved: MBZ
26:16	UV Pixel Source Width: The number of valid pixels contained in a single line of planar UV source data. This field is unused in packed modes. For planar modes it's used for the U and V source width. The width for U and V sources is required to be the same. When the last pixel is reached and the complete destination window has not been filled, this pixel will be repeated until the end of the destination window. When displaying planar YUV 4:2:0 data, this field contains the number of U pixels (same as the number of V pixels), which should be one half of the Y pixels. When displaying planar YUV 4:1:0 planar data, this field contains the number of U pixels (same as the number of V pixels), which should be one quarter of the Y pixels. The UV source width cannot be more than half the Y source width. Source formats with a Y:UV ratio of less than 2:1, such as YUV 4:4:4 with a ratio of 1:1, are not supported.
15:12	Reserved: MBZ
11:0	Y/RGB Pixel Source Width: The number of valid pixels contained in a single line of source data. In both planar and packed modes, this is the Y source width. This field should include all contributing pixel data. When the last pixel is reached and the complete destination window has not been filled, this pixel will be repeated until the end of the destination window. For source formats of YUV 4:1:0 data, the atomic unit is four pixels For source formats of YUV 4:2:2 or YUV 4:2:0 data, the atomic unit is two pixels The starting offset within the buffer must reflect these restrictions.



3.2.11.2 SWIDTHSW—Source SWORD Width Register

Memory Address Offset: 38h–3Bh (R/W)
 On-chip Reg. Mem Addr Offset: 30138h–3013Bh (RO; debug path)
 Default Value: 00h
 Access: see address offset above
 Size: 32 bits

Hardware uses values in this register to determine the number of SWORDS (32 bytes) to be fetched from the memory for each overlay source scan line. The formula below account for the memory requirement of 64byte alignments.

Bit	Description
31:24	Reserved: MBZ
23:18	UV Source SWORD Width (UV_SWIDTHSW): This field sets the number of aligned SWORDS that contains all source U/V pixels in a single line of planar U/V source data. This field is valid only in planar modes. Minimum size is 1 SWORD. Overlay hardware uses this field to make memory request for each scan line in the planar U/V source buffers. In normal overlay horizontal display order (left to right), software should calculate this field based on the pixel-aligned buffer pointer <i>UVAddress</i> and the pixel-aligned source window width <i>UVSWidth</i> by $UVSWidthSW = \{ \{ [(UVAddress + UVSWidth + 0x3F) >> 6] - [UVAddress >> 6] \} << 1 \}$ In mirrored overlay horizontal display order (for right to left), software should use $UVSWidthSW = \{ \{ [UVAddress >> 6] - [(UVAddress - (UVSWidth + 0x3F)) >> 6] \} << 1 \}$
17:9	Reserved: MBZ
8:2	Y/RGB Source SWORD Width (YRGB_SWIDTHSW): This field sets the number of aligned SWORDS that contains all source Y/RGB pixels in a single source line. In planar modes, this is the Y source SWORD width. Minimum size is 1 SWORD. Overlay hardware uses this field to make memory request for each source scan line. In normal overlay horizontal display order (left to right), software should calculate this field based on the pixel-aligned buffer pointer <i>YAddress</i> and the pixel-aligned source window width <i>YSWidth</i> by $YSWidthSW = \{ \{ [(YAddress + YSWidth + 0x3F) >> 6] - [YAddress >> 6] \} << 1 \}$ In mirrored overlay horizontal display order (for right to left), software should use $YSWidthSW = \{ \{ [YAddress >> 6] - [(YAddress - (YSWidth + 0x3F)) >> 6] \} << 1 \}$
1:0	Reserved: MBZ



3.2.11.3 SHEIGHT—Source Height Register

Memory Address Offset: 3Ch–3Fh (R/W)
 On-chip Reg. Mem Addr Offset: 3013Ch–3013Fh (RO; debug path)
 Default Value: 00h
 Access: see address offset above
 Size: 32 bits

Bit	Description
31:26	Reserved: MBZ
25:16	UV Source Height: The number of lines contained in a single planar UV source data. In packed formats this is unused. For planar YUV formats it indicates the number of lines starting at and including the base address line contained in the UV source data. When the last line is reached and the complete destination window has not been filled, this line will be repeated until the end of the destination window. The maximum UV height is 1023 lines. The minimum height is the number of vertical taps being used times the vertical chroma-subsampling ratio. Height should include the line contributing to the interpolation of the last display overlay line.
15:11	Reserved: MBZ
10:0	Y/RGB Source Height: In packed formats this indicates the number of lines starting at and including the base address line contained in the source data. In planar formats, it is the number of Y lines. This is used to determine where the end of the source is in the vertical direction to handle the edge effects related to the vertical filter. When the last line is reached and the complete destination window has not been filled, this line will be repeated until the end of the destination window. The maximum height is 2047 lines. The minimum height is the number of vertical taps currently being used. A height must not be specified for lines that are completely not on the active display. Height should include the line contributing to the interpolation of the last display overlay line.



3.2.12 Overlay Scale Factor Registers

These registers provide the scaling information that is used to specify the amount of vertical and horizontal scaling. In the case of YUV formats, there are independent scale factors for Y and UV data to compensate for the various formats that include sub-sampled UV data. Software has to ensure that the programmed scaling factors for Y and UV data match to each other precisely. Upscaling or downscaling is selected using a zero value in the integer scale factor field.

The 12 bits fractional portion of the scaling factor corresponds to a scaling accuracy of 1/4096 of a pixel.

3.2.12.1 YRGBSCALE – Y/RGB Scale Factor Register

Memory Address Offset: 40h–43h (R/W)
On-chip Reg. Mem Addr Offset: 30140h–30143h (RO; debug path)
Default Value: 00h
Access: see address offset above
Size: 32 bits

Bit	Description
31:20	Y/RGB Vertical Scale Fraction (YRGB_VFRACT) : This is a fractional positive number that represents the scale factor to be used in vertical scaling for Y/RGB components. For packed formats, it applies to all color components. For planar YUV formats, it is use for the Y (luma) component only. This field represents a value range from 0 to 1 ($0 < \text{Vertical Scale Fraction} < 1$). Defining a Ver_Scale_Factor as the ratio of source height to destination height. $\text{Ver\_Scale\_Factor} = \text{Source Image Height} / \text{Destination Height}$ For upscaling, Y/RGB Vertical Scale Fraction is Ver_Scale_Factor, which is a fractional value. The integer portion should be set to zero. For downscaling, Y/RGB Vertical Scale Fraction is the fractional portion of Ver_Scale_Factor. The integer portion is in Y/RGB Vertical Downscale Integer field. For devices with integrated panel fitting, the single line mode requires that the vertical scale factor be determined by the panel height instead of the display mode's source height.
19	Reserved: MBZ
18:16	Y/RGB Horizontal Downscale Factor Integer (YRGB_HINT) : This field is used for horizontal down scale. A value of zero is used to indicate upscale. In the case of planar formats, it specifies the integer portion of the scale factor for Y data. This is used in backward compatible mode. Only horizontal downscaling of 2 (or less) is supported through the precision filter. For greater down scale factors, it must be done through the render path.
15	Reserved: MBZ



Bit	Description
14:3	Y/RGB Horizontal Scale Fraction (YRGB_HFRACT): This is a fractional positive number that represents the scale factor to be used in horizontal scaling for Y/RGB components. For both packed formats and planar YUV formats, this field applies to all color components. This field represents a value range from 0 to 1 $(0 < X \text{ Scale Fraction} < 1)$. For the situations that are other than downscaling, we can define a Hor_Scale_Factor as the ratio of source width to destination width. $\text{Hor\_Scale\_Factor} = \text{Source Width} / \text{Destination Width}$. For upscaling, Y/RGB Horizontal Scale Fraction is Hor_Scale_Factor, which is a fractional value. For downscaling, Y/RGB Horizontal Scale Fraction is the fractional portion of Hor_Scale_Factor. The integer portion is in Y/RGB Horizontal Downscale Integer field. For the situation that is downscaling, we can define a Hor_Scale_Factor as the ratio of source width to destination width for upscaling modes: $\text{Hor\_Scale\_Factor} = \text{Destination Width} / \text{Source Width}$. For downscaling, Y/RGB Horizontal Scale Fraction is the fractional portion of Hor_Scale_Factor.
2:0	Reserved: MBZ



3.2.12.2 UVSCALE—U V Scale Factor Register

Memory Address Offset: 44h–47h (R/W)
 On-chip Reg. Mem Addr Offset: 30144h–30147h (RO; debug path)
 Default Value: 00h
 Access: see address offset above
 Size: 32 bits

Bit	Description
31:20	UV Vertical Scale Fraction (UV_VFRACT): This is a fractional positive number that represents the scale factor to be used in vertical scaling for UV components. This field represents a value range from 0 to 1 ($0 \leq \text{Vertical Scale Fraction} < 1$). Defining a Ver_Scale Factor as the ratio of source height over destination height. $\text{Ver\_Scale\_Factor} = \text{Source Image Height} / \text{Destination Height}$ For upscaling, UV Vertical Scale Fraction is Ver_Scale_Factor, which is a fractional number. For downscaling, UV Vertical Scale Fraction is the fractional portion of Ver_Scale_Factor. The integer portion is in UV Vertical Downscale Integer field. . For devices with integrated panel fitting, the single line mode requires that the vertical scale factor be determined by the panel height instead of the display mode's source height.
19	Reserved: MBZ
18:16	UV Horizontal Downscale Factor Integer (UV_HINT). This field specifies the precision filter horizontal down scale integer portion. A value of zero is used to indicate upscale. In the case of planar formats, it specifies the integer portion of the scale factor for Y data. This is used in backward compatible mode. Only horizontal downscaling of 2 (or less) is supported through the precision filter. For greater down scale factors, must be done through the render path.
15	Reserved: MBZ
14:3	UV Horizontal Scale Fraction (UV_HFRACT): This is a fractional positive number that represents the scale factor to be used in horizontal scaling for UV components. This field represents a value range from 0 to 1 ($0 < \text{Hor Scale Fraction} < 1$). Defining a Hor_Scale Factor as the ratio of source width over destination width $\text{Hor\_Scale\_Factor} = \text{Source Width} / \text{Destination Width}$ For upscaling, UV Horizontal Scale Fraction is Hor_Scale_Factor, which is a fractional number. For downscaling, UV Horizontal Scale Fraction is the fractional portion of Hor_Scale_Factor. The integer portion is in UV Horizontal Downscale Integer field.
2:0	Reserved: MBZ



3.2.13 Overlay Color Correction Registers

Intended for use for YUV sources. Adjustments are made before the YUV to RGB conversion. They take effect even when the YUV2RGB Conversion Bypass bit in Overlay Command register is set. For overlay input data in RGB format, software must set all the color correction registers to their default values (equivalent to a bypass mode).

3.2.13.1 OCLRC0—Overlay Color Correction 0 Register

Memory Address Offset: 48h–4Bh (R/W)
On-chip Reg. Mem Addr Offset: 30148h–3014Bh (RO; debug path)
Default Value: 00h
Access: see address offset above
Size: 32 bits

Bit	Description
31:27	Reserved: MBZ
26:18	Contrast: Contrast adjustment applies to YUV data. The Y channel is multiplied by the value contained in the register field. This signed fixed-point number is in 3i.6f format with the first 3 MSBs as the integer value and the last 6 LSBs as the fraction value. The allowed contrast value ranges from 0 to 7.53125 decimal. Bypassing Contrast, for YUV modes and for source data in RGB format, is accomplished by programming this field to a field value that represents 1.0 decimal or 001.000000 binary.
17:8	Reserved: MBZ
7:0	Brightness: This field provides the brightness adjustment with a 8-bit 2's complement value ranging [-128, +127]. This value is added to the Y value after contrast multiply and before YUV to RGB conversion. A value of zero disables this adjustment affect. This 8-bit signed value provides half of the achievable brightness adjustment dynamic range. A full range brightness value would have a programmable range of [-255, +255]. Bypassing Brightness for YUV formats and for source data in RGB format is accomplished by programming this field to 0.



3.2.13.2 OCLRC1—Overlay Color Correction 1 Register

Memory Address Offset: 4Ch–4Fh (R/W)
 On-chip Reg. Mem Addr Offset: 3014Ch–3014Fh (RO; debug path)
 Default Value: 00h
 Access: see address offset above
 Size: 32 bits

The sum of the absolute value of SH_SIN and SH_COS must be limited to less than 8.

$$ABS(SH\_SIN) + ABS(SH\_COS) < 8$$

Bit	Description
31:27	Reserved: MBZ
26:16	Saturation and Hue SIN (SH_SIN): This field can be used in two modes: - When full range YUV data is operated on, this field contains the saturation value. - When the range-limited YCbCr data is used, software should program this field with the sum of the saturation multiplier value added to the CbCr range scale factor (=128/112). Similar to the contrast field, there is no limit for saturation reduction – saturation = 0 means all pixels become the same value. However, increasing contrast can only be increased by a factor less than 8. For example, the largest contrast with value of 0x7.7F can bring input range [0, 32] to a full display color range of [0, 255]. Bypassing Hue, even for source data in RGB format, is accomplished by programming this field to 0.0. Format: This 11-bit signed fixed-point number is in 2's complement (s3i.7f) format with the MSB as the sign, next 3 MSBs as the integer value and the last 7 LSBs as the fraction value
15:10	Reserved: MBZ
9:0	Saturation and Hue COS (SH_COS): This field can be used in two modes: - When full range YUV data is operated on, this field contains the saturation value. - When the range-limited YCbCr data is used, software should program this field with the sum of the saturation multiplier value added to the CbCr range scale factor (=128/112). Similar to the contrast field, there is no limit for saturation reduction – saturation = 0 means all pixels become the same value. However, increasing contrast can only be increased by a factor less than 8. For example, the largest contrast with value of 0x7.7F can bring input range [0, 32] to a full display color range of [0, 255]. Bypassing Saturation, even for source data in RGB format, is accomplished by programming this field to 1.0. Format: This unsigned fixed-point number is in 3i.7f format with the first 3 MSBs be the integer value and the last 7 LSBs be the fraction value.



3.2.14 Overlay Destination Color Key Registers

These source key registers are used for YUV overlay sources only. Adjustments are made before the RGB conversion. Destination keying is based on the format of the destination surface.

3.2.14.1 DCLRKV—Destination Color Key Value Register

Memory Address Offset: 50h–53h (R/W)
 On-chip Reg. Mem Addr Offset: 30150h–30153h (RO; debug path)
 Default Value: 00000000h
 Access: see address offset above
 Size: 32 bits

Bit	Description
31:24	Reserved: MBZ
23:0	Destination Color Key Value: Destination keying causes the color of the display primary or secondary to become transparent if it is within the overlay window and matches the destination color key. The key color is specified in the format of the destination surface data (converted to 24-bits) which is limited to either the primary (Display A) or secondary display (Display B). Destination keying is used only when overlay and Display Plane A or overlay and Display Plane B are on the same pipe. If both Display A and Display B are present on the same pipe as the overlay, overlay destination keying only applies to Display A. Color Channel Value: [23:16] = Red [15:08] = Green [07:00] = Blue

Table 3-6. Destination Color Key Value Programming in Each Supported Graphics Mode

Primary/Secondary Display Mode	Color Key Programmed Value		
	Red	Green	Blue
8-bit Indexed	Indexed Color [7:0]	Indexed Color [7:0]	Indexed Color [7:0]
15-bit RGB	RedKey[4:0] <<3	GreenKey[4:0] <<3	BlueKey[4:0] <<3
16-bit RGB	RedKey[4:0] <<3	GreenKey[5:0] <<2	BlueKey[4:0] <<3
32-bit x:8:8:8	RedKey[7:0]	GreenKey[7:0]	BlueKey[7:0]



3.2.14.2 DCLRKM—Destination Color Key Mask Register

Memory Address Offset: 54h–57h (R/W)
 On-chip Reg. Mem Addr Offset: 30154h–30157h (RO; debug path)
 Default Value: 00000000h
 Access: see address offset above
 Size: 32 bits

Bit	Description
31	Overlay Destination Color Key Enable: Destination keying when enabled forces the overlay surface Z order to be below the primary display. Pixels that match the key value (see above) become transparent and the overlay becomes visible at that pixel. The combination of source key enabled and destination key enabled is not a useful operational mode. 1 = destination color key is enabled 0 = destination color key is disabled
30	Enable Constant Alpha: Overlay constant alpha provides a way to apply an alpha value to all overlay pixels. Each pixel color channel is multiplied by the constant alpha before proceeding to the blender. This can be used to create fade out effects. This is intended for CE device use where the overlay might still be used to generate video output. 0 – Overlay Constant Alpha is disabled 1 – Overlay Constant Alpha is enabled
29:24	Reserved: MBZ
23:0	Destination Color Key Mask: [23:16] = Red [15:08] = Green [07:00] = Blue 0 = Bits that are active participants in the compare. 1 = Bits that are not active participants in the compare. A mask of all ones will disable the color key compare (as if all color channel values match).



Table 3-7. Destination Color Key Mask Programming in Each Supported Graphics Mode

	Color Key Mask			
Display Mode	Red	Green	Blue	Note
8-bit Indexed	11111111	11111111	00000000	Only blue channel is compared.
15-bit RGB	00000111	00000111	00000111	Compare 5 MSBs of each color channel only
16-bit RGB	00000111	00000011	00000111	Compare 5 or 6 MSBs of color channel only
32-bit RGB	00000000	00000000	00000000	Compare 8:8:8 data of the 32-bpp data Applies to all 32 bpp data formats except formats with alpha in use.

3.2.15 Overlay Source Chroma Key Registers

There is an overlay source key per overlay stream, which is used on a pixel basis. The Source comparison occurs after the horizontal zooming, but in the YUV formats before the color space conversion. Source Chroma Key is also referred as **Source Color Key**. Unlike Destination Color Key, Source Chroma Key uses a high-key value and a low-key value to specify a range. If the source data (overlay) is within the range, then the primary display is selected.

If the overlay is in RGB mode, the most significant bits are duplicated on the least significant to form 8-bit channels before the source key comparison is made. The RGB range expansion follows the rules in this table:

Overlay Source Format	Source	Expanded Value
RGB15	R[4:0]	R[4:0 ' 4:2]
	G[4:0]	G[4:0 ' 4:2]
	B[4:0]	B[4:0 ' 4:2]
RGB16	R[4:0]	R[4:0 ' 4:2]
	G[5:0]	G[5:0 ' 5:4]
	B[4:0]	B[4:0 ' 4:2]

While it makes no sense to enable both destination and source color keying for the overlay, destination Color Key failing takes precedence over the Source Chroma Key failing. If both fail, then the primary display is selected.



3.2.15.1 SCHRKVH—Source Chroma Key Value High Register

Memory Address Offset: 58h–5Bh (R/W)
 On-chip Reg. Mem Addr Offset: 30158h–3015Bh (RO; debug path)
 Default Value: 00h
 Access: see address offset above
 Size: 32 bits

Bit	Description
31:24	Reserved: MBZ
23:16	Y/Green Source Key High (YG_SKEY_H): This 8-bit field specifies the high end Y/Green value (less than or equal to) for the overlay source exclusion pixel data. When the corresponding Y/Green Source Key Mask Enables bit in SCLRKM register is set, an overlay value greater than this field fails the comparison and the overlay pixel passes. Otherwise, the overlay pixel becomes transparent. YUV format: Y value in [0,255] range. RGB format: Green value in [0,255] range.
15:8	U/Blue Source Key High (UB_SKEY_H): This field specifies the high end U/Blue value (less than or equal to) for the overlay source exclusion pixel data. When the corresponding U/Blue Source Key Mask Enables bit in SCLRKM register is set, an overlay value greater than this field fails the comparison and the overlay pixel passes. Otherwise, the overlay pixel becomes transparent. YUV format: U value in excess-128 format. RGB format: Blue value in [0,255] range.
7:0	V/Red Source Key High (VR_SKEY_H): This field specifies the high end V/Red value (less than or equal to) for the overlay source exclusion pixel data. When the corresponding V/Red Source Key Mask Enables bit in SCLRKM register is set, an overlay value greater than this field fails the comparison and the overlay pixel passes. Otherwise, the overlay pixel becomes transparent. YUV format: V value in excess-128 format. RGB format: Red value in [0,255] range.



3.2.15.2 SCHRKVL—Source Chroma Key Value Low Register

Memory Address Offset: 5Ch–5Fh (R/W)
 On-chip Reg. Mem Addr Offset: 3015Ch–3015Fh (RO; debug path)
 Default Value: 00h
 Access: see address offset above
 Size: 32 bits

Bit	Description
31:24	Reserved: MBZ
23:16	Y/Green Source Key Low (YG_SKEY_L): This field specifies the low end Y/Green value (greater than or equal to) for the overlay source exclusion pixel data. When the corresponding Y/Green Source Key Mask Enables bit in SCLRKM register is set, an overlay value less than this field fails the comparison and the overlay pixel passes. Otherwise, the overlay pixel becomes transparent. YUV format: Y value in [0,255] range. RGB format: Green value in [0,255] range.
15:8	U/Blue Source Key Low (UB_SKEY_L): This field specifies the low end U/Blue value (greater than or equal to) for the overlay source exclusion pixel data. When the corresponding U/Blue Source Key Mask Enables bit in SCLRKM register is set, an overlay value less than this field fails the comparison and the overlay pixel passes. Otherwise, the overlay pixel becomes transparent. YUV format: U value in excess-128 format. RGB format: Blue value in [0,255] range.
7:0	V/Red Source Key Low (VR_SKEY_L): This field specifies the low end V/Red value (greater than or equal to) for the overlay source exclusion pixel data. When the corresponding V/Red Source Key Mask Enables bit in SCLRKM register is set, an overlay value less than this field fails the comparison and the overlay pixel passes. Otherwise, the overlay pixel becomes transparent. YUV format: V value in excess-128 format. RGB format: Red value in [0,255] range.



3.2.15.3 SCHRKEN—Source Chroma Key Enable Register

Memory Address Offset: 60h–63h (R/W)
On-chip Reg. Mem Addr Offset: 30160h–30163h (RO; debug path)
Default Value: 0000h
Access: see address offset above
Size: 32 bits

Bit	Description
31:27	Reserved: MBZ
26:24	Source Chroma Key Enable: Each bit enables one channel of source chroma key range comparison. If the bit is a one, the comparison result is used otherwise it is ignored. Bit 26 = Enables Y/Green Comparison [23:16] Bit 25 = Enables U/Blue Comparison [15:8] Bit 24 = Enables V/Red Comparison [7:0]
23:8	Reserved: MBZ
7:0	Overlay Constant Alpha Value: This field provides the alpha value when constant alpha is enabled. A value of FF means fully opaque and a value of zero means fully transparent. Values in between those values allow for a blending of overlay with other surfaces.



3.2.15.4 OCONFIG—Overlay Configuration Register

Memory Address Offset: 64h–67h (R/W)
 On-chip Reg. Mem Addr Offset: 30164h–30167h (RO; debug path)
 Default Value: 0000h
 Access: see address offset above
 Size: 32 bits

Bit	Description
31:27	Reserved: MBZ
26:19	Reserved for Slot Time: Programmed for UMA to allow for improved CPU performance.
18	Overlay Pipe Select: Changing this bit causes the display to take VBLANK events from both pipes (in the proper order) to complete the change. Alviso/Grantsdale-G Errata GRG04: This bit may not be toggled when BOTH pipes are enabled unless the Overlay Enable bit in the Overlay Command Register is cleared to '0'. See further description under Overlay Enable bit. 0 = Overlay is assigned to display pipe A 1 = Overlay is assigned to display pipe B
17	Reserved: MBZ
16	Overlay Secondary Gamma Enable (GAMMA2_ENBL): The secondary gamma refers to the gamma RAM in the graphics pipe that overlay is associated with. This field provides independent control of the secondary gamma logic for overlay pixels. When this bit is in use, software MUST make sure that its correspondence compatibility bit (bit 26 of register PIPEACONF at 70008h) is set to zero all the time. 0 = Overlay pixel data bypasses secondary gamma correction logic (default). 1 = Overlay pixel data is gamma corrected by the secondary correction logic.
15:6	Reserved: MBZ
5	YUV2RGB Conversion Mode (CSC_MODE): This bit sets the color space conversion mode for the overlay plane. When bit 4 (CSC_BYPASS) is set, this bit is ignored in the CSC logic since it is in bypass mode. 0 = ITU-R Recommendation BT.601 1 = ITU-R Recommendation BT.709
4	YUV2RGB Conversion Bypass (CSC_BYPASS): This bit sets the overlay data path to bypass the YUV-to-RGB conversion logic. This bit should only be set when the overlay data is output to compliant external TV encoders in overlay native YUV444 format. The color adjustment controls, programmed through brightness, contrast and saturation registers, are still active. In CSC_BYPASS mode, software should also program the overlay gamma to a linear ramp, which is the gamma default values. The output YUV data from Overlay contains 10-bit unsigned Y value (possibly in the full 0-255 range) and 10-bit UV data in excess-512 format. The exact values depend on the mode of the Overlay Color Control Output. Converting to YCrCb (range readjustment) for 656 TV out is handled separately by the TV Out control logic. 0 = Enable the YUV to RGB Color Space Conversion Logic (default) 1 = Bypass (Disable) the YUV to RGB Color Space Conversion Logic



Bit	Description
3	Reserved Color Control Output Mode (CC_OUT): Sets the output pixel resolution of the Color Control Unit (including the piecewise-linear gamma correction logic) to be either 10-bit or 8-bit. Proper rounding is applied to the pixels. CC_OUT should be set to 1 to enable correct rounding to 8-bit if the backend (including the blending unit) operates in 8-bit mode. This mode takes affect whether gamma correction is enabled or not. 0 = 10-bit output. (default) 1 = 8-bit output.
2	Single Request Mode (Test mode) 0 – Single Request disabled 1- Single Request enabled
1	Reserved: MBZ (reserved for future overlay line buffer configuration)
0	Line Buffer Configuration (LINE_CONFIG): Sets the overlay line buffer configuration. Using the 2-line mode reduces the peak bandwidth demands of the overlay when down scaling. Configuring the cache for the 2-line mode increases the source line size that will fit in the line buffers. See the appendix for details of the cache configurations and the source size limits based on format and configuration. Other limits are based on available bandwidth. 0 = Two line buffers 1 = Three line buffers



3.2.15.5 OCOMD—Overlay Command Register

Memory Address Offset: 68h–6Bh (R/W)
 On-chip Reg. Mem Addr Offset: 30168h–3016Bh (RO; debug path)
 Default Value: 00h
 Access: see address offset above
 Size: 32 bits

This register and the Overlay Configuration register provide the basic programming options that the overlay engine needs to begin its work.

Bit	Description
31:20	Reserved: MBZ
19	Tiled Surface: This bit indicates that the overlay surface data is in tiled memory. The tile pitch is specified in bytes in the OSTRIDE register. Only X tiling is supported for overlay surfaces. When this bit is set, it affects the hardware interpretation of the OBUF and OSTART registers. 0 = Overlay surfaces use linear memory 1 = Overlay surfaces use X-tiled memory
18:17	Mirroring: Mirroring cause either a horizontal or vertical reversal in the displayed image. Note that when both horizontal and vertical mirroring is enabled it is a 180 degree rotation. Mirroring affects the buffer address values used. See buffer address description for details. 00 = Normal orientation 01 = Horizontal Mirroring 10 = Vertical Mirroring 11 = Both Horizontal and Vertical Mirroring (180 degree rotate)
16	Reserved: MBZ
15:14	Packed YUV 4:2:2 Byte Order (ORDER422): Affects the byte order for Packed YUV 4:2:2 data. This allows for the support of multiple YUV 4:2:2 formats. It must be set to zero for other data formats. 00 = Normal (in YUYV order with FOURCC code "YUY2") 01 = UV Swap (in YVYU order) 10 = Y Swap (in UYVY order with FOURCC code "UYVY") 11 = Y and UV swap (in VYUY order)

Bit	Description
13:10	FORMAT: Source Format. 0XXX = Reserved 1000 = Packed YUV 4:2:2 (Field ORDER422 in this register determines the byte order of this format) 1001 = Reserved 101X = Reserved 1100 = Planar YUV 4:2:0 (MPEG-1 or 2 based on UV initial phase) 1101 = Planar YUV 4:2:2 (e.g., for DCT-domain downsampled MPEG2 video) 1110 = Planar YUV 4:1:0 or YUV 4:1:1 planar 1111 = Reserved In all YUV formats, UV source values are in excess 128 format, in which value 128 stands for intensity of 0 for the color components.
9	Reserved: (
8	Reserved: MBZ (Enabling Autoflip (AUTOFLIP):)
7	Field-Synchronized Flip Enable (TVSYNCFIP_ENBL): This bit enables the overlay flip that is synchronized with the display field. 0 = Normal Overlay Flip (standard). The TVSYNCFIP_PARITY bit is ignored. 1 = Field-Synchronized Overlay Flip
6	Reserved: MBZ
5	Buffer Display/Flip Type (BUF_TYPE): This bit affects the buffer addressing used for buffer display and the use of the initial vertical phase. Frame mode starts addressing at the value contained in the buffer address register and increments/Decrements by stride as it advances from line to line. Initial phase selection is based on the buffer and the vertical Initial phase select bit. Field mode uses the Active Field bit (bit 1 of this register) and the reverse Y bit to determine if the start address should be the value in the start address register or the start address register plus/minus stride. Field mode will increment/decrement the address by two times the stride as it increments from line to line. Initial phase selection is based on the field and the vertical Initial phase select. 0 = Frame Mode (or called Non-Interleaved Buffer Mode) 1 = Field Mode (or called Interleaved Buffer Mode)
4	Test Mode: 0 = Normal mode 1 = Test mode



Bit	Description
3:2	Active Buffer Pointer (ACT_BUF): Selects which overlay buffer for display. This determines which buffer will be displayed when the overlay is enabled and the Ignore Buffer/Field bit (bit 4) of this register is cleared (to 0). It will otherwise be ignored and the internal buffer value (with the previously loaded value) will be used. The internal buffer value is readable through the status register (OC_BUF). 00 = Buffer 0 01 = Buffer 1 10 = Reserved 11 = Reserved
1	Active Field Select (ACT_F): Selects which field in an interleaved overlay buffer for display. This determines which field will be displayed when the overlay is enabled and the Ignore Buffer/Field bit (bit 4) was cleared (to 0). It will otherwise be ignored and the internal field value (with the previously loaded value) will be used. The internal buffer value is readable through the status register (OC_FIELD). This bit is ignored if Buffer Display/Flip Type is in Frame Mode (non-Interleaved Buffer mode). 0 = Field 0 1 = Field 1
0	Overlay Enable (OV_ENBL): Changing this bit from a Zero to a One will cause the overlay to begin display after the next qualified flip event. This can only be done in conjunction with a flip packet that enables the reconfiguration of the cache using the flush flags. A disable (from 1 to 0) will cause the overlay to stop displaying on this current display/overlay VBLANK. When Overlay is flipped from one display pipe to another, it will be automatically temporarily disabled to allow for the switch. When disabled, it will cause the overlay to enter a low power state. Overlay when enabled, automatically prevents the special C3 display fetch mode from being activated. There is an override for the enable of this plane in the Pipe Configuration register. Overlay can also be disabled from the pipe control register of the pipe that overlay is assigned to. Full on/off changes to the overlay require the shared render cache to be re-configured. This must be done through the command stream according to the packet definition. You can leave the cache intact while disabling the overlay (command register). You cannot enable the overlay without having previously configured the cache. Alviso/Grantsdale-G Erratum GRG04: This bit must be manually disabled by clearing to '0' when toggling the Overlay Pipe Select bit if BOTH display pipes are enabled. This erratum will be fixed on future products. There are two cases: 1) Overlay is disabled prior to enabling and switching to other pipe: a) Issue Overlay Flip ON request with Overlay Enable bit cleared to '0' and Overlay Pipe Select bit toggled b) Issue Overlay Flip CONTINUE request with Overlay Enable bit set to '1' 2) Overlay is enabled prior to switching to other pipe: a) Issue Overlay Flip CONTINUE request with Overlay Enable bit cleared to '0' and Overlay Pipe Select bit toggled b) Issue Overlay Flip CONTINUE request with Overlay Enable bit set to '1' Note: A wait-on-flip event is always required between back-to-back overlay flip requests. 0 = Overlay Plane Disable (No display or memory fetches) (default) 1 = Overlay Plane Enable (Requires cache to be configured first)



3.2.15.6 OSTART_0Y— Overlay Surface Y 0 Base Address Register

Memory Address Offset: 70h (R/W)
On-chip Reg. Mem Addr Offset: 30170h (RO; debug path)
Default Value: 00h
Access: see address offset above
Size: 32 bits

Bit	Description
31:0	Overlay Surface Y 0 Base Address: This address specifies the surface base address. When the surface is tiled, panning is specified using (x, y) offsets in the OBUF_0Y register. When the surface is in linear memory, panning is specified using a linear offset in the OBUF_0Y register. This address must be 4K aligned. It represents an offset from the graphics memory aperture base and is mapped to physical pages through the global GTT. These registers do not have a readback path through On-chip Reg.

3.2.15.7 OSTART_1Y— Overlay Surface Y 1 Base Address Register

Memory Address Offset: 74h (R/W)
On-chip Reg. Mem Addr Offset: 30174h (RO; debug path)
Default Value: 00h
Access: see address offset above
Size: 32 bits

Bit	Description
31:0	Overlay Surface Y 1 Base Address: This address specifies the surface base address. When the surface is tiled, panning is specified using (x, y) offsets in the OBUF_0Y register. When the surface is in linear memory, panning is specified using a linear offset in the OBUF_0Y register. This address must be 4K aligned. It represents an offset from the graphics memory aperture base and is mapped to physical pages through the global GTT. These registers do not have a readback path through On-chip Reg.

3.2.15.8 OSTART_0U— Overlay Surface U 0 Base Address Register

Memory Address Offset: 78h (R/W)
On-chip Reg. Mem Addr Offset: 30178h (RO; debug path)
Default Value: 00h
Access: see address offset above
Size: 32 bits

Bit	Description
31:0	Overlay Surface U 0 Base Address: This address specifies the surface base address. When the surface is tiled, panning is specified using (x, y) offsets in the OBUF_0Y register. When the surface is in linear memory, panning is specified using a linear offset in the OBUF_0Y register. This address must be 4K aligned. It represents an offset from the graphics memory aperture base and is mapped to physical pages through the global GTT. These registers do not have a readback path through On-chip Reg.



3.2.15.9 OSTART_0V— Overlay Surface V 0 Base Address Register

Memory Address Offset: 7Ch (R/W)
On-chip Reg. Mem Addr Offset: 3017Ch (RO; debug path)
Default Value: 00h
Access: see address offset above
Size: 32 bits

Bit	Description
31:0	Overlay Surface V 0 Base Address: This address specifies the surface base address. When the surface is tiled, panning is specified using (x, y) offsets in the OBUF_0Y register. When the surface is in linear memory, panning is specified using a linear offset in the OBUF_0Y register. This address must be 4K aligned. It represents an offset from the graphics memory aperture base and is mapped to physical pages through the global GTT. These registers do not have a readback path through On-chip Reg.

3.2.15.10 OSTART_1U— Overlay Surface U 1 Base Address Register

Memory Address Offset: 80h (R/W)
On-chip Reg. Mem Addr Offset: 30180h (RO; debug path)
Default Value: 00h
Access: see address offset above
Size: 32 bits

Bit	Description
31:0	Overlay Surface U 1 Base Address: This address specifies the surface base address. When the surface is tiled, panning is specified using (x, y) offsets in the OBUF_0Y register. When the surface is in linear memory, panning is specified using a linear offset in the OBUF_0Y register. This address must be 4K aligned. It represents an offset from the graphics memory aperture base and is mapped to physical pages through the global GTT. These registers do not have a readback path through On-chip Reg.

3.2.15.11 OSTART_1V— Overlay Surface V 1 Base Address Register

Memory Address Offset: 84h (R/W)
On-chip Reg. Mem Addr Offset: 30184h (RO; debug path)
Default Value: 00h
Access: see address offset above
Size: 32 bits

Bit	Description
31:0	Overlay Surface V 1 Base Address: This address specifies the surface base address. When the surface is tiled, panning is specified using (x, y) offsets in the OBUF_0Y register. When the surface is in linear memory, panning is specified using a linear offset in the OBUF_0Y register. This address must be 4K aligned. It represents an offset from the graphics memory aperture base and is mapped to physical pages through the global GTT. These registers do not have a readback path through On-chip Reg.



3.2.15.12 OTILEOFF_0Y— Overlay Surface Y 0 Tiled Offset Register

Memory Address Offset: 88h (R/W)
On-chip Reg. Mem Addr Offset: 30188h (RO; debug path)
Default Value: 00h
Access: see address offset above
Size: 32 bits

This register specifies the panning for the overlay surface. Bit 19 of OCOMD specifies whether the overlay surfaces are in linear or tiled memory. When the surface is in linear memory, the contents of this register are ignored. When the surface is tiled, the start position is specified in this register as an (x, y) offset from the beginning of the surface.

Bit	Description
31:28	Reserved. Write as zero
27:16	Source Start Y-Position. These 12 bits specify the vertical position in lines of the beginning of the overlay source window relative to the overlay surface. When mirroring vertically (Y backward), this field specifies the vertical position of the lower right corner relative to the end of the overlay data in the unmirrored orientation.
15:12	Reserved. Write as zero
11:0	Source Start X-Position. These 12 bits specify the horizontal offset in pixels of the beginning of the overlay source window relative to the overlay surface. When mirroring horizontally (X backward), this offset specifies the last pixel of the line of the overlay data in its unmirrored orientation.

3.2.15.13 OTILEOFF_1Y— Overlay Surface Y 1 Tiled Offset Register

Memory Address Offset: 8Ch (R/W)
On-chip Reg. Mem Addr Offset: 3018Ch (RO; debug path)
Default Value: 00h
Access: see address offset above
Size: 32 bits

This register specifies the panning for the overlay surface. Bit 19 of OCOMD specifies whether the overlay surfaces are in linear or tiled memory. When the surface is in linear memory, the contents of this register are ignored. When the surface is tiled, the start position is specified in this register as an (x, y) offset from the beginning of the surface.

Bit	Description
31:28	Reserved. Write as zero
27:16	Source Start Y-Position. These 12 bits specify the vertical position in lines of the beginning of the overlay source window relative to the overlay surface. When mirroring vertically (Y backward), this field specifies the vertical position of the lower right corner relative to the end of the overlay data in the unmirrored orientation.
15:12	Reserved. Write as zero
11:0	Source Start X-Position. These 12 bits specify the horizontal offset in pixels of the beginning of the overlay source window relative to the overlay surface. When mirroring horizontally (X backward), this offset specifies the last pixel of the line of the overlay data in its unmirrored orientation.



3.2.15.14 OTILEOFF_0U— Overlay Surface U 0 Tiled Offset Register

Memory Address Offset: 90h (R/W)
On-chip Reg. Mem Addr Offset: 30190h (RO; debug path)
Default Value: 00h
Access: see address offset above
Size: 32 bits

This register specifies the panning for the overlay surface. Bit 19 of OCOMD specifies whether the overlay surfaces are in linear or tiled memory. When the surface is in linear memory, the contents of this register are ignored. When the surface is tiled, the start position is specified in this register as an (x, y) offset from the beginning of the surface.

Bit	Description
31:28	Reserved. Write as zero
27:16	Source Start Y-Position. These 12 bits specify the vertical position in lines of the beginning of the overlay source window relative to the overlay surface. When mirroring vertically (Y backward), this field specifies the vertical position of the lower right corner relative to the end of the overlay data in the unmirrored orientation.
15:12	Reserved. Write as zero
11:0	Source Start X-Position. These 12 bits specify the horizontal offset in pixels of the beginning of the overlay source window relative to the overlay surface. When mirroring horizontally (X backward), this offset specifies the last pixel of the line of the overlay data in its unmirrored orientation.

3.2.15.15 OTILEOFF_0V— Overlay Surface V 0 Tiled Offset Register

Memory Address Offset: 94h (R/W)
On-chip Reg. Mem Addr Offset: 30194h (RO; debug path)
Default Value: 00h
Access: see address offset above
Size: 32 bits

This register specifies the panning for the overlay surface. Bit 19 of OCOMD specifies whether the overlay surfaces are in linear or tiled memory. When the surface is in linear memory, the contents of this register are ignored. When the surface is tiled, the start position is specified in this register as an (x, y) offset from the beginning of the surface.

Bit	Description
31:28	Reserved. Write as zero
27:16	Source Start Y-Position. These 12 bits specify the vertical position in lines of the beginning of the overlay source window relative to the overlay surface. When mirroring vertically (Y backward), this field specifies the vertical position of the lower right corner relative to the end of the overlay data in the unmirrored orientation.
15:12	Reserved. Write as zero
11:0	Source Start X-Position. These 12 bits specify the horizontal offset in pixels of the beginning of the overlay source window relative to the overlay surface. When mirroring horizontally (X backward), this offset specifies the last pixel of the line of the overlay data in its unmirrored orientation.



3.2.15.16 OTILEOFF_1U— Overlay Surface U 1 Tiled Offset Register

Memory Address Offset: 98h (R/W)
On-chip Reg. Mem Addr Offset: 30198h (RO; debug path)
Default Value: 00h
Access: see address offset above
Size: 32 bits

This register specifies the panning for the overlay surface. Bit 19 of OCOMD specifies whether the overlay surfaces are in linear or tiled memory. When the surface is in linear memory, the contents of this register are ignored. When the surface is tiled, the start position is specified in this register as an (x, y) offset from the beginning of the surface.

Bit	Description
31:28	Reserved. Write as zero
27:16	Source Start Y-Position. These 12 bits specify the vertical position in lines of the beginning of the overlay source window relative to the overlay surface. When mirroring vertically (Y backward), this field specifies the vertical position of the lower right corner relative to the end of the overlay data in the unmirrored orientation.
15:12	Reserved. Write as zero
11:0	Source Start X-Position. These 12 bits specify the horizontal offset in pixels of the beginning of the overlay source window relative to the overlay surface. When mirroring horizontally (X backward), this offset specifies the last pixel of the line of the overlay data in its unmirrored orientation.

3.2.15.17 OTILEOFF_1V— Overlay Surface V 1 Tiled Offset Register

Memory Address Offset: 9Ch (R/W)
On-chip Reg. Mem Addr Offset: 3019Ch (RO; debug path)
Default Value: 00h
Access: see address offset above
Size: 32 bits

This register specifies the panning for the overlay surface. Bit 19 of OCOMD specifies whether the overlay surfaces are in linear or tiled memory. When the surface is in linear memory, the contents of this register are ignored. When the surface is tiled, the start position is specified in this register as an (x, y) offset from the beginning of the surface.

Bit	Description
31:28	Reserved. Write as zero
27:16	Source Start Y-Position. These 12 bits specify the vertical position in lines of the beginning of the overlay source window relative to the overlay surface. When mirroring vertically (Y backward), this field specifies the vertical position of the lower right corner relative to the end of the overlay data in the unmirrored orientation.
15:12	Reserved. Write as zero
11:0	Source Start X-Position. These 12 bits specify the horizontal offset in pixels of the beginning of the overlay source window relative to the overlay surface. When mirroring horizontally (X backward), this offset specifies the last pixel of the line of the overlay data in its unmirrored orientation.



3.2.16 Overlay Scaling Registers

3.2.16.1 Reserved FASTHSCALE—Fast Horizontal Downscale Register

Memory Address Offset: A0h (R/W)
 On-chip Reg. Mem Addr Offset: 301A0h (RO; debug path)
 Default Value: 00h
 Access: see address offset above
 Size: 32 bits

Bit	Description
31:21	Reserved: MBZ
20:16	RESERVED (was YRGB_HSCALE): The fast scale factors must be set so that the output of the fast scalar has a luminance to chrominance ratio of either 2:1 or 1:1. The original source format and the two fast scale factors determine that ratio. 0000 = No fast horizontal scaling (only precision) NNNN = Down scale by 2^N (N=1-5 for 3-line Vertical filter, N=1-6 for 2-line Vertical filter)
15:5	Reserved: MBZ
4:0	RESERVED (was UV_HSCALE): 0000 = No fast horizontal scaling (only precision) NNNN = Down scale by 2^N (N=1-5 for 3-line Vertical filter, N=1-6 for 2-line Vertical filter)

3.2.16.2 UVSCALEV—UV Vertical Downscale Integer Register

Memory Address Offset: A4h (R/W)
 On-chip Reg. Mem Addr Offset: 301A4h (RO; debug path)
 Default Value: 00h
 Access: see address offset above
 Size: 32 bits

Bit	Description
31:27	Reserved: MBZ
26:16	Y Vertical Scale Integer (Y_VINT): This field is used for vertical downscale. It specifies the integer portion of the scale factor for packed data or the Y data in planar modes. A zero in this field indicates a vertical upscale operation.
15:11	Reserved: MBZ
10:0	UV Vertical Scale Integer (UV_VINT): Used only in YUV planar modes. This field is used only for vertical downscale. It specifies the integer portion of the scale factor for U/V data. The chrominance data may be subsampled. A zero value specifies a vertical upscale operation.



3.2.17 Overlay Polyphase Filter Coefficient Registers

The filter coefficient mantissa sizes are as follows:

Coefficient Type	Number x 17	Mantissa Size in Bits
Vertical Y Center Tap	1	8
Vertical Y Others	2	6
Vertical UV	3x2	6
Horizontal Y Center Tap	1	9
Horizontal Y Others	4	7
Horizontal UV Center Tap	1x2	9
Horizontal UV Others	2x2	7

Coefficients are stored in order of taps ordered right to left (for horizontal) and top to bottom (for vertical) followed by phases starting with phase zero. For products with three line mode, vertical coefficients for the 2-line mode should use zero for the center tap and phase and 1-phase for the top and bottom coefficients.

31	30	28	27	22/21/19	21/20/18	16
Sign	Exponent	Mantissa	Reserved			

15	16	15	14	13	8/7/6	7/6/5	0
Sign	Exponent	Mantissa	Reserved				



3.2.17.1 Y_VCOEFS—Overlay Y Vertical Filter Coefficient Registers

Memory Address Offset: 200h–2FFh (R/W)
 On-chip Reg. Mem Addr Offset: 30300h–303FFh (not readable)
 Default Value: 00h
 Access: see address offset above
 Size: 3 taps x 17 phases x 32/2 bits

3.2.17.2 Y_HCOEFS—Overlay Y Horizontal Filter Coefficient Registers

Memory Address Offset: 300h–4FFh (R/W)
 On-chip Reg. Mem Addr Offset: 30400h–305FFh (not readable)
 Default Value: 00h
 Access: see address offset above
 Size: 5 taps x 17 phases x 32/2 bits

3.2.17.3 UV_VCOEFS—Overlay UV Vertical Filter Coefficient Registers

Memory Address Offset: 500h–5FFh (R/W)
 On-chip Reg. Mem Addr Offset: 30600h–306FFh (not readable)
 Default Value: 00h
 Access: see address offset above
 Size: 3 taps x 17 phases x 32/2 bits

3.2.17.4 UV_HCOEFS—Overlay UV Horizontal Filter Coefficient Registers

Memory Address Offset: 600h–6FFh (R/W)
 On-chip Reg. Mem Addr Offset: 30700h–307FFh (RO; debug path)
 Default Value: 00h
 Access: see address offset above
 Size: 3 taps x 17 phases x 32/2 bits

Bit	Description
31	Sign: 0 = Positive value 1 = Negative value
30:28	Exponent: 000 = b.bbbbbb (1.6) 001 = .bbbbbbb (0.7) 010 = .0bbbbbbb (0.8) 011 = .00bbbbbbb (0.9) 1xx = Reserved
27:21	Mantissa: The mantissa varies in size based on which filter and which tap is being specified. Vertical filters use 6 bits except Y center tap is 8-bits Horizontal filters use 7-bits except Y/UV center taps are 9-bits
20:16	Reserved: MBZ



Bit	Description
15	Sign: 0 = Positive value 1 = Negative value
14:12	Exponent: This field determines the placement of the binary point for each coefficient. Using the value of 000, the coefficient has a maximum value of just less than two. 000 = 1.6 001 = 0.7 010 = 0.8 011 = 0.9 1xx = Reserved
11:6/5	Mantissa:
5/4:0	Reserved: MBZ

§§



4 TV-Out

The TV-Out register block is offset from the MMADR by 0x68000. The registers are broken out into several sections based on function. There are many reserved testmode bits in this register block, in general software should preserve the value of those bits (unless otherwise specified) as the enabling of those functions may take place at boot time.

Address Offset	Register Name
68000	TV out Control
68004	TV DAC Control / Status
68008	Reserved
6800C	Reserved
68010	Color Space Convert Y
68014	Color Space Convert Y2
68018	Color Space Convert U
6801C	Color Space Convert U2
68020	Color Space Convert V
68024	Color Space Convert V2
68028	Color Knobs
6802C	Color Level Control
68030	H Control 1 - Hsync Htotal
68034	H Control 2 - H Burst Control
68038	H Control 3 – Blanking
6803C	V Control 1 - NBR & VI end
68040	V Control 2 - Vsync Control
68044	V Control 3 - Equalization Control
68048	V Control 4 - v burst field 1
6804C	V Control 5 - v burst field 2
68050	V Control 6 - v burst field 3
68054	V Control 7 - v burst field 4
68058	Reserved
68060	SC Control 1 - enables, Burst level, SC DDA1
68064	SC Control 2 - SC DDA2
68068	SC Control 3 - SC DDA3



Address Offset	Register Name
6806C	Reserved
68070	Window Position
68074	Window Size
68078-6807C	Reserved
68080	Filter Control 1 - Mode and H frac
68084	Filter Control 2 - Vert scale
68088	Filter Control 3 - Vert Initial Phase
6808C	Cosine ROM
68090	CC Control
68094	CC Data Field 1
68098	CC Data Field 2
6809C-680AC	Reserved
680B0	WSS Control
680B4	WSS Data
680B8-680FC	Reserved
68100	H Filter Y Coefficients
68200	H Filter C Coefficients
68300	V Filter Y Coefficients
68400	V Filter C Coefficients



4.1 TV-Out Control Registers

4.1.1 TV_CTL—TV-Out Control

Address Offset: 68000–68003h
Default Value: 00000000h

This register contains controls for the basic functional modes of the TV-Out logic. When re-enabling the encoder after disabling, software should ensure that at least one vertical blank has occurred on the attached pipe before turning the encoder back on or turning pipe off. When operating in panel fitter mode the TV encoder is disabled and the filtered video is routed back to the ports assigned to the pipe selected in bit 30.

Bit	Description
31	ENC_ENABLE: Enables the encoder and associated logic 0 = encoder is off (default) 1 = encoder is on.
30	ENC_PIPESEL: Selects the pipe to which the encoder is attached 0 = pipe A 1 = pipe B
29:28	ENC_OUTPUT_MODE[1:0]: Selects analog output format, also configures the DAC output voltage levels 00 = Composite (default) uses DAC A only 01 = SVideo (Y data on DAC B, C data on DAC C) 10 = Component (Y : DAC B, Pb : DAC A, Pr : DAC C) 11 = Combo mode: [testmode only, should not be used in production] Composite (DAC A) & SVideo (DAC B&C)
27	Reserved
26:24	Reserved. Read as zero
23	RGB: Enables RGB output. Only used in SMPTE253. ENC_OUTPUT_MODE must be set to 10 for this bit to be valid. 0 = Normal sync (default) 1 = RGB (R: DAC C, G: DAC B, B: DAC A)
22	3CH_SYNC: Enables sync generation on all channels. Used in SMPTE253, can also be used in YPbPr 0 = Normal sync (default) 1 = 3 channel sync
21	TRILEVEL_SYNC: Enables trilevel sync generation 0 = Normal sync (default) 1 = Trilevel sync
20	SLOW_SYNC: Enables slow sync generation for NTSC & PAL. 0 = Normal sync (default) 1 = Slow sync for NTSC & PAL



Bit	Description
19:18	OVERSAMPLE[1:0]: Selects between oversampling modes 00 = 4x oversample (for 480/576p) 01 = 2x oversample (for 720p & 1080i) 10 = no oversampling (for 1080p testmode) 11 = 8x oversample (for all Standard Def interlaced)
17	PROG_INT: Selects between progressive and interlaced scanning 0 = interlaced (default) 1 = progressive
16	PAL_BURST: Enables the PAL switch for colorburst phase reversal. This bit must be set in all PAL modes except PAL/M 0 = NTSC burst (default) 1 = PAL burst
15	FIELD_SWAP: (Testmode) 0 = Normal 1 = Inverted Fields
14:12	YC_SKEW[2:0]: An adjustment for Luminance timing relative to chroma. Actual skew amount depends on the output clock. Amounts shown indicate the amount of delay applied to the Y component. 000 = Normal (in phase) 001 = +0.5 pixel (advanced) 010 = +1.0 pixel 011 = +1.5 pixel 100 = +2.0 pixel 101 = +2.5 pixel others = reserved
11	PF2_ENABLE: Enable Panel Fitter 2 Mode* (PF2) [DevCL] 0 = PF2 is off (default) 1 = PF2 is on. ENC_ENABLE (bit 31) must also be set to 0.
10:9	TP_SIZE: Adjusts testpattern size for HD testing 00 = 640x480 01 = 1280x720 10 = 1920x1080 11 = reserved
8:6	VGA_THRT[1:0]: (Testmode): VGA data throttling override. Software must preserve the state of these bits: 000 = 50% Throttling (default) 001 = 33% (clk*2/3) 010 = 66% (clk/3) 011 = 75% (clk*1/4) 100 = 80% (clk*1/5) 111 = Throttling disabled



Bit	Description
5:4	FUSE_STATE: (READ-ONLY). These two bits indicate the state of the TV-Out function fuses. 00 = All features enabled (default) 01 = Reserved. 1X = TV-Out function disabled. If TV is in the this mode writing to the TV-Out registers will have no effect on the TV-Out logic.
3:0	ENC_TEST_MODE[3:0]: These bits put the encoder/DACs into one of several test modes. Software must preserve the state of these bits 0000 = Normal operation (default) 0001 = Encoder Test Pattern 1 - combo pattern 0010 = Encoder Test Pattern 2 - full screen vertical 75% color bars 0011 = Encoder Test Pattern 3 - full screen horizontal 75% color bars 0100 = Encoder Test Pattern 4 - random noise 0101 = Encoder Test Pattern 5 - linear color ramps 0110 = Reserved [DevCL] . 0111 = Monitor Detect - All DACs at 50% of full scale 1000 = DAC Test 1: Full Scale Test - all 3 DACs at full scale output 1001 = DAC Test 2: Linearity Test 1 - triangle wave on all 3 DACs 1010 = DAC Test 3: High Noise Test - square wave on all 3 DACs 1011 = DAC Test 4: Cross Talk Test - DAC A&C square wave, DAC B full scale output 1100 = DAC Test 5: SNR/Distortion Test - sine wave on all 3 DACs 1101 = DAC Test 6: Linearity Test 2 - Triangle Waves on A, B inverted, C at zero level 1110 = DAC Test 7: Linearity Test 2 - Triangle Waves on A, C inverted, B at zero level 1111 = DAC Test 8: : Linearity Test 2 - Triangle Waves on C, B inverted, A at zero level

*Notes on using the filter as a panel fitter:

The filter will be attached to the pipe specified in bit 30 of this register. Only the filter control registers will have any affect, all other TV registers must be left in the default state. It is recommended to use the auto scaling mode (0x68080[31]=1). Scaling works the same as in the primary panel fitter, pipe source size is automatically scaled fit into the area defined by the TV_WIN_POS and TV_WIN_SIZE registers within the pipe's active area. The pipe must be off when enabling or disabling the scaler (enabling/disabling Panel Fitter 2 Mode), however while the scaler is enabled the pipe source size is allowed to be changed.



4.1.2 TV_DAC—TV DAC Control / Status

Address Offset: 68004–68007h
Default Value: U0000000h

This register contains controls for the basic functional modes of the TVDAC logic. It also contains status information for detecting the connection type. TV-Out must be enabled or set to monitor sense for the Sense logic to work.

Bit	Description
31	TVDAC_STATE_CHG [R/O] : Enables DAC state change detection logic. A sticky bit cleared by state change detection being disabled. 0 = unchanged since enabled 1 = changed since enabled
30	TVDAC_A_SENSE [R/O] : Indicates the voltage level of the TV DAC A is above threshold selected in TVDAC A SENSE CTL 0 = below threshold 1 = above threshold
29	TVDAC_B_SENSE [R/O] : Indicates the voltage level of the TV DAC B is above threshold selected in TVDAC B SENSE CTL 0 = below threshold 1 = above threshold
28	TVDAC_C_SENSE [R/O] : Indicates the voltage level of the TV DAC C is above threshold selected in TVDAC C SENSE CTL 0 = below threshold 1 = above threshold
27	TVDAC_STATE_CHG_EN : Enables DAC state change detection logic 0 = disabled 1 = enabled
26	TVDAC_A_SENSE_CTL : 0 = Low sense level (used for CVBS and Y signals) 1 = High sense level (used for C/Pr/Pb signals and monitor detect)
25	TVDAC_B_SENSE_CTL : 0 = Low sense level (used for CVBS and Y signals) 1 = High sense level (used for C/Pr/Pb signals and monitor detect)
24	TVDAC_C_SENSE_CTL : 0 = Low sense level (used for CVBS and Y signals) 1 = High sense level (used for C/Pr/Pb signals and monitor detect)
23:8	Reserved. Software must preserve the state of these bits



Bit	Description
7	DAC_CTL_OVERRIDE : DAC voltage level override values. This bit also overrides the state of ENC_ENABLE. 0 = use ENC_OUTPUT_MODE[1:0] to control the DACs 1 = use DAC_A_CTL, DAC_B_CTL, DAC_C_CTL, to control the voltage levels.
6	ENC_TVDAC_SLEW : [Testmode] . Selects the DAC slew rate. Software must preserve the state of this bit. 0 = slow 1 = fast
5:4	DAC_A_CTL[1:0] : DAC voltage level override values. 00 = 1.3v 01 = 1.1v 10 = 0.7v 11 = DAC off
3:2	DAC_B_CTL[1:0] : DAC voltage level override values. 00 = 1.3v 01 = 1.1v 10 = 0.7v 11 = DAC off
1:0	DAC_C_CTL[1:0] : DAC voltage level override values. 00 = 1.3v 01 = 1.1v 10 = 0.7v 11 = DAC off



4.2 Color Conversion and Control Registers

These 5 registers contain the coefficients of the color space converter and the controls for the knob registers (Brightness, Contrast, Saturation, Hue).

The color space conversion registers are double buffered and are updated on each vertical sync following a write to CSC V2.

Attenuation registers are used to preserve dynamic range during filtering. These registers are 10 bits in b.bbbbb... format and must not be programmed to values greater than 1. CSC coefficients must be normalized in order to use these registers.

Coefficients for the CSC are stored in exponent-mantissa format similar to the filter. Two CSC coefficients are stored in each dword, the tables below show the data packing in each of the registers. Three CSC coefficients use a special 12-bit format to have the ability to reach 1.0, these are GY, RV, and BU.

Bit	Description
10:9	Exponent bits: These bits are represented as 2^{-n} 00 = 1 or mantissa is 0.bbbbbbbbb 01 = 0.5 or mantissa is 0.0bbbbbbbb 10 = 0.25 or mantissa is 0.00bbbbbbbb 11 = 0.125 or mantissa is 0.000bbbbbbbb
8:0	Mantissa:

Bit	Description
11:9	Exponent bits: These bits are represented as 2^{-n} 111 = 2 or mantissa is b.bbbbbbbb (can only use for 1.0) 000 = 1 or mantissa is 0.bbbbbbbbb 001 = 0.5 or mantissa is 0.0bbbbbbbb 010 = 0.25 or mantissa is 0.00bbbbbbbb 011 = 0.125 or mantissa is 0.000bbbbbbbb others = reserved
8:0	Mantissa:



4.2.1 TV_CSC_Y—Color Space Convert Y

Address Offset: 68010–68013h
Default Value: 00000000h

Bit	Description
31:27	Reserved
26:16	RY[10:0] : CSC coefficient
15:12	Reserved
11:0	GY[11:0] : CSC coefficient

4.2.2 TV_CSC_Y2—Color Space Convert Y2

Address Offset: 68014--68017h
Default Value: 00000000h

Bit	Description
31:27	Reserved
26:16	BY[10:0] : CSC coefficient
15:10	Reserved
9:0	AY[9:0] : Luma attenuation. Represents a number from 0 - 1.0 in 1.9 fixed point format

4.2.3 TV_CSC_U—Color Space Convert U

Address Offset: 68018–6801Bh
Default Value: 00000000h

Bit	Description
31:27	Reserved
26:16	RU[10:0] : CSC coefficient
15:11	Reserved
10:0	GU[10:0] : CSC coefficient



4.2.4 TV_CSC_U2—Color Space Convert U2

Address Offset: 6801C–6801Fh

Bit	Description
31:28	Reserved
27:16	BU[11:0] : CSC coefficient
15:10	Reserved
9:0	AU[9:0] : U attenuation (component video only). Represents a number from 0 - 1.0 in 1.9 fixed point format

4.2.5 TV_CSC_V—Color Space Convert V

Address Offset: 68020–68023h
Default Value: 00000000h

Bit	Description
31:28	Reserved
27:16	RV[11:0] : CSC coefficient
15:11	Reserved
10:0	GV[10:0] : CSC coefficient

4.2.6 TV_CSC_V2—Color Space Convert V2

Address Offset: 68024–68027h
Default Value: 00000000h

Bit	Description
31:27	Reserved
26:16	BV[10:0] : CSC coefficient
15:10	Reserved
9:0	AV[9:0] : V attenuation (component video only). Represents a number from 0 - 1.0 in 1.9 fixed point format



4.2.7 TV_CLR_KNOBS—Color Knobs

Address Offset: 68028–6802Bh
Default Value: 00000000h

If output is in RGB the CON & SAT fields must be programmed to the same value.

Bit	Description
31:24	BRT[7:0] : brightness modifier (2's comp value)
23:16	CON[7:0] : contrast modifier (2.6 fixed point value) [DevCL] Maximum programmed value is limited to 3.0.
15:8	SAT[7:0] : saturation modifier (2.6 fixed point value)
7:0	HUE[7:0] : hue modifier. 8-bit value represents a phase angle from 0 to 360 degrees.

4.2.8 TV_CLR_LEVEL—Color Level Control

Address Offset: 6802C–6802Fh
Default Value: 00000000h

Bit	Description
31:25	Reserved
24:16	BLACK_LEVEL[8:0] : The value to put out on the DAC for black
15:9	Reserved
8:0	BLANK_LEVEL[8:0] : The value to put out on the DAC for blanking



4.3 Timing Registers

The timing registers should only be modified while the encoder is disabled.

4.3.1 TV_H_CTL_1—H Control 1

Address Offset: 68030–68033h
Default Value: 00000000h

Bit	Description
31:29	Reserved
28:16	HSYNC_END[12:0] : The pixel number in which to stop the horizontal sync. Measured from the leading edge of H sync. Register is programmed with the actual pixel value.
15:13	Reserved
12:0	HTOTAL[12:0] : The total number of pixels in a line including sync and blanking. Register is programmed with one less than the actual number of pixels.

4.3.2 TV_H_CTL_2—H Control 2

Address Offset: 68034–68037h
Default Value: 00000000h

Bit	Description
31	BURST_ENA : Enable colorburst generation. 0 = color burst is disabled (for b/w or component enabled) 1 = colorburst is enabled
30:29	Reserved
28:16	HBURST_START[12:0] : The pixel number in which to start the colorburst. Measured from the leading edge of H sync. Register is programmed with one less than the actual pixel value.
15:8	Reserved
7:0	HBURST_LEN[7:0] : The length in pixels of the colorburst.



4.3.3 TV_H_CTL_3—H Control 3

Address Offset: 68038–6803Bh
Default Value: 00000000h

Bit	Description
31:29	Reserved
28:16	HBLANK_END[12:0] : The pixel number in which to end horizontal blanking. Measured from the leading edge of H sync. Register is programmed with one less than the actual pixel value.
15:13	Reserved
12:0	HBLANK_START[12:0] : The pixel number in which to start horizontal blanking. Measured from the leading edge of H sync. Register is programmed with one less than the actual pixel value.

4.3.4 TV_V_CTL_1—V Control 1

Address Offset: 6803C–6803Fh
Default Value: 00000000h

Bit	Description
31:27	Reserved
26:16	NBR_END[10:0] : Number of lines in in one field of the non-blanked region in interlace (or 1 frame in progressive mode). Register is programmed with one less than the actual number of lines.
15:14	Reserved
13:8	VI_END_F1[5:0] : The number of lines in the vertical interval of field 1. Register is programmed with one less than the actual number of lines.
7:6	Reserved
5:0	VI_END_F2[5:0] : The number of lines in the vertical interval of field 2. Register is programmed with one less than the actual number of lines.



4.3.5 TV_V_CTL_2—V Control 2

Address Offset: 68040–68043h
Default Value: 00000000h

Bit	Description
31:23	Reserved
22:16	VSYNC_LEN[6:0] : The duration of vertical sync in half lines
15	Reserved
14:8	VSYNC_START_F1[6:0] : The half line number of field 1 to start vertical sync. Register is programmed with one less than the actual number of half lines.
7	Reserved
6:0	VSYNC_START_F2[6:0] : The half line number of field 2 to start vertical sync. Register is programmed with one less than the actual number of half lines.

4.3.6 TV_V_CTL_3—V Control 3

Address Offset: 68044–68047h
Default Value: 00000000h

Bit	Description
31	EQUAL_ENA : Enable equalization generation. 0 = equalization disabled (default) 1 = equalization enabled
30:23	Reserved
22:16	VEQ_LEN[6:0] : The duration of the equalization interval in half lines
15	Reserved
14:8	VEQ_START_F1[6:0] : The half line number of field 1 to start the equalization interval. Register is programmed with one less than the actual number of half lines.
7	Reserved
6:0	VEQ_START_F2[6:0] : The half line number of field 2 to start the equalization interval. Register is programmed with one less than the actual number of half lines.



4.3.7 TV_V_CTL_4—V Control 4

Address Offset: 68048–6804Bh
Default Value: 00000000h

Bit	Description
31:22	Reserved
21:16	VBURST_START_F1[5:0] : The line number of field 1 to start inserting color burst. Measured from the start of the Vertical interval. Register is programmed with one less than the actual number of lines.
15:9	Reserved
8:0	VBURST_END_F1[8:0] : The line number of field 1 to stop inserting color burst. Measured from the start of the NBR. Register is programmed with one less than the actual number of lines.

4.3.8 TV_V_CTL_5—V Control 5

Address Offset: 6804C–6804Fh
Default Value: 00000000h

Bit	Description
31:22	Reserved
21:16	VBURST_START_F2[5:0] : The line number of field 2 to start inserting color burst. Measured from the start of the Vertical interval. Register is programmed with one less than the actual number of lines.
15:9	Reserved
8:0	VBURST_END_F2[8:0] : The line number of field 2 to stop inserting color burst. Measured from the start of the NBR. Register is programmed with one less than the actual number of lines.



4.3.9 TV_V_CTL_6—V Control 6

Address Offset: 68050–68053h
Default Value: 00000000h

Bit	Description
31:22	Reserved
21:16	VBURST_START_F3[5:0]: The line number of field 3 to start inserting color burst. Measured from the start of the Vertical interval. Register is programmed with one less than the actual number of lines. (should be programmed to the same value as VBURST_START_F1 for NTSC mode)
15:9	Reserved
8:0	VBURST_END_F3[8:0]: The line number of field 3 to stop inserting color burst. Measured from the start of the NBR. Register is programmed with one less than the actual number of lines. (should be programmed to the same value as VBURST_END_F1 for NTSC mode)

4.3.10 TV_V_CTL_7—V Control 7

Address Offset: 68054–68057h
Default Value: 00000000h

Bit	Description
31:22	Reserved
21:16	VBURST_START_F4[5:0]: The line number of field 4 to start inserting color burst. Measured from the start of the Vertical interval. Register is programmed with one less than the actual number of lines. (should be programmed to the same value as VBURST_START_F2 for NTSC mode)
15:9	Reserved
8:0	VBURST_END_F4[8:0]: The line number of field 4 to stop inserting color burst. Measured from the start of the NBR. Register is programmed with one less than the actual number of lines. (should be programmed to the same value as VBURST_END_F2 for NTSC mode)



4.4 Subcarrier Control Registers

Subcarrier generation should be disabled if encoder is set to component mode. The subcarrier registers should only be modified while the encoder is disabled.

4.4.1 TV_SC_CTL_1—SC Control 1

Address Offset: 68060–68063h
Default Value: 00000000h

Bit	Description
31	SC_DDA1_EN : Enables the primary subcarrier phase generation DDA
30	SC_DDA2_EN : Enables the secondary subcarrier phase generation DDA
29	SC_DDA3_EN : Enables the third subcarrier phase generation DDA This bit must be set for proper 625 line PAL operation
28:26	Reserved
25:24	SC_RESET_MODE : Determines the reset frequency of the subcarrier DDAs 00 = reset every other field 01 = reset every fourth field (NTSC M & J) 10 = reset every 8 fields (all PAL) 11 = never reset (NTSC 4.43)
23:16	BURST_LEVEL[7:0] : Value indicates one half of the peak amplitude of the colorburst at the DAC.
15:12	Reserved
11:0	SCDDA1_INC[11:0] : The increment value of the primary subcarrier phase generation DDA

4.4.2 TV_SC_CTL_2—SC Control 2

Address Offset: 68064–68067h
Default Value: 00000000h

Bit	Description
31	Reserved
30:16	SCDDA2_SIZE[14:0] : The rollover point of the of the secondary subcarrier phase generation DDA
15	Reserved
14:0	SCDDA2_INC[14:0] : The increment value of the secondary subcarrier phase generation DDA (this cannot be larger than SCDDA2_SIZE)



4.4.3 TV_SC_CTL_3—SC Control 3

Address Offset: 68068–6806Bh
Default Value: 00000000h

Bit	Description
31:27	Reserved
26:16	SCDDA3_SIZE[10:0] : The rollover point of the of the third subcarrier phase generation DDA
15:10	Reserved
9:0	SCDDA3_INC[9:0] : The increment value of the third subcarrier phase generation DDA

4.5 Window Control Registers

These registers are double buffered and are updated on every vertical sync following a write to the window size register. Coordinates are determined with a value of (0,0) being the upper left corner.

4.5.1 TV_WIN_POS—Window Position Register

Address Offset: 68070–68073h
Default Value: 00000000h

Bit	Description
31:29	Reserved
28:16	XPOS[12:0] : The x coordinate (in pixels) of the upper left most pixel of the display window. Measured from the end of horizontal blank.
15:12	Reserved
11:0	YPOS[11:0] : The y coordinate (in lines) of the upper left most pixel of the display window. Measured from the end of the start of the Non-Blanked Region (or end of the vertical interval, whatever).



4.5.2 TV_WIN_SIZE—Window Size Register

Address Offset: 68074–68077h
Default Value: 00000000h

Bit	Description
31:29	Reserved
28:16	XSIZE[12:0] : The horizontal size in pixels of the desired video window
15:12	Reserved
11:0	YSIZE[11:0] : The vertical size in pixels of the desired video window. lsb must be zero for interlaced modes.

4.6 Filter Control Registers

These registers are double buffered and are updated on every vertical sync following a write to the window size register.

4.6.1 TV_FILTER_CTL_1—Filter Control Register 1

Address Offset: 68080–68083h
Default Value: 00000000h

Bit	Description
31	AUTO_SCALE_MODE: 0 = the scaler will use the scaling factors in rest of this register 1 = the scaler will calculate the scale factors automatically, selected fractions can be read back in the registers.
30	AUTO_SCALE_CALC: [TESTMODE] Read-Only This read-only bit will be set while the auto scale function is in progress. It indicates that the values read back from the rest of the filter control registers should be ignored.
29	V_FILTER_BYPASS: Bypass the Vertical Filter Allows sources greater than 1024 pixels wide 0 = Vertical Filter Enabled (default) 1 = Vertical Filter Bypassed
28	VADAPT : Adaptive Vertical Filter: Puts the vertical filter into adaptive mode 0 = Adaptive filtering disabled (default) 1 = Adaptive filtering enabled



Bit	Description
27:26	VADAPT_MODE : Adaptive Vertical Filter Mode: Selects adaptive mode operation 00 = Least Adaptive (Recommended) 01 = Moderately Adaptive 10 = Reserved 11 = Most Adaptive
25:24	Reserved
23	CHR_PREF: Chroma Prefilter enable 0 = Prefilter disabled 1 = Prefilter enabled
22	CHR_POSTF: Chroma Postfilter enable (tentative)
21	VERT3TAP: Force 3 tap vertical mode (Testmode) 0 = Autodetection of 3 tap usage 1 = Force 3 tap vertical scaling
20:14	Reserved
13:0	HSCALE_FRAC[13:0]: The fractional part of the horizontal scaling factor divided by the oversampling rate. HSCALE_FRAC must be less than 1. $\text{HSCALE\_FRAC} = \text{int}((\text{src width}-1)/((\text{oversample} \times \text{dest width})-1) \times 2^{14})$

4.6.2 TV_FILTER_CTL_2—Filter Control Register 2

Address Offset: 68084-68087h
 Default Value: 00000000h

This register can be considered a single 3.15 fixed point field or two fields as described below.

Bit	Description
31:18	Reserved
17:15	VSCALE_INT[2:0]: The integer part of the vertical scale factor. $\text{VSCALE\_INT} = \text{int}((\text{src height}-1)/((\text{interlace} \times \text{dest height})-1))$ where interlace = 1/2 in interlace modes, 1 in progressive modes
14:0	VSCALE_FRAC[14:0]: The fractional part of the vertical scale factor. $\text{VSCALE\_FRAC} = \text{int}((\text{src height}-1)/((\text{interlace} \times \text{dest height})-1) - \text{VSCALE\_INT}) \times 2^{15})$ where interlace = 1/2 in interlace modes, 1 in progressive modes



4.6.3 TV_FILTER_CTL_3—Filter Control Register 3

Address Offset: 68088–6808Bh
Default Value: 00000000h

This register can be considered a single 3.15 fixed point field or two fields as described below. For progressive scan modes this register should be programmed to all zeroes.

Bit	Description
31:18	Reserved
17:15	VSCALE_INT_IP[2:0] : The integer portion of the initial phase of the vertical scaler for the bottom field $\text{VSCALE\_INT\_IP} = \text{int}((\text{source height}-1)/(1/4 \times (\text{destination height}-1)))$
14:0	VSCALE_FRAC_IP[14:0] : The fractional portion of the initial phase of the vertical scaler for the bottom field $\text{VSCALE\_FRAC\_IP} = \text{int}((\text{source height}-1)/(1/4 \times (\text{destination height}-1) - \text{VSCALE\_INT\_IP}) * 2^{15}))$

4.6.4 SIN_ROM—Sin ROM

Address Offset: 6808C–6808Fh
Default Value: 00000000h

This register can be used to calculate a sine (or cosine), one quarterwave is stored here. The intent is to enable calculation of the filter coefficients when the FPU is not available (VBIOS). Cosine angle is written as 18 bits but cosine read on [16:6] as a 11-bit fixed point 1.10 value.

Bit	Description
31:18	Reserved
17:0	Cosine[17:0] : Write the angle, read the cosine 00000 = 0 degrees 10000 = 90 degrees 20000 = 180 degrees 30000 = 270 degrees



4.7 Closed Captioning Registers

The Closed Captioning logic is programmed by the following two registers. The first register enables the feature and controls its position. The second register is used to transmit the data.

Note: TV Out hotplug will falsely detect a hotplug change when closed captioning data is being transmitted. Software should disable or ignore any TV Out hotplug indication when closed captioning is being transmitted.

4.7.1 TV_CC_CTL—CC Control Register

Address Offset: 68090–68093h
Default Value: 00000000h

Bit	Description
31	CC_ENA: Closed Captioning Enable 0 = disabled 1 = enabled
30:28	Reserved
27	CC_FID: [alviso/calistoga only] Indicates the field ID of the field to send CC data. Usually programmed to 0.
26	Reserved
25:16	CC_HOFF[9:0]: The horizontal offset of the CC run in. Usually programmed to 135.
15:6	Reserved
5:0	CC_LINE[5:0]: The line number to display the CC data minus 1. Usually programmed to 21.



4.7.2 TV_CC_DATA1—CC Data Register Field 1

Address Offset: 68094–68097h
Default Value: 00000000h

Bit	Description
31	CC_F1_RDY: [read-only] CC engine ready 1 = data fifo full 0 = CC engine can accept data
30:15	Reserved
14:8	CC_F1_D2[6:0]: CC data word 2
7	Reserved
6:0	CC_F1_D1[6:0]: CC data word 1 (transmitted first)

4.7.3 TV_CC_DATA2—CC Data Register Field 2

Address Offset: 68098–6809Bh
Default Value: 00000000h

Bit	Description
31	CC_F2_RDY: [read-only] CC engine ready 1 = data fifo full 0 = CC engine can accept data
30:15	Reserved
14:8	CC_F2_D2[6:0]: CC data word 2
7	Reserved
6:0	CC_F2_D1[6:0]: CC data word 1 (transmitted first)



4.8 PAL Wide Screen Signaling (WSS) Registers

The WSS logic is programmed by the following two registers. The first register enables the feature and controls its position. The second register is the data to transmit.

IMPORTANT: Since WSS uses line 23 to transmit data, the active region has to be downsized by two lines and the vertical blanking has to be increased by two lines.

4.8.1 TV_WSS_CTL—WSS Control Register

Address Offset: 680B0–680B3h
Default Value: 00000000h

Bit	Description
31	WSS_ENA: WSS Enable 0 = disabled 1 = enabled
30	WSS_FIELD: Sets the field in which to display WSS data. For normal PAL this should be programmed to 1 (field 1) (Assumes that standard PAL programming provided in this document is used) 0 = Field 2 1 = Field 1
29:26	Reserved
25:16	WSS_HOFF[9:0]: The horizontal offset of the WSS run in. Usually programmed to 148.
15	Reserved
14:8	WSS_BIT[6:0]: The clock divider for WSS clock. Should be 27 for 5MHz bit clock.
7:6	Reserved
5:0	WSS_LINE[5:0]: The line number to display the WSS data minus 1. Usually programmed to 25 to display on line 23.

4.8.2 TV_WSS_DATA—WSS Data Register Field 1

Address Offset: 680B4–680B7h
Default Value: 00000000h

Bit	Description
31:14	Reserved
13:0	WSS_DATA [13:0]: WSS data word

4.9 Filter Coefficient Registers

Coefficients for the filters are stored in sign-exponent-mantissa format similar to the overlay. The number of mantissa bit varies base on the filter. There are three exponent bits but not all values are allowed, ranges are specified per filter. Two filter coefficients are stored in each DWord, the tables below show the data packing in each of the words. Unused bits are considered reserved and should be written zero. The default value of all coefficient registers is 00000000h.

15	14:12	11:0
sign bit	exponent bits	mantissa bits (left justified)

Bit	Description
15	Sign bit: 0 = positive 1 = negative
14:12	Exponent bits: These bits are represented as 2^{1-n} available range will vary depending on the requirements of the filter. 000 = 2 or mantissa is b.bbbbbbb... 001 = 1 or mantissa is 0.bbbbbbb... 010 = 0.5 or mantissa is 0.0bbbbbb... 011 = 0.25 or mantissa is 0.00bbbbbb... 100 = 0.125 or mantissa is 0.000bbbbbb... etc.
11:0	Mantissa: Size of the mantissa varies based on the filter, but the MSB of the mantissa is always bit 11.

Coefficients are packed in the horizontal coefficient registers as follows (with the letter representing the tap and the number representing the coefficient set):

Address	bits [31:16]	bits[15:0]
68x00	B0	A0
68x04	D0	C0
68x08	F0	E0
68x0C	A1	G0
68x10	C1	B1

etc....



4.9.1 TV_H_LUMA—H Filter Luma Coeffs

Address Offset: 68100–681EFh

17 phases of 7 taps will require 60 DWords
center coeff is 1.2.9
other coeffs are 1.2.7

4.9.2 TV_H_CHROMA—H Filter Chroma Coeffs

Address Offset: 68200–682EFh

17 phases of 7 taps will require 60 DWords
center coeff is 1.2.9
other coeffs are 1.2.7

Coefficients are packed in the vertical coefficient registers as follows (with the letter representing the tap and the number representing the coefficient set). When the vertical filter is in 3 line mode the three taps used are A, C & E, B & C must be programmed to zero in three line mode.

Address	bits [31:16]	bits[15:0]
68x00	B0	A0
68x04	D0	C0
68x08	A1	E0
68x0C	C1	B1
68x10	E1	D1

etc....

4.9.3 TV_V_LUMA—V Filter Luma Coeffs

Address Offset: 68300–683ABh

17 phases of 5 taps will require 43 DWords
center coeff is 1.2.8
other coeffs are 1.2.6

4.9.4 TV_V_CHROMA—V Filter Chroma Coeffs

Address Offset: 68400–684ABh

17 phases of 5 taps will require 43 DWords
center coeff is 1.2.8
other coeffs are 1.2.6



5 TV-Out Programming Guide

5.1 TV-Out Register Descriptions

The TV-Out register block is offset from the MMADR by 0x68000. The registers are divided into several sections based on function. There are many reserved testmode bits in this register block; in general software should preserve the value of those bits (unless otherwise specified) as the enabling of those functions may take place at boot time.

Address Offset	Register Name
68000	TV out Control
68004	TV DAC Control / Status
68008	Reserved
6800C	Reserved
68010	Color Space Convert Y
68014	Color Space Convert Y2
68018	Color Space Convert U
6801C	Color Space Convert U2
68020	Color Space Convert V
68024	Color Space Convert V2
68028	Color Knobs
6802C	Color Level Control
68030	H Control 1 - Hsync Htotal
68034	H Control 2 - H Burst Control
68038	H Control 3 – Blanking
6803C	V Control 1 - NBR & VI end
68040	V Control 2 - Vsync Control
68044	V Control 3 - Equalization Control
68048	V Control 4 - v burst field 1
6804C	V Control 5 - v burst field 2
68050	V Control 6 - v burst field 3
68054	V Control 7 - v burst field 4
68058	Reserved
68060	SC Control 1 - enables, Burst level, SC DDA1
68064	SC Control 2 - SC DDA2
68068	SC Control 3 - SC DDA3
6806C	Reserved



Address Offset	Register Name
68070	Window Position
68074	Window Size
68078-6807C	Reserved
68080	Filter Control 1 - Mode and H frac
68084	Filter Control 2 - Vert scale
68088	Filter Control 3 - Vert Initial Phase
6808C	Cosine ROM
68090	CC Control
68094	CC Data Field 1
68098	CC Data Field 2
68098-680FC	Reserved
680B0	WSS Control
680B4	WSS Data
68098-680FC	Reserved
68100-	H Filter Y Coefficients
68200-	H Filter C Coefficients
68300-	V Filter Y Coefficients
68400-	V Filter C Coefficients



5.2 TV-Out Programming

5.2.1 Television Standards

5.2.1.1 Timing tables

NTSC-M/J/4.43 and PAL-M use the 480i timing standard; all other PAL versions use 576i timing (except PAL-N as shown below). All DTV mode timings are only available with component video and as a result do not have burst programming. Values here represent the register programming (accounting for -1s). Note that 1x mode is tentatively supported, and does not support underscan.

Timing	480i/525 (Bt470)	576i/625 (Bt470)	576i/625 (PALN)	480p	576p
Vertical Refresh	59.94	50	50	59.94	50
root clock	13.5M	13.5M	13.5M	27M	27M
oversample clock	108M	108M	108M	108M	108M
oversampling	8x	8x	8x	4x	4x
h sync end	64	64	64	64	64
hblank end	124	142	128	122	139
hblank start	836	844	844	842	859
htotal	857	863	863	857†	863
prog/int	i	i	i	p	p
trilevel	0	0	0	0	0
vsync start f1	6	5	6	12	10
vsync start f2	7	6	7	12	10
vsync len	6	5	6	12	10
veq en	1	1	1	0	0
veq start f1	0	0	0	X	X
veq start f2	1	1	1	X	X
veq len	18	15	18	X	X
vi end f1	20	24	24	44	48
vi end f2	21	25	25	44	48
nbr end	240	286	286	479	575
burst ena	cvbs/sv	cvbs/sv	cvbs/sv	0	0
hburst start	72	73	73	X	X
hburst len	34	32	34	X	X



Timing	480i/525 (Bt470)	576i/625 (Bt470)	576i/625 (PALN)	480p	576p
Vertical Refresh	59.94	50	50	59.94	50
vburst start f1	9	8	8	X	X
vburst end f1	240	285	285	X	X
vburst start f2	10	8	8	X	X
vburst end f2	240	286	286	X	X
vburst start f3	9	9	9	X	X
vburst end f3	240	286	286	X	X
vburst start f4	10	9	9	X	X
vburst end f4	240	285	285	X	X

Timing	720p	720p	1080i	1080i
Vertical Refresh	60	50	60	50
root clock	74.25M	74.25M	74.25M	74.25M
oversample clock	148.5M	148.5M	148.5M	148.5M
oversampling	2x	2x	2x	2x
h sync end	80	80	88	88
hblank end	300	300	235	235
hblank start	1580	1580	2155	2155
htotal	1649*	1979	2199*	2639
prog/int	p	p	i	i
trilevel	1	1	1	1
vsync start f1	10	10	4	4
vsync start f2	10	10	5	5
vsync len	10	10	10	10
veq en	0	0	1	1
veq start f1	X	X	4	4
veq start f2	X	X	4	4
veq len	X	X	10	10
vi end f1	29	29	21	21
vi end f2	29	29	22	22
nbr end	719	719	539	539
burst ena	0	0	0	0



5.2.1.1.1 Notes about programming POS and SIZE registers when preserving aspect ratio is NOT important...

The default POS values are 0,0.

The maximum SIZE register programming is dependant on the size of the nonblanked region. The following formulae should be useful.

if interlaced:

$$Y\text{SIZE} = 2 * (\text{nbr\_end} + 1)$$

if progressive:

$$Y\text{SIZE} = \text{nbr\_end} + 1$$

in either case

$$X\text{SIZE} = (\text{hblank\_start} - \text{hblank\_end})$$

5.2.1.1.2 Notes about programming POS and SIZE registers when preserving aspect ratio is important...

The default POS values are 0,0.

The maximum SIZE register programming is dependant on the size of the nonblanked region. The following formulae should be useful.

if interlaced:

$$Y\text{SIZE} = 2 * (\text{nbr\_end} + 1)$$

if progressive:

$$Y\text{SIZE} = \text{nbr\_end} + 1$$

in either case

$$X\text{SIZE} = (\text{hblank\_start} - \text{hblank\_end})$$

In order to keep preserve aspect ratio it may be necessary to reduce one of the two values.



Example1: 4:3 source on NTSC(Bt470).

from above XSIZE = 712, YSIZE = 482.

if XSIZE is 712, NTSC pixel has 8:9 aspect ratio (at 13.5MHz) so in square pixels this is 633. so for 4:3 the YSIZE should be 475.

Active area is then 712x475.

to center this YPOS = (YSIZEmax - YSIZE)/2 = 3 (rounded)

***Note that the XSIZE of 633 square pixels represents the destination surface size and should be taken into account for purpose of calculating coefficients.

Example 2: 4:3 source on PAL B (Bt470)

from above XSIZE = 702, YSIZE = 576

PAL has 10:9 pixel aspect ratio (at 13.5 MHz) so in square pixels this is 780.

for 4:3 aspect this the YSIZE should be 576. so YPOS is zero.

5.2.1.2 Subcarrier Programming

For high definition modes the subcarrier generator is disabled.

	NTSC (M/J)	PAL(most)	PAL Nc	PAL M	NTSC-4.43
Oversample	8	8	8	8	8
DDA2 size	27456	27648	27648	27456	27456
DDA3 size	off	625	625	off	525
DDA1 inc	135	168	135	135	168
DDA2 inc	20800	4122	23578	16704	4093
DDA3 inc	off	67	134	off	310
SCreset	4	8	8	8	Never
PAL-Burst	0	1	1	1	0



5.2.1.3 Color Space Conversion / Blank and Black Level Programming

The table below gives the register programming for the color space converter and the blank, blank and burst level registers based on output format and standard. PAL-M uses the NTSC-M programming, PAL-N is the same only with a larger burst level.

Topic	NTSC-M	NTSC-M	NTSC-J	NTSC-J
	composite	s-video	composite	s-video
blank level	225	266	225	266
black level	267	316	225	266
burst level	113	133	113	133
RY	0.2990	0.2990	0.2990	0.2990
GY	0.5870	0.5870	0.5870	0.5870
BY	0.1140	0.1140	0.1140	0.1140
AY	0.5082	0.6006	0.5495	0.6494
RU	-0.0749	-0.0885	-0.0810	-0.0957
GU	-0.1471	-0.1738	-0.1590	-0.1879
BU	0.2220	0.2624	0.2400	0.2836
AU	1.0000	1.0000	1.0000	1.0000
RV	0.3125	0.3693	0.3378	0.3992
GV	-0.2616	-0.3092	-0.2829	-0.3343
BV	-0.0508	-0.0601	-0.0549	-0.0649
AV	1.0000	1.0000	1.0000	1.0000



Topic	PAL	PAL	PAL-M	PAL-M	PAL-N	PAL-N
	composite	s-video	composite	s-video	composite	s-video
blank level	237	280	225	266	225	266
black level	237	280	267	316	267	316
burst level	118	139	113	133	118	139
RY	0.2990	0.2990	0.2990	0.2990	0.2990	0.2990
GY	0.5870	0.5870	0.5870	0.5870	0.5870	0.5870
BY	0.1140	0.1140	0.1140	0.1140	0.1140	0.1140
AY	0.5379	0.6357	0.5082	0.6006	0.5082	0.6006
RU	-0.0793	-0.0937	-0.0749	-0.0885	-0.0749	-0.0885
GU	-0.1557	-0.1840	-0.1471	-0.1738	-0.1471	-0.1738
BU	0.2350	0.2777	0.2220	0.2624	0.2220	0.2624
AU	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
RV	0.3307	0.3908	0.3125	0.3693	0.3125	0.3693
GV	-0.2769	-0.3273	-0.2616	-0.3092	-0.2616	-0.3092
BV	-0.0538	-0.0636	-0.0508	-0.0601	-0.0508	-0.0601
AV	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000

Topic	480(576) i/p	480(576) i/p	HDTV	HDTV
	YPrPb	RGB	YPrPb	RGB
blank level	279	279	279	279
black level	279	279	279	279
burst level	0	0	0	0
RY	0.2990	0	0.2126	0
GY	0.5870	1.000	0.7152	1.000
BY	0.1140	0	0.0722	0
AY	0.6364	0.7000	0.6364	0.7000
RU	-0.1687	0	-0.1146	0
GU	-0.3313	0	-0.3854	0
BU	0.5000	1.000	0.5000	1.000
AU	1.0000	0.7000	1.0000	0.7000
RV	0.5000	1.000	0.5000	1.000
GV	-0.4187	0	-0.4542	0
BV	-0.0813	0	-0.0458	0
AV	1.0000	0.7000	1.0000	0.7000

RGB modes are used for SMPTE 253 applications.



5.2.2 Underscanning and Pixel Aspect Ratios

Underscanning is used to fit the computer display inside the television's bezel. All televisions use an overscanned image to get picture to appear centered. Overscanning hides the corners of the transmitted image behind the bezel. The maximum picture area for NTSC is 720x480 pixels, the actual number of visible pixels depends on the size of the bezel. To account for this the TV-Out design is equipped with a variable scaler and underscan position control. Underscanning affects filter selection, and good quality underscanning is achievable up to 10% in all supported modes. Some modes can support further underscan if required.

Another issue in this area is the fact that the NTSC pixels aren't square, they have an aspect ratio of 9:8 (w:h) which means they are a little narrower than they are tall. This must be accounted for when calculating scale factors or when using the autoscaler. For example if a 640 pixel line is displayed on the TV the destination line size should be $640 \times 9/8 = 720$ pixels. If scaling is already being performed (from 10x7 or such) the 9/8 factor must be included or the image will look squished.

5.2.3 Programming Filter Coefficients

Programming the filter coefficients is very similar to the programming of the overlay coefficients. It uses a very similar data format, and coeffs are stored in the same ordering. However the filters are much sharper for the TV horizontal filter due to signal bandwidth limits. The intent of the vertical filter is to reduce the flicker caused by interlacing the computer output.

There are five filters implemented in the design requiring four unique filter coefficient sets, 2 sets for vertical filtering and 2 sets for horizontal filtering.

Different sets of coefficients are needed for each standard and output mode.

The following sections provide a formula (in six variables) that will generate the coefficient set for each and a table of various modes and what values to use for the six variables. The 6 variables will be selected from a table based on the standard, output type, scale factor

5.2.4 Setting the Mode

In general, the TV-Out logic should be programmed before it is enabled. The following steps should be followed.

1. disable the pipe if it is running.
2. set display PLL to the required clock frequency (from the table)
3. make sure that DPO programming is acceptable according to the rules outlined below, change them (and the planes) if needed
4. enable the pipe
5. program all TV-Out registers, then enable TV-Out

Exiting the mode is the reverse of these steps.

There are no special restrictions on planes used with TV-Out. It is okay to use double wide pixel mode if needed due to low core frequency.



5.2.4.1 Hi-Res Mode Table

TV-Out Mode Support

Source Aspect	Source Format	Destination Format						
		4:3			16:9			
		480i	576i	480p	480p	720p	1080i	1080p
4:3	640x480	Y	Y	Y	P	P	P	P
	800x600	Y	Y	Y	P	P	P	P
	1024x768	Y	Y	Y	P	P	P	P
	1280x1024	Y	Y	Y	P	P	P	P
	1600x1200	N	N	Y	P	P	P	P
16:9	848x480	L	L	L	Y	Y	Y	Y
	1280x720	L	L	L	Y	Y	Y	Y
	1600x900	N	N	L	Y	Y	Y	Y
	1920x1080	N*	N*	L	Y	Y	Y	Y
14:10	1400x1050	L	L	L	P	P	P	P

Y = yes the mode is supported

N = not supported.

P = the mode is supported in pillarbox

L = the mode is supported in letterbox

576p support is the same as 480p.

* Can be supported in letter box if source is interlaced. This results in a quality drop.

5.2.4.2 Hi-Res Modes

If TV-Out is to be used in a non-mirroring scenario Intel suggests using the native screen size for the display mode as this will present the best possible quality image. For SDTV & 480p(4:3) this is 640x480, 848x480 for 480p(16:9), 1280x720 for 720p and 1920x1080 for 1080i or 1080p. Bear in mind pixel aspect ratios when programming the horizontal scaler and window size.

DPO timing programming remains the same for all source modes, only the PLL clock frequency is changed. PLL frequency selection is described the Timing Standards section earlier in this document.



5.2.4.3 3 Tap Mode

3 tap mode enables when:

- Vertical filter bypass is disabled (not bypass)

AND

- TV in 480p and above HD modes

AND

- 3 tap mode has been set by ONE of the following scenarios:
 - HW sets 3 tap enable automatically when horizontal source size > 1024.
 - SW sets Filter Control Register 1 bit 21 (VERT3TAP) to 1 (test mode only).

When 3 tap is enabled, 3 tap coefficients need to be loaded.

Interlaced mode settings:

Field Swap (FIELD_SWAP)	Pipe interlaced	TV interlaced (PROG_INT)	Vertical Filter Bypass (V_FILTER_BYPASS)	3 tap mode
Normal	off	on (set to 0)	Not bypass	On
Inverted	on	off (set to 1)	Bypass	Off

5.2.4.4 VGA Modes

If TV-Out is to be used with VGA, the VGA Mode Disable bit in the PLLA (or PLLB depending on which pipe VGA is assigned) control register must be set to bypass the VGA clocks. Centering is not required for these modes, the TV filter will properly scale the VGA image to the size indicated in the TV Window registers if auto scale is enabled.

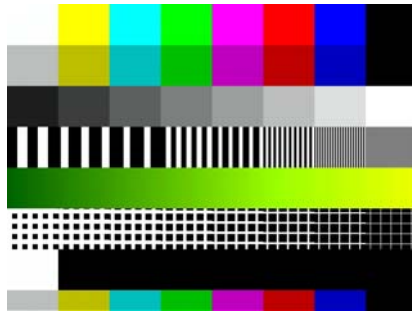
The VGA engine has tight restrictions on clocking in that the pixel clock is not allowed to exceed 45% of the core clock. The VGA throttling bits are used to ensure that the VGA does not exceed this limit.

5.2.4.5 Test Modes

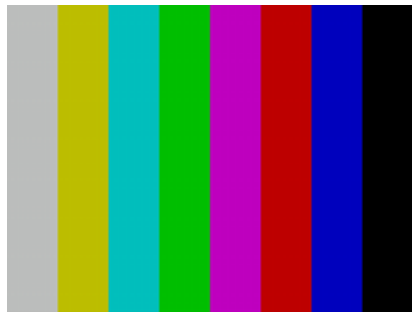
The TV-Out logic is equipped with several built in testing modes. They fall in to two categories: Video Quality Tests and DAC Quality Tests.

5.2.4.5.1 Video Quality Tests

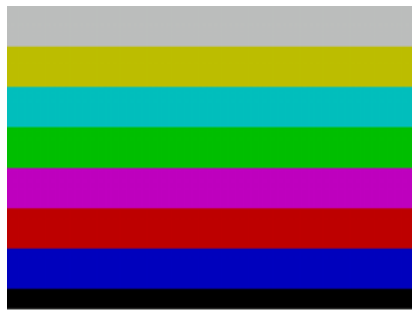
The video quality tests provide several reference images intended for use in testing the quality of the video encoder itself. They do not require the pipe or plane logic to be enabled to function. The encoder is programmed normally for the television standard, including PLL frequency. Then the encoder testmode bits are set to one of the four patterns. The first 3 images are shown below, pattern 4 is random noise which varies from frame to frame. Pattern 5 is intended for use is testing linearity.



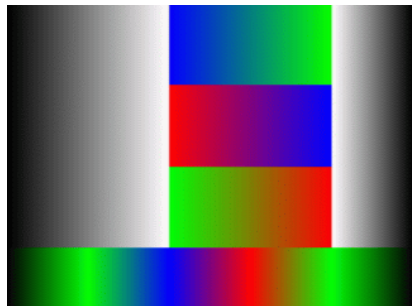
Test Image 1: Combo Pattern



Test Image 2: 75% color bars



Test Image 3: 75% horizontal bars



Test Image 5: Color Ramps



5.2.4.5.2 DAC Quality Tests

The DAC quality tests provided several reference patterns for measuring various quality standards of DAC performance. For these tests the subcarrier DDA is used as a to generate various patterns by using the sin lookup table or other function. To set the TV-Out logic for DAC testing DPLL A must be running, frequency is user selectable and is allowed to go as high as 150MHz. The subcarrier DDA1 increment is programmed to 1, DDA2 and DDA3 are disabled. The SC reset value should be set to 3 (never reset). The test modes are pretty self explanatory, check the register description for details of each test.

5.2.5 Detection and Determination of the Load

The monitor detection logic can work in two ways. A software driven polling method or and interrupt driven method. The interrupt method is the preferred method since it uses the least amount of power. The frequency of the polling in the software method will determine the power drain in that mode. In either detection mode determination of the load type is done in the same way. Alternatively, the lowest power option is to add a TV Detect button to the displays control panel (or system tray item) to force a polling mode detection loop.

5.2.5.1 Polling Based Load Connect Detection

In this mode the DACs are powered down completely, until software wishes to check for a connection change. The frequency at which this is performed is under SW control and directly affects the power consumption. The DACs are enabled in 0.7v mode. The pipe to which the TV-Out logic is assigned must have its PLL running. The encoder enable bit needs to be set to off. TESTMODE is set to monitor detect and the sense values are set to the high state. SW will need to wait for the PLL & DACs to come on, which may take some time. At this point the sense state bits are read, and if any of the bits are zero (indicating voltage level below threshold) then a load has been detected. After detection of a load the TESTMODE should immediately be set to normal. The PLL can be shut off.

5.2.5.2 Interrupt Based Load Connect Detection (with hotplug timer) (Not on [DevCL])

This feature is not on [DevCL].

The pipe PLL is turned on as in the polling mode. The software sets TEST MODE to 0. The threshold sense values are set to the high state. Enable state change detection. Finally, the TVDAC state change interrupt enable bit is set. The PLL can now be turned off. At this point any change in the sense state bits will indicate a change in the load and will generate an interrupt.



5.2.5.3 Interrupt Based Load Connect Detection (without hotplug timer)

This is not recommended since this leaves DACs enabled and drawing power, but is supplied here in case the detection mode can be changed when the power supply is connected.

In this mode two of the TVDACs must remain enabled in order to detect the change in the load (usually only DAC A&B). Using the TVDAC override controls the DACs are enabled in 0.7v mode. The pipe PLL is turned on as in the polling mode. Software places the DACs in monitor detect mode by entering TESTMODE 7. The threshold sense values are set to the high state. Enable state change detection. Finally the TVDAC state change interrupt enable bit is set. The PLL can now be turned off. At this point any change in the sense state bits will indicate a change in the load and will generate an interrupt.

5.2.5.4 Load Type Determination

The type of load can be determined after the Load Detection phase by the position of zeroes in the sense state bits. Combo mode is not detectable and also not recommended. This assumes the use of a standard pinout to the connector as recommended below.

A	B	C	Mode Detected
0	1	1	Composite
1	0	X	Svideo
0	0	0	Component

5.2.5.5 Disconnect Detection

This can be performed with the interrupt mode or the polling mode. The TV-Out logic is programmed to its normal mode. In either polling or interrupt case the channel(s) which contains either CVBS or Y data have their threshold sense levels set to low. Then the TVDAC state change enable bit can be set, and software can wait for an interrupt, or the software can periodically poll the sense state bits watching for a change. The bits are updated during vertical blank.

5.2.5.6 Alternate methods

It is also possible to remap the DAC outputs to different connector pins. In this way it would be possible to leave just one DAC on for interrupt mode detection. This is still a power drain, and would also lose connectivity with the dongle recommendation in the next section.



5.3 PCB Connector and Dongle Design

This dongle design is intended to give the best support for most of the users and conforms to the default configurations outlined in this specification. Two connector types are used, a 7 pin miniDIN connector and the RCA Phono jack. The miniDIN is expected to be used on the motherboard and is the source of all signals.

This connector is pin compatible with the standard Svideo connector (though 3 pins are not used) and so using svideo cables require no dongle at all. The two possible dongles are used to change the connector type to composite (1 connector) and component (3 connector), both of these connections rely on RCA type jacks for their connections.

Below is a diagram of the 7 pin miniDIN connector pinout, looking into the female connector which is a right angle connection mounted on the PCB. The shield is usually connected to chassis ground, (which is usually connected to PCB ground though a ferrite bead), but in either dongle the chassis ground connection is not used.

```

      _____
      / 4 7 3 \
      | 2 6 5 1 |
      \  key  /

```

The dongles described below assume the following connections between the encoder and the connector. Optionally, regular GND can be used for the DAC RTN signals, if that level of signal degradation is acceptable.

miniDIN	PCB
1	DAC B RTN
2	DAC C RTN
3	DAC B OUT
4	DAC C OUT
5	DAC A OUT
6	DAC A RTN
7	nc
shield	CHAS

The RCA connection is a simple 1 wire shielded connection with the shield connected to PCB ground (not chassis GND) and the signal carried on the center connection. For our purposes, the center connection will be pin 1 and the shield will be pin 2.

The composite dongle is simple, it has one male 7 pin miniDIN and one male RCA (usually yellow). The wire used should be 75ohm coaxial cable, the thin kind is okay. The cable can also be built with a female RCA so that users can extend as needed, this way the dongle can be very short, as small as 3-6".



miniDIN	RCA
1	nc
2	nc
3	nc
4	nc
5	1
6	2
7	nc
shield	nc

The component dongle is a little more complicated. It has one male 7 pin miniDIN and three male RCAs (usually red, green and blue). Again the wire used should be 75ohm coaxial cable. The cable can also be built with female RCAs so that users can extend as needed, this way the dongle can be very short, as small as 3-6".

miniDIN	BLUE RCA	GREEN RCA	RED RCA
1	nc	2	nc
2	nc	nc	2
3	nc	1	nc
4	nc	nc	1
5	1	nc	nc
6	2	nc	nc
7	nc	nc	nc
shield	nc	nc	nc

It should be noted that the component video dongle can be used for composite connection by using the blue RCA only and letting the green and red dongles dangle.

Cables that were distributed with the Alviso A-0 Silicon had the Red and Blue RCA's swapped. If you are using one of these cables the Composite signal will be on the red RCA. For component the red RCA has Pb and the blue RCA has Pr.



6 **VGA and Extended VGA Registers** **(00000h–00FFFh)**

This section describes the registers and the functional operation notations for the observable registers in the VGA section. This functionality is provided as a means for support of legacy applications and operating systems. It is important to note that these registers will in general have the desired effects only when running VGA display modes.

The main exceptions to this are the palette interface which will allow real mode DOS applications and full screen VGA applications under an OS control running in high resolution (non-VGA) modes to access the palette through the VGA register mechanisms and the use of the ST01 status bits that determine when the VGA enters display enable and sync periods. Other exceptions include the register bits that control the memory accesses through the A000:0000 and B000:0000 memory segments which would get used during operating system emulation of VGA for “DOS box” applications. Some of the functions of the VGA are enabled or defeated through the programming of the VGA control register bits that which are located in the MMIO register space.

Given the legacy nature of this function, it has been adapted to the changing environment that it must operate within. The three most notable changes were the addition of high resolution display mode support, new operating system support, and the use of fixed resolution display devices (such as LCD panels). Additional control bits in the PCI Config space will affect the ability to access the registers and memory aperture associated with VGA.

Mode of Operation	VGA Disable	VGA Display	VGA Registers	Palette (VGA)	VGA Memory	VGA Banking
VGA DOS	No	Yes	Yes	Yes	Yes	No
HiRes DOS	Yes	No	Yes	Yes	No	Yes
FS DOS	Yes/No	No/Yes	Yes	Yes	Yes	Yes
DOS Emulation	Yes	No	Yes	Yes	Yes	Yes



VGA Display Mode	Dot Clock Select	Dot Clock Range	Can use DVO Stall	132 Column Text Support	9-Dot Disable Support	Main Use
Native	VGA Clock Select	25/28 MHz	No	No	No	Analog CRT (VGA connector)
Centered	Fixed at display Requirements	Product Specific	No	No	Yes	DVI connector
Upper left corner	Fixed at display Requirements	Product Specific	Yes	No	Yes	Internal Panel

Even in the native VGA display operational modes, not all combinations of bit settings result in functional operating modes. VGA display modes have the restriction that they can be used only when all other display planes are disabled except for the Cursor A when used as a popup over VGA (in devices that support that function).

In most cases, these registers can be accessed via either I/O space or memory space. The I/O space resides in the PCI compatibility hole and uses only the addresses that were part of the original VGA I/O space (which includes EGA and MDA emulation). Accesses to the VGA I/O addresses are steered to the proper bus and rely on proper setup of bridge registers. Extended VGA registers such as GR10 and GR11 use additional indexes for the already defined I/O addresses. VGA register accesses are allowed as 8 or 16-bit naturally aligned transactions only. Word transactions must have the lsb of the address set to zero. DWORD I/O operations should not be performed on these registers, **DWORD Writes are IGNORED. Reads always returns 0xFFFFFFFFh.**

The memory space addresses listed are offsets from the base memory address programmed into the MMAPA register (PCI configuration offset 14h). For each register, the memory mapped address offset is the same address value as the I/O address. Several registers (GR10 and GR11) should never be accessed through the memory mapped aperture.



6.1 General Control and Status Registers

The setup, enable, and general registers are all directly accessible by the CPU. A sub indexing scheme is not used to read from and write to these registers.

Name	Function	Read		Write	
		I/O	Memory Offset	I/O	Memory Offset
ST00	VGA Input Status Register 0	3C2h	3C2h	—	—
ST01	VGA Input Status Register 1	3BAh/3DAh ¹	3BAh/3DAh ¹	—	—
FCR	VGA Feature Control Register	3CAh	3CAh	3BAh/3DAh ¹	3BAh/3DAh ¹
MSR	VGA Miscellaneous Output Register	3CCh	3CCh	3C2h	3C2h

Note:

1. The address selection for ST01 reads and FCR writes is dependent on CGA or MDA emulation mode as selected via the MSR register.

Various bits in these registers provide control over and the real-time status of the horizontal sync signal, the horizontal retrace interval, the vertical sync signal, and the vertical retrace interval. The horizontal retrace interval is the period during the drawing of each scan line containing active video data, when the active video data is not being displayed. This period includes the horizontal front and back porches, and the horizontal sync pulse. The horizontal retrace interval is always longer than the horizontal sync pulse. The vertical retrace interval is the period during which the scan lines not containing active video data are drawn. This includes the vertical front porch, back porch, and the vertical sync pulse. The vertical retrace interval is normally longer than the vertical sync pulse.

Display Enable is a status bit (bit 0) in VGA Input Status Register 1 that indicates when either a horizontal retrace interval or a vertical retrace interval is taking place. In the IBM* EGA graphics system (and the ones that preceded it, including MDA and CGA), it was important to check the status of this bit to ensure that one or the other retrace intervals was taking place before reading from or writing to the frame buffer. In these earlier systems, reading from or writing to frame buffer at times outside the retrace intervals meant that the CRT controller would be denied access to the frame buffer in while accessing pixel data needed to draw pixels on the display. This resulted in either “snow” or a flickering display.



6.1.1 ST00—Input Status 0

I/O (and Memory Offset) Address: 3C2h
 Default: 00h
 Attributes: Read-Only

7	6	5	4	3	0
CRT Interrupt	Reserved (00)		RGB Cmp / Sen	Reserved (0000)	

Bit	Descriptions
7	CRT Interrupt Pending. This bit is here for EGA compatibility and will always return zero. Note that the generation of interrupts was originally enabled, through bits [4,5] of the Vertical Retrace End Register (CR11). This ability to generate interrupts at the start of the vertical retrace interval is a feature that is typically unused by DOS software and therefore is only supported through other means for use under a operating system support. 0 = CRT (vertical retrace interval) interrupt is not pending. 1 = CRT (vertical retrace interval) interrupt is pending
6:5	Reserved. Read as 0s.
4	RGB Comparator / Sense. This bit returns the state of the output of the RGB output comparator(s). Video BIOS uses this bit during POST to determine whether the display is connected and if it is a color or monochrome CRT. BIOS blanks the screen or clears the frame buffer to display only black. Next, BIOS outputs a ramp to the D-to-A converters to test for the presence of a color display by determining which code cause the comparator to switch. Finally, if the BIOS does not detect any termination resistors on Red or Blue, it tests for the presence of a display using the Green signal. The result of each such test is read via this bit. 0 = Below threshold 1 = Above threshold
3:0	Reserved. Read as 0s.



6.1.2 ST01—Input Status 1

I/O (and Memory Offset) Address: 3BAh/3DAh

Default: 00h

Attributes: Read-Only

The address selection is dependent on CGA or MDA emulation mode as selected via the MSR register.

7	6	5	4	3	2	1	0
Reserved (0)	Reserved (0)	Video Feedback	Vertical Retrace	Reserved (00)	Display Enable		

Bit	Descriptions
7	Reserved (as per VGA specification). Read as 0s.
6	Reserved. Read as 0.
5:4	Video Feedback 1, 0. These are diagnostic video bits that are selected by the Color Plane Enable Register. These bits that are programmably connected to 2 of the 8 color bits sent to the palette. Bits 4 and 5 of the Color Plane Enable Register (AR12) selects which two of the 8 possible color bits become connected to these 2 bits of this register. The current software normally does not use these 2 bits. They exist for EGA compatibility.
3	Vertical Retrace / Video. 0 = VSYNC inactive (Indicates that a vertical retrace interval is not taking place). 1 = VSYNC active (Indicates that a vertical retrace interval is taking place). Note: VGA pixel generation is not locked to the display output but is loosely coupled. A VSYNC indication may not occur during the actual VSYNC going to the display but during the VSYNC that is generated as part of the VGA pixel generation. The exact relationship will vary with the VGA display operational mode. This status bit will remain active when the VGA is disabled and the device is running in high resolution modes (non-VGA) to allow for applications that (now it is incorrect) use these status registers bits. In this case, the status will come from the pipe that the VGA is assigned to. Bits 4 and 5 of the Vertical Retrace End Register (CR11) previously could program this bit to generate an interrupt at the start of the vertical retrace interval. This ability to generate interrupts at the start of the vertical retrace interval is a feature that is largely unused by legacy software. Interrupts are not supported through the VGA register bits.
2:1	Reserved. Read as 0s.



Bit	Descriptions
0	Display Enable Output. Display Enable is a status bit (bit 0) in VGA Input Status Register 1 that indicates when either a horizontal retrace interval or a vertical retrace interval is taking place. This was used with the IBM* EGA graphics system (and the ones that preceded it, including MDA and CGA). In those cases, it was important to check the status of this bit to ensure that one or the other retrace intervals was taking place before reading from or writing to the frame buffer. In these earlier systems, reading from or writing to frame buffer at times outside the retrace intervals meant that the CRT controller would be denied access to the frame buffer. This resulted in either “snow” or a flickering display. This bit provides compatibility with software designed for those early graphics controllers. This bit is currently used in DOS applications that access the palette to prevent the sparkle associated with read and write accesses to the palette ram with the same address on the same clock cycle. Note: This status bit will remain active when the VGA display is disabled and the device is running in high resolution modes (non-VGA) to allow for applications that (now considered incorrect) use these status registers bits. In this case, the status will come from the pipe that the VGA is assigned to. When in panel fitting VGA or centered VGA operation, the meaning of these bits will not be consistent with native VGA timings. 0 = Active display data is being sent to the display. Neither a horizontal retrace interval nor a vertical retrace interval is currently taking place. 1 = Either a horizontal retrace interval (horizontal blanking) or a vertical retrace interval (vertical blanking) is currently taking place.

6.1.3 FCR—Feature Control

I/O (and Memory Offset) Address: 3BAh/3DAh— Write; 3CAh— Read

Default: 00h

Attributes: See Address above

The I/O address used for writes is dependent on CGA or MDA emulation mode as selected via the MSR register. In the original EGA, bits 0 and 1 were used as part of the feature connector interface. Feature connector is not supported in these devices and those bits will always read as zero.

7	4	3	2	0
Reserved (0000)		VSYNC Control	Reserved (000)	

Bit	Descriptions
7:4	Reserved. Read as 0.
3	VSYNC Control. This bit is provided for compatibility only and has no other function. Reads and writes to this bit have no effect other than to change the value of this bit. The previous definition of this bit selected the output on the VSYNC pin. 0 = Was used to set VSYNC output on the VSYNC pin (default). 1 = Was used to set the logical 'OR' of VSYNC and Display Enable output on the VSYNC pin. This capability was not typically very useful.
2:0	Reserved. Read as 0.



6.1.4 MSR—Miscellaneous Output

I/O (and Memory Offset) Address: 3C2h — Write; 3CCh— Read
 Default: 00h
 Attributes: See Address above

7	6	5	4	3	2	1	0
VSYNC Polarity	HSYNC Polarity	Page Select	Reserved (0)	Clock Select		Memory Enable	I/O Address

Bit	Descriptions
7	CRT VSYNC Polarity. This is a legacy function that is used in native VGA modes. For most cases, sync polarity will be controlled by the port control bits. The VGA settings can be optionally selected for compatibility with the original VGA when used in the VGA native mode. Sync polarity was used in VGA to signal the monitor how many lines of active display are being generated. 0 = Positive Polarity (default). 1 = Negative Polarity.
6	CRT HSYNC Polarity. This is a legacy function that is used in native VGA modes. For most cases, sync polarity will be controlled by the port control bits. The VGA settings can be optionally selected for compatibility with the original VGA when used in the VGA native mode. 0 = Positive Polarity (default). 1 = Negative Polarity
5	Page Select. In Odd/Even Memory Map Mode 1 (GR6), this bit selects the upper or lower 64 KB page in display memory for CPU access: 0 = Upper page (default) 1 = Lower page. Selects between two 64KB pages of frame buffer memory during standard VGA odd/even modes (modes 0h through 5h). Bit 1 of register GR06 can also program this bit in other modes. Note that this bit is would normally set to 1 by the software.
4	Reserved. Read as 0.
3:2	Clock Select. These bits can select the dot clock source for the CRT interface. The bits should be used to select the dot clock in standard native VGA modes only. When in the centering or upper left corner modes, these bits should be set to have no effect on the clock rate. The actual frequencies that these bits select, if they have any affect at all, is programmable through the DPLL registers that default to the standard values used for VGA. 00 = CLK0, 25.175 MHz (for standard VGA modes with 640 pixel (8-dot) horizontal resolution) (default) 01 = CLK1, 28.322 MHz. (for standard VGA modes with 720 pixel (9-dot) horizontal resolution) 10 = Was used to select an external clock (now unused) 11 = Reserved



Bit	Descriptions
1	A0000–BFFFFh Memory Access Enable. VGA Compatibility bit enables access to local video memory (frame buffer) at A0000–BFFFFh. When disabled, accesses to VGA memory are blocked in this region. This bit is independent of and does not block CPU access to the video linear frame buffer at other addresses. Note that it is typical for AGP chipsets to shadow this register to allow proper steering of memory accesses to the proper bus. 0 = Prevent CPU access to memory/registers/ROM through the A0000-BFFFF VGA memory aperture (default). 1 = Allow CPU access to memory/registers/ROM through the A0000-BFFFF VGA memory aperture. This memory must be mapped as UC by the CPU; see VGA Host Access Memory Munging in <i>Display and Overlay Functions</i>.
0	I/O Address Select. This bit selects 3Bxh or 3Dxh as the I/O address for the CRT Controller registers, the Feature Control Register (FCR), and Input Status Register 1 (ST01). Presently ignored (whole range is claimed), but will “ignore” 3Bx for color configuration or 3Dx for monochrome. Note that it is typical in AGP chipsets to shadow this bit and properly steer I/O cycles to the proper bus for operation where a MDA exists on another bus such as ISA. 0 = Select 3Bxh I/O address (MDA emulation) (default). 1 = Select 3Dxh I/O address (CGA emulation).

Note:

1. In standard VGA modes using the analog VGA connector, bits 7 and 6 indicate which of the three standard VGA vertical resolutions the standard VGA display should use. Extended modes, including those with a vertical resolution of 480 scan lines, may use a setting of 0 for both of these bits. Different connector standards and timing standards specify the proper use of sync polarity. This setting was “reserved” in the VGA standard.

Table 6-1. Analog CRT Display Sync Polarities

V	H	Display	Horizontal Frequency	Vertical Frequency
P	P	200 Line	15.7 KHz	60 Hz
N	P	350 Line	21.8 KHz	60 Hz
P	N	400 Line	31.5 KHz	70 Hz
N	N	480 Line	31.5 KHz	60 Hz



6.2 Sequencer Registers

The sequencer registers are accessed via either I/O space or Memory space. To access registers the VGA Sequencer Index register (SRX) at I/O address 3C4h (or memory address 3C4h) is written with the index of the desired register. Then the desired register is accessed through the data port for the sequencer registers at I/O address 3C5 (or memory address 3C5).

6.2.1 SRX—Sequencer Index

I/O (and Memory Offset) Address: 3C4h
Default: 00h
Attributes: Read/Write

7	3	2	0
Reserved (00000)			Sequencer Index

Bit	Descriptions
7:3	Reserved. Read as 0s.
2:0	Sequencer Index. This field contains a 3-bit Sequencer Index value used to access sequencer data registers at indices 0 through 7. Notes: SR02 is referred to in the VGA standard as the Map Mask Register. However, the word “map” is used with multiple meanings in the VGA standard and was, therefore, deemed too confusing; hence, the reason for calling it the Plane Mask Register. SR07 is a standard VGA register that was not documented by IBM. It is not a graphics controller extension.

6.2.2 SR00—Sequencer Reset

I/O (and Memory Offset) Address: 3C5h(Index=00h)
Default: 00h
Attributes: Read/Write

7	2	1	0
Reserved (000000)		Reserved	Reserved

Bit	Descriptions
7:2	Reserved. Read as 0.
1	Reserved. Reserved for VGA compatibility (was reset).
0	Reserved. Reserved for VGA compatibility. (was reset)



6.2.3 SR01—Clocking Mode

I/O (and Memory Offset) Address: 3C5h (Index=01h)
 Default: 00h
 Attributes: Read/Write

7	6	5	4	3	2	1	0
Reserved (00)		Screen Off	Shift 4	Dot Clock Divide	Shift Load	Reserved (0)	8/9 Dot Clocks

Bit	Descriptions
7:6	Reserved. Read as 0s.
5	Screen Off. 0 = Normal Operation (default). 1 = Disables video output (blanks the screen) and turns off display data fetches. Synchronization pulses to the display, however, are maintained. Setting this bit to 1 had been used as a way to more rapidly update and improve CPU access performance to the frame buffer during VGA modes. In non-VGA modes (VGA Disable=1), this bit has no effect. Before the VGA is disabled through the VGA control register, this bit should be set to stop the memory accesses from the display. Programming Notes: In order to disable the VGA plane, SW must first write SR01, bit 5 = 1. It then must wait for 30us then disable the plan via Bit 31, Reg. 71400. Failure to do so will cause random VGA and CPU lockups.
4	Shift 4. 0 = Load video shift registers every 1 or 2 character clocks (depending on bit 2 of this register) (default). 1 = Load shift registers every 4th character clock.
3	Dot Clock Divide. Setting this bit to 1 stretches doubles all horizontal timing periods that are specified in the VGA horizontal CRTC registers. This bit is used in standard VGA 40-column text modes to stretch timings to create horizontal resolutions of either 320 or 360 pixels (as opposed to 640 or 720 pixels, normally used in standard VGA 80-column text modes). The effect of this is that there will actually be twice the number of pixels sent to the display per line. 0 = Pixel clock is left unaltered (used for 640 (720) pixel modes); (default). 1 = Pixel clock divided by 2 (used for 320 (360) pixel modes).
2	Shift Load. Bit 4 of this register must be 0 for this bit to be effective. 0 = Load video data shift registers every character clock (default). 1 = Load video data shift registers every other character clock.
1	Reserved. Read as 0.



Bit	Descriptions
0	8 / 9 Dot Clocks. This bit determines whether a character clock is 8 or 9 dot clocks long if clock doubling is disabled and 16 or 18 clocks if it is. This also changes the interpretation of the pixel panning values (see chart). An additional control bit determines if this bit is to be ignored and 8-dot characters are to be used always. The 9-dot disable would be used when doubling the horizontal pixels on a 1280 wide display or non-doubling on a 640 wide display. Panning however will occur according to the expected outcome. 0 = 9 dot clocks (9 horizontal pixels) per character in text modes with a horizontal resolution of 720 pixels. 1 = 8 dot clocks (8 horizontal pixels) per character in text or graphics modes with a horizontal resolution of 640 pixels.

6.2.4 SR02—Plane / Map Mask

I/O (and Memory Offset) Address: 3C5h (Index=02h)
 Default: 00h
 Attributes: Read/Write

7	4	3	0
Reserved			Memory Planes Processor Write Access Enable.

Bit	Descriptions
7:4	Reserved. Read as 0s.
3:0	Memory Planes [3:0] Processor Write Access Enable. In both the Odd/Even Mode and the Chain 4 Mode, these bits still control access to the corresponding color plane. 0 = Disable. 1 = Enable. Note: This register is referred to in the VGA standard as the Map Mask Register. However, the word “map” is used with multiple meanings in the VGA standard and was, therefore, considered too confusing; hence, the reason for calling it the Plane Mask Register.



6.2.5 SR03—Character Font

I/O (and Memory Offset) Address: 3C5h (index=03h)

Default: 00h

Attributes: Read/Write

7	6	5	4	3	2	1	0
Reserved (00)		Char Map A Select (bit 0)	Char Map B Select (bit 0)	Character Map A Select (bits 2 and 1)		Character Map B Select (bits 2 and 1)	

Bit	Descriptions																											
7:6	Reserved. Read as 0s.																											
3:2,5	Character Map Select Bits for Character Map B. These three bits are used to select the character map (character generator tables) to be used as the secondary character set (font). Note that the numbering of the maps is not sequential. <table><tr><th>Bit [3:2, 5]</th><th>Map Number</th><th>Table Location</th></tr><tr><td>00,0</td><td>0</td><td>1st 8KB of plane 2 at offset 0 (default)</td></tr><tr><td>00,1</td><td>4</td><td>2nd 8KB of plane 2 at offset 8K</td></tr><tr><td>01,0</td><td>1</td><td>3rd 8KB of plane 2 at offset 16K</td></tr><tr><td>01,1</td><td>5</td><td>4th 8KB of plane 2 at offset 24K</td></tr><tr><td>10,0</td><td>2</td><td>5th 8KB of plane 2 at offset 32K</td></tr><tr><td>10,1</td><td>6</td><td>6th 8KB of plane 2 at offset 40K</td></tr><tr><td>11,0</td><td>3</td><td>7th 8KB of plane 2 at offset 48K</td></tr><tr><td>11,1</td><td>7</td><td>8th 8KB of plane 2 at offset 56K</td></tr></table>	Bit [3:2, 5]	Map Number	Table Location	00,0	0	1st 8KB of plane 2 at offset 0 (default)	00,1	4	2nd 8KB of plane 2 at offset 8K	01,0	1	3rd 8KB of plane 2 at offset 16K	01,1	5	4th 8KB of plane 2 at offset 24K	10,0	2	5th 8KB of plane 2 at offset 32K	10,1	6	6th 8KB of plane 2 at offset 40K	11,0	3	7th 8KB of plane 2 at offset 48K	11,1	7	8th 8KB of plane 2 at offset 56K
Bit [3:2, 5]	Map Number	Table Location																										
00,0	0	1st 8KB of plane 2 at offset 0 (default)																										
00,1	4	2nd 8KB of plane 2 at offset 8K																										
01,0	1	3rd 8KB of plane 2 at offset 16K																										
01,1	5	4th 8KB of plane 2 at offset 24K																										
10,0	2	5th 8KB of plane 2 at offset 32K																										
10,1	6	6th 8KB of plane 2 at offset 40K																										
11,0	3	7th 8KB of plane 2 at offset 48K																										
11,1	7	8th 8KB of plane 2 at offset 56K																										
1:0,4	Character Map Select Bits for Character Map A. These three bits are used to select the character map (character generator tables) to be used as the primary character set (font). Note that the numbering of the maps is not sequential. <table><tr><th>Bit [1:0,4]</th><th>Map Number</th><th>Table Location</th></tr><tr><td>0,00</td><td>0</td><td>1st 8KB of plane 2 at offset 0 (default)</td></tr><tr><td>0,01</td><td>4</td><td>2nd 8KB of plane 2 at offset 8K</td></tr><tr><td>0,10</td><td>1</td><td>3rd 8KB of plane 2 at offset 16K</td></tr><tr><td>0,11</td><td>5</td><td>4th 8KB of plane 2 at offset 24K</td></tr><tr><td>1,00</td><td>2</td><td>5th 8KB of plane 2 at offset 32K</td></tr><tr><td>1,01</td><td>6</td><td>6th 8KB of plane 2 at offset 40K</td></tr><tr><td>1,10</td><td>3</td><td>7th 8KB of plane 2 at offset 48K</td></tr><tr><td>1,11</td><td>7</td><td>8th 8KB of plane 2 at offset 56K</td></tr></table>	Bit [1:0,4]	Map Number	Table Location	0,00	0	1st 8KB of plane 2 at offset 0 (default)	0,01	4	2nd 8KB of plane 2 at offset 8K	0,10	1	3rd 8KB of plane 2 at offset 16K	0,11	5	4th 8KB of plane 2 at offset 24K	1,00	2	5th 8KB of plane 2 at offset 32K	1,01	6	6th 8KB of plane 2 at offset 40K	1,10	3	7th 8KB of plane 2 at offset 48K	1,11	7	8th 8KB of plane 2 at offset 56K
Bit [1:0,4]	Map Number	Table Location																										
0,00	0	1st 8KB of plane 2 at offset 0 (default)																										
0,01	4	2nd 8KB of plane 2 at offset 8K																										
0,10	1	3rd 8KB of plane 2 at offset 16K																										
0,11	5	4th 8KB of plane 2 at offset 24K																										
1,00	2	5th 8KB of plane 2 at offset 32K																										
1,01	6	6th 8KB of plane 2 at offset 40K																										
1,10	3	7th 8KB of plane 2 at offset 48K																										
1,11	7	8th 8KB of plane 2 at offset 56K																										

NOTES:

1. In text modes, bit 3 of the video data's attribute byte normally controls the foreground intensity. This bit may be redefined to control switching between character sets. This latter function is enabled whenever there is a difference in the values of the Character Font Select A and the Character Font Select B bits. If the two values are the same, the character select function is disabled and attribute bit 3 controls the foreground intensity.
2. Bit 1 of the Memory Mode Register (SR04) must be set to 1 for the character font select function of this register to be active. Otherwise, only character maps 0 and 4 are available.



6.2.6 SR04—Memory Mode Register

I/O (and Memory Offset) Address: 3C5h (index=04h)
 Default: 00h
 Attributes: Read/Write

7	4	3	2	1	0
Reserved (0000)		Chain 4	Odd/Even	Extended Memory	Reserved (0)

Bit	Description
7:4	Reserved. Read as 0.
3	Chain 4 Mode. The selections made by this bit affect both CPU Read and write accesses to the frame buffer. 0 = The manner in which the frame buffer memory is mapped is determined by the setting of bit 2 of this register (default). 1 = The frame buffer memory is mapped in such a way that the function of address bits 0 and 1 are altered so that they select planes 0 through 3. This setting is used in mode x13 to allow all four planes to be accessed via sequential addresses.
2	Odd/Even Mode. Bit 3 of this register must be set to 0 for this bit to be effective. The selections made by this bit affect only non-paged CPU accesses to the frame buffer through the VGA aperture. 0 = The frame buffer memory is mapped in such a way that the function of address bit 0 such that even addresses select planes 0 and 2 and odd addresses select planes 1 and 3 (default). 1 = Addresses sequentially access data within a bit map, and the choice of which map is accessed is made according to the value of the Plane Mask Register (SR02).
1	Extended Memory Enable. This bit must be set to 1 to enable the selection and use of character maps in plane 2 via the Character Map Select Register (SR03). 0 = Disable CPU accesses to more than the first 64KB of VGA standard memory (default). 1 = Enable CPU accesses to the rest of the 256KB total VGA memory beyond the first 64KB.
0	Reserved. Read as 0.



6.2.7 SR07—Horizontal Character Counter Reset

I/O (and Memory Offset) Address: 3C5h (index=07h)
Default: 00h
Attributes: Read/Write

For standard VGAs, writing this register (with any data) causes the horizontal character counter to be held in reset (the character counter output will remain 0). It remained in reset until a write occurred to any other sequencer register location with SRX set to an index of 0 through 6. In this implementation that sequence has no such special effect.

The vertical line counter is clocked by a signal derived from the horizontal display enable (which does not occur if the horizontal counter is held in reset). Therefore, if a write occurs to this register during the vertical retrace interval, both the horizontal and vertical counters will be set to 0. A write to any other sequencer register location (with SRX set to an index of 0 through 6) may then be used to start both counters with reasonable synchronization to an external event via software control. Although this was a standard VGA register, it was not documented by IBM.

Bit	Description
7:0	Horizontal Character Counter.

6.3 Graphics Controller Registers

The graphics controller registers are accessed via either I/O space or Memory space. Accesses to the registers of the VGA Graphics Controller are done through the use of address 3CEh (or memory address 3CEh) written with the index of the desired register. Then the desired register is accessed through the data port for the graphics controller registers at I/O address 3CFh (or memory address 3CFh). Indexes 10 and 11 should only be accessed through the I/O space only.

6.3.1 GRX—GRX Graphics Controller Index Register

I/O (and Memory Offset) Address: 3CEh
Default: 000UUUUUb (U=Undefined)
Attributes: Read/Write

7	4	3	0
Reserved (0000)			Graphics Controller Register Index

Bit	Description
7:4	Reserved. Read as 0.
3:0	Sequencer Register Index. This field selects any one of the graphics controller registers (GR00-GR11]) to be accessed via the data port at I/O (or memory offset) location 3CFh.



6.3.2 GR00—Set / Reset Register

I/O (and Memory Offset) Address: 3CFh (index=00h)
 Default: 0Uh (U=Undefined)
 Attributes: Read/Write

7	4	3	2	1	0
Reserved (0000)		Plane 3	Plane 2	Plane 1	Plane 0

Bit	Description
7:4	Reserved. Read as 0.
3:0	Set / Reset Plane [3:0]. When the Write Mode bits (bits 0 and 1) of the Graphics Mode Register (GR05) are set to select Write Mode 0, all 8 bits of each byte of each memory plane are set to either 1 or 0 as specified in the corresponding bit in this register, if the corresponding bit in the Enable Set/Reset Register (GR01) is set to 1. When the Write Mode bits (bits 0 and 1) of the Graphics Mode Register (GR05) are set to select Write Mode 3, all CPU data written to the frame buffer is rotated, then logically ANDed with the contents of the Bit Mask Register (GR08), and then treated as the addressed data's bit mask, while value of these four bits of this register are treated as the color value.

6.3.3 GR01—Enable Set / Reset Register

I/O (and Memory Offset) Address: 3CFh (Index=01h)
 Default: 0Uh (U=Undefined)
 Attributes: Read/Write

7	4	3	2	1	0
Reserved (0000)		Enable Set/ Reset Plane 3	Enable Set/ Reset Plane 2	Enable Set/ Reset Plane 1	Enable Set/ Reset Plane 0

Bit	Description
7:4	Reserved. Read as 0.
3:0	Enable Set / Reset Plane [3:0]. This register works in conjunction with the Set/Reset Register (GR00). The Write Mode bits (bits 0 and 1) must be set for Write Mode 0 for this register to have any effect. 0 = The corresponding memory plane can be read from or written to by the CPU without any special bitwise operations taking place. 1 = The corresponding memory plane is set to 0 or 1 as specified in the Set/Reset Register (GR00).



6.3.4 GR02—Color Compare Register

I/O (and Memory Offset) Address: 3CFh (Index=02h)
 Default: 0Uh (U=Undefined)
 Attributes: Read/Write

7	4	3	2	1	0
Reserved (0000)		Color Compare			
		Plane 3	Plane 2	Plane 1	Plane 0

Bit	Description
7:4	Reserved. Read as 0.
3:0	Color Compare Plane [3:0]. When the Read Mode bit (bit 3) of the Graphics Mode Register (GR05) is set to select Read Mode 1, all 8 bits of each byte of each of the 4 memory planes of the frame buffer corresponding to the address from which a CPU read access is being performed are compared to the corresponding bits in this register (if the corresponding bit in the Color Don't Care Register (GR07) is set to 1). The value that the CPU receives from the read access is an 8-bit value that shows the result of this comparison, wherein value of 1 in a given bit position indicates that all of the corresponding bits in the bytes across all of the memory planes that were included in the comparison had the same value as their memory plane's respective bits in this register.

6.3.5 GR03—Data Rotate Register

I/O (and Memory Offset) Address: 3CFh (Index=03h)
 Default: 0Uh (U=Undefined)
 Attributes: Read/Write

7	5	4	3	2	0
Reserved		Function Select		Rotate Count	

Bit	Description
7:5	Reserved. Read as 0.
4:3	Function Select. These bits specify the logical function (if any) to be performed on data that is meant to be written to the frame buffer (using the contents of the memory read latch) just before it is actually stored in the frame buffer at the intended address location. 00 = Data being written to the frame buffer remains unchanged, and is simply stored in the frame buffer. 01 = Data being written to the frame buffer is logically ANDed with the data in the memory read latch before it is actually stored in the frame buffer. 10 = Data being written to the frame buffer is logically ORed with the data in the memory read latch before it is actually stored in the frame buffer. 11 = Data being written to the frame buffer is logically XORed with the data in the memory read latch before it is actually stored in the frame buffer.
2:0	Rotate Count. These bits specify the number of bits to the right to rotate any data that is meant to be written to the frame buffer just before it is actually stored in the frame buffer at the intended address location.



6.3.6 GR04—Read Plane Select Register

I/O (and Memory Offset) Address: 3CFh (Index=04h)
Default: 0Uh (U=Undefined)
Attributes: Read/Write

7	2	1	0
Reserved (000000)			Read Plane Select

Bit	Description
7:2	Reserved. Read as 0.
1:0	Read Plane Select. These two bits select the memory plane from which the CPU reads data in Read Mode 0. In Odd/Even Mode, bit 0 of this register is ignored. In Chain 4 Mode, both bits 1 and 0 of this register are ignored. The four memory planes are selected as follows: 00 = Plane 0 01 = Plane 1 10 = Plane 2 11 = Plane 3 These two bits also select which of the four memory read latches may be read via the Memory read Latch Data Register (CR22). The choice of memory read latch corresponds to the choice of plane specified in the table above. The Memory Read Latch Data register and this additional function served by 2 bits are features of the VGA standard that were never documented by IBM.

6.3.7 GR05—Graphics Mode Register

I/O (and Memory Offset) Address: 3CFh (Index=05h)
Default: 0UUU U0UUb (U=Undefined)
Attributes: Read/Write

7	6	5	4	3	2	1	0
Reserved (0)	Shift Register Control		Odd/Even	Read Mode	Reserved	Write Mode	

Bit	Description
7	Reserved. Read as 0.



Bit	Description																																																																																																																																							
6:5	Shift Register Control. In standard VGA modes, pixel data is transferred from the 4 graphics memory planes to the palette via a set of 4 serial output bits. These 2 bits of this register control the format in which data in the 4 memory planes is serialized for these transfers to the palette. Bits [6:5]=00 One bit of data at a time from parallel bytes in each of the 4 memory planes is transferred to the palette via the 4 serial output bits, with 1 of each of the serial output bits corresponding to a memory plane. This provides a 4-bit value on each transfer for 1 pixel, making possible a choice of 1 of 16 colors per pixel. Serial <table><tr><th>Out</th><th>1st Xfer</th><th>2nd Xfer</th><th>3rd Xfer</th><th>4th Xfer</th><th>5th Xfer</th><th>6th Xfer</th><th>7th Xfer</th><th>8th Xfer</th></tr><tr><td>Bit 3</td><td>plane 3 bit 7</td><td>plane 3 bit 6</td><td>plane 3 bit 5</td><td>plane 3 bit 4</td><td>plane 3 bit 3</td><td>plane 3 bit 2</td><td>plane 3 bit 1</td><td>plane 3 bit 0</td></tr><tr><td>Bit 2</td><td>plane 2 bit 7</td><td>plane 2 bit 6</td><td>plane 2 bit 5</td><td>plane 2 bit 4</td><td>plane 2 bit 3</td><td>plane 2 bit 2</td><td>plane 2 bit 1</td><td>plane 2 bit 0</td></tr><tr><td>Bit 1</td><td>plane 1 bit 7</td><td>plane 1 bit 6</td><td>plane 1 bit 5</td><td>plane 1 bit 4</td><td>plane 1 bit 3</td><td>plane 1 bit 2</td><td>plane 1 bit 1</td><td>plane 1 bit 0</td></tr><tr><td>Bit 0</td><td>plane 0 bit 7</td><td>plane 0 bit 6</td><td>plane 0 bit 5</td><td>plane 0 bit 4</td><td>plane 0 bit 3</td><td>plane 0 bit 2</td><td>plane 0 bit 1</td><td>plane 0 bit 0</td></tr></table> Bits [6:5]=01 Two bits of data at a time from parallel bytes in each of the 4 memory planes are transferred to the palette in a pattern that alternates per byte between memory planes 0 and 2, and memory planes 1 and 3. First the even-numbered and odd-numbered bits of a byte in memory plane 0 are transferred via serial output bits 0 and 1, respectively, while the even-numbered and odd-numbered bits of a byte in memory plane 2 are transferred via serial output bits 2 and 3. Next, the even-numbered and odd-numbered bits of a byte in memory plane 1 are transferred via serial output bits 0 and 1, respectively, while the even-numbered and odd-numbered bits of memory plane 3 are transferred via serial out bits 1 and 3. This provides a pair of 2-bit values (one 2-bit value for each of 2 pixels) on each transfer, making possible a choice of 1 of 4 colors per pixel. Serial <table><tr><th>Out</th><th>1st Xfer</th><th>2nd Xfer</th><th>3rd Xfer</th><th>4th Xfer</th><th>5th Xfer</th><th>6th Xfer</th><th>7th Xfer</th><th>8th Xfer</th></tr><tr><td>Bit 3</td><td>plane 2 bit 7</td><td>plane 2 bit 5</td><td>plane 2 bit 3</td><td>plane 2 bit 1</td><td>plane 3 bit 7</td><td>plane 3 bit 5</td><td>plane 3 bit 3</td><td>plane 3 bit 1</td></tr><tr><td>Bit 2</td><td>plane 2 bit 6</td><td>plane 2 bit 4</td><td>plane 2 bit 2</td><td>plane 2 bit 0</td><td>plane 3 bit 6</td><td>plane 3 bit 4</td><td>plane 3 bit 2</td><td>plane 3 bit 0</td></tr><tr><td>Bit 1</td><td>plane 0 bit 7</td><td>plane 0 bit 5</td><td>plane 0 bit 3</td><td>plane 0 bit 1</td><td>plane 1 bit 7</td><td>plane 1 bit 5</td><td>plane 1 bit 3</td><td>plane 1 bit 1</td></tr><tr><td>Bit 0</td><td>plane 0 bit 6</td><td>plane 0 bit 4</td><td>plane 0 bit 2</td><td>plane 0 bit 0</td><td>plane 1 bit 6</td><td>plane 1 bit 4</td><td>plane 1 bit 2</td><td>plane 1 bit 0</td></tr></table> This alternating pattern is meant to accommodate the use of the Odd/Even mode of organizing the 4 memory planes, which is used by standard VGA modes 2h and 3h. Bits [6:5]=1x Four bits of data at a time from parallel bytes in each of the 4 memory planes are transferred to the palette in a pattern that iterates per byte through memory planes 0 through 3. First the 4 most significant bits of a byte in memory plane 0 are transferred via the 4 serial output bits, followed by the 4 least significant bits of the same byte. Next, the same transfers occur from the parallel byte in memory planes 1, 2 and lastly, 3. Each transfer provides either the upper or lower half of an 8-bit value for the color for each pixel, making possible a choice of 1 of 256 colors per pixel. This is the setting used in mode x13. Serial <table><tr><th>Out</th><th>1st Xfer</th><th>2nd Xfer</th><th>3rd Xfer</th><th>4th Xfer</th><th>5th Xfer</th><th>6th Xfer</th><th>7th Xfer</th><th>8th Xfer</th></tr><tr><td>Bit 3</td><td>plane 0 bit 7</td><td>plane 0 bit 3</td><td>plane 1 bit 7</td><td>plane 1 bit 3</td><td>plane 2 bit 7</td><td>plane 2 bit 3</td><td>plane 3 bit 7</td><td>plane 3 bit 3</td></tr><tr><td>Bit 2</td><td>plane 0 bit 6</td><td>plane 0 bit 2</td><td>plane 1 bit 6</td><td>plane 1 bit 2</td><td>plane 2 bit 6</td><td>plane 2 bit 2</td><td>plane 3 bit 6</td><td>plane 3 bit 2</td></tr><tr><td>Bit 1</td><td>plane 0 bit 5</td><td>plane 0 bit 1</td><td>plane 1 bit 5</td><td>plane 1 bit 1</td><td>plane 2 bit 5</td><td>plane 2 bit 1</td><td>plane 3 bit 5</td><td>plane 3 bit 1</td></tr><tr><td>Bit 0</td><td>plane 0 bit 4</td><td>plane 0 bit 0</td><td>plane 1 bit 4</td><td>plane 1 bit 0</td><td>plane 2 bit 4</td><td>plane 2 bit 0</td><td>plane 3 bit 4</td><td>plane 3 bit 0</td></tr></table> This pattern is meant to accommodate mode 13h, a standard VGA 256-color graphics mode.	Out	1st Xfer	2nd Xfer	3rd Xfer	4th Xfer	5th Xfer	6th Xfer	7th Xfer	8th Xfer	Bit 3	plane 3 bit 7	plane 3 bit 6	plane 3 bit 5	plane 3 bit 4	plane 3 bit 3	plane 3 bit 2	plane 3 bit 1	plane 3 bit 0	Bit 2	plane 2 bit 7	plane 2 bit 6	plane 2 bit 5	plane 2 bit 4	plane 2 bit 3	plane 2 bit 2	plane 2 bit 1	plane 2 bit 0	Bit 1	plane 1 bit 7	plane 1 bit 6	plane 1 bit 5	plane 1 bit 4	plane 1 bit 3	plane 1 bit 2	plane 1 bit 1	plane 1 bit 0	Bit 0	plane 0 bit 7	plane 0 bit 6	plane 0 bit 5	plane 0 bit 4	plane 0 bit 3	plane 0 bit 2	plane 0 bit 1	plane 0 bit 0	Out	1st Xfer	2nd Xfer	3rd Xfer	4th Xfer	5th Xfer	6th Xfer	7th Xfer	8th Xfer	Bit 3	plane 2 bit 7	plane 2 bit 5	plane 2 bit 3	plane 2 bit 1	plane 3 bit 7	plane 3 bit 5	plane 3 bit 3	plane 3 bit 1	Bit 2	plane 2 bit 6	plane 2 bit 4	plane 2 bit 2	plane 2 bit 0	plane 3 bit 6	plane 3 bit 4	plane 3 bit 2	plane 3 bit 0	Bit 1	plane 0 bit 7	plane 0 bit 5	plane 0 bit 3	plane 0 bit 1	plane 1 bit 7	plane 1 bit 5	plane 1 bit 3	plane 1 bit 1	Bit 0	plane 0 bit 6	plane 0 bit 4	plane 0 bit 2	plane 0 bit 0	plane 1 bit 6	plane 1 bit 4	plane 1 bit 2	plane 1 bit 0	Out	1st Xfer	2nd Xfer	3rd Xfer	4th Xfer	5th Xfer	6th Xfer	7th Xfer	8th Xfer	Bit 3	plane 0 bit 7	plane 0 bit 3	plane 1 bit 7	plane 1 bit 3	plane 2 bit 7	plane 2 bit 3	plane 3 bit 7	plane 3 bit 3	Bit 2	plane 0 bit 6	plane 0 bit 2	plane 1 bit 6	plane 1 bit 2	plane 2 bit 6	plane 2 bit 2	plane 3 bit 6	plane 3 bit 2	Bit 1	plane 0 bit 5	plane 0 bit 1	plane 1 bit 5	plane 1 bit 1	plane 2 bit 5	plane 2 bit 1	plane 3 bit 5	plane 3 bit 1	Bit 0	plane 0 bit 4	plane 0 bit 0	plane 1 bit 4	plane 1 bit 0	plane 2 bit 4	plane 2 bit 0	plane 3 bit 4	plane 3 bit 0
Out	1st Xfer	2nd Xfer	3rd Xfer	4th Xfer	5th Xfer	6th Xfer	7th Xfer	8th Xfer																																																																																																																																
Bit 3	plane 3 bit 7	plane 3 bit 6	plane 3 bit 5	plane 3 bit 4	plane 3 bit 3	plane 3 bit 2	plane 3 bit 1	plane 3 bit 0																																																																																																																																
Bit 2	plane 2 bit 7	plane 2 bit 6	plane 2 bit 5	plane 2 bit 4	plane 2 bit 3	plane 2 bit 2	plane 2 bit 1	plane 2 bit 0																																																																																																																																
Bit 1	plane 1 bit 7	plane 1 bit 6	plane 1 bit 5	plane 1 bit 4	plane 1 bit 3	plane 1 bit 2	plane 1 bit 1	plane 1 bit 0																																																																																																																																
Bit 0	plane 0 bit 7	plane 0 bit 6	plane 0 bit 5	plane 0 bit 4	plane 0 bit 3	plane 0 bit 2	plane 0 bit 1	plane 0 bit 0																																																																																																																																
Out	1st Xfer	2nd Xfer	3rd Xfer	4th Xfer	5th Xfer	6th Xfer	7th Xfer	8th Xfer																																																																																																																																
Bit 3	plane 2 bit 7	plane 2 bit 5	plane 2 bit 3	plane 2 bit 1	plane 3 bit 7	plane 3 bit 5	plane 3 bit 3	plane 3 bit 1																																																																																																																																
Bit 2	plane 2 bit 6	plane 2 bit 4	plane 2 bit 2	plane 2 bit 0	plane 3 bit 6	plane 3 bit 4	plane 3 bit 2	plane 3 bit 0																																																																																																																																
Bit 1	plane 0 bit 7	plane 0 bit 5	plane 0 bit 3	plane 0 bit 1	plane 1 bit 7	plane 1 bit 5	plane 1 bit 3	plane 1 bit 1																																																																																																																																
Bit 0	plane 0 bit 6	plane 0 bit 4	plane 0 bit 2	plane 0 bit 0	plane 1 bit 6	plane 1 bit 4	plane 1 bit 2	plane 1 bit 0																																																																																																																																
Out	1st Xfer	2nd Xfer	3rd Xfer	4th Xfer	5th Xfer	6th Xfer	7th Xfer	8th Xfer																																																																																																																																
Bit 3	plane 0 bit 7	plane 0 bit 3	plane 1 bit 7	plane 1 bit 3	plane 2 bit 7	plane 2 bit 3	plane 3 bit 7	plane 3 bit 3																																																																																																																																
Bit 2	plane 0 bit 6	plane 0 bit 2	plane 1 bit 6	plane 1 bit 2	plane 2 bit 6	plane 2 bit 2	plane 3 bit 6	plane 3 bit 2																																																																																																																																
Bit 1	plane 0 bit 5	plane 0 bit 1	plane 1 bit 5	plane 1 bit 1	plane 2 bit 5	plane 2 bit 1	plane 3 bit 5	plane 3 bit 1																																																																																																																																
Bit 0	plane 0 bit 4	plane 0 bit 0	plane 1 bit 4	plane 1 bit 0	plane 2 bit 4	plane 2 bit 0	plane 3 bit 4	plane 3 bit 0																																																																																																																																



Bit	Description
4	Odd / Even Mode. 0 = Addresses sequentially access data within a bit map, and the choice of which map is accessed is made according to the value of the Plane Mask Register (SR02). 1 = The frame buffer is mapped in such a way that the function of address bit 0 is such that even addresses select memory planes 0 and 2 and odd addresses select memory planes 1 and 3. Note: This works in a way that is the inverse of (and is normally set to be the opposite of) bit 2 of the Memory Mode Register (SR02).
3	Read Mode. 0 = During a CPU read from the frame buffer, the value returned to the CPU is data from the memory plane selected by bits 1 and 0 of the Read Plane Select Register (GR04). 1 = During a CPU read from the frame buffer, all 8 bits of the byte in each of the 4 memory planes corresponding to the address from which a CPU read access is being performed are compared to the corresponding bits in this register (if the corresponding bit in the Color Don't Care Register (GR07) is set to 1). The value that the CPU receives from the read access is an 8-bit value that shows the result of this comparison. A value of 1 in a given bit position indicates that all of the corresponding bits in the bytes across all 4 of the memory planes that were included in the comparison had the same value as their memory plane's respective bits in this register.
2	Reserved. Read as 0.
1:0	Write Mode. 00 = Write Mode 0 — During a CPU write to the frame buffer, the addressed byte in each of the 4 memory planes is written with the CPU write data after it has been rotated by the number of counts specified in the Data Rotate Register (GR03). If, however, the bit(s) in the Enable Set/Reset Register (GR01) corresponding to one or more of the memory planes is set to 1, then those memory planes will be written to with the data stored in the corresponding bits in the Set/Reset Register (GR00). 01 = Write Mode 1 — During a CPU write to the frame buffer, the addressed byte in each of the 4 memory planes is written to with the data stored in the memory read latches. (The memory read latches stores an unaltered copy of the data last read from any location in the frame buffer.) 10 = Write Mode 2 — During a CPU write to the frame buffer, the least significant 4 data bits of the CPU write data is treated as the color value for the pixels in the addressed byte in all 4 memory planes. The 8 bits of the Bit Mask Register (GR08) are used to selectively enable or disable the ability to write to the corresponding bit in each of the 4 memory planes that correspond to a given pixel. A setting of 0 in a bit in the Bit Mask Register at a given bit position causes the bits in the corresponding bit positions in the addressed byte in all 4 memory planes to be written with value of their counterparts in the memory read latches. A setting of 1 in a Bit Mask Register at a given bit position causes the bits in the corresponding bit positions in the addressed byte in all 4 memory planes to be written with the 4 bits taken from the CPU write data to thereby cause the pixel corresponding to these bits to be set to the color value. 11 = Write Mode 3 — During a CPU write to the frame buffer, the CPU write data is logically ANDed with the contents of the Bit Mask Register (GR08). The result of this ANDing is treated as the bit mask used in writing the contents of the Set/Reset Register (GR00) are written to addressed byte in all 4 memory planes.



6.3.8 GR06—Miscellaneous Register

I/O (and Memory Offset) Address: 3CFh (Index=06h)
 Default: 0Uh (U=Undefined)
 Attributes: Read/Write

7	4	3	2	1	0
Reserved (0000)			Memory Map Mode	Chain Odd/Even	Graphics / Text Mode

Bit	Description
7:4	Reserved. Read as 0s.
3:2	Memory Map Mode. These 2 bits control the mapping of the VGA address range for frame buffer into the CPU address space as follows: Bit [3:2] Frame Buffer Address Range 00 A0000h – BFFFFh 01 A0000h – AFFFFh 10 B0000h – B7FFFh 11 B8000h – BFFFFh Note: This function is used in both in standard VGA modes, extended VGA modes (132 column text), and in non-VGA modes (hi-res). (132 column text modes are no longer supported). VGA aperture memory accesses are also controlled by the PCI configuration Memory Enable bit and the RAM enable bit in the Miscellaneous Output Register (3c2/3cc). For accesses using GR10 and GR11 to paged VGA RAM or to device MMIO registers, set these bits to 01to select the (A0000-AFFFF) range. The CPU must map this memory as uncacheable (UC); see VGA Host Access Memory Munging in <i>Display and Overlay Functions</i>.
1	Chain Odd/Even. This bit provides the ability to alter the interpretation of address bit A0, so that it may be used in selecting between the odd-numbered memory planes (planes 1 and 3) and the even-numbered memory planes (planes 0 and 2). 0 = A0 functions normally. 1 = A0 is switched with a high order address bit, in terms of how it is used in address decoding. The result is that A0 is used to determine which memory plane is being accessed (A0=0 for planes 0 and 2 and A0=1 for planes 1 and 3).
0	Graphics/Text Mode. This is one of two bits that are used to determine if the VGA is operating in text or graphics modes. The other bit is in AR10[0], these two bits need to be programmed in a consistent manner to achieve the proper results. 0 = Text mode. 1 = Graphics mode.



6.3.9 GR07—Color Don't Care Register

I/O (and Memory Offset) Address: 3CFh (Index=07h)
Default: 0Uh (U=Undefined)
Attributes: Read/Write

7	4	3	2	1	0
Reserved (0000)		Ignore Color Plane 3	Ignore Color Plane 2	Ignore Color Plane 1	Ignore Color Plane 0

Bit	Description
7:4	Reserved. Read as 0.
3:0	Ignore Color Plane [3:0]. Note that these bits have effect only when bit 3 of the Graphics Mode Register (GR05) is set to 1 to select read mode 1. 0 = The corresponding bit in the Color Compare Register (GR02) will not be included in color comparisons. 1 = The corresponding bit in the Color Compare Register (GR02) is used in color comparisons.

6.3.10 GR08—Bit Mask Register

I/O (and Memory Offset) Address: 3CFh (Index=08h)
Default: Undefined
Attributes: Read/Write

Bit	Description
7:0	Bit Mask. 0 = The corresponding bit in each of the 4 memory planes is written to with the corresponding bit in the memory read latches. 1 = Manipulation of the corresponding bit in each of the 4 memory planes via other mechanisms is enabled. Notes: This bit mask applies to any writes to the addressed byte of any or all of the 4 memory planes, simultaneously. This bit mask is applicable to any data written into the frame buffer by the CPU, including data that is also subject to rotation, logical functions (AND, OR, XOR), and Set/Reset. To perform a proper read-modify-write cycle into frame buffer, each byte must first be read from the frame buffer by the CPU (and this will cause it to be stored in the memory read latches), this Bit Mask Register must be set, and the new data then written into the frame buffer by the CPU.



6.3.11 GR10—Address Mapping

I/O (avoid MMIO access) Address: 3CFh (Index=10h)
 Default: 00h
 Attributes: R/W

This register should only be accessed using I/O operations and never be accessed through the A/B segment addressing map, I/O space register map, or direct MMIO operations.

7	4	3	2	1	0
Page Select Extension		I/O Map Enable	Paging Target		Page Mapping

Bit	Description
7:4	Page Select Extension These bits form the upper bits of a 12-bit page selection value. They when combined with the GR11 <7:0> bits they define the offset into stolen memory to the 64KB page that is accessible via the VGA Memory paging mechanism. This register provides the upper Address[27:24] bits for the access allowing for a maximum of 256MB address range. The actual range that can be used is limited to the size of the stolen graphics memory region. Addresses specified through these bits and the bits in GR11 should be legal addresses that have been limited to the size of stolen memory, accesses outside this range produce unspecified results.
3	Reserved
2:1	Paging Map Target. When paging is enabled, these bits determine the target for data cycle accesses through the VGA memory aperture. VGA Graphics pre-allocated memory is used for cases where there is no local memory and in devices that do not support local memory. Local memory is used when it exists. Memory mapped register access is only available through this mechanism in a few devices. 00 = VGA Graphics Memory (Local/pre-allocated starting at a base of local address zero) 01 = Reserved (Was Memory Mapped Registers for previous devices) 10 = Reserved (Was Video BIOS ROM Memory only for discrete graphics devices) 11 = Reserved



Bit	Description
0	Page Mapping Enable. This mode allows the mapping of the VGA memory address space to either VGA memory (pre-allocated or local). In previous devices it was used to map MMIO registers, or Video BIOS ROM. This allowed video BIOS access to the entire memory mapped register space, allowed real mode DOS applications for BIOS flash operations, and extended video mode support for DOS applications in cases where the frame buffer is greater than the 64K bytes. Some Notes on Paging. Once this is enabled, no VGA memory address swizzel will be performed; addresses are directly mapped to memory. The same thing was true for ROM or register accesses for devices that support that operation. A single paging register is used to map the 64KB [A0000:FFFF] window. An internal address is generated using GR11[7:0] as the address lines [23:16] extension to the lower address lines of the access A[15:2]. When mapping is enabled, the B0000:BFFFF area is always disabled using GR06<3:2>=01. The use of addresses in the A0000-BFFFF range requires that both the graphics device PCI configuration memory enable and MSR<1> be enabled. Graphics Mode Select (GMS). This field is used to select the amount of Main Memory that is “pre-allocated” to support the Internal Graphics device in VGA (non-linear) mode only. These 2 bits are valid only when Internal graphics is enabled. (Paging must not step out of pre-allocated memory within main memory) In discrete graphics devices, ROM pins are sometimes shared with other functions such as capture. Care must be taken to insure that ROM accesses do not conflict with other ongoing operations. In cases where SMM code is executing out of the A/B segment with both code and data cycles target the SMM memory, the programming of the target to memory mapped registers does not override SMM target selection. 0 = Disable (default) 1 = Enable



6.3.12 GR11—Page Selector

I/O Address (avoid MMIO access): 3CFh (Index=11h)

Default: 00h

Attributes: R/W

Bit	Description
7:0	Page Select. When concatenated with the GR10<7:4> bits, selects a 64KB window within target area when Page Mapping is enabled (GR10[0]=1). This requires that the graphics device PCI configuration space memory enable, the GR06<3:2> bits to be 01 (select A0000-AFFFF only), and the MSR<1:1> bit to be set. This register provides the Address[23:16] bits for the access allowing for a 256MB address range. VGA paging of frame buffer memory is for non-VGA packed modes only and should not be enabled when using basic VGA modes. This register should only be accessed using I/O operations and never be accessed through the A000 segment addressing map or direct MMIO operations. Addresses generated via this method should be restricted to within the size of the pre-allocated (stolen) memory that is currently available.

6.3.13 GR18—Software Flags

I/O (and Memory Offset) Address: 3CFh (Index=18h)

Default: 00

Attribute: R/W

Bit	Description
7:0	Software Flags. Used as scratch pad space in video BIOS. These bits are separate from the bits which appear in the memory mapped IO space. They are used specifically by the SMI BIOS which does not have access to memory mapped IO at the time they are required. These register bits have no effect on H/W operation.

6.4 Attribute Controller Registers

Unlike the other sets of indexed registers, the attribute controller registers are not accessed through a scheme employing entirely separate index and data ports. I/O address 3C0h (or memory address 3C0h) is used both as the read and write for the index register, and as the write address for the data port. I/O address 3C1h (or memory address 3C1h) is the read address for the data port.

To write to the attribute controller registers, the index of the desired register must be written to I/O address 3C0h (or memory address 3C0h), and then the data is written to the very same I/O (memory) address. A flip-flop alternates with each write to I/O address 3C0h (or memory address 3C0h) to change its function from writing the index to writing the actual data, and back again. This flip-flop may be deliberately set so that I/O address 3C0h (or memory address 3C0h) is set to write to the index (which provides a way to set it to a known state) by performing a read operation from Input Status Register 1 (ST01) at I/O address 3BAh (or memory address 3BAh) or 3DAh (or



memory address 3DAh), depending on whether the graphics system has been set to emulate an MDA or a CGA as per MSR[0].

To read from the attribute controller registers, the index of the desired register must be written to I/O address 3C0h (or memory address 3C0h), and then the data is read from I/O address 3C1h (or memory address 3C1h). A read operation from I/O address 3C1h (or memory address 3C1h) does not reset the flip-flop to writing to the index. Only a write to 3C0h (or memory address 3C0h) or a read from 3BAh or 3DAh (or memory address 3BAh or 3DAh), as described above, will toggle the flip-flop back to writing to the index.

6.4.1 ARX—Attribute Controller Index Register

I/O (and Memory Offset) Address: 3C0h

Default: 00UU UUUUb (U=Undefined)

Attributes: Read/Write

7	6	5	4	0
Reserved (00)		Video Enable	Attribute Controller Register Index	

Bit	Description
7:6	Reserved. Read as 0s.
5	Video Enable. Note that In the VGA standard, this is called the “Palette Address Source” bit. Clearing this bit will cause the VGA display data to become all 00 index values. For the default palette, this will cause a black screen. The video timing signals continue. Another control bit will turn video off and stop the data fetches. 0 = Disable. Attribute controller color registers (AR[00:0F]) can be accessed by the CPU. 1 = Enable. Attribute controller color registers (AR[00:0F]) are inaccessible by the CPU.
4:0	Attribute Controller Register Index. These five bits are used to select any one of the attribute controller registers (AR[00:14]), to be accessed. Note: AR12 is referred to in the VGA standard as the Color Plane Enable Register. The words “plane,” “color plane,” “display memory plane,” and “memory map” have been all been used in IBM* literature on the VGA standard to describe the four separate regions in the frame buffer where the pixel color or attribute information is split up and stored in standard VGA planar modes. This use of multiple terms for the same subject was deemed to be confusing; therefore, AR12 is called the Memory Plane Enable Register. Attribute Controller Register Index.



6.4.2 AR[00:0F]—Palette Registers [0:F]

I/O (and Memory Offset) Address: Read at 3C1h and Write at 3C0h; (index=00h-0Fh)

Default: 00UU UUUUb (U=Undefined)

Attributes: Read/Write

7	6	5	0
Reserved			Palette Bits P[5:0]

Bit	Description
7:6	Reserved. Read as 0.
5:0	Palette Bits P[5:0]. In each of these 16 registers, these are the lower 6 of 8 bits that are used to map either text attributes or pixel color input values (for modes that use 16 colors) to the 256 possible colors available to be selected in the palette. Note: Bits 3 and 2 of the Color Select Register (AR14) supply bits P7 and P6 for the values contained in all 16 of these registers. Bits 1 and 0 of the Color Select Register (AR14) can also replace bits P5 and P4 for the values contained in all 16 of these registers, if bit 7 of the Mode Control Register (AR10) is set to 1.

6.4.3 AR10—Mode Control Register

I/O (and Memory Offset) Address: Read at 3C1h and Write at 3C0h; (index=10h)

Default: UUh (U=Undefined)

Attributes: Read/Write

7	6	5	4	3	2	1	0
Palette Bits P5, P4 Select	Pixel Width/Clock Select	Pixel Panning Compat	Reserved (0)	Enable Blink/Select Bkgnd Int	Enable Line Graphics Char Code	Select Display Type	Graphics/Alpha Mode

Bit	Description
7	Palette Bits P5, P4 Select. 0 = P5 and P4 for each of the 16 selected colors (for modes that use 16 colors) are individually provided by bits 5 and 4 of their corresponding Palette Registers (AR[00:0F]). 1 = P5 and P4 for all 16 of the selected colors (for modes that use 16 colors) are provided by bits 1 and 0 of Color Select Register (AR14).



Bit	Description
6	Pixel Width / Clock Select. 0 = Six bits of video data (translated from 4 bits via the palette) are output every dot clock. 1 = Two sets of 4 bits of data are assembled to generate 8 bits of video data which is output every other dot clock, and the Palette Registers (AR[00:0F]) are bypassed. Note: This bit is set to 0 for all of the standard VGA modes, except mode 13h.
5	Pixel Panning Compatibility. 0 = Scroll both the upper and lower screen regions horizontally as specified in the Pixel Panning Register (AR13). 1 = Scroll only the upper screen region horizontally as specified in the Pixel Panning Register (AR13). Note: This bit has application only when split-screen mode is being used, where the display area is divided into distinct upper and lower regions which function somewhat like separate displays.
4	Reserved. Read as 0.
3	Enable Blinking / Select Background Intensity. 0 = Disables blinking in graphics modes, and for text modes, sets bit 7 of the character attribute bytes to control background intensity, instead of blinking. 1 = Enables blinking in graphics modes and for text modes, sets bit 7 of the character attribute bytes to control blinking, instead of background intensity. Note: The blinking rate is derived by dividing the VSYNC signal. The Blink Rate Control field of the VGA control register defines the blinking rate.
2	Enable Line Graphics Character Code. 0 = Every 9th pixel of a horizontal line (i.e., the last pixel of each horizontal line of each 9-pixel wide character box) is assigned the same attributes as the background of the character of which the given pixel is a part. 1 = Every 9th pixel of a horizontal line (i.e., the last pixel of each horizontal line of each 9-pixel wide character box) is assigned the same attributes as the 8th pixel of the character of which the given pixel is a part. This setting is intended to accommodate the line-drawing characters of the PC's extended ASCII character set -- characters with an extended ASCII code in the range of B0h to DFh. Note: In IBM* literature describing the VGA standard, the range of extended ASCII codes that are said to include the line-drawing characters is mistakenly specified as C0h to DFh, rather than the correct range of B0h to DFh.
1	Select Display Type. 0 = Attribute bytes in text modes are interpreted as they would be for a color display. 1 = Attribute bytes in text modes are interpreted as they would be for a monochrome display.
0	Graphics / Alphanumeric Mode. This bit (along with GR06[0]) select either graphics mode or text mode. These two bits must be programmed in a consistent manner to achieve the desired results. 0 = Alphanumeric (text) mode. 1 = Graphics mode.



6.4.4 AR11—Overscan Color Register

I/O (and Memory Offset) Address: Read at 3C1h and Write at 3C0h; (index=11h)

Default: UUh (U=Undefined)

Attributes: Read/Write

Bit	Description
7:0	Overscan. These 8 bits select the overscan (border) color index value. The actual border color will be determined by the contents of the palette at the selected index. The border color is displayed between the end of active and the beginning of blank or the end of blank and the beginning of active on CRT type devices driven from the DAC output port. For native VGA modes on digital display ports there is the option of including the border in the active region or not depending on a control bit in the port control register. For centered VGA modes, the VGA control register determines if the border is included in the centered region or not. For monochrome displays, this value should be set to 00h.

6.4.5 AR12—Memory Plane Enable Register

I/O (and Memory Offset) Address: Read at 3C1h and Write at 3C0h; (index=12h)

Default: 00UU UUUUb (U=Undefined)

Attributes: Read/Write

7	6	5	4	3	2	1	0
Reserved (00)	Video Status Mux		Enable Plane 3	Enable Plane 2	Enable Plane 1	Enable Plane 0	

Bit	Description															
7:6	Reserved. Read as 0.															
5:4	Video Status Mux. These 2 bits are used to select 2 of the 8 possible palette bits (P7-P0) to be made available to be read via bits 5 and 4 of the Input Status Register 1 (ST01). The table below shows the possible choices. <table><tr><th>Bit [5:4]</th><th>ST01 Bit 5</th><th>ST01 Bit 4</th></tr><tr><td>00</td><td>P2 (default)</td><td>P0 (default)</td></tr><tr><td>01</td><td>P5</td><td>P4</td></tr><tr><td>10</td><td>P3</td><td>P1</td></tr><tr><td>11</td><td>P7</td><td>P6</td></tr></table> These bits are typically unused by current software; they are provided for EGA compatibility.	Bit [5:4]	ST01 Bit 5	ST01 Bit 4	00	P2 (default)	P0 (default)	01	P5	P4	10	P3	P1	11	P7	P6
Bit [5:4]	ST01 Bit 5	ST01 Bit 4														
00	P2 (default)	P0 (default)														
01	P5	P4														
10	P3	P1														
11	P7	P6														
3:0	Enable Plane [3:0]. These 4 bits individually enable the use of each of the 4 memory planes in providing 1 of the 4 bits used in video output to select 1 of 16 possible colors from the palette to be displayed. 0 = Disable the use of the corresponding memory plane in video output to select colors, forcing the bit that the corresponding memory plane would have provided to a value of 0. 1 = Enable the use of the corresponding memory plane in video output to select colors. Note: AR12 is referred to in the VGA standard as the Color Plane Enable Register. The words “plane,” “color plane,” “display memory plane,” and “memory map” have been all been used in IBM™ literature on the VGA standard to describe the 4 separate regions in the frame buffer that are amongst which pixel color or attributes information is split up and stored in standard VGA planar modes. This use of multiple terms for the same subject was considered confusing; therefore, AR12 is called the Memory Plane Enable Register.															



6.4.6 AR13—Horizontal Pixel Panning Register

I/O (and Memory Offset) Address: Read at 3C1h and Write at 3C0h; (index=13h)
 Default: 0Uh (U=Undefined)
 Attributes: Read/Write

7	4	3	0
Reserved (0000)		Horizontal Pixel Shift	

Bit	Description																																																							
7:4	Reserved																																																							
3:0	Horizontal Pixel Shift 3-0. This field holds a 4-bit value that selects the number of pixels by which the image is shifted horizontally to the left. This function is available in both text and graphics modes and allows for pixel panning. In text modes with a 9-pixel wide character box, the image can be shifted up to 9 pixels to the left. In text modes with an 8-pixel wide character box, and in graphics modes other than those with 256 colors, the image can be shifted up to 8 pixels to the left. A pseudo 9-bit mode is when the 9-dot character is selected but overridden by the VGA control bit. In standard VGA mode 13h (where bit 6 of the Mode Control Register, AR10, is set to 1 to support 256 colors), bit 0 of this register must remain set to 0, and the image may be shifted up to only 4 pixels to the left. In this mode, the number of pixels by which the image is shifted can be further controlled using bits 6 and 5 of the Preset Row Scan Register (CR08). <table><tr><th colspan="5">Number of Pixels Shifted</th></tr><tr><th>Bits [3:0]</th><th>9-dot</th><th>Pseudo 9-dot</th><th>8-dot</th><th>256-Color</th></tr><tr><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>2</td><td>2</td><td>1</td><td>Undefined</td></tr><tr><td>2</td><td>3</td><td>3</td><td>2</td><td>1</td></tr><tr><td>3</td><td>4</td><td>4</td><td>3</td><td>Undefined</td></tr><tr><td>4</td><td>5</td><td>5</td><td>4</td><td>2</td></tr><tr><td>5</td><td>6</td><td>6</td><td>5</td><td>Undefined</td></tr><tr><td>6</td><td>7</td><td>7</td><td>6</td><td>3</td></tr><tr><td>7</td><td>8</td><td>7</td><td>7</td><td>Undefined</td></tr><tr><td>8</td><td>0</td><td>0</td><td>Undefined</td><td>Undefined</td></tr></table>	Number of Pixels Shifted					Bits [3:0]	9-dot	Pseudo 9-dot	8-dot	256-Color	0	1	1	0	0	1	2	2	1	Undefined	2	3	3	2	1	3	4	4	3	Undefined	4	5	5	4	2	5	6	6	5	Undefined	6	7	7	6	3	7	8	7	7	Undefined	8	0	0	Undefined	Undefined
Number of Pixels Shifted																																																								
Bits [3:0]	9-dot	Pseudo 9-dot	8-dot	256-Color																																																				
0	1	1	0	0																																																				
1	2	2	1	Undefined																																																				
2	3	3	2	1																																																				
3	4	4	3	Undefined																																																				
4	5	5	4	2																																																				
5	6	6	5	Undefined																																																				
6	7	7	6	3																																																				
7	8	7	7	Undefined																																																				
8	0	0	Undefined	Undefined																																																				



6.4.7 AR14—Color Select Register

I/O (and Memory Offset) Address: Read at 3C1h and Write at 3C0h; (index=14h)
 Default: 0Uh (U=Undefined)
 Attributes: Read/Write

7	4	3	2	1	0
Reserved (0000)				P7	P6
				Alt P5	Alt P4

Bit	Description
7:4	Reserved
3:2	Palette Bits P[7:6]. These are the 2 upper-most of the 8 bits that are used to map either text attributes or pixel color input values (for modes that use 16 colors) to the 256 possible colors contained in the palette. These 2 bits are common to all 16 sets of bits P5 through P0 that are individually supplied by Palette Registers 0-F (AR[00:0F]).
1:0	Alternate Palette Bits P[5:4]. These 2 bits can be used as an alternate version of palette bits P5 and P4. Unlike the P5 and P4 bits that are individually supplied by Palette Registers 0-F (AR[00:0F]), these 2 alternate palette bits are common to all 16 of Palette Registers. Bit 7 of the Mode Control Register (AR10) is used to select between the use of either the P5 and P4 bits that are individually supplied by the 16 Palette Registers or these 2 alternate palette bits.

6.5 VGA Color Palette Registers

In devices that have two display pipes, there are two palettes, one for each display pipe. These palettes are the same for VGA modes and non-VGA modes. Accesses through VGA register methods can optionally read or write from either one.

For each palette in the device, for each palette, the color data stored in these 256 color data positions can be accessed only through a complex sub-addressing scheme, using a data register and two index registers. The Palette Data Register at I/O address 3C9h (or memory address offset 3C1h) is the data port. The Palette Read Index Register at I/O address 3C7h (or memory address offset 3C7h) and the Palette Write Index Register at I/O address 3C8h (or memory address offset 3C8h) are the two index registers. The Palette Read Index Register is the index register that is used to choose the color data position that is to be read from via the data port, while the Palette Write Index Register is the index register that is used to choose the color data position that is to be written to through the same data port. This arrangement allows the same data port to be used for reading from and writing to two different color data positions. Reading and writing the color data at a color data position involves three successive reads or writes since the color data stored at each color data position consists of three bytes.

To read a palette color data position, the index of the desired color data position must first be written to the Palette Read Index Register. Then all three bytes of data in a given color data position may be read at the Palette Data Register. The first byte read from the Palette Data Register retrieves the 8-bit value specifying the intensity of the red color component. The second and third bytes read are the corresponding 8-bit values for the green and blue color components respectively. After completing the



third read operation, the Palette Read Index Register is automatically incremented so that the data of the next color data position becomes accessible for being read. This allows the contents of all of the 256 color data positions of the palette to be read in sequence. This is done by specifying only the index of the 0th color data position in the Palette Read Index Register, and then simply performing 768 successive reads from the Palette Data Register.

Writing a color data position, entails a very similar procedure. The index of the desired color data position must first be written to the Palette Write Index Register. Then all three bytes of data to specify a given color may be written to the Palette Data Register. The first byte written to the Palette Data Register specifies the intensity of the red color component, the second byte specifies the intensity for the green color component, and the third byte specifies the same for the blue color component. One important detail is that all three of these bytes must be written before the hardware will actually update these three values in the given color data position. When all three bytes have been written, the Palette Write Index Register is automatically incremented so that the data of the next color data position becomes accessible for being written. This allows the contents of all of the 256 color data positions of the palette to be written in sequence. This is done by specifying only the index of the 0th color data position in the Palette Write Index Register, and then simply performing 768 successive writes to the Palette Data Register.

6.5.1 DACMASK—Pixel Data Mask Register

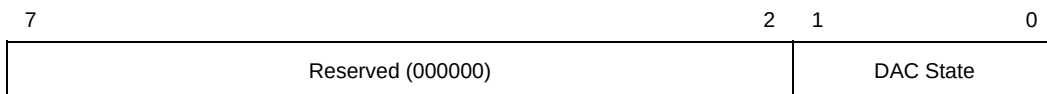
I/O (and Memory Offset) Address: 3C6h
Default: Undefined
Attributes: Read/Write

Bit	Description
7:0	Pixel Data Mask. In indexed-color mode, the 8 bits of this register are logically ANDed with the 8 bits of pixel data received from the frame buffer for each pixel. The result of this ANDing process becomes the actual index used to select color data positions within the palette. This has the effect of limiting the choice of color data positions that may be specified by the incoming 8-bit data. 0 = Corresponding bit in the resulting 8-bit index being forced to 0. 1 = Allows the corresponding bit in the resulting index to reflect the actual value of the corresponding bit in the incoming 8-bit pixel data.



6.5.2 DACSTATE—DAC State Register

I/O (and Memory Offset) Address: 3C7h
 Default: 00h
 Attributes: Read-Only



Bit	Description
7:2	Reserved. Read as 0.
1:0	DAC State. This field indicates which of the two index registers was most recently written. Bits [1:0] Index Register Indicated 00 = Palette Write Index Register at I/O Address 3C7h (default) 01 = Reserved 10 = Reserved 11 = Palette Read Index Register at I/O Address 3C8h

6.5.3 DACRX—Palette Read Index Register

I/O (and Memory Offset) Address: 3C7h
 Default: 00h
 Attributes: Write-Only

Bit	Description
7:0	Palette Read Index. The 8-bit index value programmed into this register chooses which of 256 standard color data positions within the palette are to be made accessible for being read from via the Palette Data Register (DACDATA). The index value held in this register is automatically incremented when all three bytes of the color data position selected by the current index have been read. A write to this register will abort a uncompleted palette write sequence. This register allows access to the palette even when running non-VGA display modes.



6.5.4 DACWX—Palette Write Index Register

I/O (and Memory Offset) Address: 3C8h
Default: 00h
Attributes: Write-Only

Bit	Description
7:0	Palette Write Index. The 8-bit index value programmed into this register chooses which of 256 standard color data positions within the palette are to be made accessible for being written via the Palette Data Register (DACDATA). The index value held in this register is automatically incremented when all three bytes of the color data position selected by the current index have been written. This register allows access to the palette even when running non-VGA display modes.

6.5.5 DACDATA—Palette Data Register

I/O (and Memory Offset) Address: 3C9h
Default: Undefined
Attributes: Read/Write

Bit	Description
7:0	Palette Data. This byte-wide data port provides read or write access to the three bytes of data of each color data position selected using the Palette Read Index Register (DACRX) or the Palette Write Index Register (DACWX). The three bytes in each color data position are read or written in three successive read or write operations. The first byte read or written specifies the intensity of the red component of the color specified in the selected color data position. The second byte is for the green component, and the third byte is for the blue component. When writing data to a color data position, all three bytes must be written before the hardware will actually update the three bytes of the selected color data position. When reading or writing to a color data position, ensure that neither the Palette Read Index Register (DACRX) nor the Palette Write Index Register (DACWX) are written to before all three bytes are read or written. A write to either of these two registers causes the circuitry that automatically cycles through providing access to the bytes for red, green and blue components to be reset such that the byte for the red component is the one that will be accessed by the next read or write operation via this register. This register allows access to the palette even when running non-VGA display modes. Writes to the palette can cause sparkle if not done during inactive video periods. This sparkle is caused by an attempt to write and read the same address on the same cycle. Anti-sparkle circuits will substitute the previous pixel value for the read output.



6.6 CRT Controller Register

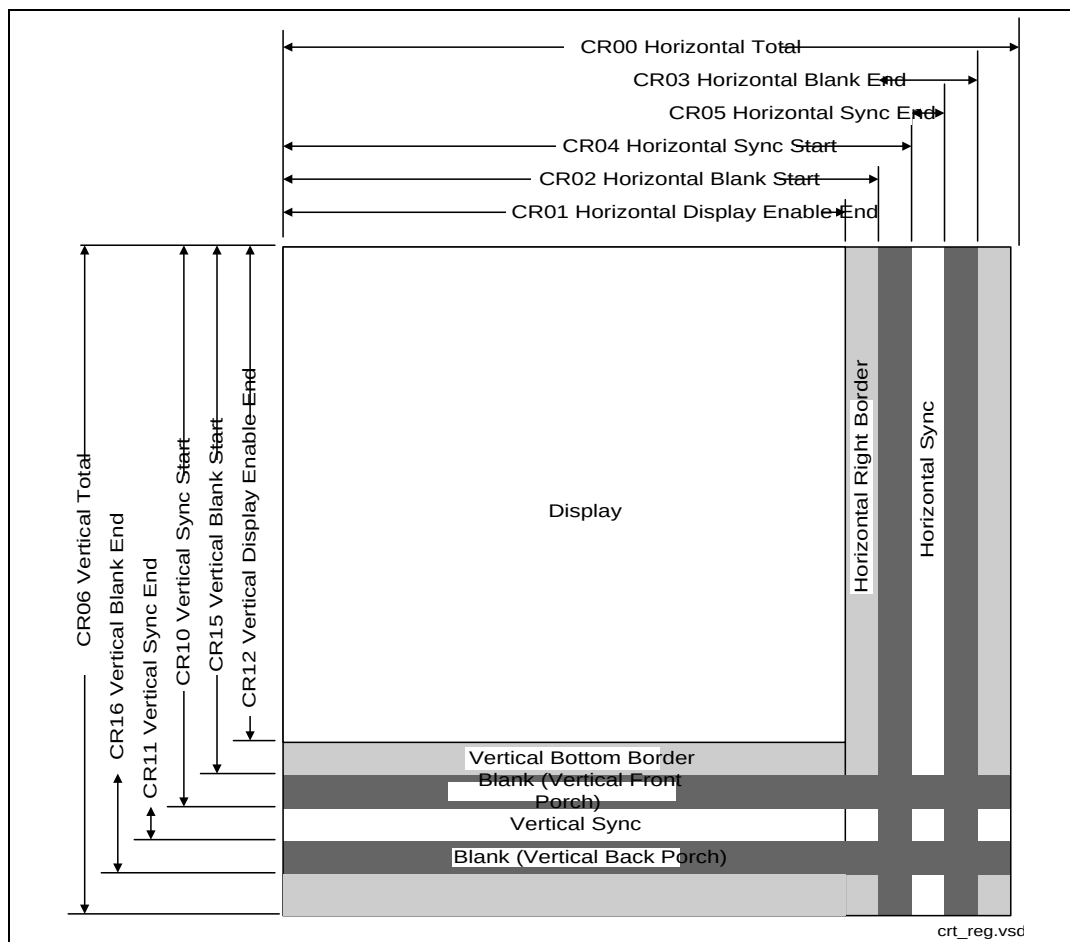
For native VGA modes, the CRT controller registers determine the display timing that is to be used. In centered VGA modes, these registers determine the size of the VGA image that is to be centered in the larger timing generator defined rectangle.

The CRT controller registers are accessed by writing the index of the desired register into the CRT Controller Index Register at I/O address 3B4h or 3D4h, depending on whether the graphics system is configured for MDA or CGA emulation. The desired register is then accessed through the data port for the CRT controller registers located at I/O address 3B5h or 3D5h, again depending upon the choice of MDA or CGA emulation as per MSR[0]. For memory mapped accesses, the Index register is at 3B4h (MDA mode) or 3D3h (CGA mode) and the data port is accessed at 3B5h (MDA mode) or 3D5h (CGA mode).

Notes:

1. **Group 0 Protection:** In the original IBM VGA, CR[0:7] could be made write-protected by CR11[7]. In BIOS code, this write protection is set following each mode change. Other protection groups have no current use, and would not be used going forward by the BIOS or by drivers. They are the result of an industry fad some years ago to attempt to write protect other groups of registers; however, all such schemes were chip specific. Only the IBM compatible write protection (Group 0 Protection) is supported.

The following figure shows display fields and dimensions and the particular CRxx register that provides the control.



6.6.1 CRX—CRT Controller Index Register

I/O (and Memory Offset) Address: 3B4h/3D4h
 Default: 0Uh (U=Undefined)
 Attributes: Read/Write

Bit	Description
7	Reserved. Read as 0.
6:0	CRT Controller Index. These 7 bits are used to select any one of the CRT controller registers to be accessed via the data port at I/O location 3B5h or 3D5h, depending upon whether the graphics system is configured for MDA or CGA emulation. The data port memory address offsets are 3B5h/3D5h.



6.6.2 CR00—Horizontal Total Register

I/O (and Memory Offset) Address: 3B5h/3D5h (index=00h)
Default: 00h
Attributes: Read/Write (Group 0 Protection)

Bit	Description
7:0	Horizontal Total. This register is used to specify the total length of each scan line. This encompasses both the part of the scan line that is within the active display area and the part that is outside of it. Programming this register to a zero has the effect of stopping the fetching of display data. This field should be programmed with a value equal to the total number of character clocks within the entire length of a scan line, minus 5.

6.6.3 CR01—Horizontal Display Enable End Register

I/O (and Memory Offset) Address: 3B5h/3D5h (index=01h)
Default: Undefined
Attributes: Read/Write (Group 0 Protection)

Bit	Description
7:0	Horizontal Display Enable End. This register is used to specify the end of the part of the scan line that is within the active display area relative to its beginning. In other words, this is the horizontal width of the active display area. This field should be programmed with a value equal to the number of character clocks that occur within the horizontal active display area, minus 1. Horizontal display enable will go active at the beginning of each line during vertical active area, it will go inactive based on the programming of this register or the programming of the horizontal total (CR00) register. When this register value is programmed to a number that is larger than the total number of characters on a line, display enable will be active for all but the last character of the horizontal display line.

6.6.4 CR02—Horizontal Blanking Start Register

I/O (and Memory Offset) Address: 3B5h/3D5h (index=02h)
Default: Undefined
Attributes: Read/Write (Group 0 Protection)

Bit	Description
7:0	Horizontal Blanking Start. This register is used to specify the beginning of the horizontal blanking period relative to the beginning of the active display area of a scan line. Horizontal blanking should always be set to start no sooner than after the end of horizontal active. This field should be programmed with a value equal to the number of character clocks that occur on a scan line from the beginning of the active display area to the beginning of the horizontal blanking.



6.6.5 CR03—Horizontal Blanking End Register

I/O (and Memory Offset) Address: 3B5h/3D5h (index=03h)
Default: 1UUU UUUUb (U=Undefined)
Attributes: Read/Write (Group 0 Protection)

7	6	5	4	0
Reserved	Display Enable Skew Control	Horizontal Blanking End Bits 4:0		

Bit	Description
7	Reserved. Values written to this bit are ignored, and to maintain consistency with the VGA standard, a value of 1 is returned when this bit is read. At one time, this bit was used to enable access to certain light pen registers. At that time, setting this bit to 0 provided this access, but setting this bit to 1 was necessary for normal operation.
6:5	Display Enable Skew Control. Defines the degree to which the start and end of the active display area are delayed along the length of a scan line to compensate for internal pipeline delays. These 2 bits describe the delay in terms of a number character clocks. Bit [6:5] Amount of Delay 00 no delay 01 delayed by 1 character clock 10 delayed by 2 character clocks 11 delayed by 3 character clocks
4:0	Horizontal Blanking End Bits [4:0]. This field provides the 5 least significant bits of a 6-bit value that specifies the end of the blanking period relative to its beginning on a single scan line. Bit 7 of the Horizontal Sync End Register (CR05) supplies the most significant bit. This 6-bit value should be programmed to be equal to the least significant 6 bits of the result of adding the length of the blanking period in terms of character clocks to the value specified in the Horizontal Blanking Start Register (CR02). End of blanking should occur before horizontal total.

6.6.6 CR04—Horizontal Sync Start Register

I/O (and Memory Offset) Address: 3B5h/3D5h (index=04h)
Default: Undefined
Attributes: Read/Write (Group 0 Protection)

Bit	Description
7:0	Horizontal Sync Start. This register is used to specify the position of the beginning of the horizontal sync pulse relative to the start of the active display area on a scan line. This field should be set equal to the number of character clocks that occur from beginning of the active display area to the beginning of the horizontal sync pulse on a single scan line. Horizontal sync should always occur between the start and end of horizontal blank. The actual start of sync will also be affected by both the horizontal sync skew register field and whether it is a text or graphics mode.



6.6.7 CR05—Horizontal Sync End Register

I/O (and Memory Offset) Address: 3B5h/3D5h (index=05h)
 Default: 00h
 Attributes: Read/Write (Group 0 Protection)

7	6	5	4	0
Hor Blank End <5>	Horizontal Sync Delay	Horizontal Sync End		

Bit	Description								
7	Horizontal Blanking End Bit 5. This bit provides the most significant bit of a 6-bit value that specifies the end of the horizontal blanking period relative to its beginning. Bits [4:0] of Horizontal Blanking End Register (CR03) supply the 5 least significant bits. See CR03[4:0] for further details. This 6-bit value should be set to the least significant 6 bits of the result of adding the length of the blanking period in terms of character clocks to the value specified in the Horizontal Blanking Start Register (CR02).								
6:5	Horizontal Sync Delay. This field defines the degree to which the start and end of the horizontal sync pulse are delayed to compensate for internal pipeline delays. This capability is supplied to implement VGA compatibility. This field describes the delay in terms of a number character clocks. Bit [6:5] Amount of Delay <table> <tr> <td>00</td><td>no delay</td></tr> <tr> <td>01</td><td>delayed by 1 character clock</td></tr> <tr> <td>10</td><td>delayed by 2 character clocks</td></tr> <tr> <td>11</td><td>delayed by 3 character clocks</td></tr> </table>	00	no delay	01	delayed by 1 character clock	10	delayed by 2 character clocks	11	delayed by 3 character clocks
00	no delay								
01	delayed by 1 character clock								
10	delayed by 2 character clocks								
11	delayed by 3 character clocks								
4:0	Horizontal Sync End. This field provides the 5 least significant bits of a 5-bit value that specifies the end of the horizontal sync pulse relative to its beginning. A value equal to the 5 least significant bits of the horizontal character counter value at which time the horizontal retrace signal becomes inactive (logical 0). Thus, this 5-bit value specifies the width of the horizontal sync pulse. To obtain a retrace signal of W, the following algorithm is used: Value of Horizontal Sync start Register (CR04) + width of horizontal retrace signal in character clock units = 5 bit result to be programmed in this field								

6.6.8 CR06—Vertical Total Register

I/O (and Memory Offset) Address: 3B5h/3D5h (index=06h)
 Default: 00h
 Attributes: Read/Write (Group 0 Protection)

Bit	Description
7:0	Vertical Total Bits [7:0]. This field provides the 8 least significant bits of either a 10-bit or 12-bit value that specifies the total number of scan lines. This includes the scan lines both inside and outside of the active display area. In standard VGA modes, the vertical total is specified with a 10-bit value. The 8 least significant bits of this value are supplied by these 8 bits of this register, and the 2 most significant bits are supplied by bits 5 and 0 of the Overflow Register (CR07).



6.6.9 CR07—Overflow Register (Vertical)

I/O (and Memory Offset) Address: 3B5h/3D5h (index=07h)
 Default: UU0U UUU0b (U=Undefined)
 Attributes: Read/Write (Group 0 Protection on bits [7:5, 3:0])

7	6	5	4	3	2	1	0
Vert Sync Start <9>	Vert Disp Enable <9>	Vert Total <9>	Line Cmp<8>	Vert Blank Start<8>	Vert Sync Start<8>	Vert Display Enable <8>	Vert Total <8>

Bit	Description
7	Vertical Sync Start Bit 9. The vertical sync start is a 10-bit that specifies the beginning of the vertical sync pulse relative to the beginning of the active display area. The 8 least significant bits of this value are supplied by bits [7:0] of the Vertical Sync Start Register (CR10), and the most and second-most significant bits are supplied by this bit and bit 2, respectively, of this register. This 10-bit value should be programmed to be equal to the number of scan lines from the beginning of the active display area to the start of the vertical sync pulse. Since the active display area always starts on the 0th scan line, this number should be equal to the number of the scan line on which the vertical sync pulse begins.
6	Vertical Display Enable End Bit 9. The vertical display enable end is a 10-bit that specifies the number of the last scan line within the active display area. In standard VGA modes, the vertical display enable end is specified with a 10-bit value. The 8 least significant bits of this value are supplied by bits [7:0] of the Vertical Display Enable End Register (CR12), and the most and second-most significant bits are supplied by this bit and bit 1, respectively, of this register. This 10-bit value should be programmed to be equal to the number of the last scan line within in the active display area. Since the active display area always starts on the 0th scan line, this number should be equal to the total number of scan lines within the active display area, minus 1.
5	Vertical Total Bit 9. The vertical total is a 10-bit value that specifies the total number of scan lines. This includes the scan lines both inside and outside of the active display area. The 8 least significant bits of this value are supplied by bits [7:0] of the Vertical Total Register (CR06), and the most and second-most significant bits are supplied by this bit and bit 0, respectively, of this register. This 10-bit value should be programmed equal to the total number of scan lines, minus 2.
4	Line Compare Bit 8. This bit provides the second most significant bit of a 10-bit value that specifies the scan line at which the memory address counter restarts at the value of 0. Bit 6 of the Maximum Scan Line Register (CR09) supplies the most significant bit, and bits 7-0 of the Line Compare Register (CR18) supply the 8 least significant bits. Normally, this 10-bit value is set to specify a scan line after the last scan line of the active display area. When this 10-bit value is set to specify a scan line within the active display area, it causes that scan line and all subsequent scan lines in the active display area to display video data starting at the very first byte of the frame buffer. The result is what appears to be a screen split into a top and bottom part, with the image in the top part being repeated in the bottom part. When used in cooperation with the Start Address High Register (CR0C) and the Start Address Low Register (CR0D), it is possible to create a split display, as described earlier, but with the top and bottom parts displaying different data. The top part will display what data exists in the frame buffer starting at the address specified in the two aforementioned start address registers, while the bottom part will display what data exists in the frame buffer starting at the first byte of the frame buffer.



Bit	Description
3	Vertical Blanking Start Bit 8. The vertical blanking start is a 10-bit that specifies the beginning of the vertical blanking period relative to the beginning of the active display area. The 8 least significant bits of this value are supplied by bits [7:0] of the Vertical Blanking Start Register (CR15), and the most and second-most significant bits are supplied by bit 5 of the Maximum Scan Line Register (CR09) and this bit of this register, respectively. This 10-bit value should be programmed to be equal to the number of scan lines from the beginning of the active display area to the beginning of the blanking period. Since the active display area always starts on the 0th scan line, this number should be equal to the number of the scan line on which the vertical blanking period begins.
2	Vertical Sync Start Bit 8. The vertical sync start is a 10-bit value that specifies the beginning of the vertical sync pulse relative to the beginning of the active display area. The 8 least significant bits of this value are supplied by bits [7:0] of the Vertical Sync Start Register (CR10), and the most and second-most significant bits are supplied by bit 7 and this bit, respectively, of this register. This 10-bit value should be programmed to be equal to the number of scan lines from the beginning of the active display area to the start of the vertical sync pulse. Since the active display area always starts on the 0th scan line, this number should be equal to the number of the scan line on which the vertical sync pulse begins.
1	Vertical Display Enable End Bit 8. The vertical display enable end is a 10-bit value that specifies the number of the last scan line within the active display area. The 8 least significant bits of this value are supplied by bits [7:0] of the Vertical Display Enable End Register (CR12), and the two most significant bits are supplied by bit 6 and this bit, respectively, of this register. This 10-bit or value should be programmed to be equal to the number of the last scan line within in the active display area. Since the active display area always starts on the 0th scan line, this number should be equal to the total number of scan lines within the active display area, minus 1.
0	Vertical Total Bit 8. The vertical total is a 10-bit value that specifies the total number of scan lines. This includes the scan lines both inside and outside of the active display area. The 8 least significant bits of this value are supplied by bits [7:0] of the Vertical Total Register (CR06), and the most and second-most significant bits are supplied by bit 5 and this bit, respectively, of this register. This 10-bit value should be programmed to be equal to the total number of scan lines, minus 2.



6.6.10 CR08—Preset Row Scan Register

I/O (and Memory Offset) Address: 3B5h/3D5h (index=08h)
 Default: 0UUU UUUUb (U=Undefined)
 Attributes: Read/Write

7	6	5	4	0
Reserved (0)	Byte Panning	Starting Row Scan Count		

Bit	Description															
7	Reserved. Read as 0s.															
6:5	Byte Panning. This field holds a 2-bit value that selects number of bytes (up to 3) by which the image is shifted horizontally to the left on the screen. This function is available in both text and graphics modes. In text modes with a 9-pixel wide character box, the image can be shifted up to 27 pixels to the left, in increments of 9 pixels. In text modes with an 8-pixel wide character box, and in all standard VGA graphics modes, the image can be shifted up to 24 pixels to the left, in increments of 8 pixels. When the Nine dot disable bit of the VGA control register is set, the pixel shift will be equivalent to the 8-dot mode. The image can be shifted still further, in increments of individual pixels, through the use of bits [3:0] of the Horizontal Pixel Panning Register (AR13). Number of Pixels Shifted <table><tr><th>Bit [6:5]</th><th>9-Pixel Text</th><th>8-Pixel Text & Graphics</th></tr><tr><td>00</td><td>0</td><td>0</td></tr><tr><td>01</td><td>9</td><td>8</td></tr><tr><td>10</td><td>18</td><td>16</td></tr><tr><td>11</td><td>27</td><td>24</td></tr></table>	Bit [6:5]	9-Pixel Text	8-Pixel Text & Graphics	00	0	0	01	9	8	10	18	16	11	27	24
Bit [6:5]	9-Pixel Text	8-Pixel Text & Graphics														
00	0	0														
01	9	8														
10	18	16														
11	27	24														
4:0	Starting Row Scan Count. This field specifies which horizontal line of pixels within the character boxes of the characters used on the top-most row of text on the display will be used as the top-most scan line. The horizontal lines of pixels of a character box are numbered from top to bottom, with the top-most line of pixels being number 0. If a horizontal line of the character boxes other than the top-most line is specified, then the horizontal lines of the character box above the specified line of the character box will not be displayed as part of the top-most row of text characters on the display. Normally, the value specified by these 5 bits should be 0, so that all of the horizontal lines of pixels within these character boxes will be displayed in the top-most row of text, ensuring that the characters in the top-most row of text do not look as though they have been cut off at the top.															



6.6.11 CR09—Maximum Scan Line Register

I/O (and Memory Offset) Address: 3B5h/3D5h (index=09h)

Default: 00h

Attributes: Read/Write

7	6	5	4	0
Double Scanning	Line Cmp <9>	Vert Blank Start <9>	Starting Row Scan Count	

Bit	Description
7	Double Scanning Enable. 0 = Disable. When disabled, the clock to the row scan counter is equal to the horizontal scan rate. This is the normal setting for many of the standard VGA modes. 1 = Enable. When enabled, the clock to the row scan counter is divided by 2. This is normally used to allow CGA-compatible modes that have only 200 scan lines of active video data to be displayed as 400 scan lines (each scan line is displayed twice).
6	Line Compare Bit 9. This bit provides the most significant bit of a 10-bit value that specifies the scan line at which the memory address counter restarts at the value of 0. Bit 4 of the Overflow Register (CR07) supplies the second most significant bit, and bits 7-0 of the Line Compare Register (CR18) supply the 8 least significant bits. Normally, this 10-bit value is set to specify a scan line after the last scan line of the active display area. When this 10-bit value is set to specify a scan line within the active display area, it causes that scan line and all subsequent scan lines in the active display area to display video data starting at the very first byte of the frame buffer. The result is what appears to be a screen split into a top and bottom part, with the image in the top part being repeated in the bottom part. When used in cooperation with the Start Address High Register (CR0C) and the Start Address Low Register (CR0D), it is possible to create a split display, as described earlier, but with the top and bottom parts displaying different data. The top part will display whatever data exists in the frame buffer starting at the address specified in the two aforementioned start address registers, while the bottom part will display whatever data exists in the frame buffer starting at the first byte of the frame buffer.
5	Vertical Blanking Start Bit 9. The vertical blanking start is a 10-bit value that specifies the beginning of the vertical blanking period relative to the beginning of the active display area. The 8 least significant bits of this value are supplied by bits [7:0] of the Vertical Blanking Start Register (CR15), and the most and second-most significant bits are supplied by this bit and bit 3 of the Overflow Register (CR07), respectively. This 10-bit value should be programmed to be equal to the number of scan line from the beginning of the active display area to the beginning of the blanking period. Since the active display area always starts on the 0th scan line, this number should be equal to the number of the scan line on which the vertical blanking period begins.
4:0	Starting Row Scan Count. This field provides all 5 bits of a 5-bit value that specifies the number of scan lines in a horizontal row of text. This value should be programmed to be equal to the number of scan lines in a horizontal row of text, minus 1.



6.6.12 CR0A—Text Cursor Start Register

I/O (and Memory Offset) Address: 3B5h/3D5h (index=0Ah)
Default: 00UU UUUUb (U=Undefined)
Attributes: Read/Write

7	6	5	4	0
Reserved (00)		Text Cursor Off	Text Cursor Start	

Bit	Description
7:6	Reserved. Read as 0.
5	Text Cursor Off. This cursor is the text cursor that is part of the VGA standard, and should not be confused with the hardware cursor and popup which are intended to be used in extended graphics modes only. This text cursor exists only in text modes, and thus, this register is entirely ignored in graphics modes. 0 = Enables the text cursor. 1 = Disables the text cursor.
4:0	Text Cursor Start. This field specifies which horizontal line of pixels in a character box is to be used to display the first horizontal line of the cursor in text mode. The horizontal lines of pixels in a character box are numbered from top to bottom, with the top-most line being number 0. The value specified by these 5 bits should be the number of the first horizontal line of pixels on which the cursor is to be shown.



6.6.13 CR0B—Text Cursor End Register

I/O (and Memory Offset) Address: 3B5h/3D5h (index=0Bh)
 Default: 0UUU UUUUb (U=Undefined)
 Attributes: Read/Write

7	6	5	4	0
Reserved (0)	Text Cursor Skew	Text Cursor End		

Bit	Description
7	Reserved. Read as 0.
6:5	Text Cursor Skew. This field specifies the degree to which the start and end of each horizontal line of pixels making up the cursor is delayed to compensate for internal pipeline delays. These 2 bits describe the delay in terms of a number character clocks. Bit [6:5] Amount of Delay 00 No delay 01 Delayed by 1 character clock 10 Delayed by 2 character clocks 11 Delayed by 3 character clocks
4:0	Text Cursor End. This field specifies which horizontal line of pixels in a character box is to be used to display the last horizontal line of the cursor in text mode. The horizontal lines of pixels in a character box are numbered from top to bottom, with the top-most line being number 0. The value specified by these 5 bits should be the number of the last horizontal line of pixels on which the cursor is to be shown.

6.6.14 CR0C—Start Address High Register

I/O (and Memory Offset) Address: 3B5h/3D5h (index=0Ch)
 Default: Undefined
 Attributes: Read/Write

Bit	Description
7:0	Start Address Bits [15:8]. This register provides either bits 15 through 8 of a 16-bit value that specifies the memory address offset from the beginning of the frame buffer at which the data to be shown in the active display area begins. (default is 0) In standard VGA modes, the start address is specified with a 16-bit value. The eight bits of this register provide the eight most significant bits of this value, while the eight bits of the Start Address Low Register (CR0D) provide the eight least significant bits.



6.6.15 CR0D—Start Address Low Register

I/O (and Memory Offset) Address: 3B5h/3D5h (index=0Dh)
Default: Undefined
Attributes: Read/Write

Bit	Description
7:0	Start Address Bits [7:0]. This register provides either bits 7 through 0 of a 16-bit value that specifies the memory address offset from the beginning of the frame buffer at which the data to be shown in the active display area begins. (default is 0) In standard VGA modes the start address is specified with a 16-bit value. The eight bits of the Start Address High Register (CR0C) provide the eight most significant bits of this value, while the eight bits of this register provide the eight least significant bits.

6.6.16 CR0E—Text Cursor Location High Register

I/O (and Memory Offset) Address: 3B5h/3D5h (index=0Eh)
Default: Undefined
Attributes: Read/Write

Bit	Description
7:0	Text Cursor Location Bits [15:8]. This field provides the 8 most significant bits of a 16-bit value that specifies the address offset from the beginning of the frame buffer at which the text cursor is located. Bit 7:0 of the Text Cursor Location Low Register (CR0F) provide the 8 least significant bits.

6.6.17 CR0F—Text Cursor Location Low Register

I/O (and Memory Offset) Address: 3B5h/3D5h (index=0Fh)
Default: Undefined
Attributes: Read/Write

Bit	Description
7:0	Text Cursor Location Bits [7:0]. This field provides the 8 least significant bits of a 16-bit value that specifies the address offset from the beginning of the frame buffer at which the text cursor is located. Bits 7:0 of the Text Cursor Location High Register (CR0D) provide the 8 most significant bits.



6.6.18 CR10—Vertical Sync Start Register

I/O (and Memory Offset) Address: 3B5h/3D5h (index=10h)

Default: Undefined

Attributes: Read/Write

Bit	Description
7:0	Vertical Sync Start Bits [7:0]. This register provides the 8 least significant bits of a 10-bit that specifies the beginning of the vertical sync pulse relative to the beginning of the active display area of a screen. In standard VGA modes, this value is described in 10 bits with bits [7,2] of the Overflow Register (CR07) supplying the 2 most significant bits. This 10-bit value should equal the vertical sync start in terms of the number of scan lines from the beginning of the active display area to the beginning of the vertical sync pulse. Since the active display area always starts on the 0th scan line, this number should be equal to the number of the scan line on which the vertical sync pulse begins.

6.6.19 CR11—Vertical Sync End Register

I/O (and Memory Offset) Address: 3B5h/3D5h (index=11h)

Default: 0U00 UUUUb (U=Undefined)

Attributes: Read/Write

7	6	5	4	3	0
Protect Regs 0:7	Reserved	Vert Int Enable	Vert Int Clear	Vertical Sync End	

Bit	Description
7	Protect Registers [0:7]. Note that the ability to write to Bit 4 of the Overflow Register (CR07) is not affected by this bit (i.e., bit 4 of the Overflow Register is always writeable). 0 = Enable writes to registers CR[00:07]. (default) 1 = Disable writes to registers CR[00:07].
6	Reserved. In the VGA standard, this bit was used to switch between 3 and 5 frame buffer refresh cycles during the time required to draw each horizontal line.
5	Vertical Interrupt Enable. This bit is reserved for compatibility only. While this bit may be written or read, its value will have no effect. Note that the VGA does not provide an interrupt signal which would be connected to an input of the system's interrupt controller. Bit 7 of Input Status Register 0 (ST00) originally indicated the status of the vertical retrace interrupt. 0 = Enable the generation of an interrupt at the beginning of each vertical retrace period. 1 = Disable the generation of an interrupt at the beginning of each vertical retrace period.
4	Vertical Interrupt Clear. This is reserved for compatibility only. Note that the VGA does not provide an interrupt signal which would be connected to an input of the system's interrupt controller. 0 = Setting this bit to 0 clears a pending vertical retrace interrupt. This bit must be set back to 1 to enable the generation of another vertical retrace interrupt.
3:0	Vertical Sync End. This 4-bit field provides a 4-bit value that specifies the end of the vertical sync pulse relative to its beginning. This 4-bit value should be set to the least significant 4 bits of the result of adding the length of the vertical sync pulse in terms of the number of scan lines that occur within the length of the vertical sync pulse to the value that specifies the beginning of the vertical sync pulse (see the description of the Vertical Sync Start Register for more details).



6.6.20 CR12—Vertical Display Enable End Register

I/O (and Memory Offset) Address: 3B5h/3D5h (index=12h)
Default: Undefined
Attributes: Read/Write

Bit	Description
7:0	Vertical Display Enable End Bits [7:0]. This register provides the 8 least significant bits of a 10-bit value that specifies the number of the last scan line within the active display area. In standard VGA modes, this value is described in 10 bits with bits [6,1] of the Overflow Register (CR07) supplying the 2 most significant bits. This 10-bit value should be programmed to be equal to the number of the last scan line within in the active display area. Since the active display area always starts on the 0th scan line, this number should be equal to the total number of scan lines within the active display area, minus 1.

6.6.21 CR13—Offset Register

I/O (and Memory Offset) Address: 3B5h/3D5h (index=13h)
Default: Undefined
Attributes: Read/Write

Bit	Description
7:0	Offset Bits [7:0]. This register provides either all 8 bits of an 8-bit value that specifies the number of words or DWords of frame buffer memory occupied by each horizontal row of characters. Whether this value is interpreted as the number of words or DWords is determined by the settings of the bits in the Clocking Mode Register (SR01). In standard VGA modes, the offset is described with an 8-bit value, all the bits of which are provided by this register. This 8-bit value should be programmed to be equal to either the number of words or DWords (depending on the setting of the bits in the Clocking Mode Register, SR01) of frame buffer memory that is occupied by each horizontal row of characters.



6.6.22 CR14—Underline Location Register

I/O (and Memory Offset) Address: 3B5h/3D5h (index=14h)
 Default: 0UUU UUUUb (U=Undefined)
 Attributes: Read/Write

7	6	5	4	0
Reserved (0)	Dword Mode	Count By 4	Underline Location	

Bit	Description															
7	Reserved. Read as 0.															
6	DWord Mode. 0 = Frame buffer addresses are interpreted by the frame buffer address decoder as being either byte addresses or word addresses, depending on the setting of bit 6 of the CRT Mode Control Register (CR17). 1 = Frame buffer addresses are interpreted by the frame buffer address decoder as being DWord addresses, regardless of the setting of bit 6 of the CRT Mode Control Register (CR17). Note that this bit is used in conjunction with bits 6 and 5 of the CRT Mode Control Register (CR17) to select how frame buffer addresses from the CPU are interpreted by the frame buffer address decoder as shown below: <table><tr><th>CR14[6]</th><th>CR17[6]</th><th>Addressing Mode</th></tr><tr><td>0</td><td>0</td><td>Word Mode</td></tr><tr><td>0</td><td>1</td><td>Byte Mode</td></tr><tr><td>1</td><td>0</td><td>DWord Mode</td></tr><tr><td>1</td><td>1</td><td>DWord Mode</td></tr></table>	CR14[6]	CR17[6]	Addressing Mode	0	0	Word Mode	0	1	Byte Mode	1	0	DWord Mode	1	1	DWord Mode
CR14[6]	CR17[6]	Addressing Mode														
0	0	Word Mode														
0	1	Byte Mode														
1	0	DWord Mode														
1	1	DWord Mode														
5	Count By 4. 0 = The memory address counter is incremented either every character clock or every other character clock, depending upon the setting of bit 3 of the CRT Mode Control Register. 1 = The memory address counter is incremented either every 4 character clocks or every 2 character clocks, depending upon the setting of bit 3 of the CRT Mode Control Register. . This is used in mode x13 to allow for using all four planes. Note that this bit is used in conjunction with bit 3 of the CRT Mode Control Register (CR17) to select the number of character clocks are required to cause the memory address counter to be incremented as shown, below: <table><tr><th>CR14[5]</th><th>CR17[3]</th><th>Address Incrementing Interval</th></tr><tr><td>0</td><td>0</td><td>every character clock</td></tr><tr><td>0</td><td>1</td><td>every 2 character clocks</td></tr><tr><td>1</td><td>0</td><td>every 4 character clocks</td></tr><tr><td>1</td><td>1</td><td>every 2 character clocks</td></tr></table>	CR14[5]	CR17[3]	Address Incrementing Interval	0	0	every character clock	0	1	every 2 character clocks	1	0	every 4 character clocks	1	1	every 2 character clocks
CR14[5]	CR17[3]	Address Incrementing Interval														
0	0	every character clock														
0	1	every 2 character clocks														
1	0	every 4 character clocks														
1	1	every 2 character clocks														
4:0	Underline Location. This field specifies which horizontal line of pixels in a character box is to be used to display a character underline in text mode. The horizontal lines of pixels in a character box are numbered from top to bottom, with the top-most line being number 0. The value specified by these 5 bits should be the number of the horizontal line on which the character underline mark is to be shown.															



6.6.23 CR15—Vertical Blanking Start Register

I/O (and Memory Offset) Address: 3B5h/3D5h (index=15h)
Default: Undefined
Attributes: Read/Write

Bit	Description
7:0	Vertical Blanking Start Bits [7:0]. This register provides the 8 least significant bits of a 10-bit value that specifies the beginning of the vertical blanking period relative to the beginning of the active display area of the screen. In standard VGA modes, the vertical blanking start is specified with a 10-bit value. The most and second-most significant bits of this value are supplied by bit 5 of the Maximum Scan Line Register (CR09) and bit 3 of the Overflow Register (CR07), respectively. This 10-bit value should be programmed to be equal the number of scan lines from the beginning of the active display area to the beginning of the vertical blanking period. Since the active display area always starts on the 0th scan line, this number should be equal to the number of the scan line on which vertical blanking begins.

6.6.24 CR16—Vertical Blanking End Register

I/O (and Memory Offset) Address: 3B5h/3D5h (index=16h)
Default: Undefined
Attributes: Read/Write

This register provides a 8-bit value that specifies the end of the vertical blanking period relative to its beginning.

Bit	Description
7:0	Vertical Blanking End Bits [7:0]. This 8-bit value should be set equal to the least significant 8 bits of the result of adding the length of the vertical blanking period in terms of the number of scan lines that occur within the length of the vertical blanking period to the value that specifies the beginning of the vertical blanking period (see the description of the Vertical Blanking Start Register for details).



6.6.25 CR17—CRT Mode Control

I/O (and Memory Offset) Address: 3B5h/3D5h (index=17h)
 Default: 0UU0 UUUUb (U=Undefined)
 Attributes: Read/Write

7	6	5	4	3	2	1	0
CRT Ctrl Reset	Word or Byte Mode	Address Wrap	Reserved (0)	Count By 2	Horizontal Retrace Select	Select Row Scan Cntr	Compat Mode Support

Bit	Description															
7	CRT Controller Reset. This bit has no effect except in native VGA modes (non-centered). 0 = Forces horizontal and vertical sync signals to be inactive. No other registers or outputs are affected. 1 = Permits normal operation.															
6	Word Mode or Byte Mode. 0 = The memory address counter's output bits are shifted by 1 bit position before being passed on to the frame buffer address decoder such that they are made into word-aligned addresses when bit 6 of the Underline Location Register (CR17) is set to 0. 1 = The memory address counter's output bits remain unshifted before being passed on to the frame buffer address decoder such that they remain byte-aligned addresses when bit 6 of the Underline Location Register (CR17) is set to 0. Note that this bit is used in conjunction with bits 6 and 5 of the CRT Mode Control Register (CR17) to control how frame buffer addresses from the memory address counter are interpreted by the frame buffer address decoder as shown below: <table><tr><th>CR14[6]</th><th>CR17[6]</th><th>Addressing Mode</th></tr><tr><td>0</td><td>0</td><td>Word Mode—Addresses from the memory address counter are shifted once to become word-aligned</td></tr><tr><td>0</td><td>1</td><td>Byte Mode—Addresses from the memory address counter are not shifted</td></tr><tr><td>1</td><td>0</td><td>DWord Mode—Addresses from the memory address counter are shifted twice to become DWord-aligned</td></tr><tr><td>1</td><td>1</td><td>DWord Mode—Addresses from the memory address counter are shifted twice to become DWord-aligned</td></tr></table>	CR14[6]	CR17[6]	Addressing Mode	0	0	Word Mode—Addresses from the memory address counter are shifted once to become word-aligned	0	1	Byte Mode—Addresses from the memory address counter are not shifted	1	0	DWord Mode—Addresses from the memory address counter are shifted twice to become DWord-aligned	1	1	DWord Mode—Addresses from the memory address counter are shifted twice to become DWord-aligned
CR14[6]	CR17[6]	Addressing Mode														
0	0	Word Mode—Addresses from the memory address counter are shifted once to become word-aligned														
0	1	Byte Mode—Addresses from the memory address counter are not shifted														
1	0	DWord Mode—Addresses from the memory address counter are shifted twice to become DWord-aligned														
1	1	DWord Mode—Addresses from the memory address counter are shifted twice to become DWord-aligned														
5	Address Wrap. Note that this bit is only effective when word mode is made active by setting bit 6 in both the Underline Location Register and this register to 0. 0 = Wrap frame buffer address at 16 KB. This is used in CGA-compatible modes. 1 = No wrapping of frame buffer addresses.															
4	Reserved. Read as 0.															



Bit	Description															
3	Count By 2. This bit is used in conjunction with bit 5 of the Underline Location Register (CR14) to select the number of character clocks are required to cause the memory address counter to be incremented. 0 = The memory address counter is incremented either every character clock or every 4 character clocks, depending upon the setting of bit 5 of the Underline Location Register. 1 = The memory address counter is incremented either every other clock. <table><tr><th>CR14[5]</th><th>CR17[3]</th><th>Address Incrementing Interval</th></tr><tr><td>0</td><td>0</td><td>every character clock</td></tr><tr><td>0</td><td>1</td><td>every 2 character clocks</td></tr><tr><td>1</td><td>0</td><td>every 4 character clocks</td></tr><tr><td>1</td><td>1</td><td>every 2 character clocks</td></tr></table>	CR14[5]	CR17[3]	Address Incrementing Interval	0	0	every character clock	0	1	every 2 character clocks	1	0	every 4 character clocks	1	1	every 2 character clocks
CR14[5]	CR17[3]	Address Incrementing Interval														
0	0	every character clock														
0	1	every 2 character clocks														
1	0	every 4 character clocks														
1	1	every 2 character clocks														
2	Horizontal Retrace Select. This bit provides a way of effectively doubling the vertical resolution by allowing the vertical timing counter to be clocked by the horizontal retrace clock divided by 2 (usually, it would be undivided). 0 = The vertical timing counter is clocked by the horizontal retrace clock. 1 = The vertical timing counter is clocked by the horizontal retrace clock divided by 2.															
1	Select Row Scan Counter. 0 = A substitution takes place, where bit 14 of the 16-bit memory address generated of the memory address counter (after the stage at which these 16 bits may have already been shifted to accommodate word or DWord addressing) is replaced with bit 1 of the row scan counter at a stage just before this address is presented to the frame buffer address decoder. 1 = No substitution takes place. See following tables.															
0	Compatibility Mode Support. 0 = A substitution takes place, where bit 13 of the 16-bit memory address generated of the memory address counter (after the stage at which these 16 bits may have already been shifted to accommodate word or DWord addressing) is replaced with bit 0 of the row scan counter at a stage just before this address is presented to the frame buffer address decoder. 1 = No substitution takes place. See following tables.															

The following tables show the possible ways in which the address bits from the memory address counter can be shifted and/or reorganized before being presented to the frame buffer address decoder. First, the address bits generated by the memory address counter are reorganized, if need be, to accommodate byte, word or DWord modes. The resulting reorganized outputs (MAOut15-MAOut0) from the memory address counter may also be further manipulated with the substitution of bits from the row scan counter (RSOut1 and RSOut0) before finally being presented to the input bits of the frame buffer address decoder (FBIn15-FBIn0).



Table 6-2. Memory Address Counter Address Bits [15:0]

	Byte Mode CR14 bit 6=0 CR17 bit 6=1 CR17 bit 5=X	Word Mode CR14 bit 6=0 CR17 bit 6=0 CR17 bit 5=1	Word Mode CR14 bit 6=0 CR17 bit 6=0 CR17 bit 5=0	DWord Mode CR14 bit 6=1 CR17 bit 6=X CR17 bit 5=X
MAOut0	0	15	13	12
MAOut1	1	0	0	13
MAOut2	2	1	1	0
MAOut3	3	2	2	1
MAOut4	4	3	3	2
MAOut5	5	4	4	3
MAOut6	6	5	5	4
MAOut7	7	6	6	5
MAOut8	8	7	7	6
MAOut9	9	8	8	7
MAOut10	10	9	9	8
MAOut11	11	10	10	9
MAOut12	12	11	11	10
MAOut13	13	12	12	11
MAOut14	14	13	13	12
MAOut15	15	14	14	13

X = Don't Care

Table 6-3. Frame Buffer Address Decoder

	CR17 bit 1=1	CR17 bit 1=1	CR17 bit 1=0	CR17 bit 1=0
	CR17 bit 0=1	CR17 bit 0=0	CR17 bit 0=1	CR17 bit 0=0
FBIn0	MAOut0	MAOut0	MAOut0	MAOut0
FBIn1	MAOut1	MAOut1	MAOut1	MAOut1
FBIn2	MAOut2	MAOut2	MAOut2	MAOut2
FBIn3	MAOut3	MAOut3	MAOut3	MAOut3
FBIn4	MAOut4	MAOut4	MAOut4	MAOut4
FBIn5	MAOut5	MAOut5	MAOut5	MAOut5
FBIn6	MAOut6	MAOut6	MAOut6	MAOut6
FBIn7	MAOut7	MAOut7	MAOut7	MAOut7
FBIn8	MAOut8	MAOut8	MAOut8	MAOut8
FBIn9	MAOut9	MAOut9	MAOut9	MAOut9
FBIn10	MAOut10	MAOut10	MAOut10	MAOut10
FBIn11	MAOut11	MAOut11	MAOut11	MAOut11
FBIn12	MAOut12	MAOut12	MAOut12	MAOut12
FBIn13	MAOut13	MAOut13	RSOut0	RSOut0
FBIn14	MAOut14	RSOut1	MAOut14	RSOut1
FBIn15	MAOut15	MAOut15	MAOut15	MAOut15



6.6.26 CR18—Line Compare Register

I/O (and Memory Offset) Address: 3B5h/3D5h (index=18h)
Default: Undefined
Attributes: Read/Write

Bit	Description
7:0	Line Compare Bits [7:0]. This register provides the 8 least significant bits of a 10-bit value that specifies the scan line at which the memory address counter restarts at the value of 0. Bit 6 of the Maximum Scan Line Register (CR09) supplies the most significant bit, and bit 4 of the Overflow Register (CR07) supplies the second most significant bit. Normally, this 10-bit value is set to specify a scan line after the last scan line of the active display area. When this 10-bit value is set to specify a scan line within the active display area, it causes that scan line and all subsequent scan lines in the active display area to display video data starting at the very first byte of the frame buffer. The result is what appears to be a screen split into a top and bottom part, with the image in the top part being repeated in the bottom part. (This register is only used in split screening modes, and this is not a problem because split screening is not actually used for extended modes. As a result, there is no benefit to extending the existing overflow bits for higher resolutions.) When used in cooperation with the Start Address High Register (CR0C) and the Start Address Low Register (CR0D), it is possible to create a split display, as described earlier, but with the top and bottom parts displaying different data. The top part will display whatever data exists in the frame buffer starting at the address specified in the two aforementioned start address registers, while the bottom part will display whatever data exists in the frame buffer starting at the first byte of the frame buffer.

6.6.27 CR22—Memory Read Latch Data Register

I/O (and Memory Offset) Address: 3B5h/3D5h (index=22h)
Default: 00h
Attributes: Read-Only

Bit	Description
7:0	Memory Read Latch Data. This field provides the value currently stored in 1 of the four memory read latches. Bits 1 and 0 of the Read Map Select Register (GR04) select which of the four memory read latches may be read via this register.



6.6.28 CR24— Test Register for Toggle State of Attribute Controller Register

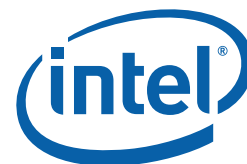
I/O (and Memory Offset) Address: 3B5h/3D5h (index=24h)

Default: 00h

Attributes: Read-Only

7	6	0
Toggle Status	Reserved (0000000)	

Bit	Description
7	Toggle Status. Indicates where the last write to attribute register was to: 0 = index port 1 = data port
6:0	Reserved. Read as 0.



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

Programmer's Reference Manual (PRM)

Volume 4: Subsystem and Cores

January 2008

Revision 1.0d

Technical queries: ilg@linux.intel.com

www.intellinuxgraphics.org



[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.

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, or life sustaining applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

The Intel® 965 Express Chipset family and Intel® G35 Express Chipset may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

I2C is a two-wire communications bus/protocol developed by Philips. SMBus is a subset of the I2C bus/protocol and was developed by Intel. Implementations of the I2C bus/protocol may require licenses from various entities, including Philips Electronics N.V. and North American Philips Corporation.

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

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

Copyright © 2008, Intel Corporation. All rights reserved.



Contents

1	Introduction	19
1.1	Notations and Conventions	21
1.1.1	Reserved Bits and Software Compatibility	21
1.2	Terminology	21
2	Subsystem Overview	33
2.1	Introduction	33
2.2	Subsystem Topology	33
2.3	Execution Units (EUs)	34
2.4	Thread Dispatching	34
2.5	Shared Functions	35
2.6	Messages	36
2.6.1	Message Register File (MRF)	38
2.6.2	Send Instruction	38
2.6.3	Creating and Sending a Message	39
2.6.4	Message Payload Containing a Header	40
2.6.5	Writebacks	40
2.6.6	Message Delivery Ordering Rules	40
2.6.7	Execution Mask and Messages	41
2.6.8	End-Of-Thread (EOT) Message	41
2.6.9	Performance	42
2.6.10	Message Description Syntax	43
2.6.11	Message Errors	43
3	Debugging	45
3.1	Introduction	45
3.2	The Snapshot Mechanism	46
3.2.1	Debug Trigger Counters	47
3.3	Fixed Function Debug Process	47
3.3.1	Overview	47
3.3.2	Vertex Shader Debug	48
3.3.3	SVG Debug	49
3.3.3.1	SVG_CTL —Debug Control	49
3.3.3.2	SVG_RDATA—Debug Return Data	49
3.3.3.3	SVG_WORK_CTL—Debug Workaround Control	50
3.3.4	Vertex Fetch	50
3.3.4.1	VF_CTL—Debug Control	51
3.3.4.2	VF_STRG_VAL—Debug Snapshot Trigger Value	52
3.3.4.3	VF_STR_VL_OVR —Debug Start Vertex Location Override	52
3.3.4.4	VF_VC_OVR —Debug Vertex Count Override	52
3.3.4.5	VF_STR_PSKIP —Debug Starting Primitives Skipped	53
3.3.4.6	VF_MAX_PRIM —Debug Max Primitives	53
3.3.4.7	VF_RDATA —Debug Return Data	53
3.3.5	Vertex Shader	54
3.3.5.1	VS_CTL —Debug Control	55
3.3.5.2	VS_STRG_VAL—Debug Snapshot Trigger Value	55



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



3.4.3	Instruction / State Cache.....	82
3.4.3.1	ISC_CTL—Instruction / State Debug Control	82
3.4.4	Instruction L1 Cache.....	83
3.4.4.1	ISC_L1CA_CTR—Instruction L1 Cache Debug Control	83
3.4.4.2	ISC_L1CA_RDATA—Instruction L1 Cache Debug Return Data	83
3.4.4.3	ISC_L1CA_BP_ADR1—Instruction L1 Cache Breakpoint Address 1 Control	84
3.4.4.4	ISC_L1CA_BP_ADR2—Instruction L1 Cache Breakpoint Address 2 Control	84
3.4.4.5	ISC_L1CA_BP_OPC1—Instruction L1 Cache Breakpoint Opcode 1 Control	85
3.4.4.6	ISC_L1CA_BP_OPC2—Instruction L1 Cache Breakpoint Opcode 2 Control	85
3.4.5	Message Arbiter	86
3.4.5.1	MA_DEBUG_1—Message Arbiter Debug Control	87
3.4.6	Sampler.....	88
3.4.6.1	SAMPLER_CTL—Sampler Debug Control	88
3.4.6.2	SAMPLER_RDATA—Sampler Debug Return Data	89
3.4.7	Data Port	89
3.4.7.1	DP_CTL—Data Port Debug Control.....	89
3.4.7.2	DP_RDATA—Data Port Debug Return Data.....	90
3.4.8	Render Cache	91
3.4.8.1	RC_CTL—RC Debug Control.....	91
3.4.8.2	RC_DEF_CLR—RC Debug Force Default Color.....	91
3.4.8.3	RC_RDATA—RC Debug Return Data.....	92
3.4.9	Unified Return Buffer (URB)	92
3.4.9.1	URB_CTL—URB Debug Control.....	92
3.4.9.2	URB_RDATA—URB Debug Return Data	93
3.4.10	Thread Spawner (TS).....	94
3.5	Attention Signaling from EU to Host	94
3.5.1	EU_CTL—EU Debug Control	95
3.5.2	EU_ATT—EU Debug Attention	95
3.5.3	EU_ATT_DATA—EU Debug Attention Data	95
3.5.4	EU_ATT_CLR—EU Debug Attention Clear.....	96
3.5.5	EU_RDATA—EU Debug Return Data	96
3.6	Breakpoints.....	97
3.6.1	Single Stepping.....	97
3.6.2	Modification of Instruction Stream	98
3.7	Message Errors	98
3.7.1.1	Error-Types Visible to the Arb.....	98
3.7.1.2	Non-pipelined S/F Operation.....	99
4	Sampling Engine.....	101
4.1	Texture Coordinate Processing	102
4.1.1	Texture Coordinate Normalization	102
4.1.2	Texture Coordinate Computation	102
4.2	Texel Address Generation	103
4.2.1	Level of Detail Computation (Mipmapping)	104
4.2.1.1	Base Level Of Detail (LOD).....	104
4.2.1.2	LOD Bias	105
4.2.1.3	LOD Pre-Clamping	105
4.2.1.4	Min/Mag Determination	105
4.2.1.5	LOD Computation Pseudocode	106



4.2.2	Inter-Level Filtering Setup	107
4.2.3	Intra-Level Filtering Setup	107
4.2.3.1	MAPFILTER_NEAREST	108
4.2.3.2	MAPFILTER_LINEAR	108
4.2.3.3	MAPFILTER_ANISOTROPIC	109
4.2.3.4	MAPFILTER_MONO	110
4.2.4	Texture Address Control	112
4.2.4.1	TEXCOORDMODE_WRAP Mode	113
4.2.4.2	TEXCOORDMODE_MIRROR Mode	113
4.2.4.3	TEXCOORDMODE_MIRROR_ONCE Mode	113
4.2.4.4	TEXCOORDMODE_CLAMP Mode	114
4.2.4.5	TEXCOORDMODE_CLAMPBORDER Mode	114
4.2.4.6	TEXCOORDMODE_CUBE Mode	114
4.3	Texel Fetch	115
4.3.1	Texel Chroma Keying	115
4.3.1.1	Chroma Key Testing	115
4.3.1.2	Chroma Key Effects	116
4.4	Shadow Prefilter Compare	116
4.5	Texel Filtering	117
4.6	Texel Color Gamma Linearization	117
4.7	State	118
4.7.1	BINDING_TABLE_STATE	118
4.7.2	SURFACE_STATE	118
4.7.2.1	For Most Messages	119
4.7.3	SAMPLER_STATE	142
4.7.3.1	For Most Messages	142
4.7.4	SAMPLER_BORDER_COLOR_STATE	153
4.7.5	3DSTATE_CHROMA_KEY	154
4.7.6	3DSTATE_SAMPLER_PALETTE_LOAD0	156
4.8	Messages	157
4.8.1	Initiating Message	157
4.8.1.1	Message Descriptor	158
4.8.1.2	Message Header	159
4.8.1.3	Payload Parameter Definition	160
4.8.1.4	Message Types	162
4.8.1.5	Parameter Types	165
4.8.1.6	SIMD16 Payload	166
4.8.1.7	SIMD8 Payload	167
4.8.1.8	SIMD4x2 Payload	167
4.8.2	Writeback Message	168
4.8.2.1	SIMD16	168
4.8.2.2	SIMD8	169
4.8.2.3	SIMD4x2	170
5	Data Port	171
5.1	Cache Agents	171
5.1.1	Render Cache	172
5.1.2	Data Cache	172
5.1.3	Sampler Cache	172
5.2	Surfaces	172
5.2.1	Surface State Model	173
5.2.2	Stateless Model	173
5.3	Write Commit	173



5.4	Read/Write Ordering	174
5.5	Accessing Buffers	174
5.6	Accessing Media Surfaces	175
5.6.1	Boundary Behavior	175
5.7	Accessing Render Targets	176
5.7.1	Single Source	176
5.7.2	Dual Source [DevCL-B]	176
5.7.3	Replicate Data	177
5.7.4	Multiple Render Targets (MRT)	177
5.8	Flushing the Render Cache	177
5.9	State	177
5.9.1	BINDING_TABLE_STATE	177
5.9.2	SURFACE_STATE	177
5.10	Messages	178
5.10.1	Global Definitions	178
5.10.1.1	Message Descriptor	179
5.10.1.2	Message Header	180
5.10.1.3	Write Commit Writeback Message	181
5.10.2	OWord Block Read/Write	181
5.10.2.1	Message Descriptor	182
5.10.2.2	Message Payload (Write)	183
5.10.2.3	Writeback Message (Read)	183
5.10.3	OWord Dual Block Read/Write	184
5.10.3.1	Message Descriptor	184
5.10.3.2	Message Payload	185
5.10.3.3	Additional Message Payload (Write)	185
5.10.3.4	Writeback Message (Read)	186
5.10.4	Media Block Read/Write	186
5.10.4.1	Message Descriptor	188
5.10.4.2	Message Header	189
5.10.4.3	Message Payload (Write)	191
5.10.4.4	Writeback Message (Read)	191
5.10.5	DWord Scattered Read/Write	192
5.10.5.1	Message Descriptor	193
5.10.5.2	Message Payload	193
5.10.5.3	Additional Message Payload (Write)	194
5.10.5.4	Writeback Message (Read)	195
5.10.6	Render Target Write	196
5.10.6.1	Subspan/Pixel to Slot Mapping	197
5.10.6.2	Message Descriptor	197
5.10.6.3	Message Header	198
5.10.6.4	Stencil and Antialias Alpha Payload	200
5.10.6.5	Color Payload: SIMD8 Single Source	203
5.10.6.6	Color Payload: SIMD16 Replicated Data	204
5.10.6.7	Color Payload: SIMD8 Dual Source [DevCL-B]	205
5.10.6.8	Depth Payload	206
5.10.6.9	Message Sequencing Summary	207
5.10.7	Flush Render Cache	209
5.10.7.1	Message Descriptor	209
5.10.7.2	Message Payload	209
6	Extended Math	211
6.1	Messages	212
6.1.1	Initiating Message	212



	6.1.1.1	Message Descriptor	212
	6.1.1.2	Scalar and Vector Mode	213
	6.1.1.3	Message Payload	215
	6.1.2	Writeback Message	216
6.2		Performance	217
6.3		Function Reference	218
	6.3.1	INV	218
	6.3.2	LOG	218
	6.3.3	EXP	219
	6.3.4	SQRT	221
	6.3.5	RSQ	222
	6.3.6	POW	222
	6.3.7	SIN	224
	6.3.8	COS	225
	6.3.9	SINCOS	226
	6.3.10	INT DIV	227
7		Message Gateway	230
	7.1	Messages	230
	7.1.1	Message Descriptor	231
	7.1.2	OpenGateway Message	231
	7.1.2.1	Message Payload	232
	7.1.2.2	Writeback Message	233
	7.1.3	CloseGateway Message	233
	7.1.3.1	Message Payload	233
	7.1.3.2	Writeback Message	234
	7.1.4	ForwardMsg Message	234
	7.1.4.1	Message Payload	235
	7.1.4.2	Writeback Message to Requester Thread	236
	7.1.4.3	Writeback Message to Recipient Thread	236
	7.1.5	GetTimeStamp Message	237
	7.1.5.1	Message Payload	237
	7.1.5.2	Writeback Message to Requester Thread	238
8		Unified Return Buffer (URB)	240
	8.1	URB Size	240
	8.2	URB Access	240
	8.3	State	241
	8.4	Messages	241
	8.4.1	Execution Mask	241
	8.4.2	Message Descriptor	242
	8.4.3	URB_WRITE	245
	8.4.3.1	URB_WRITE Message Header	245
	8.4.3.2	URB_WRITE Message Payload	246
	8.4.3.3	Writeback Message for URB Entry Allocate	250
9		Execution Unit ISA	252
	9.1	Introduction	252
	9.1.1	Objective and Scope	252
	9.1.2	Terms and Acronyms	252
	9.1.3	Formats and Conventions	255
10		EU Data Types	258
	10.1	Fundamental Data Types	258



10.2	Numerical Data Types	259
10.2.1	Unsigned Integers	260
10.2.2	Signed Integers	260
10.2.3	Single Precision Floating-Point Numbers	261
10.2.4	Packed Signed Half-Byte Integer Vector.....	261
10.2.5	Packed 8-bit Restricted Float Vector	262
10.3	Floating Point Modes.....	264
10.3.1	IEEE Floating Point Mode.....	264
10.3.1.1	Partial Listing of Honored IEEE-754 Rules	264
10.3.1.2	Complete Listing of Deviations or Additional Requirements vs. IEEE-754.....	265
10.3.1.3	Comparison of Floating Point Numbers.....	266
10.3.1.4	Min/Max of Floating Point Numbers.....	268
10.3.2	Alternative Floating Point Mode	272
10.4	Type Conversion	274
10.4.1	Float to Integer.....	274
10.4.2	Integer to Integer with Same or Higher Precision	274
10.4.3	Integer to Integer with Lower Precision	275
10.4.4	Integer to Float.....	275
11	Execution Environment	276
11.1	Overview	276
11.2	Primary Usage Models.....	277
11.2.1	AOS and SOA Data Structures.....	277
11.2.2	SIMD4 Mode of Operation.....	278
11.2.3	SIMD4x2 Mode of Operation	279
11.2.4	SIMD16 Mode of Operation	281
11.2.5	SIMD8 Mode of Operation.....	282
11.3	Registers and Register Regions.....	283
11.3.1	Register Files.....	283
11.3.2	GRF Registers	283
11.3.3	MRF Registers.....	284
11.3.4	ARF Registers	286
11.3.4.1	Overview.....	286
11.3.4.2	Access Granularity	287
11.3.4.3	Null Register	287
11.3.4.4	Address Register.....	288
11.3.4.5	Accumulator Registers.....	291
11.3.4.6	Flag Register.....	294
11.3.4.7	Mask Registers	295
11.3.4.8	Mask Stack Register	297
11.3.4.9	Mask Stack Depth Register.....	299
11.3.4.10	State Registers.....	302
11.3.4.11	Control Register	304
11.3.4.12	Notification Registers	312
11.3.4.13	IP Register.....	314
11.3.5	Immediate	315
11.3.6	Region Parameters	316
11.3.7	Region Addressing Modes	321
11.3.7.1	Direct Register Addressing	322
11.3.7.2	Register-indirect Register Addressing with a 1x1 Index Region	323
11.3.7.3	Register-indirect Register Addressing with a Vx1 Index Region	324



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



13.2	Instruction Machine Formats	366
13.2.1	Common Instruction Fields	368
13.2.2	Instruction Operation Doubleword (DW0).....	374
13.2.3	Instruction Destination Doubleword (DW1)	379
13.2.4	Instruction Source-0 Doubleword (DW2).....	383
13.2.5	Instruction Source-1 Doubleword (DW3).....	387
13.3	Opcode Encoding.....	393
13.3.1	Move and Logic Instructions.....	393
13.3.2	Flow Control Instructions.....	395
13.3.3	Miscellaneous Instructions	396
13.3.4	Parallel Arithmetic Instructions	397
13.3.5	Vector Arithmetic Instructions	398
13.3.6	Special Instructions	399
13.4	Native Instruction BNF	400
13.4.1	Instruction Groups.....	400
13.4.2	Destination Register	401
13.4.3	Source Register	402
13.4.4	Address Registers.....	403
13.4.5	Register Files and Register Numbers	403
13.4.6	Relative Location and Stack Control	404
13.4.7	Regions	404
13.4.8	Types	405
13.4.9	Write Mask	405
13.4.10	Swizzle Control	405
13.4.11	Immediate Values	405
13.4.12	Predication and Modifiers.....	406
13.4.13	Instruction Options.....	407
13.5	Deprecated Features.....	408
13.5.1	Defeatured Instructions.....	408
13.5.2	Others	408
14	Instruction Set Reference	410
14.1	Conventions	410
14.1.1	Pseudo Code Format	410
14.1.2	General Macros and Definitions	410
14.1.3	Mask Stack Operations.....	411
14.2	Instruction Description	414
14.2.1	add – Addition	415
14.2.2	and – Logical And.....	417
14.2.3	asr – Arithmetic Shift Right.....	419
14.2.4	avg – Average	421
14.2.5	break – Break	422
14.2.6	cmp – Compare	425
14.2.7	cmpn – Compare NaN	427
14.2.8	cont – Continue	429
14.2.9	do – Do	431
14.2.10	dp2 – Dot Product 2.....	432
14.2.11	dp3 – Dot Product 3.....	434
14.2.12	dp4 – Dot Product 4.....	436
14.2.13	dph –Dot Product Homogeneous.....	438
14.2.14	else.....	440
14.2.15	endif – End-If	442
14.2.16	frc – Fraction	444
14.2.17	halt – Halt.....	446



14.2.18	if – If.....	448
14.2.19	iff – Fast-If.....	450
14.2.20	jmpj – Jump Indexed	452
14.2.21	line – Line	454
14.2.22	lzd – Leading Zero Detection.....	456
14.2.23	mac – Multiply Accumulate	458
14.2.24	mach – Multiply Accumulate High	459
14.2.25	mov – Move	461
14.2.26	mrest – Mask Restore	462
14.2.27	msave – Mask Save	463
14.2.28	mul – Multiply.....	464
14.2.29	nop – No Operation	467
14.2.30	not – Logic Not	468
14.2.31	or – Logic Or	470
14.2.32	pop – Mask Stack Pop	472
14.2.33	push – Mask Stack Push	473
14.2.34	rndd – Round Down	474
14.2.35	rndu – Round Up.....	476
14.2.36	rnde – Round to Even	478
14.2.37	rndz – Round to Zero	480
14.2.38	sad2 – Sum of Absolute Difference 2	482
14.2.39	sada2 – Sum of Absolute Difference Accumulate 2	484
14.2.40	sel – Select	486
14.2.41	send – Send Message	488
14.2.42	shl – Shift Left	496
14.2.43	shr – Shift Right.....	497
14.2.44	wait – Wait Notification	498
14.2.45	while – While.....	500
14.2.46	xor – Logic Xor	502
15	EU Programming Guide.....	504
15.1	Assembler Pragmas	504
15.1.1	Declarations	504
15.1.2	Defaults and Defines.....	505
15.1.3	Example Pragma Usages	506
15.1.4	Assembly Programming Guideline	507
15.2	Usage Examples.....	509
15.2.1	Vector Immediate.....	509
15.2.1.1	Supporting Pixel Shader Indexing.....	509
15.2.1.2	Supporting OpenGL Vertex Shader Instruction SWZ	510
15.2.2	Destination Mask for DP4 and Destination Dependency Control	511
15.2.3	Null Register as the Destination.....	512
15.2.4	Use of LINE Instruction	513
15.2.5	Mask for SEND Instruction.....	515
15.2.5.1	Channel Enables for Extended Math Unit	515
15.2.5.2	Channel Enables for Scratch Memory	517
15.2.6	Flow Control Instructions.....	520
15.2.7	Execution Masking.....	521
15.2.7.1	Branching	521
15.2.7.2	Fast-If.....	521
15.2.7.3	Cascade Branching.....	522
15.2.7.4	Compound Branches	522
15.2.7.5	Looping.....	523
15.2.7.6	Indexed Jump	526





Figures

Figure 2-1. Data Flow Associated With Messages	37
Figure 4-1. Normalized vs. Unnormalized Texture Coordinates.....	102
Figure 4-2. Cube Map Coordinate Computation Example	103
Figure 4-3. Bilinear Filter Sampling.....	109
Figure 4-4. Sampling Using MAPFILTER_MONO.....	111
Figure 4-5. Texture Wrap vs. Mirror Addressing Mode.....	113
Figure 4-6. Texture Clamp Mode	114
Figure 10-1. Fundamental data types.....	258
Figure 10-2. Integer numerical data types	259
Figure 10-3. Floating point numerical data types	259
Figure 10-4. Converting a Packed Half-byte Vector to a 128-bit Signed Integer Vector.....	261
Figure 10-5. Conversion from a Restricted 8-bit Float to a Single-Precision Float ...	262
Figure 10-6. Converting a Packed Restricted Float Vector to a 128-bit Float Vector	264
Figure 11-1. AOS and SOA data structures	277
Figure 11-2. A SIMD4 Example	279
Figure 11-3. SIMD4x2 Examples with Different Emasks	280
Figure 11-4. A SIMD4x2 Example with a Constant Vector Shared by Two Program Flows	280
Figure 11-5. A SIMD16 Example.....	281
Figure 11-6. Another SIMD16 Example with an AOS Shared Constant.....	282
Figure 11-7. Format of the Mask Stack Register	299
Figure 11-8. Format of the Mask Stack Depth Register	301
Figure 11-9. Conventional SIMD Instruction Sequence.....	317
Figure 11-10. GEN4 SIMD Instruction Sequence for the Same Program.....	318
Figure 11-11. An example of a register region ($r4.1<16;8,2>:w$) with 16 elements	320
Figure 11-12. A 16-element register region with interleaved rows ($r5.0<1;8,2>:w$)	320
Figure 11-13. A region description example in direct register addressing.....	322
Figure 11-14. A region description example in direct register addressing with $\langle src0 \rangle$ as a vector of replicated scalars	323
Figure 11-15. An example illustrating register-indirect register addressing mode with a 1x1 index region.....	324
Figure 11-16. An example illustrating register-indirect-register addressing mode with a Vx1 index region (Src0).....	325
Figure 11-17. An example illustrating register-indirect register addressing mode with a VxH index region (Src0).....	326
Figure 11-18. Generation of predication mask	334
Figure 11-19. Direct addressed vector operands in a compressed instruction	341
Figure 11-20. A packed-word operand in a dword SIMD16 compressed instruction .	342
Figure 11-21. Dword and word (or byte) scalar source in a compressed instruction.	343
Figure 11-22. Indirect-addressed source/destination operand in a compressed instruction	343
Figure 15-1. Pixel Shader example using vector immediate	510



Tables

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



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



Revision History

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

§§





1 Introduction

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

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

Table 1-1. Supported Chipsets

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

NOTES:

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

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



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

- **PRM, Volume 1: Graphics Core**

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

The volume also contains a chapter on the Graphics Processing Engine (GPE). The GPE is a collective term for 3D, Media, the subsystem, and the parts of the memory interface that are used by these units. Display, blitter and their memory interfaces are *not* included in the GPE.

- **PRM, Volume 2; 3D / Media**

Volume 2 covers the 3D and Media pipelines in detail. This volume is where details for all of the "fixed functions" are covered, including commands processed by the pipelines, fixed-function state structures, and a definition of the inputs (payloads) and outputs of the threads spawned by these units.

This volume also covers the single Media Fixed Function, VLD. It describes how to initiate generic threads using the thread spawner (TS). It is generic threads which will be used for doing the majority of media functions. Programmable kernels will handle the algorithms for media functions such IDCT, Motion Compensation, and even Motion Estimation (used for encoding MPEG streams).

- **PRM, Volume 3: Display Registers**

Volume 3 describes the control registers for the display. The overlay registers and VGA registers are also cover in this volume.

- **PRM, Volume 4: Subsystem and Cores**

Volume 4 describes the GMCH programmable cores, or EUs, and the "shared functions", which are shared by more than one EU and perform functions such as I/O and complex math functions.

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

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

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



1.1 Notations and Conventions

1.1.1 Reserved Bits and Software Compatibility

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

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

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

1.2 Terminology

Term	Abbr.	Definition
3D Pipeline	—	One of the two pipelines supported in the GPE. The 3D pipeline is a set of fixed-function units arranged in a pipelined fashion, which process 3D-related commands by spawning EU threads. Typically this processing includes rendering primitives. See <i>3D Pipeline</i> .
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 “perspective 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 <i>Strips and Fans</i> .
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 <i>Graphics Processing Engine</i> .
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 <i>Geometry Shader</i> .
Graphics Address	—	The GPE virtual address of some memory-resident object. This virtual address gets mapped by a GTT or PGTT to a physical memory address. Note that many memory-resident objects are referenced not with Graphics Addresses, but instead with offsets from a "base address register".
Graphics Processing Engine	GPE	Collective name for the Subsystem, the 3D and Media pipelines, and the Command Streamer.
Guardband	GB	Region that may be clipped against to make sure objects do not exceed the limitations of the renderer's coordinate space.
Horizontal Stride	HorzStride	The distance in element-sized units between adjacent elements of a Gen4 region-based GRF access.
Immediate floating point vector	VF	A numerical data type of 32 bits, an immediate floating point vector of type VF contains 4 floating point elements with 8-bit each. The 8-bit floating point element contains a sign field, a 3-bit exponent field and a 4-bit mantissa field. It may be used to specify the type of an immediate operand in an instruction.



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



Term	Abbr.	Definition
Media Pipeline	—	Fixed function stages dedicated to media and “generic” processing, sometimes referred to as the generic pipeline.
Message	—	Messages are data packages transmitted from a thread to another thread, another shared function or another fixed function. Message passing is the primary communication mechanism of GEN4 architecture.
Message Gateway	—	Shared function that enables thread-to-thread message communication/synchronization used solely by the Media pipeline.
Message Register File	MRF	Write-only registers used by EUs to assemble messages prior to sending and as the operand of a send instruction.
Most Significant Bit	MSB	Most Significant Bit
Motion Compensation	MC	Part of the video decoding pipe.
Motion Picture Expert Group	MPEG	MPEG is the international standard body JTC1/SC29/WG11 under ISO/IEC that has defined audio and video compression standards such as MPEG-1, MPEG-2, and MPEG-4, etc.
Motion Vector Field Selection	MVFS	A four-bit field selecting reference fields for the motion vectors of the current macroblock.
Multi Render Targets	MRT	Multiple independent surfaces that may be the target of a sequence of 3D or Media commands that use the same surface state.
Normalized Device Coordinates	NDC	Clip Space Coordinates that have been divided by the Clip Space “W” component.
Object	—	A single triangle, line or point.
Parent Thread	—	A thread corresponding to a root-node or a branch-node in thread generation hierarchy. A parent thread may be a root thread or a child thread depending on its position in the thread generation hierarchy.
Pipeline Stage	—	A abstracted element of the 3D pipeline, providing functions performed by a combination of the corresponding hardware FF unit and the threads spawned by that FF unit.
Pipelined State Pointers	PSP	Pointers to state blocks in memory that are passed down the pipeline.
Pixel Shader	PS	Shader that is supplied by the application, translated by the jitter and is dispatched to the EU by the Windower (conceptually) once per pixel.
Point	—	A drawing object characterized only by position coordinates and width.
Primitive	—	Synonym for object: triangle, rectangle, line or point.



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



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



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



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



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

§§





2 Subsystem Overview

2.1 Introduction

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

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

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

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

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

2.2 Subsystem Topology

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



2.3 Execution Units (EUs)

Each EU is a vector machine capable of performing a given operation on as many as 16 pieces of data of the same type in parallel (though not necessarily on the same instant in time). In addition, each EU can support a number of execution contexts called *threads* that are used to avoid stalling the EU during a high-latency operation (external to the EU) by providing an opportunity for the EU to switch to a completely different workload with minimal latency while waiting for the high-latency operation to complete.

For example, if a program executing on an EU requires a texture read by the sampling engine, the EU may not necessarily idle while the data is fetched from memory, arranged, filtered and returned to the EU. Instead the EU will likely switch execution to another (unrelated) thread associated with that EU. If that thread encounters a stall, the EU may switch to yet another thread and so on. Once the Sampler result arrives back at the EU, the EU can switch back to the original thread and use the returned data as it continues execution of that thread.

The fact that there are multiple EU cores each with multiple threads can generally be ignored by software. There are some exceptions to this rule: e.g., for

- debugging (see *Debugging*)
- thread-to-thread communication (see *Message Gateway, Media*)
- synchronization of thread output to memory buffers (see *Geometry Shader*).

In contrast, the internal SIMD aspects of the EU are very much exposed to software.

This volume will not deal with the details of the EUs. See the *Gen4 Core* volume for details such as EU registers and instruction set.

2.4 Thread Dispatching

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

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



2.5 Shared Functions

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

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

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

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

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

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

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

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



the Graphics Processing Engine in order to blend them with other rendered objects and test for visibility of newly rendered objects. Finally, the Data Port also provides read access constant buffers (arrays of constants in memory.)

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

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

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

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

2.6 Messages

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

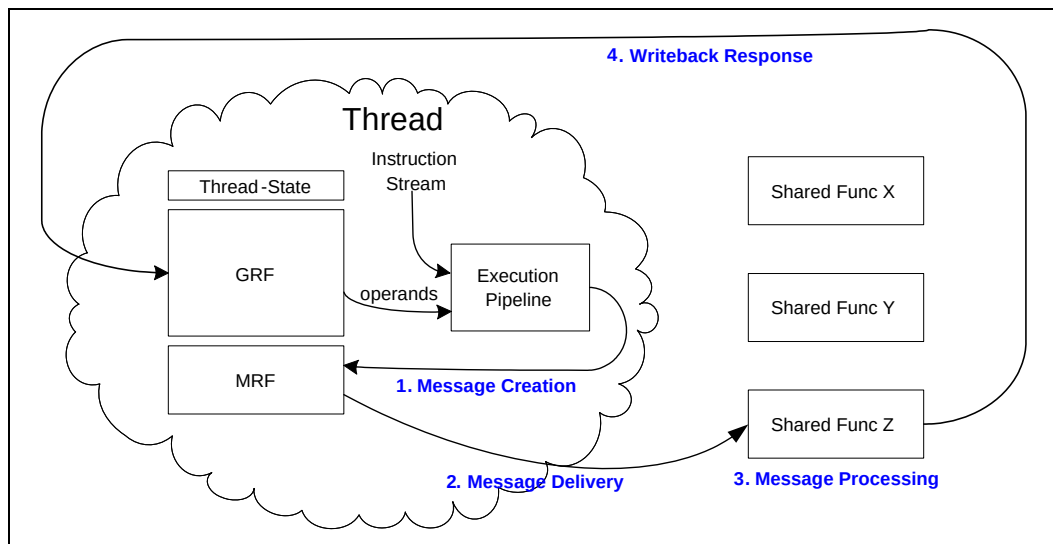
The information transmitted in a message falls into two categories:

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

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

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

Figure 2-1. Data Flow Associated With Messages





2.6.1 Message Register File (MRF)

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

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

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

2.6.2 Send Instruction

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

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

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

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



2.6.3 Creating and Sending a Message

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

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

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

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

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

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

Although more than one message may be enqueued at any point in time, care must be taken by the programmer to ensure that each message's destination GRF register region, if any, does not overlap with that of another enqueued message. This condition is not checked by HW. Due to varying latencies between two messages, and out-of-order, non-contiguous writeback cycles in the current implementation, the outcome in the GRF is indeterminate; It may be the result from the first message, or the result from the second message, or a mixture of data from both.



2.6.4 Message Payload Containing a Header

For most shared functions, the first register of the message payload contains the *header payload* of the message (or simply the *message header*). It contains the debug fields (fixed at DW6 and DW7) and state fields (such as binding table pointer, sampler state pointer, etc.) following a consistent format structure. Consequently, the rest of the message payload is referred to as the *data payload*.

Messages to Extended Math do not have a header and only contain data payload. Those messages may be referred to as header-less messages. Messages to Gateway combine the header and data payloads in a single message register.

2.6.5 Writebacks

Some messages generate return data as dictated by the 'function-control' (opcode) field of the 'send' instruction (part of the <desc> field). The Gen4 execution unit and message passing infrastructure do not interpret this field in any way to determine if writeback data is to be expected. Instead explicit fields in the 'send' instruction to the execution unit the starting GRF register and count of returning data. The execution unit uses this information to set in-flight bits on those registers to prevent execution of any instruction which uses them as an operand until the register(s) is(are) eventually written in response to the message. If a message is not expected to return data, the 'send' instruction's writeback destination specifier (<post_dest>) must be set to 'null' and the response length field of <desc> must be 0 (see 'send' instruction for more details).

The writeback data, if called for, arrives as a series of register writes to the GRF at the location specified by the starting GRF register and length as specified in the 'send' instruction. As each register is written back to the GRF, its in-flight flag is cleared and it becomes available for use as an instruction operand. If a thread was suspended pending return of that register, the dependency is lifted and the thread is allowed to continue execution (assuming no other dependency for that thread remains outstanding).

2.6.6 Message Delivery Ordering Rules

All messages between a thread and an individual shared function are delivered in the ordered they were sent. Messages to different shared functions originating from a single thread may arrive at their respective shared functions out of order.

The writebacks of various messages from the shared functions may return in any order. Further individual destination registers resulting from a single message may return out of order, potentially allowing execution to continue before the entire response has returned (depending on the dependency chain inherent in the thread).



2.6.7 Execution Mask and Messages

The Gen4 Architecture defines an Execution Mask (EMask) for each instruction issued. This 16b bit-field identifies which SIMD computation channels are enabled for that instruction. Since the 'send' instruction is inherently scalar, the EMask is ignored as far as instruction dispatch is concerned. Further the execution size has no impact on the size of the 'send' instruction's implicit move (it is always 1 register regardless of specified execution size).

The 16b EMask is forwarded with the message to the destination shared function to indicate which SIMD channels were enabled at the time of the 'send'. A shared function may interpret or ignore this field as dictated by the functionality it exposes. For instance, the Extended Math shared function observes this field and performs the specified operation only on the operands with enabled channels, while the DataPort writes to the render cache ignore this field completely, instead using the pixel mask included in-band in the message payload to indicate which channels carry valid data.

2.6.8 End-Of-Thread (EOT) Message

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

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

Target Shared Functions supporting EOT messages	Target Shared Functions <u>not</u> supporting EOT messages
Null, DataPortWrite, URB, MessageGateway, ThreadSpawner	DataPortRead, Sampler

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

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

Unlike the thread dispatcher, fixed-functions discriminate end-of-thread messages, only acting upon those from threads which they originated, as indicated by the 4b



fixed-function ID present in R0 of end-of-thread message payload. This 4b field is attached to the thread at new-thread dispatch time and is placed in its designated field in the R0 contents delivered to the GRF. Thus to satisfy the inclusion of the fixed-function ID, the typical end-of-thread message generally supplies R0 from the GRF as the first register of an end-of-thread message.

As an optimization, an end-of-thread message may be overload upon another “productive” message, saving the cost in execution and bandwidth of a dedicated end-of-thread message. Outside of the end-of-thread message, most threads issue a message just prior to their termination (for instance, a Dataport write to the framebuffer) so the overloaded end-of-thread is the common case. The requirement is that the message contains R0 from the GRF (to supply the fixed-function ID), and that destination shared function be either (a) the URB; (b) the Read or Write Dataport; or, (c) the Gateway, as these functions reside on the O-Bus. In the case where the last real message of a thread is to some other shared function, the thread must issue a separate message for the purposes of signaling end-of-thread to the “null” shared function.

2.6.9 Performance

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

Other factors to consider with regard to performance:

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



2.6.10 Message Description Syntax

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

Dispatch Messages: **Rn.d**

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

SEND Instruction Messages: **Mn.d**

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

Writeback Messages: **Wn.d**

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

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

2.6.11 Message Errors

Messages are constructed via software, and not all possible bit encodings are legal, thus there is the possibility that a message may be sent containing one or more errors in its descriptor or payload contents. There are two points of error detection in the message passing system: (a) the message delivery subsystem is capable of detecting bad FunctionIDs and some cases of bad message lengths; (b) the shared functions contain various error detection mechanisms which identify bad sub-function codes, bad message lengths, and other misc errors. The error detection capabilities are specific to each shared function. The execution unit hardware itself does not perform message validation prior to transmission.

In both cases, information regarding the erroneous message is captured and made visible through MMIO registers, and the driver notified via an interrupt mechanism (see the *Debugging* chapter for details). The set of possible errors is listed in Table 2-1 with the associated outcome. Please see the chapters on debug and error handling for detailed information.



Table 2-1. Error Cases

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

3 *Debugging*

3.1 Introduction

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

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

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

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

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

These debug features are described in more detail below.



3.2 The Snapshot Mechanism

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

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

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



3.2.1 Debug Trigger Counters

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

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

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

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

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

3.3 Fixed Function Debug Process

3.3.1 Overview

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

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



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

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

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

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

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

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

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

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

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

3.3.2 Vertex Shader Debug

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

Given a reproducible test case, one can identify the thread that shaded a given vertex. Assuming an errant vertex has been discovered in GS, the vertex sequence number from the previous enabled FF can be determined from the GS snapshot. The associated FFID captured in GS will indicate that the vertex came either from the VS



unit or (when VS is in bypass mode) directly from the VF unit. If the vertex came from the VF unit then the VS is not involved.

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

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

3.3.3 SVG Debug

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

3.3.3.1 SVG_CTL—Debug Control

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

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

3.3.3.2 SVG_RDATA—Debug Return Data

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

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



3.3.3.3 SVG_WORK_CTL—Debug Workaround Control

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

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

3.3.4 Vertex Fetch

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

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



3.3.4.1 VF_CTL—Debug Control

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

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



3.3.4.2 VF_STRG_VAL—Debug Snapshot Trigger Value

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

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

3.3.4.3 VF_STR_VL_OVR —Debug Start Vertex Location Override

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

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

3.3.4.4 VF_VC_OVR —Debug Vertex Count Override

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

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



3.3.4.5 VF_STR_PSKIP —Debug Starting Primitives Skipped

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

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

3.3.4.6 VF_MAX_PRIM —Debug Max Primitives

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

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

3.3.4.7 VF_RDATA —Debug Return Data

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

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



3.3.5 Vertex Shader

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

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

R0.7	
SS Flag	Reserved
1 bit	31 bits

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

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



3.3.5.1 VS_CTL—Debug Control

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

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

3.3.5.2 VS_STRG_VAL—Debug Snapshot Trigger Value

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

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



3.3.5.3 VS_RDATA —Debug Return Data

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

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

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

3.3.6 Geometry Shader

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

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

R0.7	
SS Flag	Reserved
1 bit	31 bits

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

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



3.3.6.1 GS_CTL —Debug Control

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

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



3.3.6.2 GS_STRG_VAL —Debug Snapshot Trigger Value

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

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

3.3.6.3 GS_RDATA —Debug Return Data

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

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



3.3.7 Clipper

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

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

R0.7	
SS Flag	Reserved
1 bit	31 bits

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

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

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



3.3.7.1 CL_CTL—Debug Control

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

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



3.3.7.2 CL_STRG_VAL—Debug Snapshot Trigger Value

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

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

3.3.7.3 CL_RDATA—Debug Return Data

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

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



3.3.8 Strips Fans

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

R0.7	
SS Flag	Reserved
1 bit	31 bits

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

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

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

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

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

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

The snapshot flag will be passed along to the Windower.



3.3.8.1 SF_CTL —Debug Control

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

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



3.3.8.2 SF_STRG_VAL—Debug Snapshot Trigger Value

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

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

3.3.8.3 SF_MIN_PR_IND—Debug Minimum Primitive Index

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

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

3.3.8.4 SF_MAX_PR_IND—Debug Maximum Primitive Index

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

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



3.3.8.5 SF_CLIP_RMIN– Debug Clip Rectangle Minimum Coordinates

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

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

3.3.8.6 SF_CLIP_RMAX—Debug Clip Rectangle Maximum Coordinates

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

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

3.3.8.7 SF_RDATA—Debug Return Data

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

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



3.3.9 Windower / Intermediate Z

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

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

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

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

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

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

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



The following pseudo code describes the Windower snapshot behavior.

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

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

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

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




3.3.9.1 WIZ_CTL—Debug Control

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

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



3.3.9.2 WIZ_STRG_VAL —Debug Snapshot Trigger Value

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

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

3.3.9.3 WIZ_RDATA—Debug Return Data

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

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



3.3.10 Video Front End

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

3.3.10.1 VFE_CTL—Debug Control

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

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

3.3.10.2 VFE_STRG_VAL—Debug Snapshot Trigger Value

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

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



3.3.10.3 VFE_RDATA—Debug Return Data

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

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

3.3.11 Thread Spawner

TS does not have a kernel counter.

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

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

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

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



3.3.11.1 TS_CTL—Debug Control

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

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



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

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

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

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

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

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

3.3.11.4 TS_RDATA—Debug Return Data

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

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



3.3.11.5 Parent Thread Recommendations

Kernel Development Guideline

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

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

```
and (1) reg32 r0.7:ud 0x80000000:ud      // save bit 31
...
// update r0.7 with new child ID
or (1) r0.7:ud r0.7:ud reg32:ud          // restore bit 31
```

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

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

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

The debug ID field of a root thread:

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

The debug ID field of a child thread:

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



3.4 Shared Function Debug

3.4.1 Thread Dispatcher

3.4.1.1 TD_CTL—Debug Control

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

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



3.4.1.2 TD_CTL2—Debug Control 2

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

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



3.4.1.3 TD_VF_VS_EMSK—Debug VF/VS Execution Mask

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

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

3.4.1.4 TD_GS_EMSK—Debug GS Execution Mask

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

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

3.4.1.5 TD_CLIP_EMSK—Debug Clipper Execution Mask

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

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



3.4.1.6 TD_SF_EMSK—Debug SF Execution Mask

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

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

3.4.1.7 TD_WIZ_EMSK — Debug WIZ Execution Mask

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

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

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

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

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



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

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

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

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

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

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

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

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

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



3.4.1.12 TD_RDATA—Debug Return Data

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

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

3.4.1.13 TD_TS_EMSK—Debug TS Execution Mask

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

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

3.4.2 Math Unit

3.4.2.1 MATH_CTL—Math Debug Control

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

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

Bit	Descriptions																																												
15:8	Debug Data Mux Select: Controls which 32 bits of the trap data for the selected SF unit are returned 0 = Error Message Header: <table border="1"> <tr><td>[31:28]</td><td>Reserved</td></tr> <tr><td>[27:24]</td><td>FFID[3:0]</td></tr> <tr><td>[23:22]</td><td>Reserved</td></tr> <tr><td>[21:20]</td><td>TID[1:0]</td></tr> <tr><td>[19:16]</td><td>EUID[3:0]</td></tr> <tr><td>[15:12]</td><td>rlen[3:0]</td></tr> <tr><td>[11:8]</td><td>milen[3:0]</td></tr> <tr><td>[7:4]</td><td>opcode[3:0]</td></tr> <tr><td>[3:0]</td><td> Error Code[3:0] -Identifies type of error [0] = bad length [1]= invalid opcode [2] = EOT bit detected [3] = Error bit detected In all cases above EM will discard message </td></tr> </table> 1 = EM Debug Data <table border="1"> <tr><td>31:31</td><td>Input Data Valid</td></tr> <tr><td>30:30</td><td>Input FIFO Full</td></tr> <tr><td>29:26</td><td>Input Opcode</td></tr> <tr><td>25:22</td><td>Input EUID</td></tr> <tr><td>21:20</td><td>Thread ID</td></tr> <tr><td>19:16</td><td>Msg Length</td></tr> <tr><td>15:9</td><td>Destination register</td></tr> <tr><td>8</td><td> Sequencer to FPU Valid & !Hold </td></tr> <tr><td>7</td><td> FPU to FPU Valid Counter </td></tr> <tr><td>6:3</td><td></td></tr> <tr><td>2</td><td> FPU to Assembler Valid & !Hold </td></tr> <tr><td>1</td><td> Output of EM Valid & Grant </td></tr> <tr><td>0</td><td>Reserved</td></tr> </table> 2 -255 = to be defined in an EM-specific document outside the PRM	[31:28]	Reserved	[27:24]	FFID[3:0]	[23:22]	Reserved	[21:20]	TID[1:0]	[19:16]	EUID[3:0]	[15:12]	rlen[3:0]	[11:8]	milen[3:0]	[7:4]	opcode[3:0]	[3:0]	Error Code[3:0] -Identifies type of error [0] = bad length [1]= invalid opcode [2] = EOT bit detected [3] = Error bit detected In all cases above EM will discard message	31:31	Input Data Valid	30:30	Input FIFO Full	29:26	Input Opcode	25:22	Input EUID	21:20	Thread ID	19:16	Msg Length	15:9	Destination register	8	Sequencer to FPU Valid & !Hold	7	FPU to FPU Valid Counter	6:3		2	FPU to Assembler Valid & !Hold	1	Output of EM Valid & Grant	0	Reserved
[31:28]	Reserved																																												
[27:24]	FFID[3:0]																																												
[23:22]	Reserved																																												
[21:20]	TID[1:0]																																												
[19:16]	EUID[3:0]																																												
[15:12]	rlen[3:0]																																												
[11:8]	milen[3:0]																																												
[7:4]	opcode[3:0]																																												
[3:0]	Error Code[3:0] -Identifies type of error [0] = bad length [1]= invalid opcode [2] = EOT bit detected [3] = Error bit detected In all cases above EM will discard message																																												
31:31	Input Data Valid																																												
30:30	Input FIFO Full																																												
29:26	Input Opcode																																												
25:22	Input EUID																																												
21:20	Thread ID																																												
19:16	Msg Length																																												
15:9	Destination register																																												
8	Sequencer to FPU Valid & !Hold																																												
7	FPU to FPU Valid Counter																																												
6:3																																													
2	FPU to Assembler Valid & !Hold																																												
1	Output of EM Valid & Grant																																												
0	Reserved																																												



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

3.4.2.2 MATH_RDATA—Math Debug Return Data

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

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

3.4.3 Instruction / State Cache

3.4.3.1 ISC_CTL—Instruction / State Debug Control

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

Bit	Descriptions
31:0	Reserved : MBZ



3.4.4 Instruction L1 Cache

3.4.4.1 ISC_L1CA_CTR—Instruction L1 Cache Debug Control

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

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

3.4.4.2 ISC_L1CA_RDATA—Instruction L1 Cache Debug Return Data

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

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



3.4.4.3 ISC_L1CA_BP_ADR1—Instruction L1 Cache Breakpoint Address 1 Control

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

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

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

3.4.4.4 ISC_L1CA_BP_ADR2—Instruction L1 Cache Breakpoint Address 2 Control

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

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



3.4.4.5 ISC_L1CA_BP_OPC1—Instruction L1 Cache Breakpoint Opcode 1 Control

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

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

3.4.4.6 ISC_L1CA_BP_OPC2—Instruction L1 Cache Breakpoint Opcode 2 Control

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

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



3.4.5 Message Arbiter

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

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

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



3.4.5.1 MA_DEBUG_1—Message Arbiter Debug Control

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

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



3.4.6 Sampler

3.4.6.1 SAMPLER_CTL—Sampler Debug Control

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

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



3.4.6.2 SAMPLER_RDATA—Sampler Debug Return Data

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

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

3.4.7 Data Port

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

3.4.7.1 DP_CTL—Data Port Debug Control

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

Bit	Descriptions
31:16	Reserved : MBZ



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

3.4.7.2 DP_RDATA—Data Port Debug Return Data

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

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



3.4.8 Render Cache

3.4.8.1 RC_CTL—RC Debug Control

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

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

3.4.8.2 RC_DEF_CLR—RC Debug Force Default Color

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

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



3.4.8.3 RC_RDATA—RC Debug Return Data

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

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

3.4.9 Unified Return Buffer (URB)

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

3.4.9.1 URB_CTL—URB Debug Control

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

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



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

3.4.9.2 URB_RDATA—URB Debug Return Data

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

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



3.4.10 Thread Spawner (TS)

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

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

3.5 Attention Signaling from EU to Host

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



3.5.1 EU_CTL—EU Debug Control

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

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

3.5.2 EU_ATT—EU Debug Attention

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

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

3.5.3 EU_ATT_DATA—EU Debug Attention Data

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

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



3.5.4 EU_ATT_CLR—EU Debug Attention Clear

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

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

3.5.5 EU_RDATA—EU Debug Return Data

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

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



3.6 Breakpoints

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

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

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

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

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

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

3.6.1 Single Stepping

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

A breakpoint is set (using any of the mechanisms above) at the point in the code where the debugger wants to begin single-stepping. The System Routine (SR) code that handles the breakpoint exception will normally communicate with host software



where the majority of the debugger code resides. When the user wants to execute the next instruction, the SR sets the **Breakpoint Suppress** bit of CR0 but does not clear the **Breakpoint Exception Status and Control** bit of CR1. Control is transferred from the system routine back to application code by writing a 0 to the **Master Exception State and Control**.

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

3.6.2 Modification of Instruction Stream

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

3.7 Message Errors

3.7.1.1 Error-Types Visible to the Arb

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



3.7.1.2 Non-pipelined S/F Operation

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

§§





4 Sampling Engine

The Sampling Engine provides the capability of advanced sampling and filtering of surfaces in memory.

The sampling engine function is responsible for providing filtered texture values to the Gen4 Core in response to sampling engine messages.. The sampling engine uses SAMPLER_STATE to control filtering modes, address control modes, and other features of the sampling engine. A pointer to the sampler state is delivered with each message, and an index selects one of 16 states pointed to by the pointer. Some messages do not require SAMPLER_STATE. In addition, the sampling engine uses SURFACE_STATE to define the attributes of the surface being sampled. This includes the location, size, and format of the surface as well as other attributes.

Although data is commonly used for “texturing” of 3D surfaces, the data can be used for any purpose once returned to the execution core.

The following table summarizes the various subfunctions provided by the Sampling Engine. After the appropriate subfunctions are complete, the 4-component (reduced to fewer components in some cases) filtered texture value is provided to the Gen4 Core in order to complete the *sample* instruction.

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



4.1 Texture Coordinate Processing

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

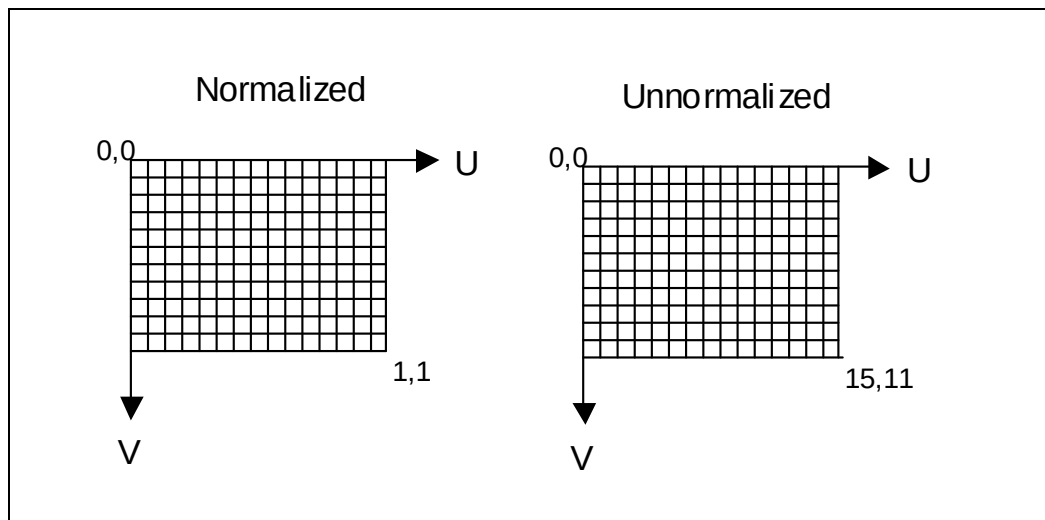
4.1.1 Texture Coordinate Normalization

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

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

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

Figure 4-1. Normalized vs. Unnormalized Texture Coordinates



4.1.2 Texture Coordinate Computation

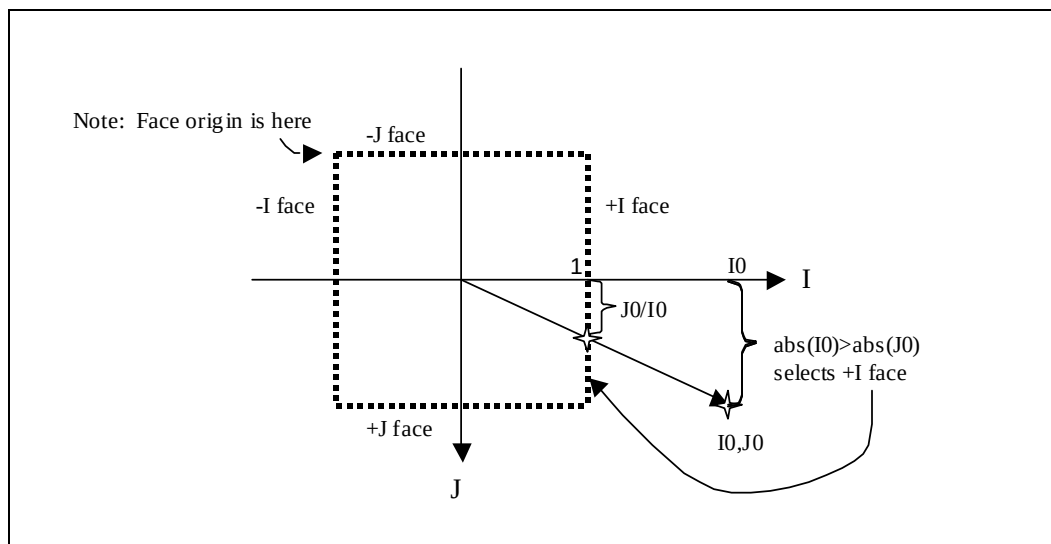
Cartesian (2D) and homogeneous (projected) texture coordinate values are projected from (interpolated) screen space back into texture coordinate space by dividing the pixel's S and T components by the Q component. This operation is done as part of the pixel shader kernel in the Gen4 Core.

Vector (cube map) texture coordinates are generated by first determining which of the 6 cube map faces (+X, +Y, +Z, -X, -Y, -Z) the vector intersects. The vector component (X, Y or Z) with the largest absolute value determines the proper (major)

axis, and then the sign of that component is used to select between the two faces associated with that axis. The coordinates along the two minor axes are then divided by the coordinate of the major axis, and scaled and translated, to obtain the 2D texture coordinate $[0,1]$ within the chosen face. Note that the coordinates delivered to the sampling engine must already have been divided by the component with the largest absolute value.

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

Figure 4-2. Cube Map Coordinate Computation Example



4.2 Texel Address Generation

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

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



4.2.1 Level of Detail Computation (Mipmapping)

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

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

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

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

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

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

4.2.1.1 Base Level Of Detail (LOD)

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

The ideal LOD computation is included below.

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

where :

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



4.2.1.2 LOD Bias

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

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

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

4.2.1.3 LOD Pre-Clamping

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

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

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

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

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

4.2.1.4 Min / Mag Determination

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

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



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

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

4.2.1.5 LOD Computation Pseudocode

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

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

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

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

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

MagMode = (LOD - Base <= 0)
If (MagMode or MipFlt = None)
    LOD = 0
    LOD = min(LOD, ceil(MaxLod))
    LOD = max(LOD, floor(MinLod))
else if (MipFlt = Nearest)
    LOD += 0.5
    LOD = min(LOD, ceil(MaxLod))
    LOD = max(LOD, floor(MinLod))
    LOD = floor(LOD)
else // MipFlt = Linear
    LOD = min(LOD, MaxLod)
    LOD = max(LOD, MinLod)
    TriBeta = frac(LOD)
    LOD0 = floor(LOD)
    LOD1 = LOD0 + 1
```



4.2.2 Inter-Level Filtering Setup

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

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

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

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

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

4.2.3 Intra-Level Filtering Setup

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

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



The following table summarizes the intra-level filtering modes.

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

4.2.3.1 MAPFILTER_NEAREST

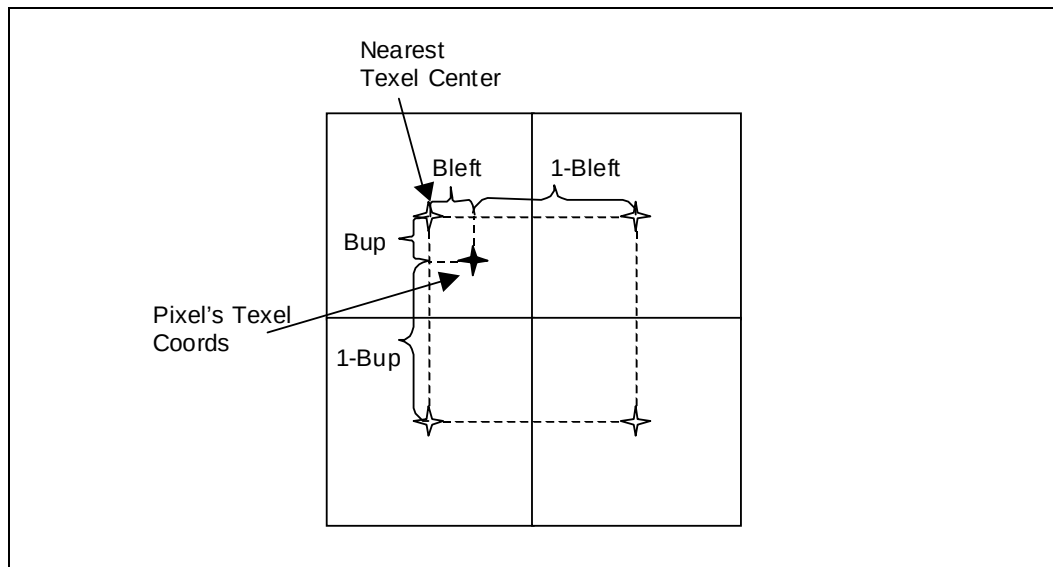
When the MAPFILTER_NEAREST is selected, the texel with coordinates nearest to the pixel's texture coordinate is selected and output as the single texel sample coordinates for the level.

4.2.3.2 MAPFILTER_LINEAR

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

When the MAPFILTER_LINEAR filter is selected on a 2D surface, the 2x2 region of texels surrounding the pixel's texture coordinate are sampled and later bilinearly filtered.

Figure 4-3. Bilinear Filter Sampling

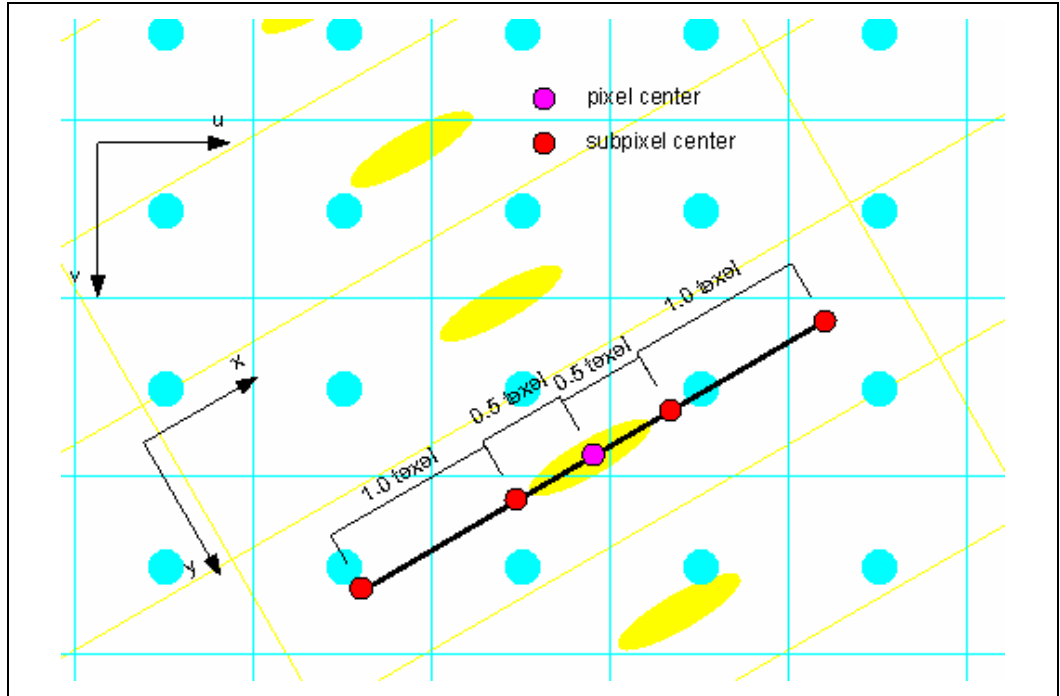


The four texels surrounding the pixel center are chosen for the bilinear filter. The filter weights each texel's contribution according to its distance from the pixel center. Texels further from the pixel center receive a smaller weight.

4.2.3.3 MAPFILTER_ANISOTROPIC

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

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



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

4.2.3.4 MAPFILTER_MONO

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

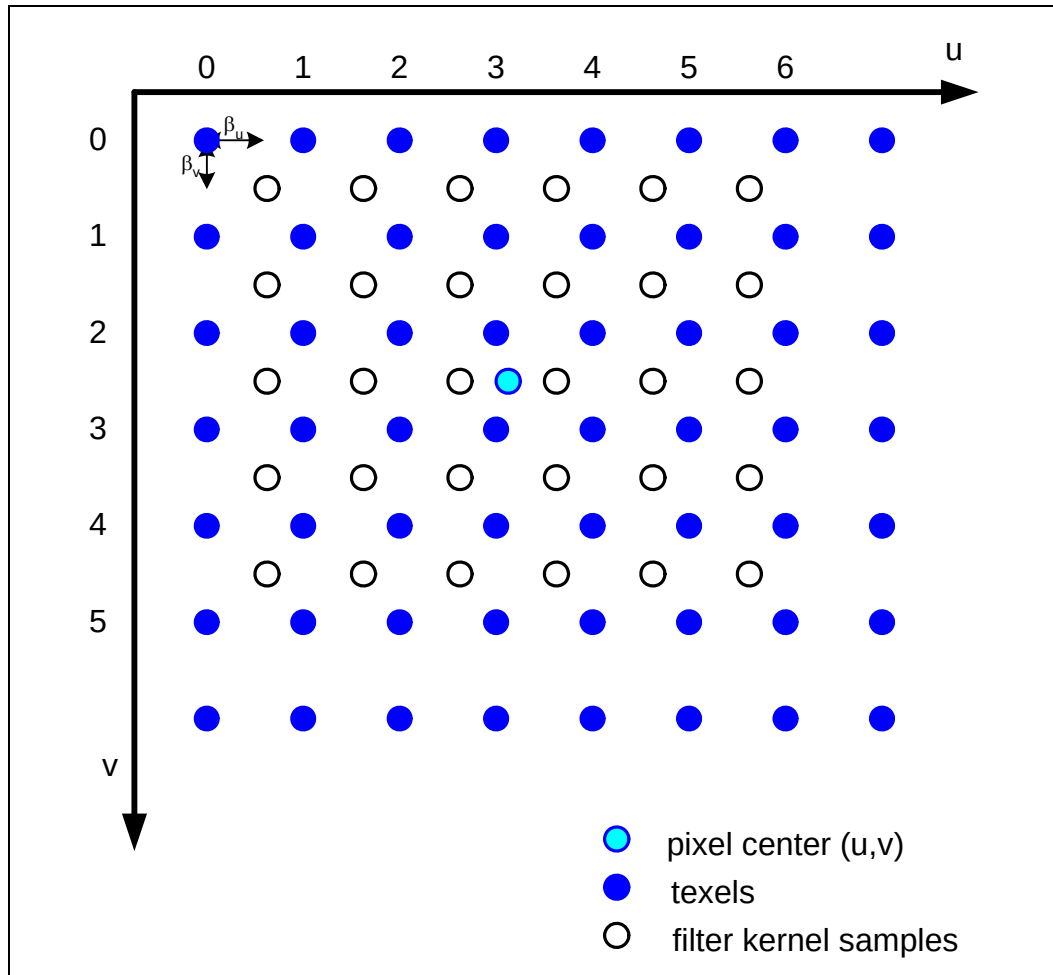
The figure below shows a 6x5 filter kernel as an example. The footprint of the filter (filter kernel samples) is equal to the size of the filter and the pixel center lies at the exact center of this footprint. The position of the upper left filter kernel sample (u_f , v_f) relative to the pixel center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

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

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

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

Figure 4-4. Sampling Using MAPFILTER_MONO



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

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

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



4.2.4 Texture Address Control

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

Note: For **Wrap Shortest** mode, the setup kernel has already taken care of correctly interpolating the texture coordinates. Software will need to specify `TEXCOORDMODE_WRAP` mode for the sampler that is provided with wrap-shortest texture coordinates, or artifacts may be generated along map edges.

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

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

Maximum Wraps/Mirrors

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

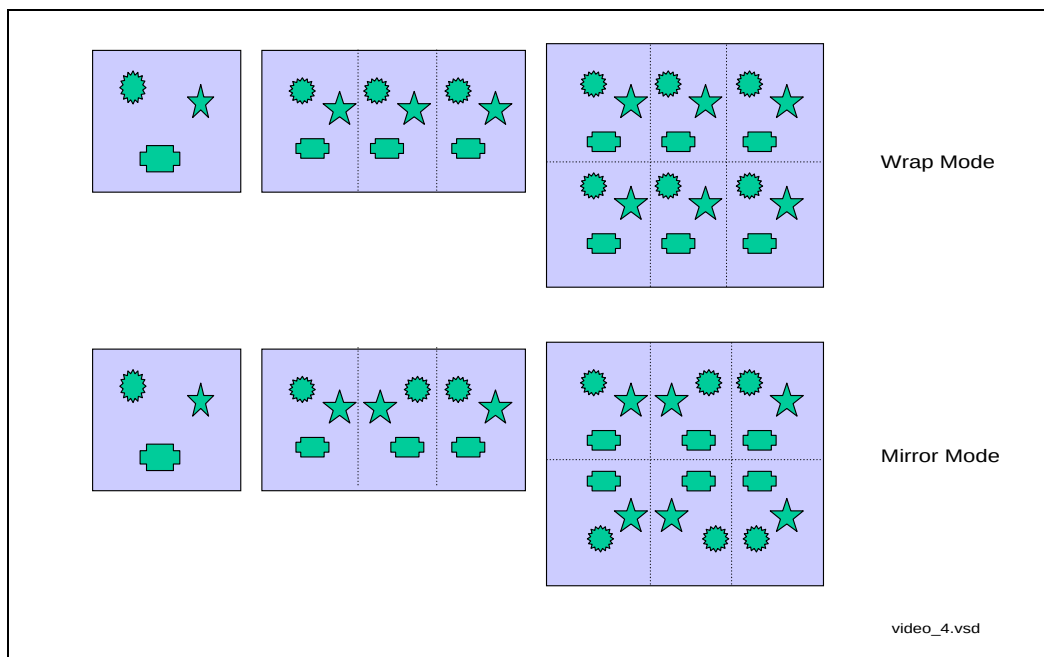
4.2.4.1 TEXCOORDMODE_WRAP Mode

In TEXCOORDMODE_WRAP addressing mode, the integer part of the texture coordinate is discarded, leaving only a fractional coordinate value. This results in the effect of the base map ([0,1)) being continuously repeated in all (axes-aligned) directions. Note that the interpolation between coordinate values 0.1 and 0.9 passes through 0.5 (as opposed to WrapShortest mode which interpolates through 0.0).

4.2.4.2 TEXCOORDMODE_MIRROR Mode

TEXCOORDMODE_MIRROR addressing mode is similar to Wrap mode, though here the base map is flipped at every integer junction. For example, for U values between 0 and 1, the texture is addressed normally, between 1 and 2 the texture is flipped (mirrored), between 2 and 3 the texture is normal again, and so on. The second row of pictures in the figure below indicate a map that is mirrored in one direction and then both directions. You can see that in the mirror mode every other integer map wrap the base map is mirrored in either direction.

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



4.2.4.3 TEXCOORDMODE_MIRROR_ONCE Mode

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

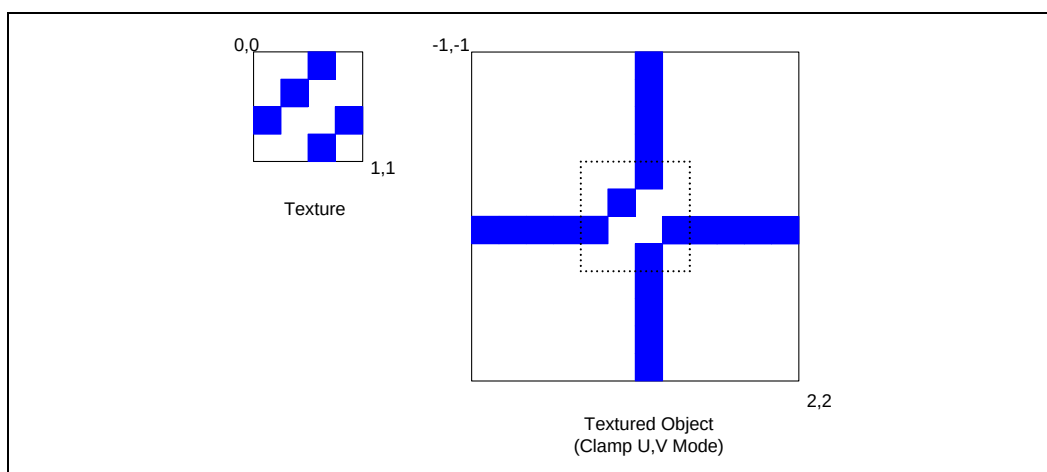


4.2.4.4 TEXCOORDMODE_CLAMP Mode

The TEXCOORDMODE_CLAMP addressing mode repeats the “edge” texel when the texture coordinate extends outside the $[0,1)$ range of the base texture map. This is contrasted to TEXCOORDMODE_CLAMPBORDER mode which defines a separate texel value for off-map samples. TEXCOORDMODE_CLAMP is also supported for cube maps, where texture samples will only be obtained from the intersecting face (even along edges).

The figure below illustrates the effect of clamp mode. The base texture map is shown, along with a texture mapped object with texture coordinates extending outside of the base map region.

Figure 4-6. Texture Clamp Mode



4.2.4.5 TEXCOORDMODE_CLAMPBORDER Mode

For non-cube map textures, TEXCOORDMODE_CLAMPBORDER addressing mode specifies that the texture map’s border value *BorderColor* is to be used for any texel samples that fall outside of the base map. The border color is specified via a pointer in SAMPLER_STATE.

4.2.4.6 TEXCOORDMODE_CUBE Mode

For cube map textures TEXCOORDMODE_CUBE addressing mode can be set to allow inter-face filtering. When texel sample coordinates that extend beyond the selected cube face (e.g., due to intra-level filtering near a cube edge), the correct sample coordinates on the adjoining face will be computed. This will eliminate artifacts along the cube edges, though some artifacts at cube corners may still be present.



4.3 Texel Fetch

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

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

4.3.1 Texel Chroma Keying

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

4.3.1.1 Chroma Key Testing

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

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

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



4.3.1.2 Chroma Key Effects

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

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

- KEYFILTER_KILL_ON_ANY_MATCH: Kill the pixel if any contributing texel sample matches the key
- KEYFILTER_REPLACE_BLACK: Here the sample is replaced with (0,0,0,0).

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

4.4 Shadow Prefilter Compare

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

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

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

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



Programming Notes:

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

4.5 Texel Filtering

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

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

4.6 Texel Color Gamma Linearization

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

This function is enabled on a per-texture basis by use of a surface format with “_SRGB” in its name. If enabled, the post-filtered texel RGB color to be converted from gamma=2.4 space to gamma=1.0 space by applying a $^{1/2.4} = ^{0.4167}$ exponential function.



4.7 State

4.7.1 BINDING_TABLE_STATE

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

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

4.7.2 SURFACE_STATE

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

Surface state defines the state needed for the following objects:

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



4.7.2.1 For Most Messages

SURFACE_STATE																																			
Project:		All																																	
This is the normal surface state used by all messages that use SURFACE_STATE except <i>deinterlace</i> and <i>sample_8x8</i> .																																			
DWord	Bit	Description																																	
0	31:29	Surface Type Project: All Format: U3 enumerated type FormatDesc This field defines the type of the surface.																																	
		<table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td>SURFTYPE_1D</td><td>Defines a 1-dimensional map or array of maps</td><td>All</td></tr><tr><td>1h</td><td>SURFTYPE_2D</td><td>Defines a 2-dimensional map or array of maps</td><td>All</td></tr><tr><td>2h</td><td>SURFTYPE_3D</td><td>Defines a 3-dimensional (volumetric) map</td><td>All</td></tr><tr><td>3h</td><td>SURFTYPE_CUBE</td><td>Defines a cube map or array of cube maps</td><td>All</td></tr><tr><td>4h</td><td>SURFTYPE_BUFFER</td><td>Defines an element in a buffer</td><td>All</td></tr><tr><td>5h-6h</td><td>Reserved</td><td></td><td>All</td></tr><tr><td>7h</td><td>SURFTYPE_NULL</td><td>Defines a null surface</td><td>All</td></tr></table>	Value	Name	Description	Project	0h	SURFTYPE_1D	Defines a 1-dimensional map or array of maps	All	1h	SURFTYPE_2D	Defines a 2-dimensional map or array of maps	All	2h	SURFTYPE_3D	Defines a 3-dimensional (volumetric) map	All	3h	SURFTYPE_CUBE	Defines a cube map or array of cube maps	All	4h	SURFTYPE_BUFFER	Defines an element in a buffer	All	5h-6h	Reserved		All	7h	SURFTYPE_NULL	Defines a null surface	All	
		Value	Name	Description	Project																														
		0h	SURFTYPE_1D	Defines a 1-dimensional map or array of maps	All																														
		1h	SURFTYPE_2D	Defines a 2-dimensional map or array of maps	All																														
		2h	SURFTYPE_3D	Defines a 3-dimensional (volumetric) map	All																														
		3h	SURFTYPE_CUBE	Defines a cube map or array of cube maps	All																														
		4h	SURFTYPE_BUFFER	Defines an element in a buffer	All																														
		5h-6h	Reserved		All																														
		7h	SURFTYPE_NULL	Defines a null surface	All																														
		Programming Notes A null surface will be used in instances where an actual surface is not bound. When a write message is generated to a null surface, no actual surface is written to. When a read message (including any sampling engine message) is generated to a null surface, the result is all zeros. All of the remaining fields in surface state are ignored for null surfaces, with the following exceptions: Width, Height, Depth, LOD, MIP Map Layout Mode, and Render Target View Extent fields must match the depth buffer’s corresponding state for all render target surfaces, including null.																																	
		The Surface Type of a surface used as a render target (accessed via the Data Port’s Render Target Write message) must be the same as the Surface Type of all other render targets and of the depth buffer (defined in 3DSTATE_DEPTH_BUFFER), unless either the depth buffer or render targets are SURFTYPE_NULL.																																	
28	Reserved	Project: All	Format: MBZ																																



SURFACE_STATE

27

Data Return Format

Project: All

Format: U1 enumerated type FormatDesc

For Sampling Engine Surfaces:

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

For Other Surfaces:

This field is ignored.

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

Programming Notes

The S1.14 return format is only legal for returning data from normalized (UNORM, or SNORM) map formats where *all* channels have ≤ 8 bits. *It is not legal to use this format with any floating point or integer map format.*

S1.14 return format is only used for SIMD16 and SIMD8 messages from the sampling engine. For SIMD4x2 messages, FLOAT32 format will be used for surfaces specifying S1.14 data return format.

Data returned in format S1.14 will be converted to FLOAT32 before reaching the GRF register, thus the state of this bit does not affect the kernel.

It is recommended that S1.14 format be used wherever it is legal, as the performance will generally be improved.

26:18

Surface Format

Project: All

Format: U32 FormatDesc

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

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

Programming Notes

Tile Walk TILEWALK_YMAJOR is UNDEFINED for *render target* formats that have 128 bits-per-element (BPE).

YUV (YCRCB) surfaces used as render targets can only be rendered to using 3DPRIM_RECTLIST with even X coordinates on all of its vertices, and the pixel shader cannot kill pixels.

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



SURFACE_STATE

17:14

Color Buffer Component Write Disables

Project: All

Format: U4 bit mask of disables (0 or logical OR of any of the enumerated values) FormatDesc

For Render Target Surfaces:

This field contains a bitmask that controls the writing of individual color components into the Color Buffer. If a component is disabled (bit set) writes to the color buffer will not modify that component. If enabled (bit clear), that component can be overwritten.

For Other Surfaces:

this field is ignored.

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

Programming Notes

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

Errata	Description	Project
#	Desc	All

13

Color Blend Enable

Project: All

Format: Enable FormatDesc

For Render Target Surfaces:

Specifies that color blend is enabled for this particular render target. The Color Buffer Blend Enable state in COLOR_CALC_STATE provides global control over blending. See Color Buffer Blending (Windower) for details.

For Other Surfaces:

is ignored.

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



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



SURFACE_STATE

10		MIP Map Layout Mode Project: All Format: U1 enumerated type FormatDesc For 1D and 2D Surfaces and This field specifies which MIP map layout mode is used, whether the map for LOD 1 is stored to the right of the LOD 0 map, or stored below it. See Memory Data Formats for details on the specifics of each layout mode. For Other Surfaces: This field is reserved : MBZ <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <th style="width: 15%;">Value</th> <th style="width: 35%;">Name</th> <th style="width: 35%;">Description</th> <th style="width: 15%;">Project</th> </tr> <tr> <td style="text-align: center;">0h</td> <td>MIPLAYOUT_BELOW</td> <td></td> <td style="text-align: center;">All</td> </tr> <tr> <td style="text-align: center;">1h</td> <td>MIPLAYOUT_RIGHT</td> <td></td> <td style="text-align: center;">All</td> </tr> </table> Programming Notes MIPLAYOUT_RIGHT is legal only for 2D non-array surfaces <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <th style="width: 15%;">Errata</th> <th style="width: 60%;">Description</th> <th style="width: 25%;">Project</th> </tr> <tr> <td style="text-align: center;">#</td> <td>MIPLAYOUT_RIGHT is not supported with “ld” sampler message</td> <td style="text-align: center;">All</td> </tr> <tr> <td style="text-align: center;">#</td> <td>MIPLAYOUT_RIGHT is not supported with sample_c/sample_l_c/sample_b_c sampler messages.</td> <td style="text-align: center;">[DevCL]</td> </tr> </table>	Value	Name	Description	Project	0h	MIPLAYOUT_BELOW		All	1h	MIPLAYOUT_RIGHT		All	Errata	Description	Project	#	MIPLAYOUT_RIGHT is not supported with “ld” sampler message	All	#	MIPLAYOUT_RIGHT is not supported with sample_c/sample_l_c/sample_b_c sampler messages.	[DevCL]
Value	Name	Description	Project																				
0h	MIPLAYOUT_BELOW		All																				
1h	MIPLAYOUT_RIGHT		All																				
Errata	Description	Project																					
#	MIPLAYOUT_RIGHT is not supported with “ld” sampler message	All																					
#	MIPLAYOUT_RIGHT is not supported with sample_c/sample_l_c/sample_b_c sampler messages.	[DevCL]																					
9		Reserved : MBZ																					



SURFACE_STATE

8

Render Cache Read Write Mode

Project: All

Format: U1 enumerated type FormatDesc

For Surfaces accessed via the Data Port to Render Cache:

This field specifies the way Render Cache treats a write request. If unset, Render Cache allocates a write-only cache line for a write miss. If set, Render Cache allocates a read-write cache line for a write miss.

For Surfaces accessed via the Sampling Engine or Data Port to Texture Cache or Data Cache:

This field is reserved : MBZ

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

Programming Notes

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

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



SURFACE_STATE

7:6

Media Boundary Pixel Mode

Project: All

Format: U2 enumerated type FormatDesc

For 2D Non-Array Surfaces accessed via the Data Port Media Block Read Message:

This field enables control of which rows are returned on vertical out-of-bounds reads using the Data Port Media Block Read Message. In the description below, frame mode refers to **Vertical Line Stride** = 0, field mode is **Vertical Line Stride** = 1 in which only the even or odd rows are addressable. The frame refers to the entire surface, while the field refers only to the even or odd rows within the surface. Refer to Section 5.6.1 for more details.

For Other Surfaces:

Reserved : MBZ

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



SURFACE_STATE

5:0

Cube Face Enables

Project: All

Format: U6 bit mask of enables FormatDesc

For SURFTYPE_CUBE Surfaces accessed via the Sampling Engine:

Bits 5:0 of this field enable the individual faces of a cube map. Enabling a face indicates that the face is present in the cube map, while disabling it indicates that that face is represented by the texture map's border color. Refer to Memory Data Formats for the correlation between faces and the cube map memory layout. Note that storage for disabled faces must be provided.

For other surfaces:

This field is reserved : MBZ

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

Programming Notes

When TEXCOORDMODE_CLAMP is used when accessing a cube map, this field must be programmed to 111111b (all faces enabled).

This field is ignored unless the **Surface Type** is SURFTYPE_CUBE.



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



SURFACE_STATE																				
2	31:19	Height Project: All Format: U13 FormatDesc Range SURFTYPE_1D: must be zero SURFTYPE_2D: height of surface – 1 (y/v dimension) [0,8191] SURFTYPE_3D: height of surface – 1 (y/v dimension) [0,2047] SURFTYPE_CUBE: height of surface – 1 (y/v dimension) [0,8191] SURFTYPE_BUFFER: contains bits [19:7] of the number of entries in the buffer – 1 [0,8191] This field specifies the height of the surface. If the surface is MIP-mapped, this field contains the height of the base MIP level. For buffers, this field specifies a portion of the buffer size. <table border="1"> <tr> <th colspan="3">Programming Notes</th></tr> <tr> <td colspan="3">For buffer surfaces, the number of entries in the buffer ranges from 1 to 2^{27}. After subtracting one from the number of entries, software must place the fields of the resulting 27-bit value into the Height, Width, and Depth fields as indicated, right-justified in each field. Unused upper bits must be set to zero.</td></tr> <tr> <td colspan="3">If Vertical Line Stride is 1, this field indicates the height of the field, not the height of the frame</td></tr> <tr> <td colspan="3">The Height of a render target must be the same as the Height of the other render targets and the depth buffer (defined in 3DSTATE_DEPTH_BUFFER), unless Surface Type is SURFTYPE_1D or SURFTYPE_2D with Depth = 0 (non-array) and LOD = 0 (non-mip mapped).</td></tr> </table> <table border="1"> <tr> <th>Errata</th><th>Description</th><th>Project</th></tr> <tr> <td>#</td><td>The number of entries in a SURFTYPE_BUFFER is restricted to $2^{27} - 1$</td><td>[DevBW-A,B]</td></tr> </table>	Programming Notes			For buffer surfaces, the number of entries in the buffer ranges from 1 to 2^{27} . After subtracting one from the number of entries, software must place the fields of the resulting 27-bit value into the Height , Width , and Depth fields as indicated, right-justified in each field. Unused upper bits must be set to zero.			If Vertical Line Stride is 1, this field indicates the height of the field, not the height of the frame			The Height of a render target must be the same as the Height of the other render targets and the depth buffer (defined in 3DSTATE_DEPTH_BUFFER), unless Surface Type is SURFTYPE_1D or SURFTYPE_2D with Depth = 0 (non-array) and LOD = 0 (non-mip mapped).			Errata	Description	Project	#	The number of entries in a SURFTYPE_BUFFER is restricted to $2^{27} - 1$	[DevBW-A,B]
Programming Notes																				
For buffer surfaces, the number of entries in the buffer ranges from 1 to 2^{27} . After subtracting one from the number of entries, software must place the fields of the resulting 27-bit value into the Height , Width , and Depth fields as indicated, right-justified in each field. Unused upper bits must be set to zero.																				
If Vertical Line Stride is 1, this field indicates the height of the field, not the height of the frame																				
The Height of a render target must be the same as the Height of the other render targets and the depth buffer (defined in 3DSTATE_DEPTH_BUFFER), unless Surface Type is SURFTYPE_1D or SURFTYPE_2D with Depth = 0 (non-array) and LOD = 0 (non-mip mapped).																				
Errata	Description	Project																		
#	The number of entries in a SURFTYPE_BUFFER is restricted to $2^{27} - 1$	[DevBW-A,B]																		



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



SURFACE_STATE															
	5:2	MIP Count / LOD													
		Project:	All												
		Format:	Sampling Engine Surfaces: U4 in (LOD units – 1) FormatDesc Render Target Surfaces: U4 in LOD units												
		Range	Sampling Engine Surfaces: [0,13] representing [1,14] MIP levels Render Target Surfaces: [0,13] representing LOD Other Surfaces: [0]												
		For Sampling Engine Surfaces:													
		This field indicates the number of MIP levels allowed to be accessed starting at Surface Min LOD , which must be less than or equal to the number of MIP levels actually stored in memory for this surface.													
		Force the mip map access to be between the mipmap specified by the integer bits of the Min LOD and the ceiling of the value specified here.													
		For Render Target Surfaces:													
		This field defines the MIP level that is currently being rendered into. This is the absolute MIP level on the surface and is not relative to the Surface Min LOD field, which is ignored for render target surfaces.													
		For Other Surfaces:													
This field is reserved : MBZ															
		<table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td>Disable</td><td>Desc</td><td>All</td></tr><tr><td>1h</td><td>Enable</td><td>Desc</td><td>All</td></tr></table>	Value	Name	Description	Project	0h	Disable	Desc	All	1h	Enable	Desc	All	
Value	Name	Description	Project												
0h	Disable	Desc	All												
1h	Enable	Desc	All												
		<table><tr><th colspan="4">Programming Notes</th></tr><tr><td colspan="4">The LOD of a render target must be the same as the LOD of the other render target(s) and of the depth buffer (defined in 3DSTATE_DEPTH_BUFFER).</td></tr><tr><td colspan="4">For render targets with YUV surface formats, the LOD must be zero.</td></tr></table>		Programming Notes				The LOD of a render target must be the same as the LOD of the other render target(s) and of the depth buffer (defined in 3DSTATE_DEPTH_BUFFER).				For render targets with YUV surface formats, the LOD must be zero.			
Programming Notes															
The LOD of a render target must be the same as the LOD of the other render target(s) and of the depth buffer (defined in 3DSTATE_DEPTH_BUFFER).															
For render targets with YUV surface formats, the LOD must be zero.															
		<table><tr><th>Errata</th><th>Description</th><th>Project</th></tr><tr><td>#</td><td>Desc</td><td>All</td></tr></table>	Errata	Description	Project	#	Desc	All							
Errata	Description	Project													
#	Desc	All													
1:0	Reserved : MBZ														



SURFACE_STATE			
3	31:21	Depth Project: All Format: U11 FormatDesc Range SURFYPE_1D: number of array elements – 1 [0,511] SURFYPE_2D: number of array elements – 1 [0,511] SURFYPE_3D: depth of surface – 1 (z/r dimension) [0,2047] SURFYPE_CUBE: number of array elements – 1 [see programming notes for range] SURFYPE_BUFFER: contains bits [26:20] of the number of entries in the buffer – 1 [0,127] This field specifies the total number of levels for a volume texture or the number of array elements allowed to be accessed starting at the Minimum Array Element for arrayed surfaces. If the volume texture is MIP-mapped, this field specifies the depth of the base MIP level. For buffers, this field specifies a portion of the buffer size.	
		Programming Notes The Depth of a render target must be the same as the Depth of the other render target(s) and of the depth buffer (defined in 3DSTATE_DEPTH_BUFFER). For SURFYPE_CUBE: for all cube surfaces, this field must be zero as cube arrays are not supported.	
	20	Reserved Project: All Format: MBZ	
	19:3	Surface Pitch Project: All Format: U17 pitch in (#Bytes – 1) FormatDesc Range For surfaces of type SURFYPE_BUFFER: [0,2047] -> [1B, 2048B] For other linear surfaces: [0, 131071] -> [1B, 128KB] For X-tiled surface: [511, 131071] -> [512B, 128KB] = [1tile, 256 tiles] For Y-tiled surfaces: [127, 131071]->[128B,128KB] = [1 tile, 1024 tiles] This field specifies the surface pitch in (#Bytes - 1). For surfaces of type SURFYPE_BUFFER, this field indicates the size of the structure.	
		Programming Notes For linear <i>render target</i> surfaces, the pitch must be a multiple of the element size for non-YUV surface formats. Pitch must be a multiple of 2 * element size for YUV surface formats. For other linear surfaces, the pitch can be any multiple of bytes. For tiled surfaces, the pitch must be a multiple of the tile width.	
	2	Reserved Project: All Format: MBZ	



SURFACE_STATE

1

Tiled Surface

Project: All

Format: U1 enumerated type

FormatDesc

This field specifies whether the surface is tiled.

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

Programming Notes

Linear surfaces can be mapped to Main Memory (uncached) or System Memory (cacheable, snooped). Tiled surfaces can only be mapped to Main Memory.

The corresponding cache(s) must be invalidated before a previously accessed surface is accessed again with an altered state of this bit.

If **Surface Type** is SURFTYPE_BUFFER, this field must be FALSE (buffers are supported only in linear memory)

If the target cache via the Data Port is the Data Cache, this field must be disabled (zero). The data cache only supports access to linear memory.

If **Surface Type** is SURFTYPE_NULL, this field must be TRUE

0

Tile Walk

Project: All

Format: U1 enumerated type

FormatDesc

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

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

Programming Notes

Refer to *Memory Data Formats* for restrictions on *TileWalk* direction for the various buffer types. (Of particular interest is the fact that YMAJOR tiling is **not** supported for display/overlay buffers).

The corresponding cache(s) must be invalidated before a previously accessed surface is accessed again with an altered state of this bit.

Use of TILEWALK_YMAJOR is UNDEFINED for render target formats that have 128 bits-per-element (BPE).

This field is ignored when the surface is linear.



SURFACE_STATE											
4	31:28	Surface Min LOD Project: All Format: U4 in LOD units FormatDesc Range [0,13] For Sampling Engine Surfaces: This field indicates the most detailed LOD that can be accessed as part of this surface. This field is added to the delivered LOD (sample_l, ld, or resinfo message types) before it is used to address the surface. For Other Surfaces: This field is ignored. <table><tr><td colspan="3">Programming Notes</td></tr><tr><td colspan="3">This field must be zero if the Surface Format is MONO8</td></tr><tr><td colspan="3">[DevBW-A,B]: this field must be zero</td></tr></table>	Programming Notes			This field must be zero if the Surface Format is MONO8			[DevBW-A,B]: this field must be zero		
	Programming Notes										
This field must be zero if the Surface Format is MONO8											
[DevBW-A,B]: this field must be zero											
	27:17	Minimum Array Element Project: All Format: U11 FormatDesc Range 1D/2D/cube surfaces: [0,511] 3D surfaces: [0,2047] For Sampling Engine and Render Target 1D and 2D Surfaces: This field indicates the minimum array element that can be accessed as part of this surface. This field is added to the delivered array index before it is used to address the surface. For Render Target 3D Surfaces: This field indicates the minimum 'R' coordinate on the LOD currently being rendered to. This field is added to the delivered array index before it is used to address the surface. For Other Surfaces: This field must be set to zero. <table><tr><th>Errata</th><th>Description</th><th>Project</th></tr><tr><td>#</td><td>This field must be zero.</td><td>[DevBW-A,B]</td></tr><tr><td>#</td><td>For sample_c/sample_b_c/sample_l_c instructions this field is ignored. If it is tiled surface and not at a 4k boundary it must be copied to a 4k aligned surface. Then for any case it must be pointed to by the Surface Base Address.</td><td>[DevBW-A,B,C,D], [DevCL-A,B]</td></tr></table>	Errata	Description	Project	#	This field must be zero.	[DevBW-A,B]	#	For sample_c/sample_b_c/sample_l_c instructions this field is ignored. If it is tiled surface and not at a 4k boundary it must be copied to a 4k aligned surface. Then for any case it must be pointed to by the Surface Base Address .	[DevBW-A,B,C,D], [DevCL-A,B]
Errata	Description	Project									
#	This field must be zero.	[DevBW-A,B]									
#	For sample_c/sample_b_c/sample_l_c instructions this field is ignored. If it is tiled surface and not at a 4k boundary it must be copied to a 4k aligned surface. Then for any case it must be pointed to by the Surface Base Address .	[DevBW-A,B,C,D], [DevCL-A,B]									



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

4.7.2.1.1 Surface Formats

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

Support of each format and capability is as follows:

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



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



Sampling Engine	Sampling Engine Filtering	Sampling Engine Shadow Map	Sampling Engine Chroma Key	Render Target	Alpha Blend Render Target	Input Vertex Buffer	Streamed Output Vertex Buffers	Color Processing	Surface Format Encoding (Hex)	Format Name	Bits Per Element (BPE)
Y									091	L32X32_FLOAT	64
Y									092	I32X32_FLOAT	64
						Y			093	R16G16B16A16_SSCEALED	64
						Y			094	R16G16B16A16_USCEALED	64
						Y			095	R32G32_SSCEALED	64
						Y			096	R32G32_USCEALED	64
Y	Y		Y	Y	Y	Y			0C0	B8G8R8A8_UNORM	32
Y	Y			Y	Y				0C1	B8G8R8A8_UNORM_SRGB	32
Y	Y			Y	Y	Y			0C2	R10G10B10A2_UNORM	32
Y	Y								0C3	R10G10B10A2_UNORM_SRGB	32
Y				Y		Y			0C4	R10G10B10A2_UINT	32
Y	Y					Y			0C5	R10G10B10_SNORM_A2_UNORM	32
Y	Y			Y	Y	Y			0C7	R8G8B8A8_UNORM	32
Y	Y			Y	Y				0C8	R8G8B8A8_UNORM_SRGB	32
Y	Y			Y		Y			0C9	R8G8B8A8_SNORM	32
Y				Y		Y			0CA	R8G8B8A8_SINT	32
Y				Y		Y			0CB	R8G8B8A8_UINT	32
Y	Y			Y		Y			0CC	R16G16_UNORM	32
Y	Y			Y		Y			0CD	R16G16_SNORM	32
Y				Y		Y			0CE	R16G16_SINT	32
Y				Y		Y			0CF	R16G16_UINT	32
Y	Y			Y	Y	Y			0D0	R16G16_FLOAT	32
Y	Y			Y	Y				0D1	B10G10R10A2_UNORM	32
Y	Y			Y	Y				0D2	B10G10R10A2_UNORM_SRGB	32
Y	Y			Y	Y	Y			0D3	R11G11B10_FLOAT	32
Y				Y		Y	Y		0D6	R32_SINT	32
Y				Y		Y	Y		0D7	R32_UINT	32
Y		Y		Y	Y	Y	Y		0D8	R32_FLOAT	32
Y		Y							0D9	R24_UNORM_X8_TYPELESS	32
Y									0DA	X24_TYPELESS_G8_UINT	32
Y	Y								0DF	L16A16_UNORM	32
Y		Y							0E0	I24X8_UNORM	32
Y		Y							0E1	L24X8_UNORM	32



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



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

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



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

4.7.2.1.2 Sampler Output Channel Mapping

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

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



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



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

4.7.3 SAMPLER_STATE

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

4.7.3.1 For Most Messages

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



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



SAMPLER_STATE

21:20

Mip Mode Filter

Project: All

Format: U2 enumerated type FormatDesc

This field determines if and how mip map levels are chosen and/or combined when texture filtering.

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

Programming Notes

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



SAMPLER_STATE

SAMPLER_STATE			
	19:17	Mag Mode Filter	
		Project: All	
		Format: U2 enumerated type	



SAMPLER_STATE																																							
	13:3	Texture LOD Bias Project: All Format: S4.6 2's complement FormatDesc Range [-16.0, 16.0) This field specifies the signed bias value added to the calculated texture map LOD prior to min-vs-mag determination and mip-level clamping. Assuming mipmapping is enabled, a positive LOD bias will result in a somewhat blurrier image (using less-detailed mip levels) and possibly higher performance, while a negative bias will result in a somewhat crisper image (using more-detailed mip levels) and may lower performance.																																					
		Programming Notes There is <u>no</u> requirement or need to offset the LOD Bias in order to produce a correct LOD for texture filtering (as was required for correct bilinear and anisotropic filtering in some legacy devices).																																					
	2:0	Shadow Function Project: All Format: U3 enumerated type FormatDesc This field is used for shadow mapping support via the sample_c message type, and specifies the specific comparison operation to be used. The comparison is between the texture sample red channel (except for alpha-only formats which use the alpha channel), and the "ref" value provided in the input message.																																					
		<table> <tr> <th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr> <tr> <td>0h</td><td>PREFILTEROP_ALWAYS</td><td></td><td>All</td></tr> <tr> <td>1h</td><td>PREFILTEROP_NEVER</td><td></td><td>All</td></tr> <tr> <td>2h</td><td>PREFILTEROP_LESS</td><td></td><td>All</td></tr> <tr> <td>3h</td><td>PREFILTEROP_EQUAL</td><td></td><td>All</td></tr> <tr> <td>4h</td><td>PREFILTEROP_LEQUAL</td><td></td><td>All</td></tr> <tr> <td>5h</td><td>PREFILTEROP_GREATER</td><td></td><td>All</td></tr> <tr> <td>6h</td><td>PREFILTEROP_NOTEQUAL</td><td></td><td>All</td></tr> <tr> <td>7h</td><td>PREFILTEROP_GEQUAL</td><td></td><td>All</td></tr> </table>		Value	Name	Description	Project	0h	PREFILTEROP_ALWAYS		All	1h	PREFILTEROP_NEVER		All	2h	PREFILTEROP_LESS		All	3h	PREFILTEROP_EQUAL		All	4h	PREFILTEROP_LEQUAL		All	5h	PREFILTEROP_GREATER		All	6h	PREFILTEROP_NOTEQUAL		All	7h	PREFILTEROP_GEQUAL		All
Value	Name	Description	Project																																				
0h	PREFILTEROP_ALWAYS		All																																				
1h	PREFILTEROP_NEVER		All																																				
2h	PREFILTEROP_LESS		All																																				
3h	PREFILTEROP_EQUAL		All																																				
4h	PREFILTEROP_LEQUAL		All																																				
5h	PREFILTEROP_GREATER		All																																				
6h	PREFILTEROP_NOTEQUAL		All																																				
7h	PREFILTEROP_GEQUAL		All																																				



SAMPLER_STATE												
1	31:22	Min LOD Project: All Format: U4.6 in LOD units FormatDesc Range [0.0, 13.0], where the upper limit is also bounded by the Max LOD . This field specifies the minimum value used to clamp the computed LOD after LOD bias is applied. Note that the minification-vs.-magnification status is determined after LOD bias and <u>before</u> this maximum (resolution) mip clamping is applied. The integer bits of this field are used to control the “maximum” (highest resolution) mipmap level that may be accessed (where LOD 0 is the highest resolution map). The fractional bits of this value effectively clamp the inter-level trilinear blend factor when trilinear filtering is in use.										
		<table><tr><th colspan="3">Programming Notes</th></tr><tr><td colspan="3">If Min LOD is greater than Max LOD, Min LOD takes precedence, i.e. the resulting LOD will always be Min LOD.</td></tr><tr><td colspan="3">This field must be zero if the Min or Mag Mode Filter is set to MAPFILTER_MONO</td></tr></table>		Programming Notes			If Min LOD is greater than Max LOD , Min LOD takes precedence, i.e. the resulting LOD will always be Min LOD .			This field must be zero if the Min or Mag Mode Filter is set to MAPFILTER_MONO		
		Programming Notes										
If Min LOD is greater than Max LOD , Min LOD takes precedence, i.e. the resulting LOD will always be Min LOD .												
This field must be zero if the Min or Mag Mode Filter is set to MAPFILTER_MONO												
<table><tr><th>Errata</th><th>Description</th><th>Project</th></tr><tr><td>#</td><td>If the Mip Mode Filter is set to MIPFILTER_NEAREST and the fractional portion of Min LOD is < 0.5 but > 0.0, the LOD chosen is one too large. Zeroing the fractional portion of Min LOD in these cases gives the correct behavior as a software workaround.</td><td>All</td></tr></table>	Errata	Description	Project	#	If the Mip Mode Filter is set to MIPFILTER_NEAREST and the fractional portion of Min LOD is < 0.5 but > 0.0, the LOD chosen is one too large. Zeroing the fractional portion of Min LOD in these cases gives the correct behavior as a software workaround.	All						
Errata	Description	Project										
#	If the Mip Mode Filter is set to MIPFILTER_NEAREST and the fractional portion of Min LOD is < 0.5 but > 0.0, the LOD chosen is one too large. Zeroing the fractional portion of Min LOD in these cases gives the correct behavior as a software workaround.	All										
	21:12	Max LOD Project: All Format: U4.6 in LOD units FormatDesc Range [0.0, 13.0] This field specifies the maximum value used to clamp the computed LOD after LOD bias is applied. Note that the minification-vs.-magnification status is determined after LOD bias and <u>before</u> this minimum (resolution) mip clamping is applied. The integer bits of this field are used to control the “minimum” (lowest resolution) mipmap level that may be accessed. The fractional bits of this value effectively clamp the inter-level trilinear blend factor when trilinear filtering is in use. Force the mip map access to be between the mipmap specified by the integer bits of the Min LOD and the ceiling of the value specified here.										
		<table><tr><th colspan="3">Programming Notes</th></tr><tr><td colspan="3">If Min LOD is greater than Max LOD, Min LOD takes precedence, i.e. the resulting LOD will always be Min LOD.</td></tr></table>		Programming Notes			If Min LOD is greater than Max LOD , Min LOD takes precedence, i.e. the resulting LOD will always be Min LOD .					
Programming Notes												
If Min LOD is greater than Max LOD , Min LOD takes precedence, i.e. the resulting LOD will always be Min LOD .												
	11:10	Reserved	Project: All Format: MBZ									



SAMPLER_STATE

9

Cube Surface Control Mode

Project: All

Format: U1 enumerated type FormatDesc

When sampling from a SURFTYPE_CUBE surface, this field controls whether the **TC* Address Control Mode** fields are interpreted as programmed or overridden to TEXCOORDMODE_CUBE.

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

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



SAMPLER_STATE

8:6	TCX Address Control Mode Project: All Format: U3 enumerated type



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



SAMPLER_STATE															
	24:23	ChromaKey Index Project: All Format: U2 FormatDesc Range [0,3] This field specifies the index of the ChromaKey Table entry associated with this Sampler. This field is a "don't care" unless ChromaKey Enable is ENABLED.													
	22	ChromaKey Mode Project: All Format: U1 enumerated type FormatDesc This field specifies the behavior of the device in the event of a ChromaKey match. This field is ignored if ChromaKey is disabled. KEYFILTER_KILL_ON_ANY_MATCH: In this mode, if any contributing texel matches the chroma key, the corresponding pixel mask bit for that pixel is cleared. The result of this operation is observable only if the Killed Pixel Mask Return flag is set on the input message. KEYFILTER_REPLACE_BLACK: In this mode, each texel that matches the chroma key is replaced with (0,0,0,0) (black with alpha=0) prior to filtering. For YCrCb surface formats, the black value is A=0, R(Cr)=0x80, G(Y)=0x10, B(Cb)=0x80. This will tend to darken/fade edges of keyed regions. Note that the pixel pipeline must be programmed to use the resulting filtered texel value to gain the intended effect, e.g., handle the case of a totally keyed-out region (filtered texel alpha==0) through use of alpha test, etc.													
		<table><tr><th>Value</th><th>Name</th><th>Description</th><th>Project</th></tr><tr><td>0h</td><td>KEYFILTER_KILL_ON_ANY_MATCH</td><td></td><td>All</td></tr><tr><td>1h</td><td>KEYFILTER_REPLACE_BLACK</td><td></td><td>All</td></tr></table>	Value	Name	Description	Project	0h	KEYFILTER_KILL_ON_ANY_MATCH		All	1h	KEYFILTER_REPLACE_BLACK		All	
Value	Name	Description	Project												
0h	KEYFILTER_KILL_ON_ANY_MATCH		All												
1h	KEYFILTER_REPLACE_BLACK		All												



SAMPLER_STATE

SAMPLER_STATE			
	21:19	Maximum Anisotropy Project: All Format: U3 enumerated type	



4.7.4 SAMPLER_BORDER_COLOR_STATE

This structure is pointed to by a field in SAMPLER_STATE.

The interpretation of the border color is as follows:

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

Programming Notes:

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

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



4.7.5 3DSTATE_CHROMA_KEY

3DSTATE_CHROMA_KEY		
Project:	All	Length Bias: 2
The 3DSTATE_CHROMA_KEY instruction is used to program texture color/chroma-key key values. A table containing four set of values is supported. The ChromaKey Index sampler state variable is used to select which table entry is associated with the map. Texture chromakey functions are enabled and controlled via use of the ChromaKey Enable texture sampler state variable. Texture Color Key (keying on a paletted texture index) is not supported.		
DWord	Bit	Description
0	31:29	Command Type Default Value: 3h GFXPIPE Format: OpCode
	28:27	Command SubType Default Value: 3h GFXPIPE_3D Format: OpCode
	26:24	3D Command Opcode Default Value: 1h 3DSTATE Format: OpCode
	24:16	3D Command Sub Opcode Default Value: 04h 3DSTATE_CHROMA_KEY Format: OpCode
	15:8	Reserved Project: All Format: MBZ
	7:0	DWord Length Default Value: 2h Excludes DWord (0,1) Format: =n Total Length - 2
1	31:30	ChromaKey Table Index Project: All Format: U2 index Range 0..3 Selects which entry in the ChromaKey table is to be loaded
	29:0	Reserved Project: All Format: MBZ
2	31:0	ChromaKey Low Value This field specifies the "low" (minimum) value of the chroma key range. Texel samples are considered "matching the key" if each component of the texel falls within the (inclusive) chroma range. See ChromaKey High Value for further format, programming info.



3DSTATE_CHROMA_KEY

3

31:0

ChromaKey High Value

This field specifies the "high" (maximum) value of the chroma key range. Texel samples are considered "matching the key" if each component of the texel falls within the (inclusive) chroma range.

Programming Notes

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

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

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

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

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

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

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

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



4.7.6 3DSTATE_SAMPLER_PALETTE_LOAD0

3DSTATE_SAMPLER_PALETTE_LOAD0			
Project: All		Length Bias:	2
The 3DSTATE_SAMPLER_PALETTE_LOAD0 instruction is used to load 24-bit values into the first texture palette. The texture palette is used whenever a texture with a paletted format (containing "Px [palette0]") is referenced by the sampler. This instruction is used to load all or a subset of the 16 entries of the first palette. Partial loads always start from the first (index 0) entry.			
DWord	Bit	Description	
0	31:29	Command Type Default Value: 3h GFXPIPE Format: OpCode	
	28:27	Command SubType Default Value: 3h GFXPIPE_3D Format: OpCode	
	26:24	3D Command Opcode Default Value: 1h 3DSTATE Format: OpCode	
	24:16	3D Command Sub Opcode Default Value: 02h 3DSTATE_SAMPLER_PALETTE_LOAD0 Format: OpCode	
	15:8	Reserved Project: All Format: MBZ	
	7:0	DWord Length Default Value: 0h Excludes DWord (0,1) Format: =n Total Length - 2	
1..n	31:24	Reserved : MBZ	
	23:0	Palette Color[0:N-1] Project: All Colors loaded into the first N entries of the texture color palette. Format = Bits 23:0 = U24 interpreted as RGB_888 color as follows: [23:16] Red [15:8] Green [7:0] Blue	



4.8 Messages

Restrictions:

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

4.8.1 Initiating Message

Execution Mask

SIMD16. The 16-bit execution mask forms the valid pixel signals. This determines which pixels are sampled and results returned to the GRF registers. Samples for invalid pixels are not overwritten in the GRF. However, if LOD needs to be computed for a subspan based on the message type and MIP filter mode and at least one pixel in the subspan being valid, the sampling engine assumes that the parameters for the upper left, upper right, and lower left pixels in the subspan are valid regardless of the execution mask, as these are needed for the LOD computation.

SIMD8. The lower 8 bits of the execution mask forms the valid pixel signals. If LOD needs to be computed based on MIP filter mode and at least one pixel in the subspan being valid, the sampling engine assumes that the parameters for the upper left, upper right, and lower left pixels in the subspan are valid regardless of the execution mask, as these are needed for the LOD computation.

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



4.8.1.1 Message Descriptor

Bit	Description
15:14	Message Type: Specifies the type of message being sent, along with the message length (in the general message descriptor) Format = U2 Refer to the table in section 4.8.1.3 for encoding details.
13:12	Data Return Format: Specifies the format of the data returned to the requesting thread. 00 = FLOAT32 – return a signed 32-bit IEEE Float to the thread. Required for all UNORM, SNORM, and FLOAT surface formats. Can be used by resinfo messages regardless of surface format. 01 = Reserved 10 = UINT32 – return an unsigned 32-bit integer. Required for all UINT surface formats. Can be used by resinfo messages regardless of surface format. 11 = SINT32 – return a signed 32-bit 2’s complement integer. Required for all SINT surface formats.
11:8	Sampler Index: Specifies the index into the sampler state table. Ignored for “ld” and “resinfo” type messages. Format = U4 Range = [0,15]
7:0	Binding Table Index: Specifies the index into the binding table. Format = U8 Range = [0,255]



4.8.1.2 Message Header

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

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



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

4.8.1.3 Payload Parameter Definition

The table below shows all of the messages supported by the sampling engine. The message type field in the message descriptor in combination with the message length determines which message is being sent. The table defines all of the *parameters* sent for each message type. The position of the parameters in the payload is given in the section following corresponding to the *SIMD mode* given in the table. The instruction column indicates the shader instructions expected to be translated to each message type.

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



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

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



4.8.1.4 Message Types

The behavior of each message type is as follows:

Message Type	Description
sample	The surface is sampled using the indicated sampler state. LOD is computed using gradients between adjacent pixels. One, two, or three parameters may be specified depending on how many coordinate dimensions the indicated surface type uses. Extra parameters specified are ignored. Missing parameters are defaulted to 0. Programming Notes: • The Surface Type of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_CUBE. • The Surface Format of the associated surface cannot be MONO8 or any UINT or SINT format. • sample is not supported in SIMD4x2 mode.
sample+killpix	The surface is sampled as in the sample message type. An additional register is returned after the sample results which contains the kill pixel mask. This message type is required to allow the result of a chroma key enabled sampler in KEYFILTER_KILL_ON_ANY_MATCH mode to affect the final pixel mask. Programming Notes: • The Surface Type of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_CUBE. • The Surface Format of the associated surface cannot be MONO8 or any UINT or SINT format. • sample+killpix is supported only in SIMD8 mode.
sample_b	The surface is sampled using the indicated sampler state. LOD is computed using gradients between adjacent pixels, then the value in the "bias" parameter is added to the LOD for each pixel. All four coordinates must be specified, however v and r may not be used depending on the indicated surface type. The LOD bias delivered in the "bias" parameter is restricted to a range of [-16.0, +16.0). Values outside this range produce undefined results. Programming Notes: • The Surface Type of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_CUBE. • The Surface Format of the associated surface cannot be MONO8 or any UINT or SINT format. • sample_b is not supported in SIMD4x2 mode.
sample_l	The surface is sampled using the indicated sampler state. LOD is not computed, but instead is taken from the "lod" parameter. All four coordinates must be specified, however v and r may not be used depending on the indicated surface type. Programming Notes: • The Surface Type of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_CUBE. • The Surface Format of the associated surface cannot be a UINT or SINT format.

Message Type	Description
sample_c	The surface is sampled using the indicated sampler state. All four coordinates must be specified, however v and r may not be used depending on the indicated surface type. The “ai” parameter indicates the array index for a cube surface. The “ref” parameter specifies the reference value that is compared against the red channel of the sampled surface, and the texel is replaced with either white or black depending on the result of the comparison. The WGF sample_c_lz instruction is implemented by issuing the sample_c message with Force LOD to Zero enabled in the message header or by issuing the sample_l_c message with the LOD parameter set to zero. Programming Notes: • The Surface Type of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, or SURFTYPE_CUBE. • 1D and 2D arrays are not supported (Depth of the associated surface must be 0). • The Surface Format of the associated surface must be indicated as supporting shadow mapping as indicated in the surface format table. • With sample_c, MIPFILTER_LINEAR, MAPFILTER_LINEAR, MAPFILTER_ANISOTROPIC are allowed <i>even for surface formats that are listed as not supporting filtering</i> in the surface formats table. • Use of the SIMD4x2 form of sample_c without Force LOD to Zero enabled in the message header is not allowed, as it is not possible for the hardware to compute LOD for SIMD4x2 messages. • Use of sample_c with SURFTYPE_CUBE surfaces is undefined with the following surface formats: I24X8_UNORM, L24X8_UNORM, A24X8_UNORM, I32_FLOAT, L32_FLOAT, A32_FLOAT. • [DevBW, DevCL] Errata: When sample_c is used on a texture map with A16_FLOAT surface format, any value read in from the texture map that is a NaN will be treated like a + inf.
sample_b_c	This is a combination of sample_b and sample_c. Both the LOD bias and reference values are delivered. All restrictions applying to both sample_b and sample_c must be honored.
sample_l_c	This is a combination of sample_l and sample_c. Both the LOD and reference values are delivered. All restrictions applying to both sample_l and sample_c must be honored. However, unlike sample_c, sample_l_c is allowed as a SIMD4x2 message. Programming Notes: • [DevBW, DevCL] Errata: SIMD4x2 sample_l_c is not allowed and must be worked around using SIMD8 sample_l_c.
sample_g (sample_d)	The surface is sampled using the indicated sampler state. LOD is computed using the gradients present in the message. The r coordinate and its gradients are required only for surface types that use the third coordinate. Usage of this message type on cube surfaces assumes that the u, v, and gradients have already been transformed onto the appropriate face, but still in [-1,+1] range. The r coordinate contains the faceid, and the r gradients are ignored by hardware. Programming Notes: • The Surface Type of the associated surface must be SURFTYPE_1D, SURFTYPE_2D, SURFTYPE_3D, or SURFTYPE_CUBE. • The Surface Format of the associated surface cannot be MONO8 or any UINT or SINT format.



Message Type	Description
resinfo	The surface indicated in the surface state is not sampled. Instead, the width, height, depth, and MIP count of the surface are returned in the red, green, blue, and alpha channels respectively (UINT32 format). The width, height, and depth are shifted right, per pixel, by the LOD value provided in the "lod" parameter to give the dimensions of the specified mip level. The "lod" parameter is an unsigned 32-bit integer in this mode (note that sending a signed 32-bit integer always has the same effect, as negative values are out-of-range when interpreted as unsigned integers). The Sampler State Pointer and Sampler Index are ignored. Programming Notes: • [DevBW-A,B] Errata: if lod is > 0xf it must be forced to 0xf.
ld (includes ld2dms)	The surface is sampled using a default sampler state, indicated below. The "lod" parameter contains the LOD of the mip map to be sampled. The v and r channel may also be ignored depending on the indicated surface type. All incoming values are unsigned 32-bit integers in this mode. The u, v, and r parameters contain integer texel addresses on the LOD indicated in the "lod" parameter. The Sampler State Pointer and Sampler Index are ignored. For the <i>ld</i> message type, the sampler state is defaulted as follows: • min, mag, and mip filter modes are "nearest" • all address control modes are "zero" (a special mode in which any texel off the map or outside the MIP range of the surface has a value of zero in all channels, except for surface formats without an alpha channel, which will return a value of one in the alpha channel) Programming Notes: • The Surface Type of the associated surface must be SURFYPE_1D, SURFYPE_2D, SURFYPE_3D, or SURFYPE_BUFFER. • [DevBW-A,B] Errata: Only non-array (Depth = 0) SURFYPE_1D and SURFYPE_2D are supported with "ld". • The Surface Format of the associated surface cannot be MONO8. • [DevBW, DevCL] Errata: For ld with SURFYPE_BUFFER the lod channel MBZ. • Errata: Surface formats with 8 bits per channel and no alpha channel will return zero in the alpha channel.

Programming Notes:

- For surfaces of type SURFYPE_CUBE, the sampling engine requires u, v, and r parameters that have already been divided by the absolute value of the parameter (u, v, or r) with the largest absolute value.



4.8.1.5 Parameter Types

sample*, LOD, and load4 messages

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

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

ld messages

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

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



4.8.1.6 SIMD16 Payload

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

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



4.8.1.7 SIMD8 Payload

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

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

4.8.1.8 SIMD4x2 Payload

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



4.8.2 Writeback Message

Corresponding to the four input message definitions are four writeback messages. Each input message generates a corresponding writeback message of the same type (SIMD16, SIMD8 or SIMD4x2).

4.8.2.1 SIMD16

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

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



4.8.2.2 SIMD8

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

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

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



4.8.2.3 SIMD4x2

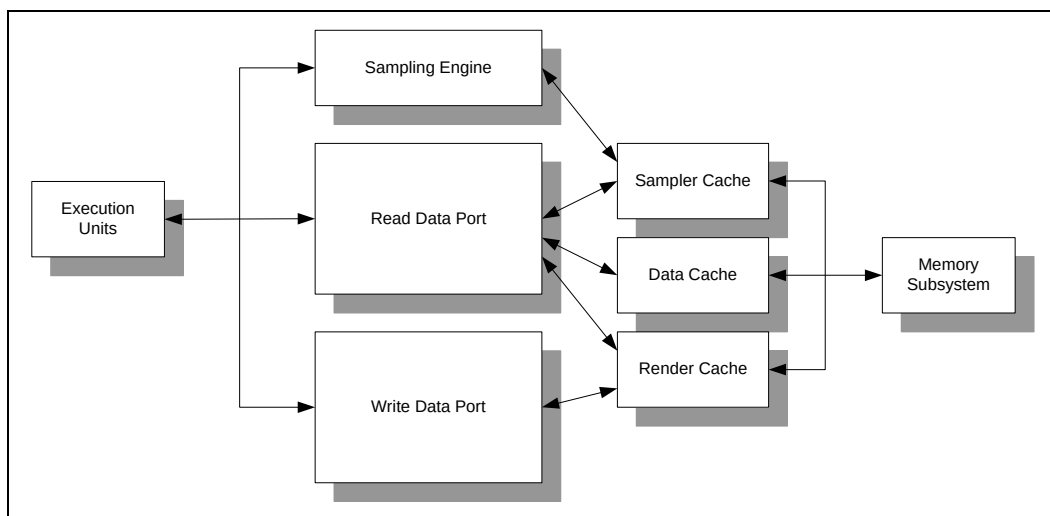
A SIMD4x2 writeback message always consists of a single message register containing all four channels of each of the two “pixels” (called “samples” here, as they are not really pixels) of data. The write channel mask bits as well as the execution mask on the “send” instruction are used to determine which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

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

5 Data Port

The Data Port provides all memory accesses for the Gen4 subsystem other than those provided by the sampling engine. These include render target writes, constant buffer reads, scratch space reads/writes, and media surface accesses.

The diagram below shows the two parts of the Data Port (Read and Write) and how they connect with the caches and memory subsystem. The execution units and sampling engine are shown for clarity.



The kernel programs running in the execution units communicate with the data port via messages, the same as for the other shared function units. The read and write data ports are considered to be separate shared functions, each with its own shared function identifier.

5.1 Cache Agents

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

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



5.1.1 Render Cache

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

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

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

5.1.2 Data Cache

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

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

5.1.3 Sampler Cache

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

5.2 Surfaces

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



5.2.1 Surface State Model

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

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

5.2.2 Stateless Model

The stateless model is used when a **Binding Table Index** (specified in the message descriptor) of 255 is specified. In this model, the binding table is not accessed, and the parameters that define the surface state are overloaded as follows:

- Surface Type = SURFTYPE_BUFFER
- Surface Format = R32G32B32A32_FLOAT
- Vertical Line Stride = 0
- Surface Base Address = **General State Base Address + Immediate Base Address**
- Buffer Size = checked only against **General State Access Upper Bound**
- Surface Pitch = 16 bytes
- Utilize Fence = false
- Tiled = false

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

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

5.3 Write Commit

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



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

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

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

5.4 Read / Write Ordering

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

5.5 Accessing Buffers

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

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

Message	Applications
OWord Block Read/Write	- constant buffer reads of a single constant or multiple contiguous constants - scratch space reads/writes where the index for each pixel/vertex is the same - block constant reads, scratch memory reads/writes for media
OWord Dual Block Read/Write	- SIMD4x2 constant buffer reads where the indices of each vertex/pixel are different (if there are two indices and they are the same, hardware will optimize the cache accesses and do only one cache access) - SIMD4x2 scratch space reads/writes where the indices are different.
DWord Scattered Read/Write	- SIMD8/16 constant buffer reads where the indices of each pixel are different (read one channel per message) - SIMD8/16 scratch space reads/writes where the indices are different (read/write one channel per message) - general purpose DWord scatter/gathering, used by media

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



5.6 Accessing Media Surfaces

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

5.6.1 Boundary Behavior

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

Media Boundary Pixel Mode	Vertical Line Stride	Vertical Line Stride Offset	Usage Model	Behavior
0	0	X	normal frame	00000 123 3333
0	1	0	normal field even	00000 2222222
0	1	1	normal field odd	11111 3333333
2	0	X	frame / progressive	00000 123 3333
2	1	0	field even / progressive	00000 2 333333
2	1	1	field odd / progressive	0000 1 3333333
3	0	X	frame / interlaced	01010 123 2323
3	1	0	field even / interlaced	00000 2222222
3	1	1	field odd / interlaced	11111 3333333



5.7 Accessing Render Targets

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

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

5.7.1 Single Source

The “normal” render target messages are single source. There are two forms, SIMD16 and SIMD8, intended for the equivalent-sized pixel shader threads. A single color (4 channels) is delivered for each of the 16 or 8 pixels in the message payload. Optional depth, stencil, and antialias alpha information can also be delivered with these messages.

The pixel scoreboard bits corresponding to the dispatched pixel mask (or half of the mask in the case of SIMD8 messages) are cleared only if the **Pixel Scoreboard Clear** bit is set in the message descriptor.

5.7.2 Dual Source [DevCL-B]

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

The dual source render target messages only have SIMD8 forms due to maximum message length limitations. SIMD16 pixel shaders must send two of these messages to cover all of the pixels. Each message contains two colors (4 channels each) for each pixel in the message payload. In addition to the first source, the second source can be selected as a blend factor (BLENDFACTOR_*_SRC1_* options in the blend factor fields of COLOR_CALC_STATE or BLEND_STATE). Optional depth, stencil, and antialias alpha information can also be delivered with these messages.

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

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



5.7.3 Replicate Data

The replicate data render target message is intended to be used for “fast clear” functionality in cases where the color data for each pixel is identical. This message performs better than the other messages due to its smaller message length. This message does not support depth, stencil, or antialias alpha data being sent with it. This message must target only tiled memory. Access of linear memory using this message type is UNDEFINED. The depth buffer can be cleared through the “early depth” function in conjunction with a pixel shader using this message. Refer to the *Windower* chapter for more details on the early depth function.

The pixel scoreboard bits corresponding to the dispatched pixel mask are cleared only if the **Pixel Scoreboard Clear** bit is set in the message descriptor.

5.7.4 Multiple Render Targets (MRT)

Multiple render targets are supported with the single source and replicate data messages. Each render target is accessed with a separate Render Target Write message, each with a different surface indicated (different binding table index). The depth buffer is written only by the message(s) to the last render target, indicated by the **Last Render Target Select** bit set to clear the pixel scoreboard bits.

5.8 Flushing the Render Cache

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

5.9 State

5.9.1 BINDING_TABLE_STATE

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

5.9.2 SURFACE_STATE

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



5.10 Messages

5.10.1 Global Definitions

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

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

Restrictions:

- **[DevBW-A,B,C0, DevCL-A0] Errata:** use of any Sampling Engine message in the same workload (between pipeline flushes) with any Data Port read messages utilizing the Sampler Cache is not allowed.
- Data port messages may not have the **End of Thread** bit set in the message descriptor other than the following exceptions:
 - The Render Target Write message may have **End of Thread** set for pixel shader threads dispatched by the windower in non-contiguous dispatch mode.
 - The Render Target UNORM Write message may have **End of Thread** set for pixel shader threads dispatched by the windower in contiguous dispatch mode.



5.10.1.1 Message Descriptor

Bit	Description		
DATA PORT READ TARGET		DATA PORT WRITE TARGET	
15:14	Target Cache 00 = Data Cache 01 = Render Cache 10 = Sampler Cache 11 = Reserved	15	Send Write Commit Message. Indicates that a write commit message will be sent back to the thread when the write has been committed. See section 5.3 for more details. Format = Enable
13:12	Read Message Type 00 = OWord Block Read 01 = OWord Dual Block Read 10 = Media Block Read 11 = DWord Scattered Read	14:12	Write Message Type 000 = OWord Block Write 001 = OWord Dual Block Write 010 = Media Block Write 011 = DWord Scattered Write 100 = Render Target Write 111 = Flush Render Cache All other encodings are reserved.
11:8	Message Specific Control. Refer to the specific message section for the definition of these bits.		
7:0	Binding Table Index. Specifies the index into the binding table for the specified surface. A binding table index of 255 indicates that a stateless model is to be used. Refer to section 5.2.2 for details on the stateless model. Programming Notes: [DevBW-A,B] Erratum BWT006: Using a binding table index of 255 is UNDEFINED unless the DAP Stateless Access ECO bit in the SVG-Debug Workaround Control register is set. This is a DEBUG ONLY mode that lacks necessary security checks. Format = U8 Range = [0,255]		



5.10.1.2 Message Header

This header applies to the following data port messages:

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

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

DWord	Bit	Description
M0.7	31:0	Debug
M0.6	31:0	Debug
M0.5	31:10	Immediate Buffer Base Address. Specifies the surface base address for messages in which the Binding Table Index is 255 (stateless model), otherwise this field is ignored. This pointer is relative to the General State Base Address . Format = GeneralStateOffset[31:10]
	9:8	Ignored
	7:0	Dispatch ID. This ID is assigned by the fixed function unit and is a unique identifier for the thread. It is used to free up resources used by the thread upon thread completion.
M0.4	31:0	Ignored (reserved for hardware delivery of binding table pointer)
M0.3	31:0	Ignored
M0.2	31:0	Global Offset. Specifies the global byte offset into the buffer. • For the OWord messages, this offset must be OWord aligned (bits 3:0 MBZ) • For the DWord messages, this offset must be DWord aligned (bits 1:0 MBZ) Format = U32 Range = [0,FFFFFFFF0h] for OWord messages Range = [0,FFFFFFFFCh] for DWord messages
M0.1	31:0	Ignored
M0.0	31:0	Ignored



5.10.1.3 Write Commit Writeback Message

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

DWord	Bit	Description
W0.7:0		Reserved

5.10.2 OWord Block Read / Write

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

Restrictions:

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

Applications:

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

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

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



preceding paragraph. **[DevBW], [DevCL] Errata:** Execution mask bits outside of those corresponding to the OWord being read/written cannot be asserted.

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

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

5.10.2.1 Message Descriptor

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



5.10.2.2 Message Payload (Write)

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

The **Offset** referred to below is the **Global Offset** and is in units of OWords (discard low 4 bits). The **OWord** array index is also in units of OWords.

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

5.10.2.3 Writeback Message (Read)

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

The **Offset** referred to below is the **Global Offset** and is in units of OWords (discard low 4 bits). The **OWord** array index is also in units of OWords.

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



5.10.3 OWord Dual Block Read / Write

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

Restrictions:

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

Applications:

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

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

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

5.10.3.1 Message Descriptor

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



5.10.3.2 Message Payload

DWord	Bit	Description
M1.7	31:0	Ignored
M1.6	31:0	Ignored
M1.5	31:0	Ignored
M1.4	31:0	Block Offset 1. Specifies the byte offset of OWord Block 1 into the surface. Must be OWord aligned (bits 3:0 MBZ). Format = U32 Range = [0,FFFFFFFFh]
M1.3	31:0	Ignored
M1.2	31:0	Ignored
M1.1	31:0	Ignored
M1.0	31:0	Block Offset 0

5.10.3.3 Additional Message Payload (Write)

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

The **Offset1 / 0** referred to below is the **Global Offset** added to the corresponding **Block Offset 1 / 0** and is in units of OWords (discard low 4 bits). The **OWord** array index is also in units of OWords.

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



5.10.3.4 Writeback Message (Read)

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

The **Offset1 / 0** referred to below is the **Global Offset** added to the corresponding **Block Offset 1 / 0** and is in units of OWords (discard low 4 bits). The **OWord** array index is also in units of OWords.

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

5.10.4 Media Block Read / Write

The read form of this message enables a rectangular block of data samples to be read from the source surface and written into the GRF. The write form enables data from the GRF to be written to a rectangular block.

Restrictions:

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



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

Applications:

- Block reads/writes for media

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

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

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

- For a surface with 8bpp pixels, the boundary byte is replicated. For example, for a boundary dword B0B1B2B3, to replicate the left boundary byte pixel, the out of bound dwords have the format of B0B0B0B0, and that for right boundary is B3B3B3B3.
 - This rule applies to all surface formats with BPE of 8. As the data port does not perform format conversion, the most likely used surface formats are R8_UINT and R8_SINT.
- For any other surfaces with 16bpp pixels, boundary pixel replication is on words. For example, for a boundary dword B0B1B2B3, to replicate the left boundary word pixel, the out of bound dwords have the format of B0B1B0B1, and that for right boundary is B2B3B2B3.
 - This rule applies to all surface formats with BPE of 16. As the data port does not perform format conversion, only the formats with integer data types may be useful in practice.
- For special surfaces with 16bpp pixels YUV422 packed format, there are two basic cases depending on the Y location: YUYV (surface format YCRCB_NORMAL) and UYVY (surface format YCRCB_SWAPY). Boundary handling for VYU (surface format YCRCB_SWAPUV) is the same as that for YUYV. Similarly, boundary handling for VYU (surface format YCRCB_SWAPUVY) is the same as that for UYVY. Note that these four surface formats have 16bpp pixels, even though the BPE fields are set to zero according to the table in the Surface Formats Section.
 - For a boundary dword Y0U0Y1V0, to replicate the left boundary, we get Y0U0Y0V0, and to replicate the right boundary, we get Y1U0Y1V0.
 - For a boundary dword U0Y0V0Y1, to replicate the left boundary, we get U0Y0V0Y0, and to replicate the right boundary, we get U0Y1V0Y1.
- For a surface with 32bpp pixels, the boundary dword pixel is replicated.
 - This rule applies to all surface formats with BPE of 32. As the data port does not perform format conversion, some of the formats may not be useful in practice.

Hardware behavior for any other surface types is undefined.



5.10.4.1 Message Descriptor

Bit	Description															
12	Cache Allocation Method This field is only allowed to be 1 only if resulting Vertical Line Stride (from surface state or being overridden by this message) is 1. This field is only valid for Sampler Cache read messages. This field is ignored for Render Cache messages (read/write). 0 = frame cache lines 1 = field cache lines this bit is part of the Message Type fields															
11	Pixel Scoreboard Clear. Reserved : MBZ															
10	Vertical Line Stride Override Specifies whether the Vertical Line Stride and Vertical Line Stride Offset fields in the surface state should be replaced by bits 9 and 8 below. If this field is 1, Height in the surface state (see SURFACE_STATE section of Sampling Engine chapter) is modified according the following rules: <table><tr><th>Vertical Line Stride (in surface state)</th><th>Override Vertical Line Stride</th><th>Derived 1-based surface height (As a function of the 0-based Height in surface state)</th></tr><tr><td>0</td><td>0</td><td>Height + 1 (Normal)</td></tr><tr><td>0</td><td>1</td><td>(Height + 1) / 2 <i>Restriction: (Height + 1) must be an even number.</i></td></tr><tr><td>1</td><td>0</td><td>(Height + 1) * 2</td></tr><tr><td>1</td><td>1</td><td>Height + 1 (Normal)</td></tr></table> For example, for a 720x480 standard resolution video buffer, if Vertical Line Stride in surface state is 0, i.e. a frame, Height (of the frame) should be 479. When accessing the bottom field of this frame video buffer, both Override Vertical Line Stride and Override Vertical Line Stride Offset will be set to 1, then the derived surface height (of the field) will be 240 ((Height + 1) / 2). In contrary, if Vertical Line Stride in surface state is 1 and Vertical Line Stride Offset in surface state is 0, the surface state represents the top field of the video buffer. In this case, Height (of the top field) should be programmed as 239. Accessing the bottom video field will use the same surface height of 240. Accessing the video frame (with Override Vertical Line Stride and Override Vertical Line Stride Offset set to 0) will result in a derived surface height of 480 ((Height + 1) * 2). 0 = Use parameters in the surface state and ignore bits 9:8 1 = Use bits 9:8 to provide the Vertical Line Stride and Vertical Line Stride Offset [DevBW-A] Erratum: This field is ignored by hardware.	Vertical Line Stride (in surface state)	Override Vertical Line Stride	Derived 1-based surface height (As a function of the 0-based Height in surface state)	0	0	Height + 1 (Normal)	0	1	(Height + 1) / 2 <i>Restriction: (Height + 1) must be an even number.</i>	1	0	(Height + 1) * 2	1	1	Height + 1 (Normal)
Vertical Line Stride (in surface state)	Override Vertical Line Stride	Derived 1-based surface height (As a function of the 0-based Height in surface state)														
0	0	Height + 1 (Normal)														
0	1	(Height + 1) / 2 <i>Restriction: (Height + 1) must be an even number.</i>														
1	0	(Height + 1) * 2														
1	1	Height + 1 (Normal)														



Bit	Description
9	Override Vertical Line Stride Specifies number of lines (0 or 1) to skip between logically adjacent lines – provides support of interleaved (field) surfaces as textures. Format = U1 in lines to skip between logically adjacent lines [DevBW-A] Erratum: This field is ignored by hardware.
8	Override Vertical Line Stride Offset Specifies the offset of the initial line from the beginning of the buffer. Ignored when Override Vertical Line Stride is 0. Format = U1 in lines of initial offset (when Vertical Line Stride == 1) [DevBW-A] Erratum: This field is ignored by hardware.

5.10.4.2 Message Header

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



DWord	Bit	Description										
	21:16	Block Height. Height in rows of block being accessed. Programming Notes: The Block Height is restricted to the following maximum values depending on the Block Width: <table><thead><tr><th>Block Width (bytes)</th><th>Maximum Block Height (rows)</th></tr></thead><tbody><tr><td>1-4</td><td>64</td></tr><tr><td>5-8</td><td>32</td></tr><tr><td>9-16</td><td>16</td></tr><tr><td>17-32</td><td>8</td></tr></tbody></table> Format = U6 Range = [0,63] representing 1 to 64 rows	Block Width (bytes)	Maximum Block Height (rows)	1-4	64	5-8	32	9-16	16	17-32	8
	Block Width (bytes)	Maximum Block Height (rows)										
	1-4	64										
5-8	32											
9-16	16											
17-32	8											
15:5	Ignored											
	4:0	Block Width. Width in bytes of the block being accessed. Programming Notes: - Must be DWord aligned for the write form of the message. - This field must also be DWord aligned for the read form of the message. Format = U5 Range = [0,31] representing 1 to 32 Bytes										
M0.1	31:0	Y offset. The Y offset of the upper left corner of the block into the surface. Format = S31										
M0.0	31:0	X offset. The X offset of the upper left corner of the block into the surface. Must be DWord aligned (Bits 1:0 MBZ) for the write form of the message. The X offset field defines the offset in the input message block. This may differ from the offset in the surface if Color Processing is enabled due to format conversion. This field must also be DWord aligned for the read form of the message. Format = S31										



5.10.4.3 Message Payload (Write)

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

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

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

5.10.4.4 Writeback Message (Read)

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



5.10.5 DWord Scattered Read / Write

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

For read messages with X/Y offsets that are outside the bounds of the surface, the address is clamped to the nearest edge of the surface. For write messages with X/Y offsets that are outside the bounds of the surface, the behavior is undefined.

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

Restrictions:

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

Applications:

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

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

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



5.10.5.1 Message Descriptor

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

5.10.5.2 Message Payload

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



5.10.5.3 Additional Message Payload (Write)

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

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

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



5.10.5.4 Writeback Message (Read)

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

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

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



5.10.6 Render Target Write

This message takes four subspans of pixels for write to a render target. Depending on parameters contained in the message and state, it may also perform a depth and stencil buffer write and/or a render target read for a color blend operation. Additional operations enabled in the Color Calculator state will also be initiated as a result of issuing this message (depth test, alpha test, logic ops, etc.). This message is intended only for use by pixel shader kernels for writing results to render targets.

Restrictions:

- All surface types are allowed.
- Dual Source messages are not supported on **DevBW** and **DevCL-A**
- For SURFTYPE_BUFFER and SURFTYPE_1D surfaces, only the X coordinate is used to index into the surface. The Y coordinate must be zero.
- For SURFTYPE_1D, 2D, 3D, and CUBE surfaces, a **Render Target Array Index** is included in the input message to provide an additional coordinate. The **Render Target Array Index** must be zero for SURFTYPE_BUFFER.
- The surface format is restricted to the set supported as render target. If source/dest color blend is enabled, the surface format is further restricted to the set supported as alpha blend render target.
- Only one pair of dual source messages is allowed per thread, as these messages implicitly clear the pixel scoreboard. In addition, a thread sending dual source messages is not allowed to send any other render target write messages.
- The last message sent to the render target by a thread must have the **End Of Thread** bit set in the message descriptor and the dispatch mask set correctly in the message header to enable correct clearing of the pixel scoreboard.
- The stateless model cannot be used with this message (**Binding Table Index** cannot be 255).
- This message can only be issued from a kernel specified in WM_STATE or 3DSTATE_WM (pixel shader kernel), dispatched in non-contiguous mode. Any other kernel issuing this message will cause undefined behavior.
- The dual source message cannot be used if the **Antialias Alpha Present to Render Target** bit in the message header is enabled.
- The dual source message cannot be used if the **Alpha Test Enable** bit in COLOR_CALC_STATE is enabled.
- This message cannot be used on a surface in field mode (**Vertical Line Stride** = 1)

Execution Mask. The execution mask for render target messages is ignored. Control of which pixels are active is controlled by the **Pixel / Sample Enables** fields in the message header.

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



5.10.6.1 Subspan / Pixel to Slot Mapping

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

Pixels are numbered as follows within a subspan:

- 0 = upper left
- 1 = upper right
- 2 = lower left
- 3 = lower right

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

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

5.10.6.2 Message Descriptor

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



5.10.6.3 Message Header

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

DWord	Bit	Description
M0.7	31:0	Debug
M0.6	31:0	Debug
M0.5	31:8	Ignored
	7:0	FFTID. The Fixed Function Thread ID is assigned by the fixed function unit and is a unique identifier for the thread. It is used to free up resources used by the thread upon thread completion.
M0.4	31:0	Ignored (reserved for hardware delivery of binding table pointer)
M0.3	31:0	Ignored
M0.2	31:0	Ignored
M0.1	31:6	Color Calculator State Pointer. Specifies the 64-byte aligned pointer to the color calculator state. This pointer is relative to the General State Base Address . Format = GeneralStateOffset[31:6]
	5:0	Ignored
M0.0	31:16	Dispatched Pixel Enables. One bit per pixel indicating which pixels were originally enabled when the thread was dispatched. This field is only required for the end-of-thread message and on all dual-source messages. The Dispatched Pixel Enables <i>must be unmodified</i> from the ones sent when the pixel shader thread was initiated. If the Dispatched Pixel Enables are modified, behavior is undefined.
	15:0	Pixel Enables. One bit per pixel indicating which pixels are still lit based on kill instruction activity in the pixel shader. This mask is used to control actual writes to the color buffer.
M1.7	31	Ignored
	30:27	Viewport Index. Specifies the index of the viewport currently being used. Format = U4 Range = [0,15]

DWord	Bit	Description														
	26:16	Render Target Array Index. Specifies the array index to be used for the following surface types: SURFTYPE_1D: specifies the array index. Range = [0,511] SURFTYPE_2D: specifies the array index. Range = [0,511] SURFTYPE_3D: specifies the "z" or "r" coordinate. Range = [0,2047] SURFTYPE_CUBE: specifies the face identifier. Range = [0,5] SURFTYPE_BUFFER: must be zero. <table><tr><th>face</th><th>Render Target Array Index</th></tr><tr><td></td><td>0</td></tr><tr><td>x</td><td>1</td></tr><tr><td>x</td><td>2</td></tr><tr><td>y</td><td>3</td></tr><tr><td>y</td><td>4</td></tr><tr><td>z</td><td>5</td></tr></table> Format = U11^z The Render Target Array Index used by hardware for access to the Render Target is overridden with the Minimum Array Element defined in SURFACE_STATE if it is out of the range between Minimum Array Element and Depth. For cube surfaces, a depth value of 5 is used for this determination.	face	Render Target Array Index		0	x	1	x	2	y	3	y	4	z	5
	face	Render Target Array Index														
	0															
x	1															
x	2															
y	3															
y	4															
z	5															
	15:0	Ignored														
M1.6	31	Front / Back Facing Polygon. Determines whether the polygon is front or back facing. Used by the render cache to determine which stencil test state to use. 0 = Front Facing 1 = Back Facing														
	30	Ignored														
	29	Source Depth Present to Render Target. Indicates that source depth is included in the message. If Destination Depth Present is also set, the depth test and conditional write of the depth buffer must be performed. If Destination Depth Present is not set, no depth test is performed but the source depth value is conditionally written to the depth buffer.														
	28	Destination Depth Present to Render Target. Indicates that destination depth is included in the message, and that the depth test and conditional write of the depth buffer must be performed. It is not valid to have Destination Depth Present without Source Depth Present.														
	27	Destination Stencil Present to Render Target. Indicates that destination stencil is included in the message, and that the stencil test and conditional write of the stencil buffer must be performed.														
	26	Antialias Alpha Present to Render Target. Indicates that antialias alpha is included in the message, and that the antialias function must be performed.														
	25:0	Ignored														



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

5.10.6.4 Stencil and Antialias Alpha Payload

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

Note that the Antialias Alpha values are U0.4.

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



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



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

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



5.10.6.5 Color Payload: SIMD8 Single Source

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

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



5.10.6.6 Color Payload: SIMD16 Replicated Data

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

This message is legal with color data only. The registers for depth, stencil, and antialias alpha data cannot be included with this message, and the corresponding bits in the message header must indicate that these registers are not present.

The value of 'm' here is equal to 2.

Programming Notes:

- This message is allowed only on tiled surfaces

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



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

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

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

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



5.10.6.8 Depth Payload

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

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

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



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

5.10.6.9 Message Sequencing Summary

This section summarizes the sequencing that occurs for each legal render target write message. All messages have the M0 and M1 header registers, thus they are not shown in the table. All cases not shown in this table are illegal.

Key:

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



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



5.10.7 Flush Render Cache

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

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

5.10.7.1 Message Descriptor

Bit	Description
12	Ignored
11:8	Ignored

5.10.7.2 Message Payload

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

§§





6 *Extended Math*

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

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

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

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



6.1 Messages

Restrictions:

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

6.1.1 Initiating Message

6.1.1.1 Message Descriptor

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



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

6.1.1.2 Scalar and Vector Mode

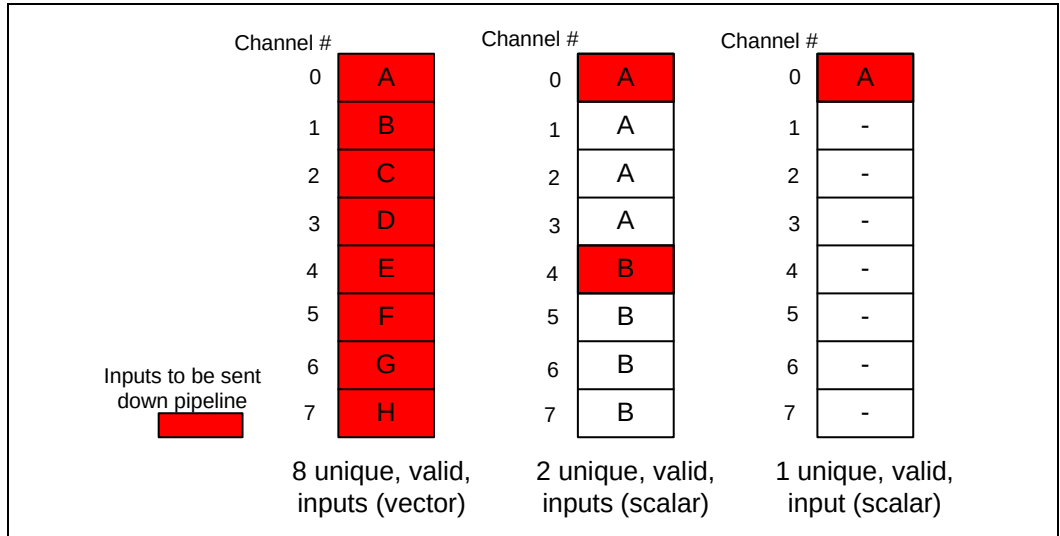
For a given request message, the Extended Math examines the 8-bit channel enable field and the Source Structure field in the message descriptor to determine which dwords contain valid inputs. There are two general cases that EM sees.

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



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

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





6.1.1.3 Message Payload

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

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

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



6.1.2 Writeback Message

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

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

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



6.2 Performance

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

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

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

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

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



6.3 Function Reference

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

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

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

6.3.1 INV

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

Format: INV <dst0> <src0>

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

Precision: 1 ULP

Src->	+inf	+0 / +Denorm	- 0 / - Denorm	-inf	NaN
Dest – IEEE mode	+0	+inf	-inf	-0	NaN
Dest – ALT mode		+FLT_MA X	- FLT_MA X		NaN

6.3.2 LOG

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



Format: LOG <dst0> <src0>

Pseudocode:

```

for (n = 0; n < 8; n++) {
    int srcCh = (vector mode) ? n : ((n < 4) ? 0 : 4)
    if (EMask.channel[n] == 1) {
        dst0.channel[n] = Log2(src0.channel[srcCh])
    }
}

```

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

Notes: In ALT mode log is computed as Log₂ (abs (src0))

Src->	+inf	+0 / +Deno rm	-0 / - Denor m	-inf	-F	NaN
Dest – IEEE mode	+inf	-inf	-inf	NaN	NaN	NaN
Dest – ALT mode		- FLT_M AX	- FLT_M AX		+F	NaN

6.3.3 EXP

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

Format: EXP <dst0> <src0>

Pseudocode:

```

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

```




```

dst0.channel[n] = 2src0.channel[srcCh]
    }
}

```

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

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



6.3.4 SQRT

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

Format: SQRT <dst0> <src0>

Pseudocode:

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

Precision: 1 ULP

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

Src->	+ in f	+0 / +Denorm	-0 / - Denorm	-inf	-F	NaN
Dest – IEEE mode	+ i n f	0	-0	NaN	NaN	NaN
Dest – ALT mode		0	0		+F	NaN



6.3.5 RSQ

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

32-bit FP values

Format: RSQ <dst0> <src0>

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

Precision: 1 ULP

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

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

6.3.6 POW

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

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

```
Pseudocode:
    for (n = 0; n < 8; n++) {
        int srcCh = (vector mode) ? n : ((n < 4) ? 0 : 4)
        if (EMask.channel[n] == 1) {
            dst0.channel[n] = 2src1·log2(abs(src0.channel[srcCh]))
        }
    }
```

Precision: 2⁻¹⁵ relative error



IEEE Mode:

Src0->

Src1	abs(F > 1)	abs (F < 0)	abs(+F == 1)	+inf f	+0 / +Denorm	-Denorm / -0	- inf	N a
+inf	+inf	0	NaN	+inf f	0	0	+i nf	N a
+0 / Denorm	1	1	1	NaN	NaN	NaN	NaN	NaN
-0 / Denorm	1	1	1	NaN	NaN	NaN	NaN	NaN
-inf	0	+inf	NaN	0	+inf	+inf	0	NaN
-F	+F	+F	+F	0	+inf	+inf	0	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
+F	+F	N	NaN	+inf f	0	0	NaN	NaN

ALT Mode:

Src0->

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



6.3.7 SIN

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

Format: SIN <dst0> <src0>

Pseudocode:

```

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

```

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

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



6.3.8 COS

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

Format: SIN <dst0> <src0>

Pseudocode:

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

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

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



6.3.9 SINCOS

Description:	Computes the sine of src0 (in radians) and stores computed result in dst0. Computes the cosine of src0 (in radians) and returns the result to dst1. Src0, dst0 and dst1 are 32-bit FP values.
Format:	SINCOS <dst0> <dst1> <src0>
Pseudocode:	<pre>for (n = 0; n < 8; n++) { int srcCh = (vector mode) ? n : ((n < 4) ? 0 : 4) if (EMask.channel[n] == 1) { if(dst0 != NULL){ dst0.channel[n] = Sin(src0.channel[srcCh]) } if(dst1 != NULL){ dst1.channel[n] = Cos(src0.channel[srcCh]) } } }</pre>
Precision:	Max absolute error of 0.0008 for the range of +/- 100 * pi. Outside of the above range the function will remain periodic, producing values between -1 and 1. However, the period of SINCOS is determined by the internal representation of Pi, meaning that as the magnitude of input increases the absolute error will, in general, also increase.
Notes:	See individual Sin and Cos tables for error handling



6.3.10 INT DIV

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

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

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

Precision: 32-bit integer

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

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



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

§§





7 Message Gateway

The Message Gateway shared function provides a mechanism for active thread-to-thread communication. Such thread-to-thread communication is based on direct register access. One thread, a **requester thread**, is capable of writing into the GRF register space of another thread, a **recipient thread**. Such direct register access between two threads in a multi-processor environment some time is referred to as **remote register access**. Remote register access may include read or write. GEN4 architecture supports **remote register write**, but not remote register read (natively). Message Gateway facilitates such remote register write via message passing. The requester thread sends a message to Message Gateway requesting a write to the recipient thread's GRF register space. Message Gateway sends a writeback message to the recipient thread to complete the register write on behave of the requester. The requester thread and the recipient thread may be on the same EU or on different EUs.

7.1 Messages

Message Gateway supports such thread-to-thread communication with the following three messages:

- **OpenGateway:** opens a gateway for a requester thread. Once a thread successfully opens its gateway, it can be a recipient thread to receive remote register write.
- **CloseGateway:** closes the gateway for a requester thread. Once a thread successfully closes its gateway, Message Gateway will block any future remote register writes to this thread.
- **ForwardMsg:** forwards a formatted message (remote register write) from a requester thread to a recipient thread.



7.1.1 Message Descriptor

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

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

7.1.2 OpenGateway Message

The OpenGateway message opens a communication channel between the requesting thread and other threads. It specifies a key for other threads to access its gateway, as well as the GRF register range allowed to be written. The message consists of a single 256-bit message payload.



If the AckReq bit is set, a single 256-bit payload writeback message is sent back to the requesting thread after completion of the OpenGateway function. Only the least significant DWord in the post destination register is overwritten.

If the EOT is set for this message, Message Gateway will ignore this message; instead, it will close the gateway for the requesting thread regardless of the previous state of the gateway.

It is software's policy to determine how to generate the key.

7.1.2.1 Message Payload

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



7.1.2.2 Writeback Message

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

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

7.1.3 CloseGateway Message

The CloseGateway message closes a communication channel for the requesting thread that was previously opened with OpenGateway. Each thread is allowed to have only one open gateway at a time, thus no additional information in the message payload is required to close the gateway. The message consists of a single 256-bit message payload.

If the AckReq bit is set, a single 256-bit payload writeback message is sent back to the requesting thread after completion of the CloseGateway function. Only the least significant DWord in the post destination register is overwritten.

7.1.3.1 Message Payload

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



DWord	Bit	Description
M0.4:0		Ignored

7.1.3.2 Writeback Message

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

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

7.1.4 ForwardMsg Message

The ForwardMsg message gives the ability for a requester thread to write a **data segment** in the form of a byte, a dword, 2 dwords, or 4 dwords to a GRF register in a recipient thread. The message consists of a single 256-bit message payload, which contains the specially formatted data segment.

The ForwardMsg message utilizes a communication channel previously opened by the recipient thread. The recipient thread has communicated its EUID, TID, and key to the requester thread previously via some other mechanism. Generally, this is done through the thread spawn message from parent to child thread, allowing each child (requester) to then communicate with its parent through a gateway opened by the parent (recipient). The child could then use ForwardMsg message to communicate its own EUID, TID, and key back to the parent to enable bi-directional communication after opening its own gateway.

If the AckReq bit is set, a single 256-bit payload writeback message is sent back to the requester thread after completion of the ForwardMsg function. Only the least significant DWord in the post destination register is overwritten.

If the Notify bit in the message descriptor is set, a 'notification' is sent to the recipient thread in order to increment the recipient thread's notification counter. This allows multiple messages to be sent to the recipient without waking up the recipient thread. The last message, having this bit set, will then wake up the recipient thread.



7.1.4.1 Message Payload

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



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

7.1.4.2 Writeback Message to Requester Thread

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

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

7.1.4.3 Writeback Message to Recipient Thread

This message contains the byte or dwords data segment indicated in the message written to the GRF register offset indicated. Only the byte/dword(s) will be enabled, all other data in the GRF register is untouched.



7.1.5 GetTimeStamp Message

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

AbsoluteTimeLap is based on an absolute wall clock in unit of nSec/uSec that is independent of context switch or GPU frequency adjustment. Message Gateway shares the same GPU timestamp. Details can be found in the **TIMESTAMP** register section in *vol1 Memory Interface and Command Stream*.

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

7.1.5.1 Message Payload

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



7.1.5.2 Writeback Message to Requester Thread

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

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

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

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

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

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



§§



8 Unified Return Buffer (URB)

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

8.1 URB Size

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

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

8.2 URB Access

The URB can be written by the following agents:

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

The URB can be read by the following agents:

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

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



8.3 State

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

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

8.4 Messages

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

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

Programming Notes:

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

8.4.1 Execution Mask

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



8.4.2 Message Descriptor

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

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



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

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



8.4.3 URB_WRITE

8.4.3.1 URB_WRITE Message Header

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



8.4.3.2 URB_WRITE Message Payload

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

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

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



8.4.3.2.1 URB_NOSWIZZLE

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

Programming Notes:

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

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

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



8.4.3.2.2 URB_INTERLEAVED

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

Programming Restrictions:

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

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



8.4.3.2.3 URB_TRANSPOSE

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

Programming Notes:

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

Table 8-1. URB_TRANSPOSE Payload

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



Table 8-2.URB_TRANSPOSE URB Destination Layout

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

8.4.3.3 Writeback Message for URB Entry Allocate

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

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

§§





9 *Execution Unit ISA*

9.1 Introduction

9.1.1 Objective and Scope

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

9.1.2 Terms and Acronyms

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



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



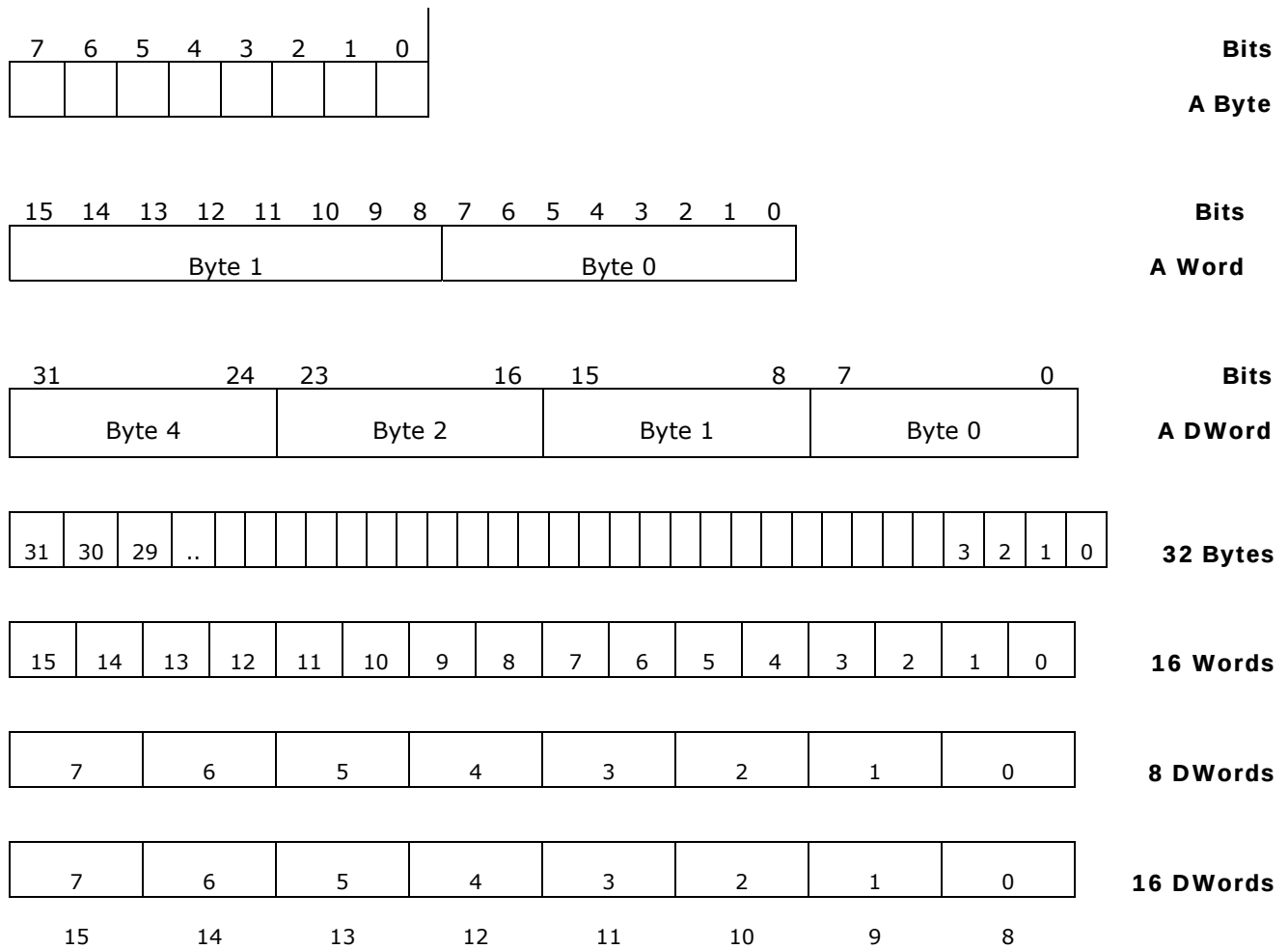
DQ	Double Quad word (DQword). As a fundamental data type, DQ represents 16 bytes.
POR	Plan Of Record
QW	Quad Word (QWord). As a fundamental data type, QW represents 8 bytes.
QQ	Quad Quad word (QQword). As a fundamental data type, QQ represents 32 bytes.
Sub-Register	Subfield of a SIMD register. A SIMD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



VF	Immediate floating point vector. As a numerical data type of 32 bits, an immediate floating point vector of type VF contains 4 floating point elements with 8-bit each. The 8-bit floating point element contains a sign field, a 3-bit exponent field and a 4-bit mantissa field. It may be used to specify the type of an immediate operand in an instruction.
W	Word. As a numerical data type of 16 bits, W represents a signed word integer. It is used to specify the type of an operand in an instruction.
URB	Unified Return Buffer. The on-chip memory managed/shared by GEN4 Fixed Functions. Threads use the URB to return data that will be consumed either by a Fixed Function or other threads.
UB	Unsigned Byte integer. A numerical data type of 8 bits. It may be used to specify the type of an operand in an instruction.
UD	Unsigned Double Word integer. A numerical data type of 32 bits. It may be used to specify the type of an operand in an instruction.
UW	Unsigned Word integer. A numerical data type of 16 bits. It may be used to specify the type of an operand in an instruction.
VFE	Video Front End. VFE is the first fixed function stage of the media pipeline.

9.1.3 Formats and Conventions

In order to conveniently (and without ambiguity) describe the register files with 256-bit wide registers that may contain various data types with different data element widths, it is important to use a consistent table format to represent the registers. Throughout this document, we will adopt the following table formats and conventions. When a register or a number is presented by a row, increasing order is always **from right to left** and then **top down** pictorially. In other words, for a bit field, the LSB to MSB is from right to left; for a byte sequence, the least significant byte to the most significant byte is also from right to left. This is consistent with the 'Little Endian' convention used by IA-32 machines. The following tables depict the layout formats for different data units.



With this convention, we note that the execution channels are logically viewed as from right to left too, which is a little bit unconventional. However, as shown in the *GEN4 Execution Environment* Chapter, it matches with the bit order of the flag registers. This also impacts the view of a GRF register region, now the region origin is located at the upper-right corner and a region row is viewed from right to left.



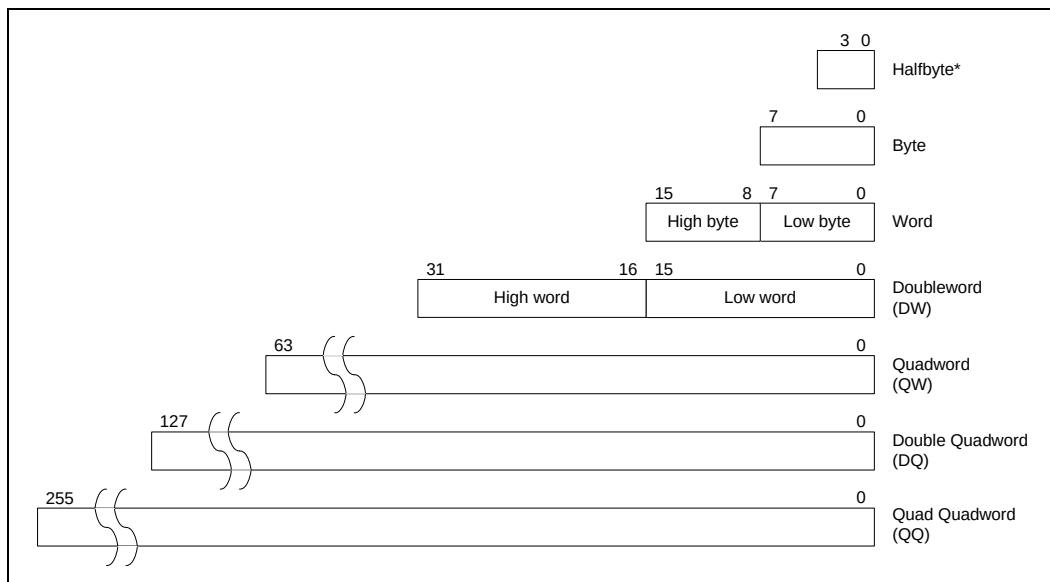


10 EU Data Types

10.1 Fundamental Data Types

The fundamental data types in the GEN4 architecture are halfbyte, byte, word, doubleword (DW), quadword (QW), double quadword (DQ) and quad quadword (QQ). They are defined based on the number of bits of the data type, ranging from 4 bits to 256 bits. As shown in Figure 10-1, a halfbyte contains 4 bits, a byte contains 8 bits, a word contains two bytes, and a doubleword (dword) contains two words, and so on. Halfbyte is a special data type such that it cannot be accessed directly as standalone data element. It is only allowed as a subfield of the numerical data type of “packed signed halfbyte integer vector” described in the next section.

Figure 10-1. Fundamental data types



With the exception of halfbyte, the access of a data element to/from a GEN4 register or to/from memory must be aligned on the natural boundaries of the data type. The natural boundary for a word has an even-numbered address in unit of byte. The natural boundary for a doubleword has an address divisible by 4 bytes. Similarly, the natural boundary for a quadword, double quadword and quad quadword has an address divisible by 8, 16, and 32 bytes, respectively. Quadword, double quadword and quad quadword do not have corresponding numerical data type. Instead, they are used to describe a group (a vector) of numerical data elements of smaller size align to larger natural boundaries.

10.2 Numerical Data Types

The numerical data types defined in the GEN4 architecture include signed and unsigned integers and floating-point numbers (floats) of various numbers of bits. These numerical data types are pictorially illustrated in Figure 10-2 and Figure 10-3. Table 10-1 details the notation, size and numerical range of each data type. The largest numerical data type has 32 bits.

Figure 10-2. Integer numerical data types

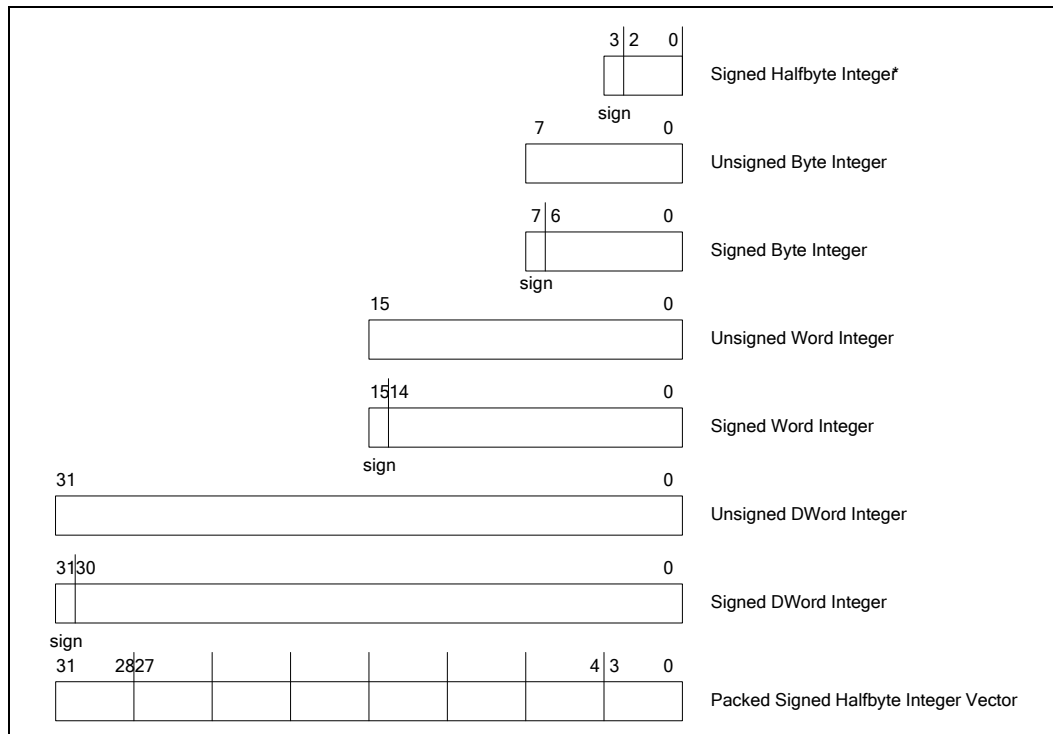


Figure 10-3. Floating point numerical data types

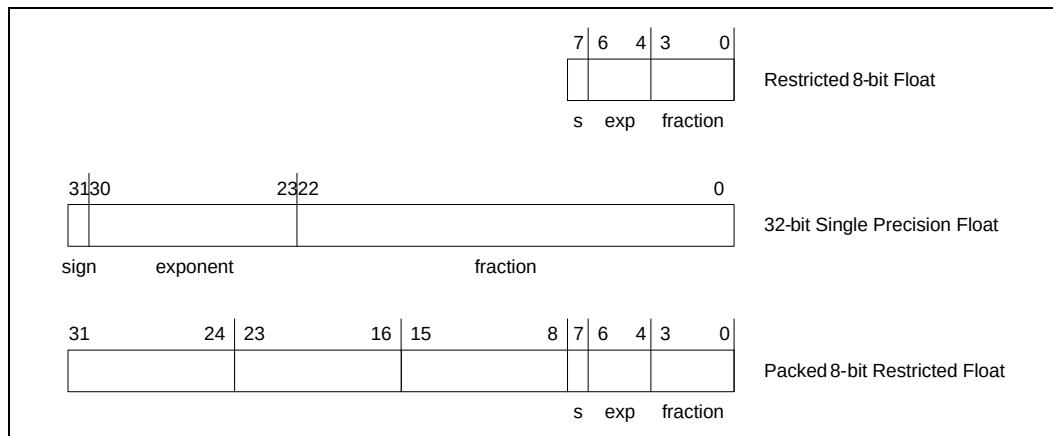




Table 10-1. Formats and ranges of numerical data types

Notation	Numerical Data Types	Fundamental Data Type	Range
UB	Unsigned Byte Integer	Byte	[0, 255]
B	Signed Byte Integer	Byte	[-128, 127]
UW	Unsigned Word Integer	Word	[0, 65535]
W	Signed Word Integer	Word	[-32768, 32767]
UD	Unsigned Doubleword Integer	Doubleword	[0, $2^{32} - 1$]
D	Signed Doubleword Integer	Doubleword	$[-2^{31}, 2^{31} - 1]$
F	Single Precision Float	Doubleword	$[-(2^{-23})^{127} \dots -2^{-149}, 0.0, 2^{-149} \dots (2^{-23})^{127}]$
n/a	Signed Halfbyte Integer	Halfbyte	[-8, 7]
V	Packed Signed Halfbyte Integer Vector	Doubleword	[-8, 7]
n/a	Restricted 8-bit Float	Byte	[-31...-0.125, 0, 0.125... 31]
VF	Packed Restricted Float Vector	Doubleword	[-31...-0.125, 0, 0.125... 31]

10.2.1 Unsigned Integers

Unsigned integers are unsigned binary numbers contained in a byte, a word or a doubleword. The range for an unsigned byte integer is from 0 to 255. The range for an unsigned word integer is from 0 to 65535. The range for an unsigned doubleword integer is from 0 to $2^{32} - 1$.

The short hand notation for an unsigned byte integer, an unsigned word integer, and an unsigned doubleword integer is **UB**, **UW**, **UD**, respectively.

10.2.2 Signed Integers

Signed integers are signed binary number in 2's complement form contained in a halfbyte, a byte, a word or a doubleword. A signed halfbyte integer has a numerical range from -8 to 7 with the sign bit at bit 3. A signed byte integer has a range from -128 to 127 with the sign at bit 7. A signed word integer is has a range from -32768 to 32767 with the sign at bit 15. A signed doubleword integer has a range from -2^{31} to $2^{31} - 1$ with the sign at bit 31.

The short hand notation for a signed byte integer, a signed word integer, and a signed doubleword integer is **B**, **W**, **D**, respectively.

10.2.3 Single Precision Floating-Point Numbers

The single precision floating point numbers is contained in a doubleword. Floating point format is as defined in IEEE Standard 754 for Binary Floating-Point Arithmetic. Maximal representable number is $(2-2^{-23})^{127}$ and the minimal number is $-(2-2^{-23})^{127}$. The smallest fractional negative number -2^{-149} and the smallest fractional positive number is 2^{-149} . Value 0.0 has no fractional parts.

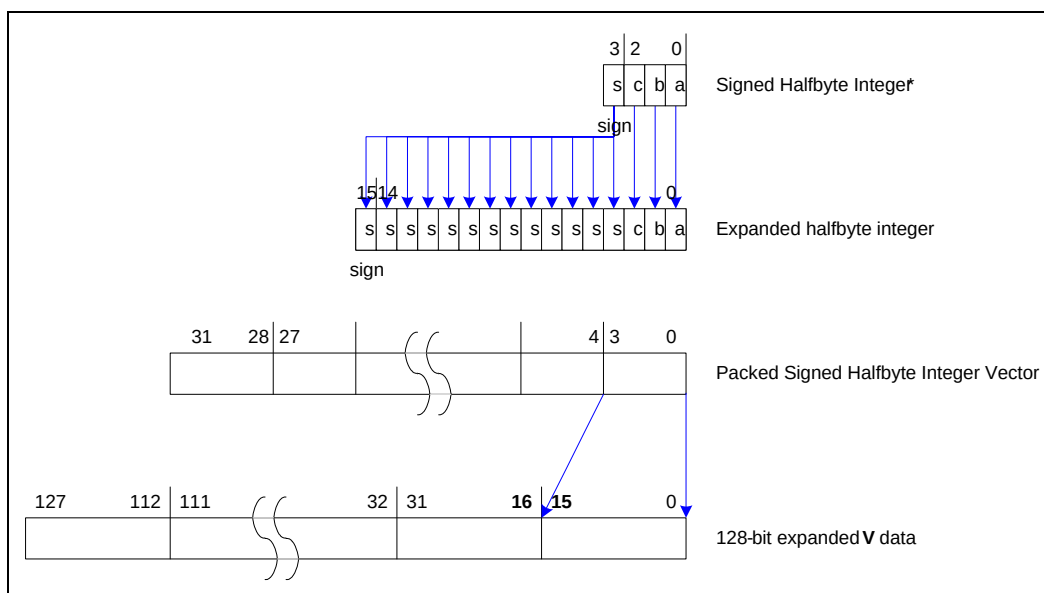
The short hand format notation for a single precision floating-point number is **F**.

10.2.4 Packed Signed Half-Byte Integer Vector

A packed signed halfbyte integer vector consists of 8 signed halfbyte integers contained in a doubleword. Each signed halfbyte integer element has a range from -8 to 7 with the sign on bit 3. This numerical data type is only used by an immediate source operand of doubleword in a GEN4 instruction. It cannot be used for the destination operand or a non-immediate source operand. GEN4 hardware converts the 32-bit vector into 8-element signed word vector by sign extension. This is illustrated in Figure 10-4.

The short hand format notation for a packed signed half-byte vector is **V**.

Figure 10-4. Converting a Packed Half-byte Vector to a 128-bit Signed Integer Vector





10.2.5 Packed 8-bit Restricted Float Vector

A packed restricted float vector consists of 4 8-bit restricted floats contained in a doubleword. Each restricted float has the sign at bit 7, a 3-bit coded exponent in bits 4 to 6, a 4-bit fraction in bits 0 to 3, and an implied integer 1. The exponent is in excess-3 format – having a bias of 3. Restricted float provides zero, positive/negative normalized numbers with a small range (3-bit exponent) and small precision (4-bit fraction). This numerical data type is only used by an immediate source operand of doubleword in a GEN4 instruction. It cannot be used for the destination operand, or a non-immediate source operand.

Figure 10-5 shows how to convert an 8-bit restricted float into a single precision float. Converting a 3-bit exponent with a bias of 3 to an 8-bit exponent with a bias of 127 is by adding 4, or equivalently copying bit 2 to bit 7 and putting the inverted bit 2 to bits 6:2. A special logic is also needed to take care of positive/negative zeros.

Figure 10-5. Conversion from a Restricted 8-bit Float to a Single-Precision Float

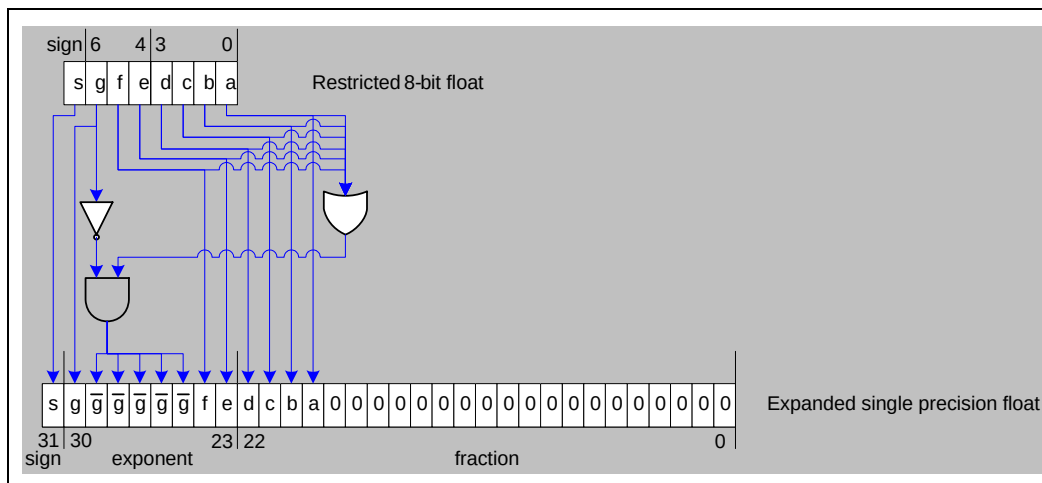


Table 10-2 shows all possible numbers of the restricted 8-bit float. Only normalized float numbers can be represented, including positive and negative zero, and positive and negative finite numbers. Normalized infinities, NaN and denormalized float numbers cannot be represented by this type. It should be noted that this 8-bit floating point format does not follow IEEE-754 convention in describing numbers with small magnitudes. Specifically, when the exponent field is zero and the fraction field is not zero, an implied one is still present instead of taking a denormalized form (without an implied one). This results in a simple implementation but with a smaller dynamic range – the magnitude of the smallest non-zero number is 0.125.

Table 10-2. Example of restricted 8-bit float numbers

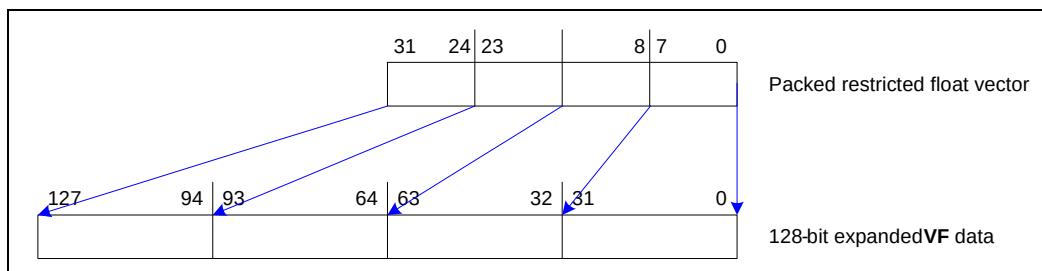
Class	Restricted 8-bit Float				Extended 8-bit Exponent	Floating number in decimal
	Hex #	Sign [7]	Exponent [6:4]	Fraction [3:0]		
Positive Normalized Float	0x70- 0x7F	0	111	0000 ... 1111	1000 0011	16 ... 31
	0x60- 0x6F	0	110	0000 ... 1111	1000 0010	8 ... 15.5
	0x50- 0x5F	0	101	0000 ... 1111	1000 0001	4 ... 7.75
	0x40- 0x4F	0	100	0000 ... 1111	1000 0000	2 ... 3.875
	0x30- 0x3F	0	011	0000 ... 1111	0111 1111	1 ... 1.9375
	0x20- 0x2F	0	010	0000 ... 1111	0111 1110	0.5 ... 0.96875
	0x10- 0x1F	0	001	0000 ... 1111	0111 1101	0.25 ... 0.484375
	0x01- 0x0F	0	000	0001 ... 1111	0111 1100	0.125 ... 0.2421875
	0x00	0	000	0000	0000	0 (+zero)
					0000	
Negative Normalized Float	0xF0- 0xFF	1	111	0000 ... 1111	1000 0011	-16 ... -31
	0xE0- 0xEF	1	110	0000 ... 1111	1000 0010	-8 ... -15.5
	0xD0- 0xDF	1	101	0000 ... 1111	1000 0001	-4 ... -7.75
	0xC0- 0xCF	1	100	0000 ... 1111	1000 0000	-2 ... -3.875
	0xB0- 0xBF	1	011	0000 ... 1111	0111 1111	-1 ... -1.9375
	0xA0- 0xAF	1	010	0000 ... 1111	0111 1110	-0.5 ... - 0.96875
	0x90- 0x9F	1	001	0000 ... 1111	0111 1101	-0.25 ... - 0.484375
	0x81- 0x8F	1	000	0001 ... 1111	0111 1100	-0.125 ... - 0.2421875
					0000	
	0x80	1	000	0000	0000	-0 (-zero)

Figure 10-6 shows the conversion of a packed exponent-only float to a 4-element vector of single precision floats.

The short hand format notation for a packed signed half-byte vector is **VF**.



Figure 10-6. Converting a Packed Restricted Float Vector to a 128-bit Float Vector



10.3 Floating Point Modes

GEN4 architecture supports two floating point operation modes, namely IEEE floating point mode (IEEE mode) and alternative floating point mode (ALT mode). Both modes follow mostly the requirements in IEEE-754 but with different deviations. The deviations will be described in details in later sections. The primary difference between these modes is on the handling of Infs, NaNs and denorms. The IEEE floating point mode may be used to support newer versions of 3D graphics API Shaders and the alternative floating point mode may be used to support early Shader versions.

These two modes are supported by all units that perform floating point computations, including GEN4 execution units, GEN4 shared functions like Extended Math, the Sampler and the Render Cache color calculator, and fixed functions like VF, Clipper, SF and WIZ. Host software sets floating point mode through the fixed function state descriptors for 3D pipeline and the interface descriptor for media pipeline. Therefore different modes may be associated with different threads running concurrently. Floating point mode control for EU and shared functions are based on the floating point mode field (bit 0) of *cr0* register.

10.3.1 IEEE Floating Point Mode

10.3.1.1 Partial Listing of Honored IEEE-754 Rules

Here is a summary of expected 32-bit floating point behaviors in GEN4 architecture. Refer to IEEE-754 for topics not mentioned.

- $\text{INF} - \text{INF} = \text{NaN}$
- $0 * (+/-)\text{INF} = \text{NaN}$
- $1 / (+\text{INF}) = +0$ and $1 / (-\text{INF}) = -0$
- $(+/-)\text{INF} / (+/-)\text{INF} = \text{NaN}$ as $A/B = A * (1/B)$
- $\text{INV}(+0) = \text{RSQ}(+0) = +\text{INF}$, $\text{INV}(-0) = \text{RSQ}(-0) = -\text{INF}$, and $\text{SQRT}(-0) = -0$
- $\text{RSQ}(-\text{finite}) = \text{SQRT}(-\text{finite}) = \text{NaN}$
- $\text{LOG}(+0) = \text{LOG}(-0) = -\text{INF}$, $\text{LOG}(-\text{finite}) = \text{LOG}(-\text{INF}) = \text{NaN}$
- $\text{NaN}(\text{any OP}) \text{ any-value} = \text{NaN}$ with one exception for min/max mentioned below. Resulting NaN may have different bit pattern than the source NaN.



- Normal comparison with conditional modifier of EQ, GT, GE, LT, LE, when either or both operands is NaN, returns FALSE. Normal comparison of NE, when either or both operands is NaN, returns TRUE.
 - Note: Normal comparison is either a **cmp** instruction or an instruction with conditional modifier
- Special comparison **cmpn** with conditional modifier of EQ, GT, GE, LT, LE, when the second source operand is NaN, returns TRUE, regardless of the first source operand, and when the second source operand is not NaN, but first one is, returns FALSE. **Cmpn** of NE, when the second source operand is NaN, returns FALSE, regardless of the first source operand, and when the second source operand is not NaN, but first one is, returns TRUE.
 - This is used to support the proposed IEEE-754R rule on **min** or **max** operations. For which, if only one operand is NaN, min and max operations return the other operand as the result.
- Both normal and special comparisons of any non-NaN value against +/- INF return exact result according to the conditional modifier. This is because that infinities are exact representation in the sense that +INF = +INF and -INF = -INF.
 - NaN is unordered in the sense that NaN != NaN.

10.3.1.2 Complete Listing of Deviations or Additional Requirements vs. IEEE-754

For a result that cannot be represented precisely by the floating point format, GEN4 execution unit uses rounding toward zero (which is a bit-field truncation of the magnitude portion of a floating point data in sign-magnitude form) to produce a result to the closest representable value. This ends up with a result that is within 1 Unit-Last-Place (1 ULP) of the infinitely precise result.

- GEN4 execution unit can report floating point overflow and NaN into conditional flags. However, there is no support for floating point exceptions, status bits or traps.
- Denorms are flushed to sign-preserved zero on input and output of any floating point mathematical operation.
 - The exception to the above point about flushing denorms is that any I/O or data movement operation that does not manipulate the data (such as point sampling float data, or executing any raw "mov" instruction, or any sort of conditional raw "mov" if present) must not alter data at all (so a denorm remains denorm). Note that doing something that amounts to just moving data, but isn't explicitly, such as multiplying a number by 1.0f is not detected as just a raw "mov", and in this case a denorm flush would happen.
- NaN input to an operation obviously always produces NaN on output, however the exact bit pattern of the NaN is not required to stay the same (unless the operation is a raw "mov" instruction which does not alter data at all.)
- $x * 1.0f$ must always result in x (except denorm flushed and possible bit pattern change for NaN).
- $x +/- 0.0f$ must always result in x (except denorm flushed and possible bit pattern change for NaN). But $-0 + 0 = +0$.
- Fused operations (such as mac, dp4, dp3, etc.) may produce intermediate results out of 32-bit float range, but whose final results would be within 32-bit float range if intermediate results were kept at greater precision. In this case,



implementations are permitted to produce either the correct result, or else +/-INF. Thus, compatibility between a fused operation, such as “mac”, with the unfused equivalent, “mul” followed by “add” in this case, is not guaranteed.

— As the accumulator registers have more precision than 32-bit float, any instruction with accumulator as a source/destination operand may produce a different result than that using GRF/MRF registers.

- API Shader divide operations are implemented as $x*(1.0f/y)$. With the two-step method, $x*(1.0f/y)$, the multiply and the divide each independently operate at the 32-bit floating point precision level (accuracy to 1 ULP).
- See the Type Conversion section below for rules on converting to/from float representations.

10.3.1.3 Comparison of Floating Point Numbers

The following tables (Table 10-3 through Table 10-8) detail the rules for floating point comparison. In the tables, “+/-Fin” stands for a positive or negative finite precision floating point number. Result is either a true (T) or false (FALSE or F). Each row corresponds to a fixed <src0> and each column corresponds to a fixed <src1>. When comparing two positive finite numbers (or two negative finite numbers), the result can be T or F depending on the values. Therefore, the corresponding fields in the following tables are marked as T/F.

Table 10-3. Results of “Greater-Than” Comparison – CMP.G

src0	src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf		FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
-Fin		T	T/F	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
-denorm		T	T	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
-0		T	T	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
+0		T	T	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
+denorm		T	T	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
+Fin		T	T	T	T	T	T	T/F	FALSE	FALSE
+inf		T	T	T	T	T	T	T	FALSE	FALSE
NaN		FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE



Table 10-4. Results of “Less-Than” Comparison – CMP.L

src0	src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf		FALSE	T	T	T	T	T	T	T	FALSE
-Fin		FALSE	T/F	T	T	T	T	T	T	FALSE
-denorm		FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T	T	FALSE
-0		FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T	T	FALSE
+0		FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T	T	FALSE
+denorm		FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T	T	FALSE
+Fin		FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T/F	T	FALSE
+inf		FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
NaN		FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

Table 10-5. Results of “Equal-To” Comparison – CMP.E

src0	src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf		T	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
-Fin		FALSE	T/F	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
-denorm		FALSE	FALSE	T	T	T	T	FALSE	FALSE	FALSE
-0		FALSE	FALSE	T	T	T	T	FALSE	FALSE	FALSE
+0		FALSE	FALSE	T	T	T	T	FALSE	FALSE	FALSE
+denorm		FALSE	FALSE	T	T	T	T	FALSE	FALSE	FALSE
+Fin		FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T/F	FALSE	FALSE
+inf		FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T	FALSE
NaN		FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

Table 10-6. Results of “Not-Equal-To” Comparison – CMP.NE

src0	src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf		FALSE	T	T	T	T	T	T	T	T
-Fin		T	T/F	T	T	T	T	T	T	T
-denorm		T	T	FALSE	FALSE	FALSE	FALSE	T	T	T
-0		T	T	FALSE	FALSE	FALSE	FALSE	T	T	T
+0		T	T	FALSE	FALSE	FALSE	FALSE	T	T	T
+denorm		T	T	FALSE	FALSE	FALSE	FALSE	T	T	T
+Fin		T	T	T	T	T	T	T/F	T	T
+inf		T	T	T	T	T	T	T	FALSE	T
NaN		T	T	T	T	T	T	T	T	T



Table 10-7. Results of “Less-Than Or Equal-To” Comparison – CMP.LE

src0	src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf		T	T	T	T	T	T	T	T	FALSE
-Fin		FALSE	T/F	T	T	T	T	T	T	FALSE
-denorm		FALSE	FALSE	T	T	T	T	T	T	FALSE
-0		FALSE	FALSE	T	T	T	T	T	T	FALSE
+0		FALSE	FALSE	T	T	T	T	T	T	FALSE
+denorm		FALSE	FALSE	T	T	T	T	T	T	FALSE
+Fin		FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T/F	T	FALSE
+inf		FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T	FALSE
NaN		FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

Table 10-8. Results of “Greater-Than or Equal-To” Comparison – CMP.GE

src0	src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf		T	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
-Fin		T	T/F	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
-denorm		T	T	T	T	T	T	FALSE	FALSE	FALSE
-0		T	T	T	T	T	T	FALSE	FALSE	FALSE
+0		T	T	T	T	T	T	FALSE	FALSE	FALSE
+denorm		T	T	T	T	T	T	FALSE	FALSE	FALSE
+Fin		T	T	T	T	T	T	T/F	FALSE	FALSE
+inf		T	T	T	T	T	T	T	T	FALSE
NaN		FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

10.3.1.4 Min / Max of Floating Point Numbers

A special comparison called Compare-NaN is introduced in GEN4 architecture to handle the difference of above mentioned floating point comparison and the rules on supporting MIN/MAX. To compute the MIN or MAX of two floating point numbers, if one of the numbers is NaN and the other one is not, MIN or MAX of the two numbers returns the one that is not NaN. When two numbers are NaN, MIN or MAX of the two numbers returns a NaN, which may not have the same binary form as any of the two numbers.



When using CMPN for MIN/MAX, the flag polarity for CMPN and SEL instructions must be the same:

Evaluations	GEN4 Instructions
$\text{MIN}(\text{src0}, \text{src1}) = (\text{src0} < \text{src1}) ? \text{src0} : \text{src1}$	cmpn.l.f0.0 null src0 src1 (f0.0) sel dst src0 src1
$\text{MAX}(\text{src0}, \text{src1}) = (\text{src0} >= \text{src1}) ? \text{src0} : \text{src1}$	cmpn.ge.f0.0 null src0 src1 (f0.0) sel dst src0 src1

The following tables (Table 10-9 through Table 10-14) detail the rules for this special compare-NaN operation for floating point numbers. Notice that excepting “Not-Equal-To” comparison-NaN, last columns in all other tables have ‘T’.

Table 10-9. Results of “Greater-Than” Comparison-NaN – CMPN.G

src0	src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf		FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T
-Fin		T	T/F	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T
-denorm		T	T	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T
-0		T	T	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T
+0		T	T	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T
+denorm		T	T	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T
+Fin		T	T	T	T	T	T	T/F	FALSE	T
+inf		T	T	T	T	T	T	T	FALSE	T
NaN		FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T



Table 10-10. Results of “Less-Than” Comparison-NaN – CMPN.L

src0	src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf		FALSE	T	T	T	T	T	T	T	T
-Fin		FALSE	T/F	T	T	T	T	T	T	T
-denorm		FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T	T	T
-0		FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T	T	T
+0		FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T	T	T
+denorm		FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T	T	T
+Fin		FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T/F	T	T
+inf		FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T
NaN		FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T

Table 10-11. Results of “Equal-To” Comparison-NaN – CMPN.E

src0	src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf		T	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T
-Fin		FALSE	T/F	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T
-denorm		FALSE	FALSE	T	T	T	T	FALSE	FALSE	T
-0		FALSE	FALSE	T	T	T	T	FALSE	FALSE	T
+0		FALSE	FALSE	T	T	T	T	FALSE	FALSE	T
+denorm		FALSE	FALSE	T	T	T	T	FALSE	FALSE	T
+Fin		FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T/F	FALSE	T
+inf		FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T	T
NaN		FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T

Table 10-12. Results of “Not-Equal-To” Comparison-NaN – CMPN.NE

src0	src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf		FALSE	T	T	T	T	T	T	T	FALSE
-Fin		T	T/F	T	T	T	T	T	T	FALSE
-denorm		T	T	FALSE	FALSE	FALSE	FALSE	T	T	FALSE
-0		T	T	FALSE	FALSE	FALSE	FALSE	T	T	FALSE
+0		T	T	FALSE	FALSE	FALSE	FALSE	T	T	FALSE
+denorm		T	T	FALSE	FALSE	FALSE	FALSE	T	T	FALSE
+Fin		T	T	T	T	T	T	T/F	T	FALSE
+inf		T	T	T	T	T	T	T	FALSE	FALSE
NaN		T	T	T	T	T	T	T	T	FALSE



Table 10-13. Results of “Less-Than Or Equal-To” Comparison-NaN – CMPN.LE

src0	src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf		T	T	T	T	T	T	T	T	T
-Fin		FALSE	T/F	T	T	T	T	T	T	T
-denorm		FALSE	FALSE	T	T	T	T	T	T	T
-0		FALSE	FALSE	T	T	T	T	T	T	T
+0		FALSE	FALSE	T	T	T	T	T	T	T
+denorm		FALSE	FALSE	T	T	T	T	T	T	T
+Fin		FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T/F	T	T
+inf		FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T	T
NaN		FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T

Table 10-14. Results of “Greater-Than or Equal-To” Comparison-NaN – CMPN.GE

src0	src1	-inf	-Fin	-denorm	-0	+0	+denorm	+Fin	+inf	NaN
-inf		T	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T
-Fin		T	T/F	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T
-denorm		T	T	T	T	T	T	FALSE	FALSE	T
-0		T	T	T	T	T	T	FALSE	FALSE	T
+0		T	T	T	T	T	T	FALSE	FALSE	T
+denorm		T	T	T	T	T	T	FALSE	FALSE	T
+Fin		T	T	T	T	T	T	T/F	FALSE	T
+inf		T	T	T	T	T	T	T	T	T
NaN		FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	T



10.3.2 Alternative Floating Point Mode

The key characteristics of the alternative floating point mode is that NaN, Inf and denorm are not expected for an application to pass into the graphics pipeline, and the graphics hardware must not generate NaN, Inf or denorm as computation result. For example, a result that is larger than the maximum representable floating point number is expected to be flushed to the largest representable floating point number, i.e., +FLT_MAX. The FLT_MAX has an exponent of 0xFE and a mantissa of all one's, which is the same for IEEE floating point mode.

Here is the complete list of the differences of legacy graphics mode from the relaxed IEEE-754 floating point mode.

- Any +/- INF result must be flushed to +/- FLT_MAX, instead of being output as +/- INF.
- Extended mathematics functions of log(), rsq() and sqrt() take the absolute value of the sources before computation to avoid generating INF and NaN results.

Table 10-15 shows the support of these differences in various hardware units.

Table 10-15. Supported Legacy Float Mode and Impacted Units

IEEE-754 Deviations	VF	Clipper	SF	WIZ	EU	EM	Sampler	RC
Any +/- INF result flushed to +/- FLT_MAX	Y	Y	Y	Y	Y	Y	Y	Y
Log, rsq, sqrt take abs() of sources	N/A	N/A	N/A	N/A	N/A	Y	N/A	N/A

Table 10-16 shows some of the desired or recommended alternative floating point mode behaviors that do not have hardware design impact. The reasons of not needing special hardware support for these items are also provided.



Table 10-16. Dismissed legacy behaviors

Suggested IEEE-754 Deviations	Reason for Dismiss
Mov forces (+/-)INF to (+/-)FLT_MAX	(+/-)INF is never present as input
(+/-)INF - (+/-)INF = +/- FLT_MAX instead of NaN	(+/-)INF is never present as input
Denorm must be flushed to zero in all cases (including trivial mov and point sampling)	Denorm is never present as input
Anything*0=0 (including NaN*0=0 and INF*0=0)	NaN and INF are never present as input
Except propagated NaN, NaN is never generated	NaN is never present as input and GEN4 never generates NaN based on rules in the previous table
An input NaN gets propagated excepting (a)-(d)	NaN is never present as input
(a) Rcp (and rsq) of 0 yields FLT_MAX	N/A, as it is already covered by the general rule "Any +/- INF result flushed to +/- FLT_MAX"
(b) Sampler honors 0/0 = 0 as if (1/0)*0	There is no divide in Sampler
I Rcp (and rsq) of INF yields +/- 0	(+/-)INF is never present as input
(d) Sampler honors INF/INF = 0 as if (1/INF)=0 followed by Anything*0 = 0	There is no divide in Sampler



10.4 Type Conversion

10.4.1 Float to Integer

Converting from float to integer is based on rounding toward zero. If the floating point value is +0, -0, +Denorm, -Denorm, +NaN or -NaN, the resulting integer value is always 0. If the floating point value is positive infinity (or negative infinity), the conversion result takes the largest (or the smallest) represent-able integer value. If the floating point value is larger (or smaller) than the largest (or the smallest) represent-able integer value, the conversion result takes the largest (or the smallest) represent-able integer value. The following table shows these special cases. The last two rows are just examples. They can be any number outside the represent-able range of the output integer type (UD, D, UW, W, UB and B).

Input Format	Output Format					
F	UD	D	UW	W	UB	B
+/- Zero	00000000	00000000	00000000	00000000	00000000	00000000
+/- Denorm	00000000	00000000	00000000	00000000	00000000	00000000
NAN	00000000	00000000	00000000	00000000	00000000	00000000
-NAN	00000000	00000000	00000000	00000000	00000000	00000000
INF	FFFFFFFF	7FFFFFFF	0000FFFF	00007FFF	000000FF	0000007F
-INF	00000000	80000000	00000000	00008000	00000000	00000080
+2 ³² (*)	FFFFFFFF	7FFFFFFF	0000FFFF	00007FFF	000000FF	0000007F
-2 ³² -1 (*)	00000000	80000000	00000000	00008000	00000000	00000080

10.4.2 Integer to Integer with Same or Higher Precision

Converting an unsigned integer to a signed or an unsigned integer with higher precision is based on zero extension.

Converting an unsigned integer to a signed integer with the same precision is based on modular wrap-around. Without saturation, a larger than represent-able number becomes a negative number. With saturation, a larger than represent-able number is saturated to the largest positive represent-able number.

Converting a signed integer to a signed integer with higher precision is based on sign extension.

Converting a signed integer to an unsigned integer with higher precision is based on zero extension. Without saturation, a negative number becomes a large positive number with the sign bit wrapped-up. With saturation, a negative number is saturated to zero.



10.4.3 Integer to Integer with Lower Precision

Converting a signed or an unsigned integer to a signed or an unsigned integer with lower precision is based on bit truncation. Without saturation, only the lower bits are kept in the output regardless of the sign-ness of input and output. With saturation, a number that is outside the represent-able range is saturated to the closest represent-able value.

10.4.4 Integer to Float

Converting a signed or an unsigned integer to a single precision float number is to round to the closest representable float number. For any integer number with magnitude less than or equal to 24 bits, resulting float number is a precise representation of the input. However, if it is more than 24 bits, LSBs are truncated. This truncation is performed in sign-magnitude domain, thus, is equivalent to floating point rounding toward zero operation.



11 *Execution Environment*

11.1 Overview

GEN4 instruction set is a general-purpose data-parallel instruction set optimized for graphics and media computations. Supports for 3D graphics API (application program interface) Shader instructions are mostly native, meaning that GEN4 provides efficient execution for Shader programs. Depending on the Shader program operation modes (for example, a Vertex Shader may be executed on a base of a vertex-pair, while a Pixel Shader may be executed on a base of a 16-pixel group), translation from 3D graphics API Shader instruction streams into GEN4 native instructions may be required. In addition, there are many specific capabilities to accelerate media applications. The following list provides a summary of the GEN4 instruction set.

- GEN4 ISA support SIMD (single instruction multiple data) instructions. The number of data elements per instruction depends on the data type.
- GEN4 ISA supports SIMD parallel arithmetic, vector arithmetic, logical, and SIMD control/branch instructions.
- GEN4 ISA supports instruction level variable-width SIMD execution.
- GEN4 ISA supports conditional SIMD execution via destination mask, predication, and execution mask.
- GEN4 ISA supports in-place format conversion and mixed data type computations.
- GEN4 ISA supports instruction compression.
- A GEN4 instruction may be executed in multiple cycles over a SIMD execution pipeline.
- Most GEN4 instructions have three operands. Some instructions have additional implied source and destination operands. Some instructions have explicit dual destinations.
- GEN4 ISA supports region-based register addressing.
- GEN4 ISA supports direct and indirect (indexed) register addressing.
- GEN4 instructions may have a scalar and vector immediate source operand.
- Higher precision accumulator registers are architecturally visible.
- Self-modifying code is not allowed (instruction streams, including instruction caches, are read-only).

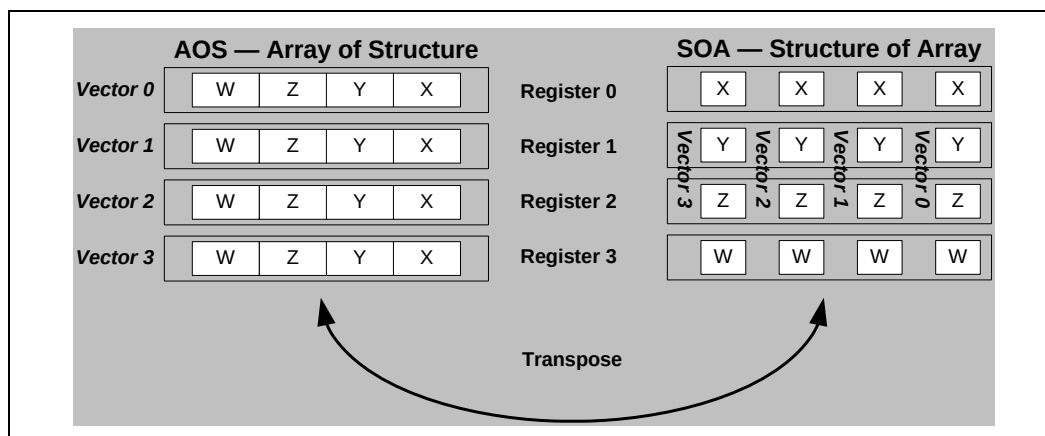
11.2 Primary Usage Models

In describing the usage models of GEN instruction set, it is inevitable to forward reference terminology, syntax and instructions detailed later in this specification. For clarity reasons, not all forward references will be provided in this section as well as subsequent sections. For example, reference to binary instruction fields such as **Align1**, **Align16**, **Compr**, **SecHalf**, etc., can be found in the Instruction Summary chapter. And assembly instruction syntax can be found in the Instruction Summary chapter and Instruction Reference chapter.

11.2.1 AOS and SOA Data Structures

With the Align1 and Align16 access modes, GEN4 instruction set provides effective SIMD computation regardless whether data are arranged in array of structure (AOS) form or in structure of array (SOA) form. The AOS and SOA data structures are illustrated by the examples in Figure 11-1. The example shows two different ways of storing four vectors in four SIMD registers. For simplicity, data vector and SIMD register both have four data elements. The four data elements in a vector are denoted by X, Y, Z and W just as for a vertex in 3D geometry. The AOS structure stores one vector in a register and the next vector in another register. The SOA structure stores one data element of each vector in a register and the next element of each vector in the next register and so on. It is obvious in this case the two structures can be related by a matrix transpose operation.

Figure 11-1. AOS and SOA data structures



GEN4 3D and media applications take advantage of such broad architecture support and use both AOS and SOA data arrangements.

- Vertices in 3D Geometry (Vertex Shader and Geometry Shader) are arranged in AOS structure and run on SIMD4x2 and SIMD4 modes, respectively, as detailed below.
- Pixels in 3D Rasterization (Pixel Shader) are arranged in SOA structure and run on SIMD8 and SIMD16 modes as detailed below.
- Pixels in media are primarily arranged in SOA structure, and occasionally in AOS structure with possible mixed mode of operations that use region-based addressing extensively.



These are preferred methods; alternative arrangements may also be possible. Shared function resources provide data transpose capability to support both modes of operations: The sampler has a transpose for sample reads, the data port has a transpose for render cache writes, and the URB unit has a transpose for URB writes.

The following 3D graphics API Shader instruction will be used in the following sections to illustrate various modes of operations:

```
add <dst>.xyz <src0>.yxzw <src1>.zwxy
```

This example is an SIMD instruction that takes two source operands <src0> and <src1>, performs addition operation (add), and store the additions to the destination operand <dst>. Each operand contains four floating point data elements. The data type is determined by the instruction opcode. This instruction also uses source swizzle modifier (.yxzw for <src0> and .zwxy for <src1> and destination mask modifier (.xyz). Please refer to programming specifications of 3D graphics API Shader instructions for more details.

A physical GRF register has 256 bits, which may be used to store 8 floating point data elements. For 3D graphics usage, the mode of operation is (loosely) termed after the data structure as SIMD $m \times n$, where “ m ” is a numerical term describing the size of vector and “ n ” is the number of concurrent program flows executed in SIMD.

- Execution with AOS data structures
 - **SIMD4 (short for SIMD4x1)** stands for the mode of operation where a SIMD instruction operates on 4-element vectors stored packed in the registers. There is only one program flow.
 - **SIMD4x2** standards for the SIMD operation based on a pair of 4-element vectors stored in a register. There are effectively two programs running side by side with one vector per program.
- Execution with SOA data structures – also referred to as “channel serial” execution
 - **SIMD8 (short for SIMD1x8)** standards for the SIMD operation based on the SOA data structure where one register contains one data element (the same one) of 8 vectors. Effectively, there are 8 concurrent program flows.
 - **SIMD16 (short for SIMD1x16)** is a special term indicating the use of instruction compression whereas each compressed SIMD instruction operates on a pair of registers that contains one data element (the same one) of 16 vectors. SIMD16 has 16 concurrent program flows.

11.2.2 SIMD4 Mode of Operation

With a register mapping of <src0> to doublewords 0-3 of $r2$, <src1> to doublewords 4-7 of $r2$ and <dst> to doublewords 0-3 of $r3$, the example 3D graphics API Shader instruction can be translated into the following GEN4 instruction:

```
add (4) r3<4>.xyz:f r2<4>.yzwx:f r2.4<4>.zwxy:f {NoMask}
```

Without diving too much into the syntax definition of a GEN4 instruction, it is clear that a GEN4 instruction also takes two source operands and one destination operands. The second term, (4), is the execution size that determines the number of data elements processed by the SIMD instruction. It is similar to the term SIMD Width used in the literature. Each operand is described by the register region parameters such as ‘<4>’ and data type (e.g. “:f”). These will be detailed in Section 11.3. The instruction option field, {NoMask}, ensure that the execution occurs for the execution channels shown in the instruction, instead of, possibly, being masked out by the conditional

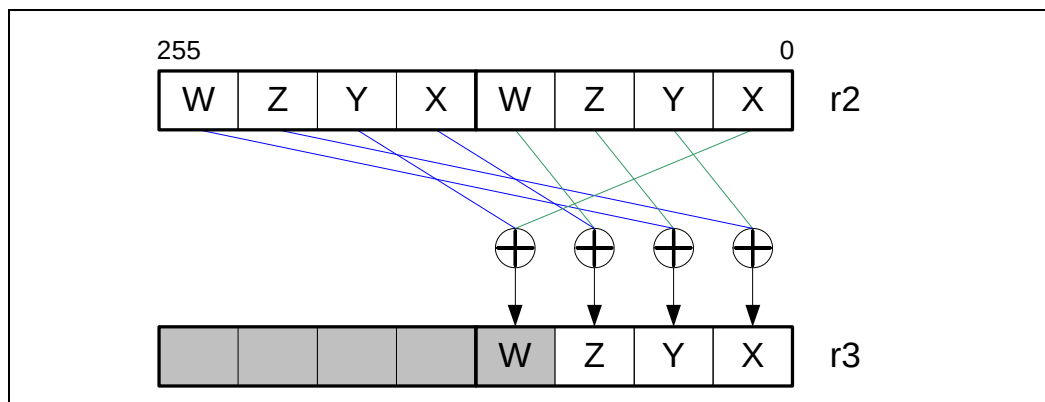
masks of the thread (See Instruction Summary chapter for definition of *MaskCtrl* instruction field).

The operation of this GEN4 instruction is illustrated in Figure 11-2. In this example, both source operands share the same physical GRF register r2. The two are distinguished by the subregister number. The source swizzles control the routing of source data elements to the parallel adders corresponding to the destination data elements. The shaded areas in the destination register r3 are not modified. In particular, doublewords 4-7 are unchanged as the execution size is 4; doubleword 3 is unchanged due to the destination mask setting.

In this mode of operation, there is only one program flow – any branch decision will be based on a scalar condition and apply to the whole vector of four elements. Option *{NoMask}* ensures that the instruction is not subject to the masks. In fact, most of the instructions in a thread should have *{NoMask}* set.

Even though the execution only performs four parallel add operations, the GEN4 instruction still executes in 2 cycles (with no useful computation in the second cycle).

Figure 11-2. A SIMD4 Example



11.2.3 SIMD4x2 Mode of Operation

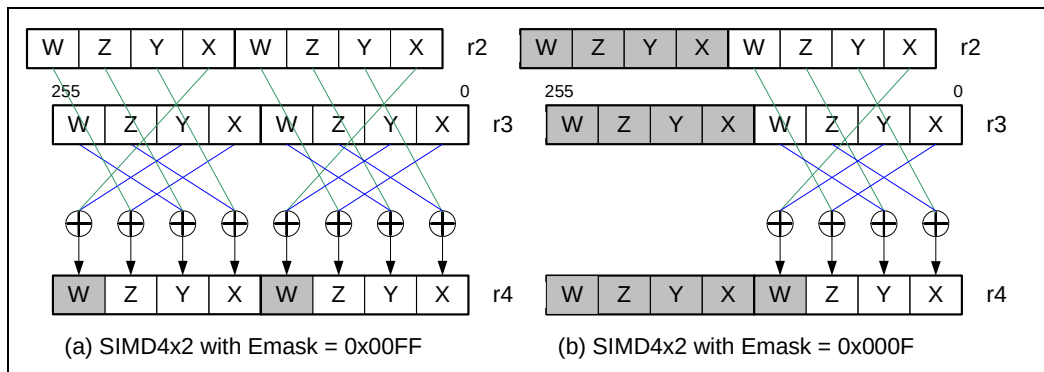
In this mode, two corresponding vectors from the two program flows fill a GEN4 physical register. With a register mapping of *<src0>* to r2, *<src1>* to r3 and *<dst>* to r4, the example 3D graphics API Shader instruction can be translated into the following GEN4 instruction:

```
add (8) r4<4>.xyz:f r2<4>.yxzw:f r3<4>.zwxy:f
```

This instruction is subject to the execution mask, which initiated from the dispatch mask. If both program flows are available (e.g. Vertex Shader executed with two active vertices), the dispatch mask is set to 0x00FF. The operation of this GEN4 instruction is illustrated in Figure 11-3 (a). The source swizzles control the routing of source data elements to the parallel adders corresponding to the destination data elements. The shaded areas in the destination register r3 (doublewords 3 and 7) are unchanged due to the destination mask setting. If only one program flow is available (e.g. the same SIMD4x2 Vertex Shader with only one active vertex), the dispatch mask is set to 0x000F. The operation of the same instruction is shown in Figure 11-3 (b).



Figure 11-3. SIMD4x2 Examples with Different Emasks

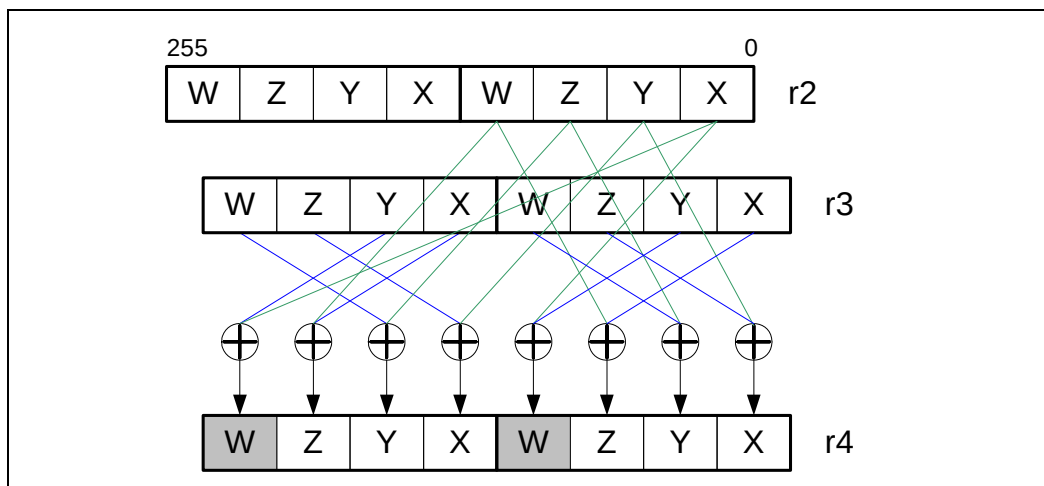


The two source operands only need to be 16-byte aligned, not have to be GRF register aligned. For example, the first source operand could be a 4-element vector (e.g. a constant) stored in doublewords 0-3 in r2, which is shared by the two program flows. The example 3D graphics API Shader instruction can then be translated into the following GEN4 instruction:

```
add(8) r4<4>.xyz:f r2<0>.yzwx:f r3<4>.zwxxy:f
```

The only difference here is that the vertical stride of the first source is 0. The operation of this GEN4 instruction is illustrated in Figure 11-4.

Figure 11-4. A SIMD4x2 Example with a Constant Vector Shared by Two Program Flows



11.2.4 SIMD16 Mode of Operation

With 16 concurrent program flows, one element of a vector would take two GRF registers. In this mode, two corresponding vectors from the two program flows fill a GEN4 physical register.

With the following register mappings,

```
<src0>:      r2-r9 (with 16 X data elements in r2-r3, Y in r4-5, Z in r6-7 and W in r8-9),
<src1>:      r10-r17,
<dst>:       r18-r25,
```

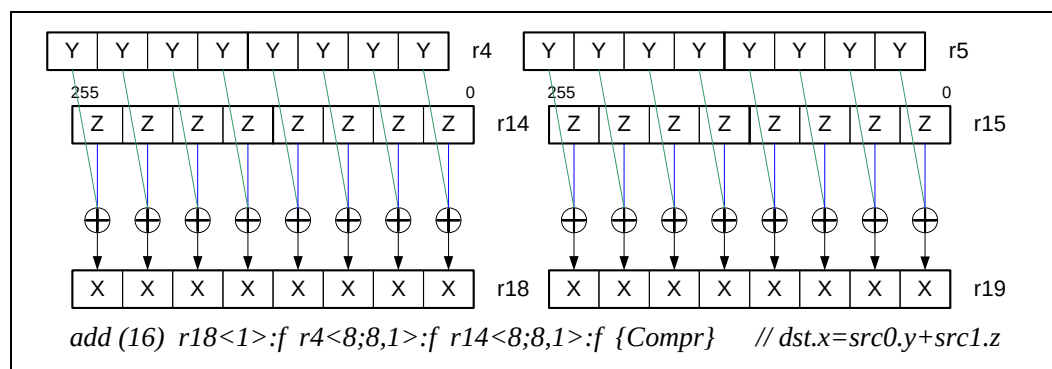
the example 3D graphics API Shader instruction can be translated into the following three GEN4 instructions:

```
add (16) r18<1>:f r4<8;8,1>:f r14<8;8,1>:f {Compr}      // dst.x = src0.y + src1.z
add (16) r20<1>:f r6<8;8,1>:f r16<8;8,1>:f {Compr}      // dst.y = src0.z + src1.w
add (16) r22<1>:f r8<8;8,1>:f r10<8;8,1>:f {Compr}      // dst.z = src0.w + src1.x
```

The three GEN4 instructions correspond to the three enabled destination masks. All instructions are compressed instructions with instruction option of {Compr} (See Instruction Summary chapter for definition of **ComprCtrl** field in GEN4 instruction word). All operands are even-aligned GRF registers. As there is no output for the W elements of <dst>, no instruction is needed for that element. The first instruction inputs the Y elements of <src0> and the Z elements of <src1> and outputs the X elements of <dst>. The operation of this instruction is shown in Figure 11-5.

With the number of program flows more than one, the above instructions also subject to execution mask. The 16-bit dispatch mask is partitioned into four groups with four bits each. For Pixel Shader generated by the Windower, each 4-bit group corresponds to a 2x2 pixel subspan. If a subspan is not valid for a Pixel Shader instance, the corresponding 4-bit group in the dispatch mask is not set. Therefore, the same instructions can be used independent of the number of available subspans without creating bogus data in the subspans that are not valid.

Figure 11-5. A SIMD16 Example



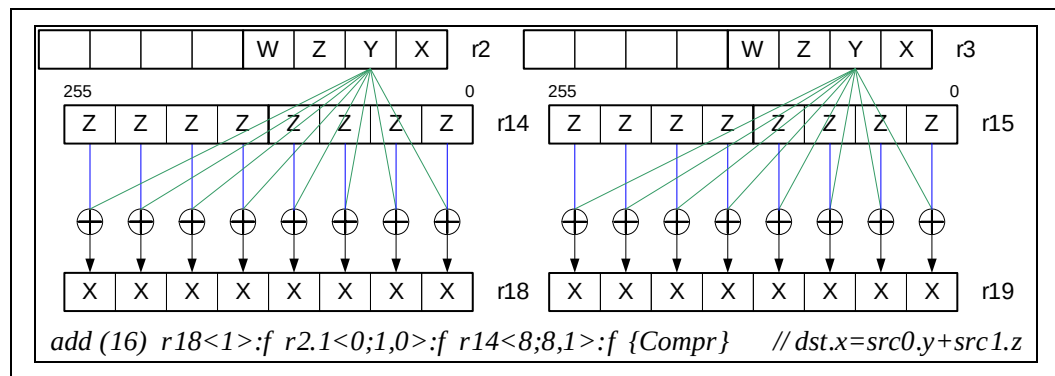


Similar to SIMD4x2 mode, a constant may also be shared for the 16 program flows. For example, the first source operand could be a 4-element vector (e.g. a constant) stored in doublewords 0-3 in r2 (AOS format). The example 3D graphics API Shader instruction can then be translated into the following GEN4 instruction:

```
add (16) r18<1>:f r2.1<0;1,0>:f r14<8;8,1>:f {Compr} // dst.x = src0.y + src1.z
add (16) r20<1>:f r2.2<0;1,0>:f r16<8;8,1>:f {Compr} // dst.y = src0.z + src1.w
add (16) r22<1>:f r2.3<0;1,0>:f r10<8;8,1>:f {Compr} // dst.z = src0.w + src1.x
```

The register region of the first source operand represents a replicated scalar. The operation of the first GEN4 instruction is illustrated in Figure 11-6.

Figure 11-6. Another SIMD16 Example with an AOS Shared Constant



11.2.5 SIMD8 Mode of Operation

Each compressed instruction has two correspond uncompressed instructions. Taking the example instruction shown in Figure 11-6, it is equivalent to the following two instructions.

```
add (8) r18<1>:f r4<8;8,1>:f r14<8;8,1>:f // dst.x[7:0] = src0.y + src1.z
add (8) r19<1>:f r5<8;8,1>:f r15<8;8,1>:f {SecHalf} // dst.x[15:8] = src0.y + src1.z
```

Therefore, SIMD8 can be viewed as a special case for SIMD16.

There are other reasons that SIMD8 instructions may be used. Within a program with 16 concurrent program flows, some time SIMD8 instruction must be used due to architecture restrictions. For example, the address register a0 only have 8 elements, if an indirect GRF addressing is used, SIMD16 instructions are not allowed.



11.3 Registers and Register Regions

11.3.1 Register Files

GEN4 registers are grouped into different name spaces called register files. There are three different register files defined: General Register File, Message Register File, and Architecture Register File. In addition, immediate operands also have a unique encoding of the register file field, even though they come inline in the instruction word and do not have dedicated physical storages.

- General Register File (GRF): GRF contains general-purpose read-write registers.
- Message Register File (MRF): MRF contains special purpose registers used for message passing only. MRF registers are write-only.
- Architecture Register File (ARF): ARF contains all other architectural registers, including the address registers (*a#*), accumulators (*acc#*), flags (*f#*), masks (*mask#*), mask stack (*ms#*), mask stack depth (*msd#*), notification count (*n#*), instruction pointer (*ip*), etc. Null register (*null*) is also encoded as an ARF register.
- Immediate: Certain instructions take immediate terms as the source operands. These immediate terms have a distinct register file encoding.

Each thread executed in an EU has its own thread context, i.e. dedicated register space, which is not shared between threads executing on a common EU or on a different EU. In the rest of the Chapters, register access are in respect to a given thread.

11.3.2 GRF Registers

Number of Registers:	Various
Default Value:	None
Normal Access:	RW
Elements:	Various
Element Size:	Various
Element Type:	Various
Access Granularity:	Byte
Write Mask Granularity:	Byte
Index-ability:	Yes

Registers in the General Register File are the most commonly used read-write registers. During the execution of a thread, GRF registers are used to store the temporary data, and serve as the destination to receive data from shared function units (and some times from a fixed function unit). They are also used to store the input (initialization) data when a thread is created. By allowing fixed function hardware to initialize some portion of GRF registers during thread dispatch time, GEN4 architecture can achieve better parallelism. A thread's execution efficiency can also be improved as some data are already in the register to be executed upon. Besides these registers containing thread's payload, the rest of GRF registers of a thread are not initialized.



Table 11-1. Summary of GRF Registers

Register File	Register Name	Description
General Register File (GRF)	<i>r#</i>	General purpose read write registers

Each execution unit has a fixed size physical GRF register RAM. The GRF register RAM is shared by all threads on the EU. GRF space for a thread is allocated at thread dispatch time, allowing the amount of GRF space to adapt to the need of a given thread.

Mapping of a thread's GRF registers to the physical GRF RAM is through a translation table. Therefore, a thread's access to GRF is always through the 0-based logical view. For example, the GRF registers of a thread with 64 GRF register allocation are *r0* through *r63*.

GRF registers can be accessed using region-based addressing at byte granularity (both read and write). A source operand must be contained within two adjacent physical registers. A destination operand must be contained within one physical register. GRF registers support direct addressing and register-indirect addressing. Register-indirect addressing uses the address registers (ARF registers *a#*) and an immediate address offset value.

When accessing (read and/or write) outside the GRF register range allocated for a given thread either through direct or indirect addressing, the result is unpredictable.

Table 11-2. GRF Registers Available in Device Hardware

Device	Physical Register Size	Allocation Granularity	Number per Thread	Number per EU
[DevBW]	256 bits	16 registers	128 registers	256 registers
[DevCL]	256 bits	16 registers	128 registers	256 registers

11.3.3 MRF Registers

Number of Registers:	Fixed
Default Value:	None
Normal Access:	WO
Elements:	Various
Element Size:	Various
Element Type:	Various
Access Granularity:	Byte
Write Mask Granularity:	Byte
Index-ability:	See Table 11-4

Registers in the Message Register File are used to store the header and payload for out-going messages from a thread to a shared function such as the Sampler and Extended Math unit. There are fixed number of MRF registers for each thread.

MRF registers are write-only, and therefore, can only be the destination operand of an instruction.



MRF registers support write-enable at byte granularity. When an MRF register is used as the current destination operand of the *send* instruction, only 256-bit register aligned access is supported.

When accessing (write) outside the MRF register range for a given thread, the result is unpredictable.

Table 11-3. Summary of MRF Registers

Register File	Register Name	Description
Message Register File (MRF)	<i>m</i> #	Special purpose output write-only registers

Table 11-4. MRF Registers Available in Device Hardware

Device	Physical Register Size	Number per Thread	Indirect Addressing?
[DevBW]	256 bits	16 registers	No
[DevCL]	256 bits	16 registers	No

Note for Programmers: As a software usage policy, m0 register is reserved for debug. Normal thread should access MRF starting at m1.



11.3.4 ARF Registers

11.3.4.1 Overview

Besides GRF and MRF registers that are directly indicated by unique register file coding, all other registers belong to the general Architecture Register File (ARF). Encoding of architecture register types are based on the MSBs of the register number field, RegNum, in the instruction word. RegNum field has 8 bits. The 4 MSBs, RegNum[7:4], represent the architecture register type. This is summarized in Table 11-5.

Table 11-5. Summary of Architecture Registers

Register Type (RegNum [7:4])	Register Name	Register Count	Description
0000	<i>null</i>	1	Null register
0001	<i>a0.#</i>	1	Address register
0010	<i>acc#</i>	2	Accumulator register
0100	<i>mask0.#</i>	1	Mask register (active, branch, loop). Note that dispatch mask is RO and in sr#
0101	<i>ms0.#</i>	1	Mask stack register
0110	<i>msd0.#</i>	1	Mask stack depth register
0111	<i>sr0.#</i>	1	State register
1000	<i>cr0.#</i>	1	Control register
1001	<i>n#</i>	1	Notification count register
1010	<i>ip</i>	1	Instruction pointer register
1011-1111	<i>reserved</i>		

The remaining register number field RegNum[3:0] is used to identify the register number of a given architecture register type. Therefore, maximum number of registers for a given architecture register type is limited to 16. The subregister number field, SubRegNum, in instruction word has 5 bits. It is used for addressing subregister region for an architecture register supporting register subdivision. SubRegNum field is in unit of byte. Therefore, maximum number of bytes of an architecture register is limited to 32. Depending on alignment restriction of a register type, only certain encodings of SubRegNum field is applicable for an architecture register. The detailed definitions are provided in the following sections.



11.3.4.2 Access Granularity

ARF registers may be accessed with subregister granularity according to the descriptions below and following the same rule of region-based addressing for GRF and MRF. The machine code for register number and subregister number of ARF follows the same rule as for other register files with byte granularity. For an ARF as a source operand, the region-based address controls the source swizzle mux. The destination subregister number and destination horizontal stride can be used to control to generate the destination write mask at byte level.

A special restriction on region-based addressing for ARF is that the register region cannot cross register boundary. This rule in fact only applies to the accumulator as it is the only ARF register containing multiple registers (two).

Subregister fields of an ARF register may not all populated (indicated by the subregister indicated as reserved). Write to an unpopulated subregister will be dropped, there is no side effect. Read from an unpopulated subregister, if not specified, will return unpredictable data.

Some of ARF registers are read-only. Write to a read-only ARF register is dropped and there is no side effect.

11.3.4.3 Null Register

ARF Register Type Encoding (RegNum[7:4]):	0000b
Number of Registers:	1
Default Value:	N/A
Normal Access:	N/A
Elements:	N/A
Element Size:	N/A
Element Type:	N/A
Access Granularity:	N/A
Write Mask Granularity:	N/A
SecHalf Control:	N/A
Index-ability:	No

The null register is a special encoding for an operand that does not have physical map. It is primarily used in the instruction to indicate the non-existence of an operand. Write to the null register has no side effect. Read from the null register returns undefined result.

The null register can be used in the place when a source operand is absent. For example, for a single source operand instruction such as MOV, NOT, the second source operand <src1> must be a null register.

When the null register is used as the destination operand of an instruction, it indicates the computed results are not stored in any physical registers. However, implied writes to the accumulator register, if applicable, may still occur for the instruction. When the conditional modifier is present, update to the selected flag register also happens. In this case, the register region fields of the 'null' operand are valid.

Another example use is to use the null register as the posted destination of a *send* instruction for data write to indicate that there is no write completion acknowledgement required. In this case, however, the register region fields are still



valid. The null register can also be the first source operand for a *send* instruction indicating the absent of the implied move. See *send* instruction for details.

11.3.4.4 Address Register

ARF Register Type Encoding (RegNum[7:4]): 0001b
 Number of Registers: 1
 Default Value: None
 Normal Access: RW
 Elements: 8
 Element Size: 16 bits
 Element Type: UW or UD
 Access Granularity: Word
 Write Mask Granularity: Word
 SecHalf Control: N/A
 Index-ability: No

There are eight address subregisters forming an 8-element vector. Each address subregister contains 16 bits. Address subregisters can be used as regular source and destination operands, as the indexing addresses for register-indirect-addressed access of GRF registers, and also as the source of the message descriptor for the *send* instruction.

Table 11-6. Register and Subregister Numbers for Address Register

RegNum[3:0]	SubRegNum[4:0]
0000 = a0 All other encodings are reserved.	When register a0 or subregisters in a0 is used as the address register for register-indirect addressing, the address subregisters must be accessed as unsigned word integers. Therefore, the subregister number field must also be word-aligned. 00000 = a0.0:uw 00010 = a0.1:uw 00100 = a0.2:uw 00110 = a0.3:uw 01000 = a0.4:uw 01010 = a0.5:uw 01100 = a0.6:uw 01110 = a0.7:uw All other encodings are reserved. However, when register a0 or subregisters in a0 is an explicit source and/or destination register, other data types are allowed as long as the register region falls in the 128-bit range.



Table 11-7. Address Register Fields

Dword	Bits	Subfield Description
3	31:16	Address subregister a0.7:uw. This field, with only the lower 12 bits populated, can be used as an unsigned integer for register-indirect register addressing. Format: U12
	15:0	Address subregister a0.6:uw. This field, with only the lower 12 bits populated, can be used as an unsigned integer for register-indirect register addressing. Format: U12
2	31:16	Address subregister a0.5:uw. This field, with only the lower 12 bits populated, can be used as an unsigned integer for register-indirect register addressing. Format: U12
	15:0	Address subregister a0.4:uw. This field, with only the lower 12 bits populated, can be used as an unsigned integer for register-indirect register addressing. Format: U12
1	31:16	Address subregister a0.3:uw. This field, with only the lower 12 bits populated, can be used as an unsigned integer for register-indirect register addressing. Format: U12
	15:0	Address subregister a0.2:uw. This field, with only the lower 12 bits populated, can be used as an unsigned integer for register-indirect register addressing. Format: U12
0	31:16	Address subregister a0.1:uw. This field can be used for register-indirect register addressing or serve as message descriptor for <i>send</i> instruction. When used for register-indirect register addressing, it is a 12-bit unsigned integer. For <i>send</i> instruction, it provides the higher 16 bits of <desc>. Format: U12 or U16.
	15:0	Address subregister a0.0:uw. This field can be used for register-indirect register addressing or serve as message descriptor for <i>send</i> instruction. When used for register-indirect register addressing, it is a 12-bit unsigned integer. For <i>send</i> instruction, it provides the lower 16 bits of <desc>. Format: U12 or U16.

When used as a source or destination operand, the address subregisters can be accessed individually or as a group. In the following example, the first instruction moves all 8 address subregisters to the first half of GRF register r1, the second instruction replicates a0.4:uw as an unsigned word to the second half of r1, the third instruction moves the first 4 words in r1 into the first 4 address subregisters, and the fourth instruction replicates r1.4 as an unsigned word to the last 4 address subregisters.

```

mov (8) r1.0<1>:uw a0.0<8;8,1>:uw // r1.n = a0.n for n = 0 to 7 in words
mov (8) r1.8<1>:uw a0.4<0;1,0>:uw // r1.m = a0.4 for m = 8 to 15 in words
mov (4) a0.0<1>:uw r1.0<4;4,1>:uw // a0.n = r1.n for n = 0 to 3 in words
mov (4) a0.4<1>:uw r1.4<0;1,0>:uw // a0.m = r1.4 for m = 4 to 7 in words

```



When used as the register-indirect addressing for GRF registers, the address subregisters can be accessed also individually or in group. When accessed in group, the address subregisters must be group-aligned. For example, when two address subregisters are used for register indirect addressing, they must be aligned to even address subregisters. In the following example, the first instruction is legal. However, the second one is not. As ExecSize = 8 and the width of <src0> is 4, two address subregisters will be used as row indices, each pointing to 4 data elements spaced by HorzStride = 1 dword. For the first instruction, the two address subregisters are a0.2:uw and a0.3:uw. The two align to a dword group in the address register. However, the two address subregisters for the second instruction are a0.3:uw and a0.4:uw. They are not dword aligned in the address register and therefore violate the above mentioned alignment rule.

```
mov (8) r1.0<1>:d r[a0.2]<4,1>:d    // a0.2 and a0.3 is used for src1
mov (8) r1.0<1>:d r[a0.3]<4,1>:d    // ILLEGAL use of register indirect
```

Implementation restriction: GEN4 ISA supports per channel indexing for a source operand. As there are only 8 sub-fields in the address register (to save hardware cost), the execution size of an instruction using per-channel indexing is limited to 8. Software may reload the address register and use compression control **SecHalf** to complete a 16-channel computation.

Implementation restriction: When used as the source operand <desc> for the send instruction, only the first dword subregister of a0 register is allowed (i.e. a0.0:ud, which can be viewed as the combination of a0.0:uw and a0.1:uw). In addition, it must be of UD type and in the following form <desc> = a0.0<0;1,0>:ud.

Implementation restriction: Elements a0.0 and a0.1 have 16 bits each, but the rest of elements (a0.2:uw through a0.7:uw) only have 12 bits populated each. 12-bit precision supports full indirect-addressing capability for the largest GRF register range. Software must observe the asymmetry of the implementation. When a0.0:uw and a0.1:uw are the source or destination, full 16-bit precision is preserved. However, when a0.2:uw to a0.7:uw are the destination, the higher 4 bits for each element will be dropped; when they are the source, hardware inserts zero to the higher 4 bits for each element.

Performance Note: There is only one scoreboard for the whole address register. When a write to some subregisters is in flight, hardware will stall any instruction writing to other subregisters. Software may use the destination dependency control {NoDDChk, NoDDClr} to improve performance in this case. Similarly, when a write to some subregisters is in flight, hardware will stall any instruction sourcing other subregisters until the write retires.



11.3.4.5 Accumulator Registers

ARF Register Type Encoding (RegNum[7:4]):	0010b
Number of Registers:	2
Default Value:	None
Normal Access:	RW
Elements:	8 or 16
Element Size:	Various
Element Type:	Various
Access Granularity:	Word
Write Mask Granularity:	N/A
SecHalf Control:	Yes
Index-ability:	No

There are two accumulator registers, *acc0* and *acc1*. They can be accessed either as explicit source and/or destination registers or as implied source and/or destination registers. To a programmer, each accumulator register may contain either 8 doublewords or 16 words of data elements. However, as shown in

Table 11-9, each data element may have higher precision with additional guard bits than that indicated by the numerical data type.

Table 11-8. Register and Subregister Numbers for Accumulate Register

RegNum[3:0]	SubRegNum[4:0]
0000 = acc0 0001 = acc1 All other encodings are reserved.	Reserved: MBZ The accumulator subfields are individually addressable at word granularity. When an accumulator register is an explicit destination, it follows the rules for a destination register. If an accumulator is an explicit source operand, its register region must match with that of the destination register.

The accumulators are implied destination for arithmetic instructions, including parallel and vector instructions. For all other instructions, if accumulator is not specified as the destination operand, the content in the accumulator registers are unaltered. Details can be found in Instruction Reference chapter. There is a control field called Accumulator Disable in control register *cr0.0* allowing software to turn on (which is the default) and off the implicit update of accumulators.

When an accumulator register is used as an implicit source or destination operand, it is *acc0* by default. For a compressed instruction, both *acc0* and *acc1* are used. If ComprCtrl is set to **SecHalf**, the implicit accumulator is then *acc1*. When an accumulator register is used as an explicit source or destination operand, the **SecHalf** compression control is ignored. In other words, the implied accumulator (source or destination), if present, is the same as the explicit one.

It is illegal to specify different accumulator registers for source and destination operands in an instruction (e.g. "*add (8) acc1:f acc0:f*"). Result of such instruction is unpredictable.



For a compressed instruction, if an accumulator register is used as an explicit source or destination operand, it must be `acc0`.

When an accumulator register is used as an explicit source operand, it must be the first source operand `<src0>`. Meanwhile, source operand modifiers (absolute, negate) are not allowed, as there are ignored by hardware.

When an accumulator register is explicitly or implicitly specified as the destination, destination channel enables do not apply to the accumulator register. In other words, accumulator registers cannot be masked out and the content of the 'disabled' channels of the accumulator register is unpredictable.

When an accumulator register is used as an explicit destination operand, saturation (`.sat`) is not allowed.

Whether the accumulator register is updated for a given instruction depends on several conditions: it can be an explicit destination operand, it can be an implicit destination for arithmetic and logic instructions and the implicit update is also subject to the control register bit mentioned above. For an instruction in the form like, `opcode <dst> <src0> [<src1>]`, the accumulator register is updated if the any of the following conditions is true

- `<dst>` is an accumulator register
- `cr0.0[1]` is cleared and `opcode` indicates that the instruction implicitly update accumulator register

Bit field `cr0.0[1]` is the Accumulator Disable that controls the implied update.

Implementation Restriction due to Floating Point Precision: When a floating point value is stored in the accumulator, it is stored in a non-normalized form with extra precision in mantissa. For an instruction involving addition operation sourcing accumulators, the addition is performed in non-normalized space. Therefore, the results may vary depending on the order of the operations. This is commonly referred to as 'fused' operations. For example, instructions like `mac` and `dp#` are fused operation. A group of back-to-back add instructions sourcing accumulators is also a fused operation. Though accumulator may be used as a temporary register with reduced pipeline compute latency, caution must be taken when using accumulator for floating point computation. In general, floating point computation explicitly and/or implicitly involving accumulator should be only used for fused operations where result deviation due to operation order is acceptable. Otherwise, accumulator should not be used as a temporary register.

Errata: When `acc1` is used as explicit operands of two back-to-back instructions, the results may be nondeterministic. This can be worked around using `acc1` together with **SecHalf** compression control. This provides almost equivalent behavior except that the second half of the flag register may be used (including when `ExecSize = 16`). If for certain reason that first half of the flag (or the whole 16-bit of the flag) needs to be used together with `acc1`, an alternative workaround is to use 'switch' instruction control on the first instruction.

Performance Note: GEN4 hardware cannot support write followed by read on the same accumulator register back to back. A thread stall (equivalent of having a 'switch' instruction control) may occur before an instruction that uses an accumulator register as an (implicit or explicit) source and the previous instruction has the same accumulator register as the (implicit or explicit) destination. This commonly occurs in signal processing algorithms. For example, a multi-tap FIR filter can be implemented



by a sequence of **mac** instructions; a matrix computation in DCT transform also consists of a sequence of **mac** instructions. A program with a non-compressed instruction stream may choose to interleave the use of **acc0** and **acc1** to achieve better performance, if accumulators are used explicitly. This is not true if the accumulator is addressed implicitly based on the **SecHalf** compression control field as flags and masks are also affected by **SecHalf**.

Implementation Precision Restriction: As there are only 64 bits per channel in dword mode (D and UD), it is sufficient to store multiplication result of two dword operands as long as the post source modified sources are still within 32 bits. If any one source is type UD and is negated, the negated result becomes 33 bits. The dword multiplication results will be 65 bits, bigger than the storage capacity of accumulators. Consequently, the results are unpredictable.

Implementation Precision Restriction: As there are only 33 bits per channel in word mode (W and UW), it is sufficient to store multiplication result of two word operands with and without source modifier as the result is up to 33 bits. Integer is stored in accumulator in 2's complement form with bit 32 as the sign bit. As there is no guard bit left, the accumulator can only be sourced once before running into risk of overflowing. When overflow occurs, only modular addition can generate correct result. But in this case, conditional flags may be incorrect. When saturation is used, the output is unpredictable. This is also true for other operations that may result in more than 33 bits of data. For example, adding UDW (FFFFFFFF) with DW (00000001) results in (1FFFFFFFFE). The sign bit is now at bit 34 and is lost when stored in the accumulator. When it is read out later from the accumulator, it becomes a negative number as bit 32 now becomes the sign bit.

Table 11-9. Accumulator Channel Precision

Data Type	# Channel	Bits / Channel	Description
F	8	54+8	When the internal execution data type is float, each accumulator register contains 8 channels of (extended) single precision floating point numbers. The data is in non-normalized format with an 8-bit exponent and a 54-bit mantissa in 2's complement form. The 54-bit mantissa provides 5 extra guard bits over the precision required to store the multiplication result of two 32-bit single precision floats.
D (UD)	8	64	When the internal execution data type is doubleword integer, each accumulator register contains 8 channels of (extended) doubleword integer values. The data are always stored in accumulator in 2's complement form with 64 bits total regardless of the source data type. This is sufficient to construct the 64-bit D or UD multiplication results using an instruction macro sequence consisting mul, mach and shr (or mov). [Open: may mention negating a UD may result in unpredictable numbers.]
W (UW)	16	33	When the internal execution data type is doubleword integer, each accumulator register contains 16 channels of (extended) word integer values. The data are always stored in accumulator in 2's complement form with 33 bits total. This supports single instruction multiplication of two word source in W and/or UW format.
B (UB)	N/A	N/A	Not supported data type.



Implementation Restriction about Denorm: In general, for a floating point arithmetic instruction, hardware converts a denormalized number to sign-preserved zero before performing the computation. However, there is no such a conversion for an accumulator source operand. When accumulator is used as a temporary register for floating point computation, it is software's responsibility to ensure that such conversion is performed when storing floating point data in the accumulator. This is illustrated in the example for the following three-source non-Gen4 'mad' instruction:

```
mad dst src0 src1 src2 // dst = src0 + src1 * src2
```

One might intuitively consider translating this into two Gen4 instructions: a mov to accumulator followed by a mac:

```
mov acc0.0:f src0:f // acc0.0 = src0 (this is a 'raw' move)
```

```
mac dst src1 src2 // dst = acc0.0 + src1 * src2
```

This may generate incorrect floating results as the first move instruction doesn't flush denorms in src0 into zeros in acc0.0. Consequently, these denorm numbers in accumulator may create undefined results for the mac instruction. A correct translation should use an instruction that forces the denorm-to-zero flush, such as a multiplication of 1 or an addition of negative zero as show below. Yes, it must be a negative zero to preserve the sign of the source.

```
add acc0.0:f src0:f -0.0:f // acc0.0 = src0 (with denorm-to-zero flush)
```

```
mac dst src1 src2 // dst = acc0.0 + src1 * src2
```

11.3.4.6 Flag Register

ARF Register Type Encoding (RegNum[7:4]): 0011b

Number of Registers:	1
Default Value:	None
Normal Access:	RW
Elements:	2
Element Size:	16 bits
Element Type:	UW
Access Granularity:	Word
Write Mask Granularity:	Word
SecHalf Control:	Yes
Index-ability:	No

There is one flag register that consists of two 16-bit subregisters. Each flag subregister can be individually addressed. Each bit of a flag subregister corresponds to a data channel. (See Table 11-10 for details). Furthermore, each 16-bit subregister may be split to half when **ComprCtrl** field is set to **SecHalf** in the instruction.

The two flag subregisters (*f0.0:uw* and *f0.1:uw*) can be used as the destination of the conditional modifier and can also be the source of the predication. As both predication flag source and conditional flag destination share the same instruction field, when both are enabled, they use the same flag subregister.

The values held in the individual bits of a flag subregister are the result of the most recent instruction which performed a condition-code evaluation with that flag register



specified as the destination of the evaluation result. For example “*add.nz.f0.0 ...*” updates flag *f0.0* with the per-channel results of the not-zero condition. The flag subregister has per-bit write enables. When being updated as the secondary destination associated with conditional modifier, only the bits corresponding to the enabled channels in *EMask* are updated. The other bits in the flag subregister are unchanged.

The flag register as a whole or as two subregisters can also be an explicit source and/or destination operand.

Table 11-10. Register and Subregister Numbers for Flag Register

RegNum[3:0]	SubRegNum[4:0]
0000 = f0 All other encodings are reserved.	00000 = f0.0:uw 00010 = f0.1:uw All other encodings are reserved.

Table 11-11. Flag Register Fields

Dword	Bits	Subfield Description
0	31:16	Flag subregister f0.1:uw. This field contains 16 bits of conditional flags. It can be used for predication and branch instructions. This field can be updated as the regular destination operand of an instruction or as the secondary destination operands associated with conditional modifier. This field can serve as regular source operand of an instruction or serve as the source for predication used for regular instructions or branch instructions. Format: U16
	15:0	Flag subregister f0.0:uw Same as f0.1:uw. Format: U16

11.3.4.7 Mask Registers

ARF Register Type Encoding (RegNum[7:4]): 0100b
 Number of Registers: 1
 Default Value: DMask
 Normal Access: RW
 Elements: 4
 Element Size: 16 bits
 Element Type: UW
 Access Granularity: Word
 Write Mask Granularity: Word
 SecHalf Control: Yes
 Index-ability: No



There is one mask register that contains four subfields with 16-bit each. Each 16-bit mask subfield can be split into two halves (8 bits each). The four mask subregisters, namely, *amask*, *imask*, *lmask* and *cmask*, are used to form the *emask* to control the channel enables for SIMD instructions. They can be updated using the branch instructions such as *if*, *else*, *endif*, *do*, *while*, *break*, etc. A value one at a bit location of a mask subfield indicates that the channel is active with respect to the mask subfield. A value zero at a bit location indicates that the channel is inactive with respect to the mask subfield. For example, an *if* instruction may change some bits in *imask* from one to zero, effectively turning these channels off for the if-endif-block. In the mean time, the previous value in *imask* is pushed in the mask stack register. The corresponding *endif* instruction pops the mask stack register and restores the *imask* value, effectively re-enabled the channels turned off by the *if* instruction.

The four mask subfields can be accessed individually or as a group using region based register addressing. Explicit access of the second half of the mask subfield is through the compression control of **SecHalf**.

Table 11-12. Register and Subregister Numbers for Mask Register

RegNum[3:0]	SubRegNum[4:0]
0000 = mask0 All other encodings are reserved.	00000 = mask0.0:uw (amask) . It contains the active mask register 00010 = mask0.1:uw (imask) . It contains the if-mask register 00100 = mask0.2:uw (lmask) . It contains the loop-mask register 00110 = mask0.3:uw (cmask) . It contains the continue-mask register All other encodings are reserved.

Table 11-13. Mask Register Fields

Dword	Bits	Subfield Description
1	31:16	Continue-Mask (mask0.3:uw or cmask:uw) . This field contains the 16-bit continue mask. Format: U16
	15:0	Loop-Mask (mask0.2:uw or lmask:uw) . This field contains the 16-bit loop mask. Format: U16
0	31:16	If-Mask (mask0.1:uw or imask:uw) . This field contains the 16-bit if-mask. Format: U16
	15:0	Active-Mask (mask0.0:uw or amask:uw) . This field contains the 16-bit active mask. Format: U16

Implementation Restriction on Register Access: When a mask register is used as an explicit source and/or destination, hardware doesn't ensure execution pipeline coherency. Software must set the thread control field to '**switch**' for an instruction that uses mask registers as an explicit operand.



Relaxed Restriction on Register Access: Software must ensure that there is a delay of at least four clock cycles between an instruction, A, that explicitly read the mask register and another instruction, B, that implicitly writes the mask register. This may be achieved, for example, by setting the thread control field to **'switch'** for instruction A or by inserting two nop instructions between instructions A and B. Note that one GEN4 instruction, at minimal, takes two clock cycles.

11.3.4.8 Mask Stack Register

ARF Register Type Encoding (RegNum[7:4]): 0101b
 Number of Registers: 1
 Default Value: 0 (See Erratum below)
 Normal Access: RW
 Elements: 32
 Element Size: 8 bits
 Element Type: UB
 Access Granularity: DQ
 Write Mask Granularity: DQ
 SecHalf Control: No
 Index-ability: No

127	120	119	112	15	8	7	0
isc15				isc1		isc0	
255	148	147	140	143	136	135	128
lsc15				lsc1		lsc0	

The Mask Stack register consists of two stacks: *istack* and *lstack*, 128-bit each. The If-Stack, *istack*, is for 'if' conditional code blocks. The Loop Stack, *lstack*, is for looping code blocks. Unlike traditional hardware stacks, the Mask Stack is implemented by counters, with one counter per execution channel. There are 16 bytes in each stack, corresponding to the 16 counters. The lower 4 bits within each byte is the counter value. The upper 4 bits is forced to zero when the Mask Stack register is the source and are ignored when the Mask Stack register is the destination.

The counters in the mask stack register are initialized to zero at thread dispatch. When a mask is pushed to a mask stack (e.g. an *if* instruction pushes *imask* to *istack*), the counters in the mask stack subfield are updated based on the following rules:

- If a bit of the mask is one, the corresponding mask stack counter is not changed
- If a bit of the mask is zero, the corresponding mask stack counter is incremented by one

When a mask is restored from popping a mask stack (e.g. an *endif* instruction pops *istack* and restores *imask*), the counters in the mask stack subfield are updated based on the following rules:

- If a mask stack counter is zero, it remains zero; and a one is written to the corresponding bit location of the mask
- If a mask stack counter is not zero, it is decremented by one; and a zero is written to the corresponding bit location of the mask



The hardware mask stack is provided to support nested loops/branches. The premise of a counter based mask stack is that once an execution channel is disabled (the mask bit is set to zero, for example, by a *do* instruction), it will remain disabled for subsequent nested branches. It can only be re-enabled after all nested loops/branches within a loop/branch block are completed (the mask bit is restored to one, for example, by the matching *while* instruction to the above mentioned *do* instruction). Therefore, only the number of nesting for a disabled channel needed to be stored in the mask stack. The counter in the mask stack register counts this number.

The bit layout of the Mask Stack register is shown in Figure 11-7. *IStack* takes the lower 128 bits and *LStack* is in the upper 128 bits. Each 128-bit subfield should not be further subdivided.

When SPF bit in *cr0* register is set, this register becomes read-only – contents cannot be modified implicitly or explicitly.

Table 11-14. Register and Subregister Numbers for Mask Stack Register

RegNum[3:0]	SubRegNum[4:0]
0000 = ms0 All other encodings are reserved.	00000 = ms0.0:ub (istack) . It contains the if-stack register 10000 = ms0.16:ub (lstack) . It contains the loop-stack register All other encodings are reserved.

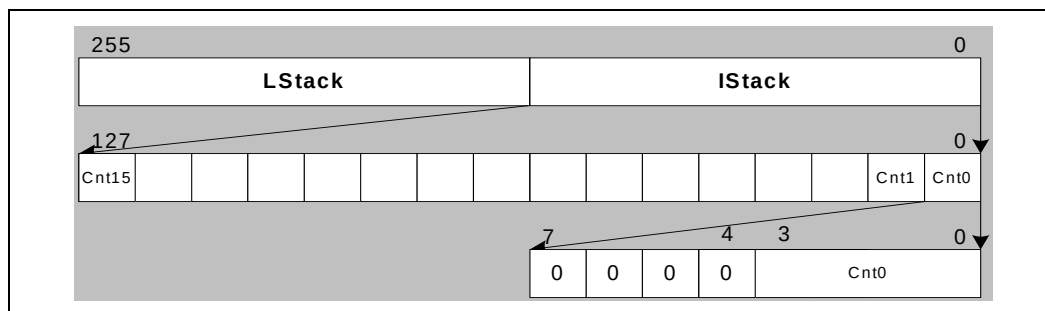
Table 11-15. Mask Stack Register Fields

Subregister	Bits	Subfield Description
isc# (# = 0 to 15)	7:4	Reserved: MBZ.
	3:0	If-Stack Count # : This is the 4-bit If-stack count value for execution channel #, counting the number of 1's has pushed into the If-stack. The 4-bit count per channel provides maximum nesting depth of 15. For nesting beyond 15 levels deep, software must manually save/restore the Mask Stack register. Format: U4
lsc# (# = 0 to 15)	7:4	Reserved: MBZ.
	3:0	Loop Stack Count # : This is the 4-bit loop stack count value for execution channel #, counting the number of 1's has pushed into the loop stack. The 4-bit count per channel provides maximum nesting depth of 7 as one <i>loop</i> -instruction pushes both <i>cmask</i> and <i>bmask</i> in the same mask stack. For nesting beyond 7 levels deep, software must manually save/restore the MaskStack register. Format: U4

The IStack, LStack, or the Mask Stack register, as a whole, can be an explicit source or destination operand. Smaller grain accesses to the register are not allowed. For example, the stacks can be saved/restored using 'mov' instruction via GRF and/or MRF registers as shown in the next instruction.

```
mov (16) r5.0<1>:w ms:w // save mask stack to r5
... // other instructions
mov (16) ms:w r5.0<1>:w // restore mask stack from r5
```

Figure 11-7. Format of the Mask Stack Register



Implementation Restriction on Register Access: When a mask stack register is used as an explicit source and/or destination, hardware doesn't ensure execution pipeline coherency. Software must set the thread control field to **'switch'** for an instruction that uses mask registers as an explicit operand.

This register access restriction is not applicable, hardware does ensure execution pipeline coherency, when a mask stack register is used as an explicit source and/or destination.

[DevBW, DevCL] Erratum: The subfields in mask stack register are reset to zero during graphics reset, however, they are not initialized at thread dispatch. These subfields will retain the values from the previous thread. Software should make sure the mask stack is empty (reset to zero) before terminating the thread. In case that this is not practical, software may have to reset the mask stack at the beginning of each kernel, which will impact the performance.

11.3.4.9 Mask Stack Depth Register

ARF Register Type Encoding (RegNum[7:4]): 0110b
 Number of Registers: 1
 Default Value: 0 (See erratum below)
 Normal Access: R/W
 Elements: 2
 Element Size: 16 bits
 Element Type: UW
 Access Granularity: Word
 Write Mask Granularity: Word
 SecHalf Control: No
 Index-ability: No

The Mask Stack Depth register, *msd0*, allows a software view of the current depth of the Mask Stack. Depths for both the IStack and the LStack are maintained in a single 32-bit register, with individual depths accessible using region-based register addressing. Each depth value occupies a word. Read and write access is provided to allow for extending the Mask Stack Depth beyond 7 or 15 through manual stack management.

Both depths are initialized to 0 at thread-load time. When SPF bit in *cr0* register is set, this register becomes read-only – contents cannot be modified implicitly or explicitly.

The depth value is incremented by hardware each time a value is pushed to the associated Mask Stack. The value is decremented each time a value is popped from the Mask Stack. The addition/subtraction operates in modular math on the 4-bit



unsigned integer fields. Whenever, the Mask Stack Depth value **exceeds** the overflow/underflow trigger values, a mask stack overflow or underflow exception will be generated. Upon a mask stack underflow/overflow exception, software may manually update the mask stack and mask stack depth registers based on the amount of underflow/overflow.

Table 11-16. Overflow / Underflow Exception Trigger Value

Device	Underflow	Overflow
[DevBW]	0	15
[DevCL]	0	7

For example, if the current depth value is 15 and the overflow trigger value is 15, an instruction that pushes two masks into a mask stack will cause mask stack overflow exception and the resulting depth value is 1. Upon an overflow exception, software may compute the amount of overflow by adding 1 to the depth value in the register. In this case, the amount of overflow is 2.

Another example, if the current depth value is 5 (0101b) and the underflow trigger value is 0, an instruction that pops a mask stack register by 9 (1001b) will cause mask stack underflow exception with a resulting depth value of 12 (1100b). By subtracting the depth value from 16, software can determine the amount of underflow. In this case, it is 4.

Implementation Restrictions: When the Overflow trigger value equals to the maximum mask stack value, the overflow exception mechanism is not sufficient to support extended stack managed by software. This is because that software must reset the Mask Stack Depth value for the overflowed mask stack to 0 and reset the mask stack values to 0 or 1. If the original value of a channel is 0, it remains 0; if the original value of a channel is non-zero, it is reset to 1. Continuing pushing into the stack will eventually cause the mask stack counter to wrap before the Mask Stack Depth value to exceed the overflow trigger value. This will cause functional error. Software may choose to use different workarounds for the device that has the overflow trigger value equals to the maximum mask stack value. Here are a few workaround examples:

- For loops/branch within the hardware limit, software can use the hardware stack
- For loops/branch beyond the limit, there are several solutions
 - o Use single program flow - processing a pixel at a time. May not be a benchmark or real performance issue
 - o Use software jitted stack management (without using the overflow exception mechanism)
- For example, software may manually manage the overflow cases, ensuring that overflow exception will never be reached. This solution may use the underflow exception though.



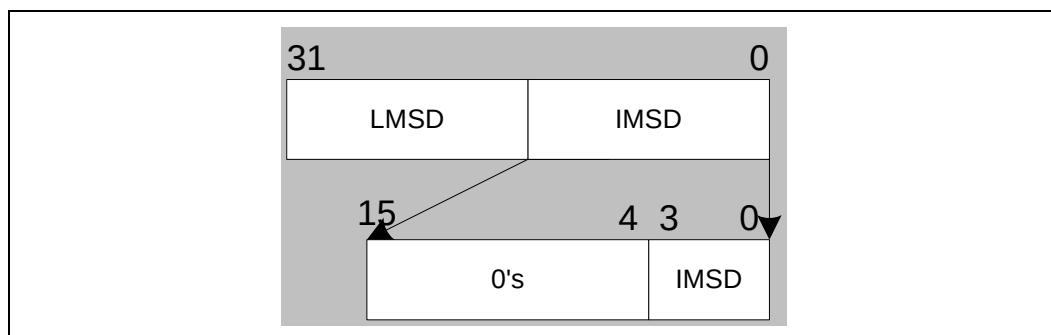
Table 11-17. Register and Subregister Numbers for Mask Stack Depth Register

RegNum[3:0]	SubRegNum[4:0]
0000 = msd0 All other encodings are reserved.	00000 = msd0.0:w (imsd) . It contains the if-stack depth register 00010 = msd0.1:w (lmsd) . It contains the loop-stack depth register All other encodings are reserved.

Table 11-18. Mask Stack Depth Register Fields

DWord	Bits	Subfield Description
0 (lmsd)	31:20	Reserved: MBZ.
	19:16	Loop-Stack Depth: Current depth of the LStack. The maximum allowed nesting of <i>loop</i> -instructions is half of the overflow trigger value before a stack overflow or underflow. This is because that loop-instructions such as <i>do</i> (or <i>while</i> / <i>break</i> / <i>cont</i>) push (or pop) both <i>lmask</i> and <i>cmask</i> to (or from) the LStack. When overflow/underflow occurs, this field contains the modular residue of the amount of overflow/underflow. Initialized to 0 at thread load time. Format: U4
1 (imsd)	15:4	Reserved: MBZ.
	3:0	If-Stack Depth: Current depth of the IStack. The maximum allowed nesting of <i>if</i> -instructions equals to the overflow trigger value before a stack overflow or underflows. When overflow/underflow occurs, this field contains the modular residue of the amount of overflow/underflow. Initialized to 0 at thread load time. Format: U4

Figure 11-8. Format of the Mask Stack Depth Register



Implementation Restriction on Register Access: When a mask stack depth register is used as an explicit source and/or destination, hardware doesn't ensure execution pipeline coherency. Software must set the thread control field to **'switch'** for an instruction that uses mask registers as an explicit operand.



This register access restriction is not applicable, hardware does ensure execution pipeline coherency, when a mask stack depth register is used as an explicit source and/or destination.

Erratum: The subfields in mask stack depth register are reset to zero during graphics reset, however, they are not initialized at thread dispatch. These subfields will retain the values from the previous thread. Software should make sure the mask stack depth values are zero before terminating the thread. In case that this is not practical, software may have to reset the mask stack depth at the beginning of each kernel, which will impact the performance.

HW stack overflow and underflow exception are not supported for both IStack and LStack. Software has to maintain stack depth to ensure stack will not be overflowed and underflowed. Both stack overflow in HW when they reach 16 for BWR and 7 for CLN, and underflow when they pass 0.

11.3.4.10 State Registers

ARF Register Type Encoding (RegNum[7:4]): 0111b
 Number of Registers: 1
 Default Value: Provided by the Dispatcher
 Normal Access: RO
 Elements: 2
 Element Size: 32 bits
 Element Type: UD
 Access Granularity: Byte
 Write Mask Granularity: N/A
 SecHalf Control: No
 Index-ability: No

Thread state registers are read-only registers. They can be accessed as a whole or individually using region-based addressing with byte granularity.

Table 11-19. Register and Subregister Numbers for State Register

RegNum[3:0]	SubRegNum[4:0]
0000 = sr0 All other encodings are reserved.	Valid encoding range: 00000 – 00111 (in unit of byte) For example, 00000 = sr0.0:ud . It contains general thread states such as EUID/TID 00100 = sr0.1:ud . It contains the dispatch mask All other encodings are reserved.



Table 11-20. State Register Fields

DWord	Bits	Subfield Description
0 (sr0.0:ud)	31:28	Reserved: MBZ
	27:24	FFID (Fixed Function Identifier). Specifies which fixed function unit generates the current thread. This field is set at thread dispatch and is forwarded on the message bus for all out-going messages from this thread.
	23:19	Reserved: MBZ
	18	Priority Class. This field, when set, indicates the thread belongs to the high priority class, which has higher scheduling priority over any thread with this field cleared. The priority field below determines the relative priority within the same priority class. This field is initialized by the thread dispatcher at thread dispatch time and stays unchanged throughout the life span of the thread. This field is forwarded on the message bus to the message bus arbiter for all out-going messages. It serves as a priority hint for the target shared function. See shared function chapters whether and how a shared function uses this priority hint. 0 = Low priority class 1 = High priority class
	17:16	Priority. This field is the relative aging priority of the thread. This field indicates the 'age' of this thread relative to other thread within the EU. No two threads within the same EU can have the same priority number (independent of the priority class value). Within the same priority class, an older thread (with a larger priority number) has higher schedule priority over a younger thread. This field is set to zero at a thread's dispatch. During a thread's run time, this field may or may not be incremented when a new thread is dispatched to the same EU. It is only incremented when another thread's priority number is incremented and reaches the same value. For example, if currently there is a thread with priority 0 on an EU, dispatching a new thread to this EU will cause the old thread's priority number being incremented to 1. However, if the active thread (assuming for simplicity there is only one) on an EU has a priority number 1 (or 2 or 3), dispatching a new thread to this EU will not change the old thread's priority number. As threads on an EU may terminate in arbitrary orders, the exact number for a thread depends on the dynamic execution of threads.
	15:12	Reserved: MBZ
	11:8	EUID (Execution Unit Identifier). Specifies which execution unit the current thread is at.
	7:3	Reserved: MBZ
	2:0	TID (The thread identifier). Specifies which thread slot the current thread is assigned to. This field is set at thread dispatch.
1 (sr0.1:ud)	31:24	Reserved: MBZ
	23:20	GRF Register Blocks. This field contains a number of GRF register count in unit of 16 registers. The valid range is from 16 to 128 GRF registers. 0000: Reserved 0001 – 1000: 16 GRF registers to 128 GRF registers 1001 – 1111: Reserved
	19:16	Reserved : MBZ



DWord	Bits	Subfield Description
	15:0	Dispatch Mask (DMask). The 16-bit field specifies which channels are active at Dispatch time. This field is used by hardware to initialize the mask register. Format: U16

11.3.4.11 Control Register

ARF Register Type Encoding (RegNum[7:4]): 1000b
 Number of Registers: 1
 Default Value: Provided by the Dispatcher
 Normal Access: RW
 Elements: 4
 Element Size: 32 bits
 Element Type: UD
 Access Granularity: Dword
 Write Mask Granularity: Dword
 SecHalf Control: No
 Index-ability: No

The Control register is a read-write register. It contains four 32-bit subregisters that can be accessed individually.

Subregister *cr0.0:ud* contains normal operation control fields such as the floating point mode and the accumulator disable. It also contains the master exception status/control field that allows software to switch back to the application thread from the system routine. Debug control field Breakpoint Suppress is also in *cr0.0:ud*.

Subregister *cr0.1:ud* contains the mask and status/control fields for all exceptions. The exception fields are arranged in significance-decreasing order from MSB to LSB. This allows the system routine to use *lzd* instruction to find the high priority exceptions and handles them first. As each exception is mapped to a single bit, other exception priority order may be implemented by software. System routine may choose to handle one exception at a time, by handle the exception detected by a *lzd* instruction and return to application thread. Or it may choose to handle all the concurrent exceptions, by looping through the exception fields until all outstanding exceptions are handled before returning back to the application thread.

Exception enable bits (bits 15:0 in *cr0.1:ud*) control whether an exception will cause hardware to jump to system routine or not. Exception status and control bits (bits 31:16 in *cr0.1:ud*) indicate which exceptions have occurred and are used for system routine to clear the exception. Even if a given exception is disabled, the corresponding exception status and control bit still reflects the status whether an exception event has occurred or not.

cr0.2:ud contains the **Application IP (AIP)** indicating the current thread IP when an exception occurs.

cr0.3:ud is reserved. Writing to this subregister is dropped; result of reading from this subregister is unpredictable.

Fields in Control registers also refer to a virtual register called **System IP (SIP)**. SIP is the virtual register holding the global System IP, which is the initial instruction pointer for the system routine. There is only one SIP for the whole system. It is virtual



only from a thread's point of view, as it is not visible (i.e. not readable and not writeable) to the thread software executed on a GEN4 EU. It can only be accessed indirectly by the hardware to response to exception events. Upon an exception, hardware performs some book keepings (e.g. saving the current IP into AIP) and then jumps to SIP. Upon finishing exception handling, the system routine may return back to the application by clearing the Master Exception Status and Control field in *cr0*, which will cause the hardware to load AIP to IP register. See STATE_SIP command for how to set SIP.

Details about exception handling and debug can be found in the debugging and exceptions chapters.

Table 11-21. Register and Subregister Numbers for Control Register

RegNum[3:0]	SubRegNum[4:0]
0000 = cr0 All other encodings are reserved.	00000 = cr0.0:ud . It contains general thread control fields 00100 = cr0.1:ud . It contains exception status and control 01000 = cr0.2:ud . It contains AIP.10100 (reserved) All other encodings are reserved.

Table 11-22. Control Register Fields

DWord	Bits	Subfield Description
0 (cr0.0:ud)	31	Master Exception State and Control. This field is the master state and control for all exceptions. Reading a 0 indicates that the thread is in normal operation state and a 1 means the thread is in exception handle state. Upon an exception event, hardware sets this field to 1 and switch to SIP. Writing a 1 to this field has no effect. Writing a 0 to this field also has no effect if the previous value is 0. In both cases, the field keeps the previous value. If the previous value of this field is 1, software writing a 0 causes the thread to return to AIP. This transition is automatic – software does not have to move AIP to IP. The value of this field then stays as 0. This field is initialized to 0. 0 = Indicate that the thread is in normal state 1 = Indicate that the thread is in exception state
	30:16	Reserved: MBZ



DWord	Bits	Subfield Description
	15	Breakpoint Suppress. This field specifies whether breakpoint exception is suppressed or not. This field is normally set by software and cleared by hardware. If Master Exception Status and Control field is 1, this field is ignored by hardware. If Master Exception Status and Control field is 0 (i.e. not in system routine) and Breakpoint is enabled: If this field is set, breakpoint is temporally ignored (suppressed); Upon a breakpoint condition, the instruction is executed and this bit is automatically reset by hardware. This field is provided to prevent infinite loop of jumping to the system routine on a breakpoint condition. The system routine must set this bit (and also clear the corresponding status and control field) before returning to the application thread. This field has no effect when Breakpoint Enable bits is cleared. This field is initialized to 0. 0 = Breakpoint exception is not suppressed 1 = Breakpoint exception is suppressed
	14:9	Reserved : MBZ
	8	Host Notification Data. This field is forwarded to the debug MMIO registers (EU Debug Register 4 and 5 – Attention Data registers) when a host notification signal is sent by the EU upon executing instruction “WAIT n1”. Combining host notification signaling mechanism, this data field provides a one-way communication interface from the thread to the host. Even though it may be slow, this thread/host interface is independent of the message bus structure and any shared function. Note that this communication interface does not allow the host to return data to the thread. More details can be found in Debugging chapter. This field is initialized to 0.
	7:3	Reserved : MBZ
	2	Single Program Flow (SPF). Specifies whether the thread has a single program flow (SIMDn_{xm} with m = 1) or multiple program flows (SIMDn_{xm} with m > 1). This field affects the operation of all branch instructions. If SPF is not set, branch instructions such as <i>if</i>, <i>do</i>, <i>while</i>, push the current branch masks into the corresponding mask stack registers or pop the mask stack and updates the corresponding masks. Use of the mask stack supports concurrent execution of multiple program flows in a single thread using SIMD instructions. If SPF is set, the mask stack push or pop actions are inhibited for these branch instructions. Therefore, the top of stack will not be modified by any branch instruction. In Single Program Flow mode, all execution channels branch and/or loop identically. By hold the mask stack unchanged for all branch instructions, infinite level of nesting of branch instructions can be supported. This field is initialized by the Thread Dispatch. 0 = Multiple Program Flows 1 = Single Program Flow



DWord	Bits	Subfield Description
	1	Accumulator Disable. This field controls the implicit update of the accumulator. If this field is cleared, the accumulator is updated for all instructions in which it is designated implicitly. If set, the accumulator is disabled for all implicit update operations, maintaining its value prior to being disabled. Setting this field has no effect if the accumulator is the explicit destination operand for an instruction. This field is initialized to 0. 0 = Enable accumulator update 1 = Disable accumulator update <i>Usage Notes:</i> This control bit is primarily designed for the System Routine. That routine is not expected to use the accumulator, though it may need to use instructions which include implicit update of the accumulator. In order to use those instructions within the System Routine, but still preserving the accumulator contents upon return to the application kernel, the System Routine would either (a) save and restore the accumulator, or (b) prevent the accumulator from being unintentionally modified. This control bit has been added for the latter method. Software has the option to limit the setting of this control bit strictly within the System Routine. If, by convention, this bit is clear within application kernels, the System Routine can simply set the bit upon entry and clear it prior to returning control to the application kernel. This usage model would not necessarily require cr0.0 to be saved/restored in the System Routine. However, if by convention application kernels are permitted to set this bit, then the System Routine would be required to preserve the content of this bit.
	0	Floating Point Mode (FPMODE). This field specifies whether the current floating point operation mode is in IEEE standard mode or the alternative mode. It is used to control the floating operation of the Execution Unit. It is also forwarded on the message sideband for all out-going messages, for example, to control the floating point mode of the Extended Math unit or the Sampler unit. Software may modify this field to dynamically switch between the two floating point modes. This field is initialized by the Thread Dispatch. 0 = IEEE floating point mode 1 = Alternative floating point mode



DWord	Bits	Subfield Description
1 (cr0.1:ud) [31:16] StatCtrl [15:0] Masking	31	Breakpoint Exception Status and Control. This field, when set, indicates breakpoint exception. Under breakpoint condition, hardware sets this bit upon entering the system routine. This field is used for kernel debug in both breakpoint and single stepping modes. For normal breakpoint handle, the system (debug) routine should reset this field and also set the Breakpoint Suppress field before returning back to the application thread. If this bit is not reset in the system routine, upon returning to application routine, hardware executes one instruction (if Breakpoint Suppress field was set in the system routine), and then jumps to system routine again. Therefore, by not clearing this field, single stepping can be emulated. This bit may be set and cleared by software. This field is initialized to 0. There is a restriction associated with this field when emulating single stepping debug on – it does not work on compressed instructions. When system routine sets this field as well as the Breakpoint Suppress field for a compressed instruction before returning to application routine, the instruction pointer is not automatically advanced by hardware. This prohibits the intended single-stepping mechanism supplied by hardware. Two software workarounds are available, which emulate the single stepping debugging capability with certain drawbacks. 1. The host debugger software may expose to the debug user interface with breakpoint and single stepping methods but set the breakpoint for all instructions in kernel binary. For breakpoint events that don't match with any user exposed breakpoint/single-stepping instructions, the host debugger software may silently skip the false breakpoint and simply let the system routine to return back to application routine. This approach has low performance. 2. Alternatively, the host debugger software may set/clear the breakpoint field on an instruction by instruction basis to emulate single stepping. After servicing a breakpoint and before letting the system routine to return to the application routine, the host debugger may perform the following steps: flush the instruction cache, restore the breakpoint field for the current instruction, store the breakpoint field for the next instruction, and then set the breakpoint field for the next instruction. This process is repeated each step of source code stepping. A side effect is that the kernel instructions in memory are dynamically modified, resulting other threads under debug (with breakpoint enabled) to cause breakpoint exception on an unintended instruction. Host debugger software must detect and skip these false exceptions (e.g. based on identifier of the thread).
	30	External Halt Exception Status and Control. This field indicates the External Halt exception. It is set by EU hardware upon receiving the broadcast External Halt signal. System routine should reset this field before returning to application routine in order to avoid infinite loop. This bit may be set or cleared by software. This field is initialized to 0.



DWord	Bits	Subfield Description
	29	Software Exception Control. This is the control field of software exception. Setting this field to 1 in application routine will cause an exception. Clearing this field in application routine has no effect. Upon entering system routine, the hardware maintains this field as one to signify software exception. System routine should reset this field before returning to application routine. This field may be set or cleared by software. This field is initialized to 0.
	28	Illegal Opcode Exception Status. This field, when set, indicates illegal opcode exception. The exception handle routine normally does not return back to the application thread upon an illegal opcode exception. Leaving this bit set, has no effect on hardware – if system software adversely returns to application routine leaving this field set, it doesn't cause any exception. This field should not be set by software or left set by system routine to avoid confusion. This field is initialized to 0.
	27	LStack Overflow Exception Status. This field, when set, indicates the LStack overflow exception. System routine should clear this field before returning back to the application thread. Leaving this bit set, has no effect on hardware – if system software adversely returns to application routine leaving this field set, it doesn't cause any exception. This field should not be set by software or left set by system routine to avoid confusion. This field is initialized to 0.
	26	LStack Underflow Exception Status and Control. This field, when set, indicates the LStack Underflow exception. The exception handle routine should clear this field before returning back to the application thread. Leaving this bit set, has no effect on hardware – if system software adversely returns to application routine leaving this field set, it doesn't cause any exception. This field should not be set by software or left set by system routine to avoid confusion. This field is initialized to 0.
	25	IStack Overflow Exception Status and Control. This field, when set, indicates the IStack Underflow exception. The exception handle routine should clear this field before returning back to the application thread. Leaving this bit set, has no effect on hardware – if system software adversely returns to application routine leaving this field set, it doesn't cause any exception. This field should not be set by software or left set by system routine to avoid confusion. This field is initialized to 0.
	24	IStack Underflow Exception Status and Control. This field, when set, indicates the IStack Underflow exception. The exception handle routine should clear this field before returning back to the application thread. Leaving this bit set, has no effect on hardware – if system software adversely returns to application routine leaving this field set, it doesn't cause any exception. This field should not be set by software or left set by system routine to avoid confusion. This field is initialized to 0.
	23	Reserved: MBZ.
	22:16	Reserved: MBZ



DWord	Bits	Subfield Description
	15	Breakpoint Enable. Specifies whether breakpoint exception is enabled or not. This field is initialized by the Thread Dispatcher. Format = ENABLED 0 = Disabled 1 = Enabled
	14	External Halt Exception Enable. This field specifies whether the External Halt Exception is enabled or not. When this bit is set, the thread allows to be interrupted by an external halt signal. The usage of this exception is for a host debug software to halt the execution of threads in GEN4 EUs at any selected time. With proper debug handling in SIP, execution may resume normally after thread being halted. This field is initialized by the Thread Dispatcher. Format = ENABLED
	13	Software Exception Enable. This field enables or disables the software exception. Enabling or disabling this field may allow host software to turn on/off certain features (such as profiling) without changing the kernel program. This field is initialized by the Thread Dispatcher. Format = ENABLED
	12	Illegal Opcode Exception Enable. This field specifies whether illegal opcode exception is enabled or not. Illegal opcode exception includes illegal opcode and undefined opcode, caused by bad program or run time data corruption. This field is initialized by the Thread Dispatcher. Software should normally set it in the interface descriptor. Even though the mechanism is provided to disable illegal opcode exception, it should be used with extreme caution. Format = ENABLED
	11	MaskStack Exception Enable. This field specifies whether MaskStack Exceptions are enabled or not. It is used to control all four MaskStack Exceptions (LStack Overflow and Underflow, IStack Overflow and Underflow). This field is initialized by the MaskStack Exception Enable bit from the Thread Dispatcher. Software should normally set it in the interface descriptor. Even though the mechanism is provided to disable stack overflow/underflow exception, it should be used with extreme caution. For the kernel that is known to not overflow/underflow mask stacks, enable this exception has no adverse effect. Format = ENABLED Erratum: This field is reserved: MBZ.
	10	Reserved: MBZ
	9:0	Reserved : MBZ



DWord	Bits	Subfield Description												
2 (cr0.2:ud)	31:4	Application IP (AIP). This is the register storing the instruction pointer before an exception is handled. Upon an exception, hardware automatically saves the current IP into the AIP register, and then sets the Master Exception State and Control field to 1, which forces a switch to the System IP (SIP). AIP register may contain either the pointer to the instruction that causes the exception (such as breakpoint for debug), or the one after (such as mask stack overflow/underflow exceptions). This is shown in the following table, where IP is the instruction which generated the exception. <table><thead><tr><th>Exception Type</th><th>AIP Value</th></tr></thead><tbody><tr><td>Breakpoint</td><td>IP</td></tr><tr><td>External Halt</td><td>n/a ⁽¹⁾</td></tr><tr><td>Software Exception</td><td>IP + 1</td></tr><tr><td>Illegal Opcode</td><td>IP</td></tr><tr><td>Mask Stack Overflow / Underflow</td><td>IP + 1</td></tr></tbody></table> (1) External Halt exception is asynchronous and not associated with an instruction. When the system routine changes the Master Exception State and Control field from 1 to 0. Hardware restores IP from this register. This field is writable allowing returning IP to be altered after an exception handle.	Exception Type	AIP Value	Breakpoint	IP	External Halt	n/a ⁽¹⁾	Software Exception	IP + 1	Illegal Opcode	IP	Mask Stack Overflow / Underflow	IP + 1
	Exception Type	AIP Value												
Breakpoint	IP													
External Halt	n/a ⁽¹⁾													
Software Exception	IP + 1													
Illegal Opcode	IP													
Mask Stack Overflow / Underflow	IP + 1													
	3:0	Reserved : MBZ												

Implementation Restriction on Register Access: When the control register is used as an explicit source and/or destination, hardware doesn't ensure execution pipeline coherency. Software must set the thread control field to **'switch'** for an instruction that uses control register as an explicit operand. This is important as the control register is an implicit source for most instructions. For example, fields like FPMODE and Accumulator Disable control the arithmetic and/or logic instructions. Therefore, if the instruction updating the control register doesn't set 'switch', subsequent instructions may have indeterministic results.



11.3.4.12 Notification Registers

ARF Register Type Encoding (RegNum[7:4]): 1001b
 Number of Registers: 3
 Default Value: No
 Normal Access: RO
 Elements: 3
 Element Size: 32 bits
 Element Type: UD
 Access Granularity: Dword
 Write Mask Granularity: Dword
 SecHalf Control: No
 Index-ability: No

There are two notification registers (*n0:ud* and *n1:ud*) used by the *wait* instruction. These registers are read-only and can be accessed in 32-bit granularity.

It should be noted that in the extreme case, it is possible to have more notifications to a thread than the maximal allowable of concurrent threads in the system. Therefore, the range of the thread-to-thread notification count in *n0*, is larger than the maximum number of threads computed by $EUID * TID$.

There is only one bit for the host-to-thread notification count in *n1*. That means if there are multiple host write to EU Debug Register 6 or 7 (Attention Clear registers) may be collapsed into a single notification event to the target thread, if only one wait instruction is executed during that time period.

When directly accessed, this register is read-only. If the value is none zero, the only way to alter the value is to use the wait instruction to decrement the value until zero is reach. A wait instruction on a zero notification subregister will cause the thread to stall, waiting for a notification signal from outside targeting to the same subregister. See wait instruction for details.

Implementation Restrictions: The notification registers are initialized to 0 after hardware/software reset. However, it is not reset at thread dispatch time.

Table 11-23. Register and Subregister Numbers for Notification Register

RegNum[3:0]	SubRegNum[4:0]
0000 = <i>n0</i> 0001 = <i>n1</i> All other encodings are reserved.	Reserved.



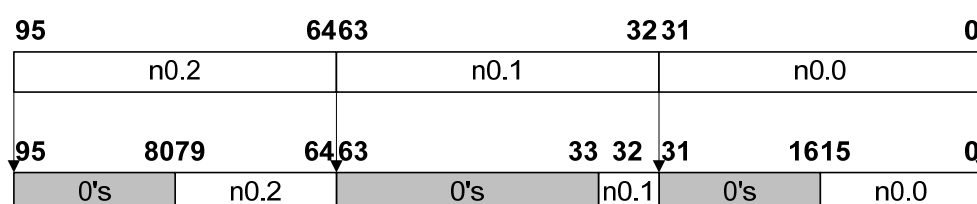
Table 11-24. Fields of Notification Register n0

DWord	Bits	Subfield Description
0	31:7	Reserved: MBZ
	6:0	Thread to Thread Notification Count. This register is used by the WAIT instruction for thread-to-thread synchronization. The value read from this register specifies the outstanding notifications received from other threads. It can be changed indirectly by using the WAIT instruction. See WAIT instruction for details. Format: U7

Table 11-25. Fields of Notification Register n1

DWord	Bits	Subfield Description
0	31:1	Reserved : MBZ
	0	Host to Thread Notification. This register is used by the WAIT instruction for host-to-thread synchronization via MMIO registers EU Debug Register 6 and 7 (Attention Clear registers). See Debugging chapter for details. Format: U1
	15:0	Thread to Thread Notification Count. This register is used by the WAIT instruction for thread-to-thread synchronization. The value read from this register specifies the outstanding notifications received from other threads. It can be changed indirectly by using the WAIT instruction. See WAIT instruction for details. Format: U16

Table 11-26. Format of the Notification Register





11.3.4.13 IP Register

ARF Register Type Encoding (RegNum[7:4]):	1010b
Number of Registers:	1
Default Value:	Provided by the Dispatcher
Normal Access:	RW
Elements:	1
Element Size:	32 bits
Element Type:	UD
Access Granularity:	Dword
Write Mask Granularity:	Dword
SecHalf Control:	No
Index-ability:	No

The ip register can be accessed as a 32-bit quantity. It is a read-write register, containing the current instruction pointer, which is relative to the **Generate State Base Address**. Reading this register returns the instruction pointer of the current instruction. The 3 LSBs are always read as zero. Writing this register forces the program flow to jump to the new address. When it is written, the 3 LSBs are dropped by hardware.

Table 11-27. Register and Subregister Numbers for IP Register

RegNum[3:0]	SubRegNum[4:0]
0000 = <i>ip</i> All other encodings are reserved.	00000 = <i>ip:ud</i> All other encodings are reserved.

Table 11-28. IP Register Fields

DWord	Bits	Subfield Description
0	31:4	Ip. Specifies the current instruction pointer. This pointer is relative to the General State Base Address .
	3:0	Reserved : MBZ
	2:0	Reserved : MBZ

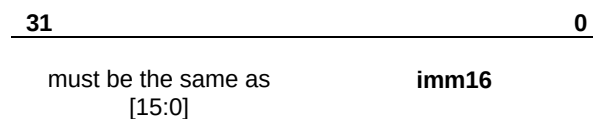
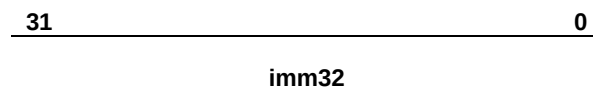


11.3.5 Immediate

Two forms of immediate are provided as a source operand: scalar and vector.

For a scalar immediate, it can be of any of the specified numerical data types from a word to a dword. Byte and unsigned byte are not supported as the smallest internal type of the execution pipeline is word. These two numerical types are reserved for future extensions.

The immediate field in a GEN4 instruction has 32 bits as shown below. For a word or an unsigned word immediate data, software must replicate the same 16-bit immediate value to both the lower word and the high word of the 32-bit immediate field in a GEN4 instruction.



The immediate form of vector allows a constant vector to be in-lined in the instruction stream. Both integer and float immediate vectors are supported.

An immediate integer vector is denoted by type **v** as *imm32:v*, where the 32-bit immediate field is partitioned into 8 4-bit subfields. Each 4-bit subfield contains a signed integer value in 2's complement form (halfbyte). Therefore each 4-bit subfield has a range of [-8, +7]. This is depicted in the following table.

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0
immV7	immV6	immV5	immV4	immV3	immV2	immV1	immV0

An immediate float vector is denoted by type **vf** as *imm32:vf*, where the 32-bit immediate field is partitioned into 4 8-bit subfields. Each 8-bit subfield contains a signed float with 3-bit exponent and 4-bit fraction. Each 8-bit subfield provides signed floating point values with restricted range and precision. This is depicted in the following table.

31 24	23 16	15 8	7 0
immVF3	immVF1	immVF1	immVF0



*Restriction: When an immediate vector is used in an instruction, the destination must be 128-bit aligned with destination horizontal stride equivalent to a word for an immediate integer vector (**v**) and equivalent to a dword for an immediate float vector (**vf**).*

11.3.6 Region Parameters

Unlike conventional SIMD architectures where an N-bit wide SIMD instruction can only operate on N-bit aligned SIMD data registers, a region-based register addressing scheme is employed in GEN4 architecture. The region-based register addressing capability significantly improves the SIMD computation efficiency by providing per-instruction-based multiple data gathering from register file. This avoids instruction overhead to perform data pack, unpack, and shuffling, which has been observed on other SIMD architectures. One benefit of such capability is allowing various kinds of 3D Graphics API Shader compute models to run efficiently on GEN4. Another benefit is allowing high throughput of media applications, which tend to operate on byte or word data elements.

This can be illustrated by the example shown in Figure 11-9 and Figure 11-10. As shown in Figure 11-9, a sequence of SIMD instruction is executed on a conventional load/store based superscalar machine with SIMD instruction extension. The data parallelism can be achieved by first level of loop unrolling. As shown, there is a second level of loop for the task. Before a given SIMD compute instruction, *Process (i)*, can proceed, there might be a load, a data rearrange and a data unpack (and conversion) instruction to load and prepare the input data. After the compute instruction is complete, it might also require pack, re-arrange and store instructions, to format and save the same to memory. At the loop, other scalar computations such as loop count and address generation may be needed. For the same program, when the data can fit in the large GEN4 GRF register file, the outer loop may be unrolled for GEN4. Here one or a few block loads (using *send* instruction) may be sufficient to move the working set into GRF. Then the data shuffle can be combined with the processing operation with region-based addressing capability. Per operand float type and mixed data type operation may also allow GEN4 to combine data conditioning operations with computing operations. These techniques in GEN4 architecture help to achieve high compute efficiency and throughput for graphics and media applications.

Figure 11-9. Conventional SIMD Instruction Sequence

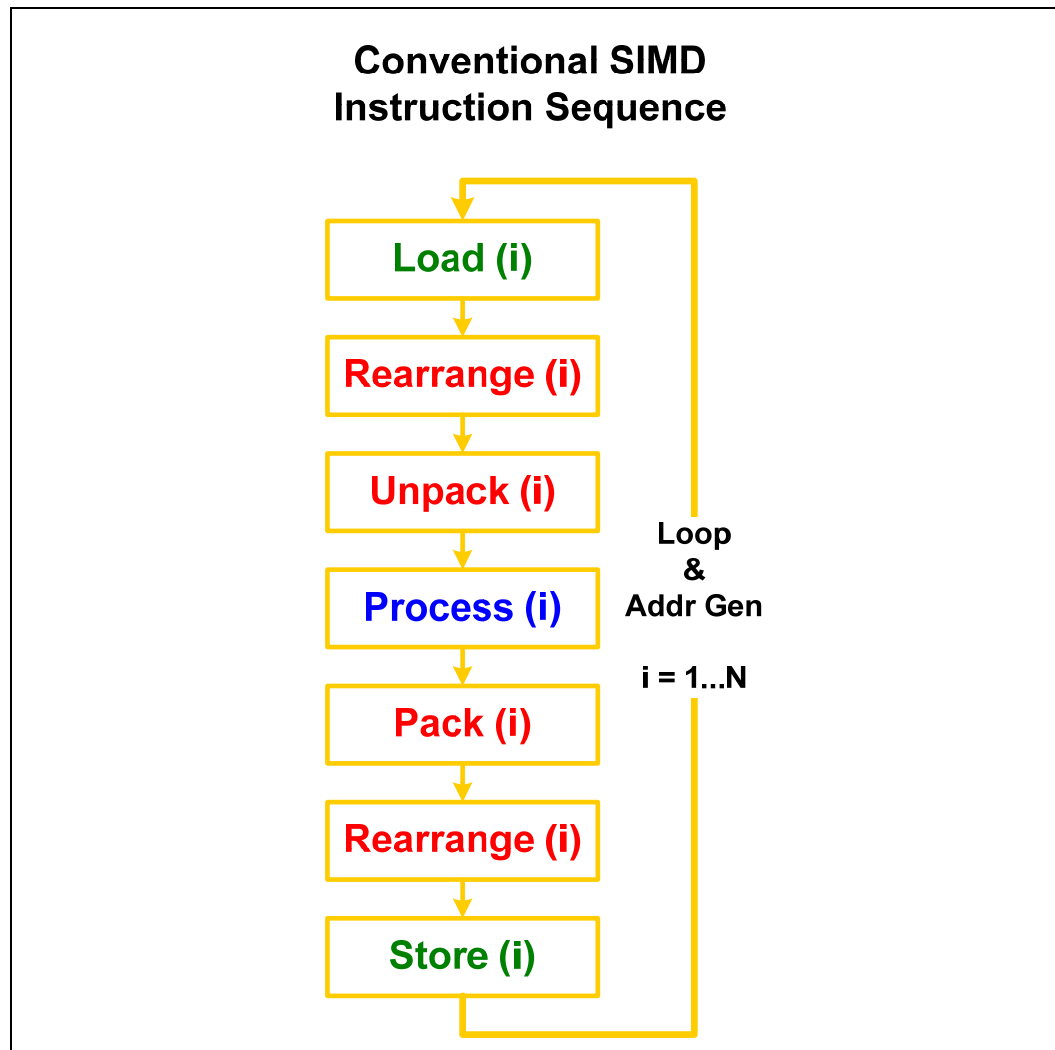
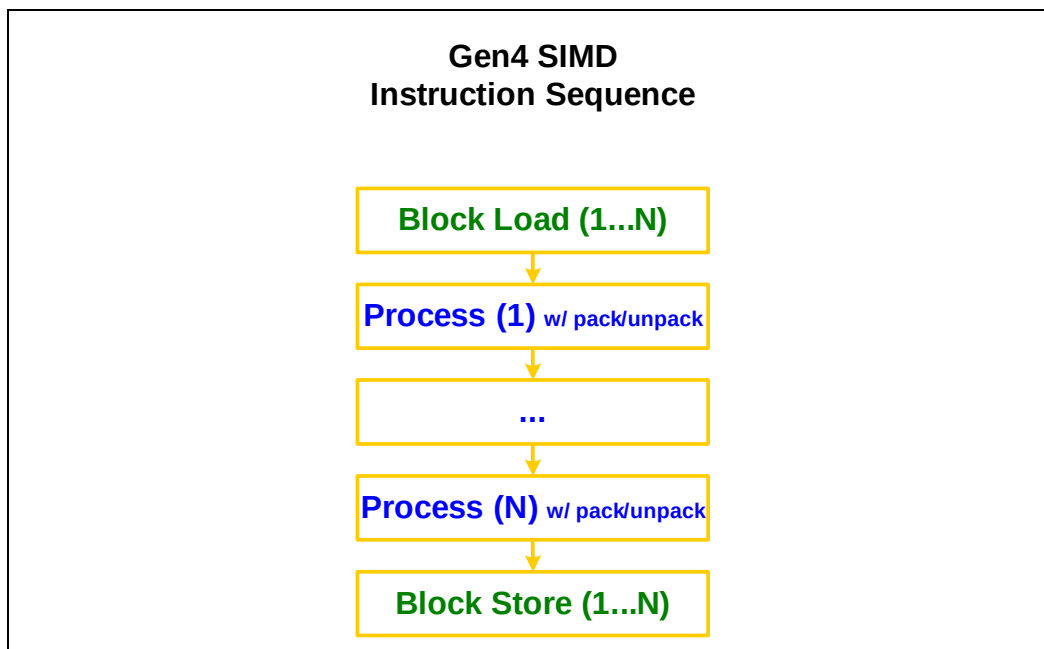




Figure 11-10. GEN4 SIMD Instruction Sequence for the Same Program



In a GEN4 instruction, each operand defines a region in the register file. A region may contain multiple data elements. Each data element is assigned to an execution channel in the EU. The total number of data elements of a region is called the **size** of the region, or the size of the operand. The number of execution channels is called the **execution size (ExecSize)**, which is specified in the instruction word. ExecSize determines the size of region for source and destination operands in an instruction.

- For an instruction with two source operands, the sizes of the two source operands must be the same.
- The size of a destination operand generally matches the execution size, therefore equals to the number of source operand(s) in the same instruction.
 - Exception of this rule is present for the integer reduction instructions (such as sad2 and sada2) where the destination area is smaller than the source area.

Regions are **generalized 2-dimensional** (2D) arrays in row-major order. The first dimension is named the **horizontal** dimension (data elements within a row) and the second dimension is termed the **vertical** dimension (data elements in a column). Here, horizontal/vertical and row/column are just symbolic notations. When the GRF or MRF registers are viewed as a row-major 2D array of memory, such a notation normally matches well with the geometric locations of the data elements of an operand. However, as the register region is fully described by the parameters discussed below, the data elements of a register region may not form a regular rectangular shape. For example, Vertical Stride parameter is allowed to be smaller than Horizontal Stride, making the rows of a register region interleave with each other. It should also note that the meanings of horizontal/vertical here is different than that used for the flag control in Section 11.3.4.6.



Specifically, a region-based description of a source operand can take the following format

RegFile RegNum.SubRegNum<VertStride;Width,HorzStride>:type

Parameters are as the follows.

- Register Region Origin (*RegFile*, *RegNum* and *SubRegNum*): This set of parameters, including the register file, *RegFile*, the register number, *RegNum*, and the subregister number, *SubRegNum*, describes the register region origin, which is the location of the first data element of the operand. *RegNum* is in unit of 256-bit and *SubRegNum* is in unit of the data element size.
- Width (*Width*): *Width* specifies the number of data elements along the horizontal dimension, or the number of data elements of a row.
- Horizontal Stride (*HorzStride*): *HorzStride* specifies the step size between two adjacent data elements within a row. It is in unit of data element size, which is determined by the data element *Type*.
- Vertical Stride (*VertStride*): *VertStride* specifies the step size between two adjacent data elements along the vertical dimension (or the step size between two rows). It is again in unit of data element size, which is determined by the data element *Type*.
- Data Element Type (*Type*): *Type* specifies numerical data type (float, word, byte, etc.) of the data elements. All data elements within a region must have the same type.

In GEN4, both GRF and MRF register files consist of a sequence of 256-bit physical registers. When viewing the register file (GRF for example) as a sequence of 256-bit aligned physical registers, *RegNum* field provides the physical register number, thus for the name. *SubRegNum* provides the sub-field addressing within a physical register. However, when viewing the register file as a byte addressable memory array, (*RegNum* and *SubRegNum*) is just a byte address within the register file with *SubRegNum* providing the lower 5 bits and *RegNum* providing the higher bits.

The execution size is specified for each instruction by the parameter *ExecSize*. The size of the vertical dimension is *ExecSize/Width*, based on the rule that the size of regions must equal to the execution size.

Figure 11-11 depicts the register region description. The example shows a register region of *r4.1<16;8,2>:w*, where the shaded fields denote the data elements in the region and the numbers in these fields are the execution channel assignments. The register region assumes that an *ExecSize* of 16 is set for the instruction. Each data element is a word (as noted by the type field “:w”). The origin of the region is at the second word of *r4*, denoted by *r4.1*. Each row of the region has 8 data elements (words) that are 2 data elements (words) apart. The distance between two rows is 16 words. Note that the region shown is for illustration purpose only. It does not represent a typical usage model nor a performance optimized mode.



Figure 11-11. An example of a register region ($r4.1<16;8,2>:w$) with 16 elements

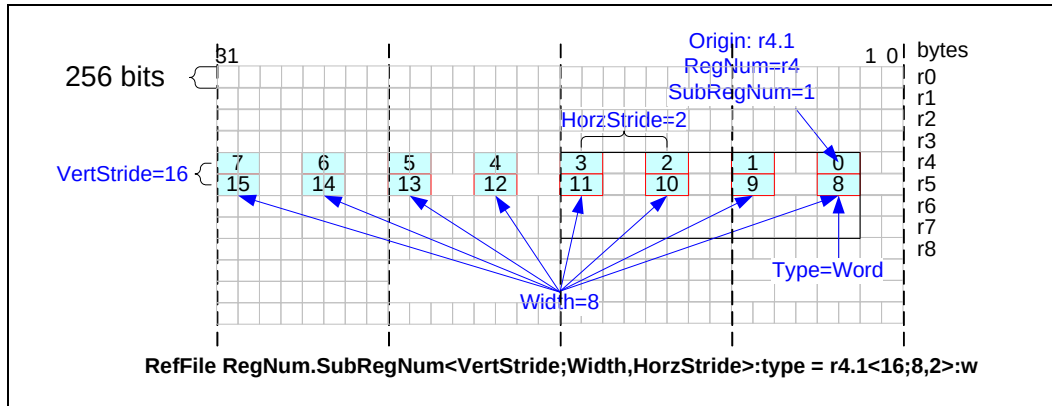
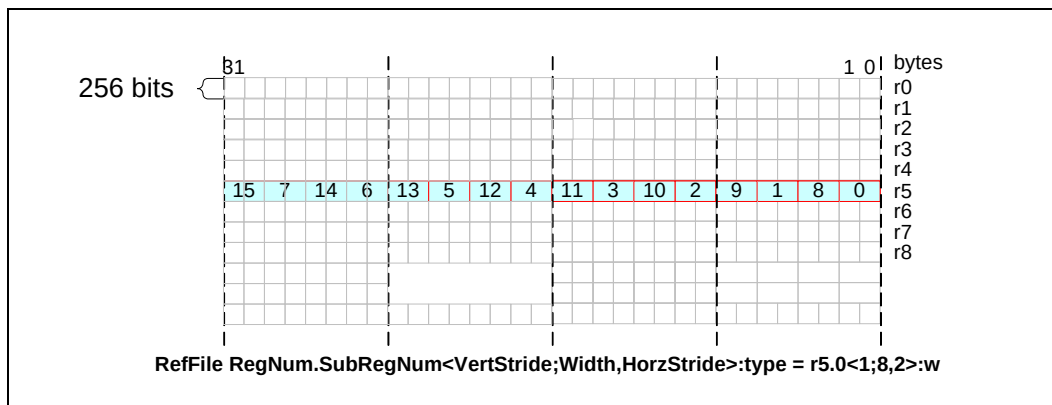


Figure 11-12 shows another example where the rows are interleaved. The region, having word data elements, starts at location $r5.0:w$. HorzStride, the distance within a row, is 2 words. So the second element (channel number 1) is at location $5.2:w$. And there are 8 elements per row. VertStride, the distance between two rows, is only 1 word, which is less than HorzStride. Therefore, the first element of the second row (channel number 8) is at $r5.1:w$, just next to channel number 0. It is clear from the picture that the two rows are interleaved.

By varying the region parameters, reader may construct other configurations. The next section provides more details on the region-based register addressing. However, there are restrictions imposed by hardware implementation, which can be found in the later sections of this chapter.

Figure 11-12. A 16-element register region with interleaved rows ($r5.0<1;8,2>:w$)



Without considering the source channel swizzle and destination register region description, the above row-major-order region description provides the data assignment to each execution channel. The following pseudo code computes the addresses of data elements assigned to execution channels for a special case when the destination register is aligned to 256-bit register boundary.



```

// Input:      Type: ub | b | uw | w | ud | d | f | v
//            RegNum: In unit of 256-bit register
//            SubRegNum: In unit of data element size
//            ExecSize, Width, VertStride, HorzStride: In unit of data elements
// Output: Address[0:ExecSize-1] for execution channels

int ElementSize = (Type=="b"||Type=="ub") ? 1 : (Type=="w"||Type=="uw") ? 2 : 4;
int Height = ExecSize / Width;
int Channel = 0;
int RowBase = RegNum<<5 + SubRegNum * ElementSize;
for (int y=0; y<Height; y++) {
    int Offset = RowBase;
    for (int x=0; x<Width; x++) {
        Address [Channel++] = Offset;
        Offset += HorzStride*ElementSize;
    }
    RowBase += VertStride * ElementSize;
}

```

As *HorzStride* and *VertStride* are specified independently (note that *VertStride* might be smaller than or equal to *HorzStride*), the region may take various shapes from a replicated scalar, a replicated vector, a vector of replicated scalars, a sliding window, to a non-overlapped 2D array.

A region-based description of a destination operand can take the following simplified format

RegFile RegNum.SubRegNum<HorzStride>:type

The destination operand is only allowed to have a 1 dimensional region. The Register Region Origin and Type are the same as for a source operand. The total number of elements is given by *ExecSize*. However, only *HorzStride* is required to describe the 1D region, not *VertStride* and *Width*.

As a source register region may across multiple physical GRF register, an instruction with such source operands may take more than two execution cycles to gather source data elements for execution. The destination register region is restricted to be within a physical GRF register. In other words, destination scatter writes over multiple physical registers are not supported.

11.3.7 Region Addressing Modes

There are two different register addressing modes: Direct register addressing and register-indirect register addressing. Depending on the register region description, the register-indirect register addressing mode can be further divided into three usages: 1x1 index region where only the origin of register region is provided by the address register, Vx1 index region where the offset of each row of the register region is provided by an address register, VxH index region where the offset of each data element is provided by an address register.



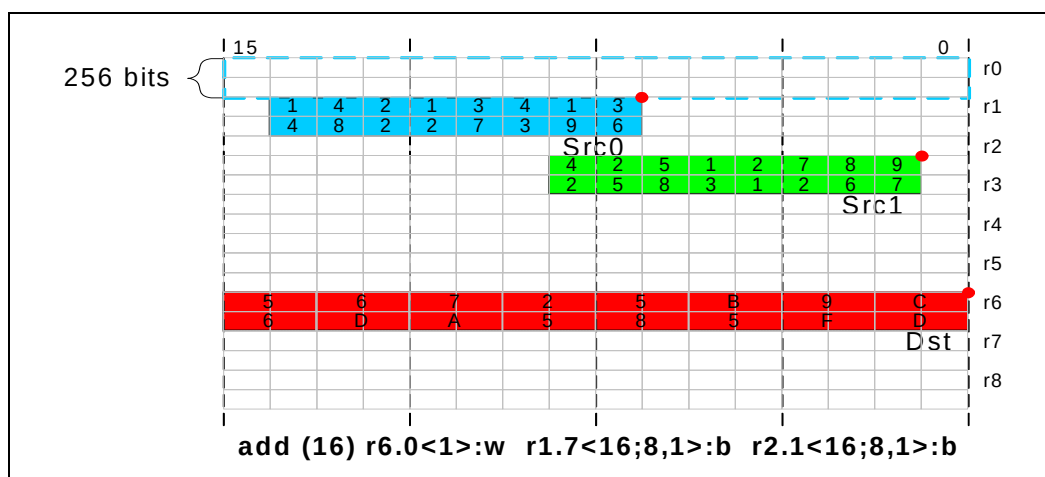
11.3.7.1 Direct Register Addressing

In this mode, all register region parameters are specified for an operand using fields in the instruction word.

Figure 11-13 and Figure 11-14 are two examples of direct register addressing.

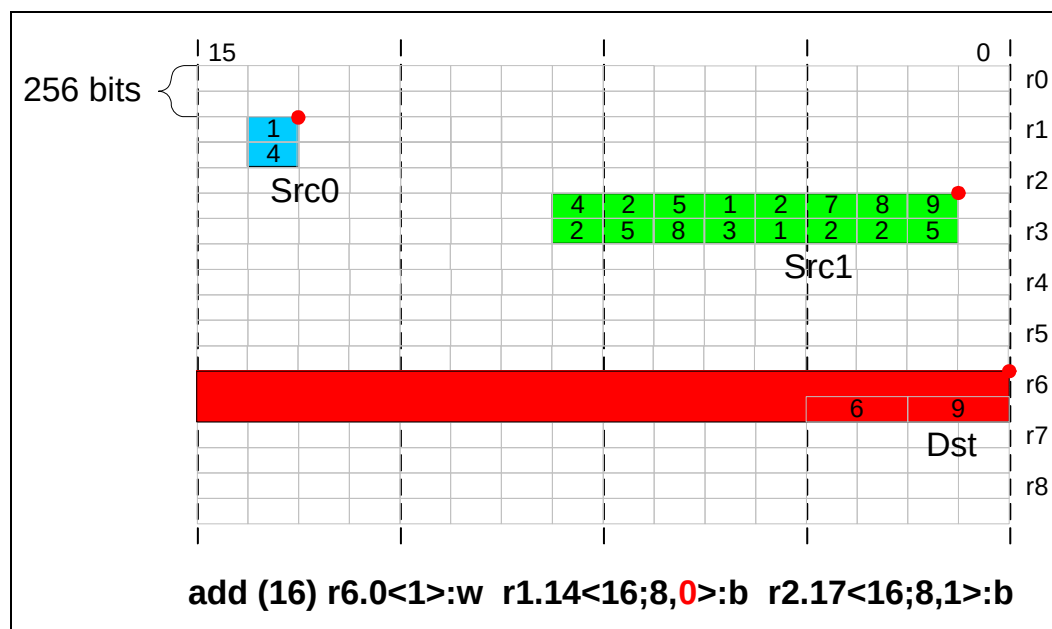
For the example in Figure 11-13, all operands are 2D rectangular regions having the same size of 16 data elements. The two source operands, *Src0* and *Src1*, have 16 bytes. The destination operand, *Dst*, has 16 words. There are 8 elements in a row for *Src0* and *Src1*. The vertical stride of 16 bytes for *Src0* and *Src1* indicates that the first element and the 9'th element are 16 bytes apart in the register file. Note that *Src0* falls into the 256-bit physical GRF register starting at r1.0, but *Src1* crosses the 256-bit physical GRF register boundary between r2 and r3. The numbers in the shaded regions are the values of the data elements. Observing the upper right corners of the source/destination regions (first data element), we have $C = 3 + 9$.

Figure 11-13. A region description example in direct register addressing



For the example in Figure 11-14, the sizes of areas of *Src0* and *Src1* are the same, but *Src0* contains a vector of replicated scalars. With $\text{HorzStride} = 0$ and $\text{Width} = 8$, the first row of 8 elements in *Src0* is a replication of the byte at r1.14. Comparing ExecSize of 16 to Width of 8 indicates that there is a second row of 8 elements in *Src0*. With $\text{VertStride} = 16$, the second row in *Src0* is a replication of the byte at r1.20 ($20 = 14 + 16$). Effectively, the 16 data elements of *Src0* are $\{1, 1, 1, 1, 1, 1, 1, 1, 4, 4, 4, 4, 4, 4, 4, 4\}$.

Figure 11-14. A region description example in direct register addressing with <src0> as a vector of replicated scalars



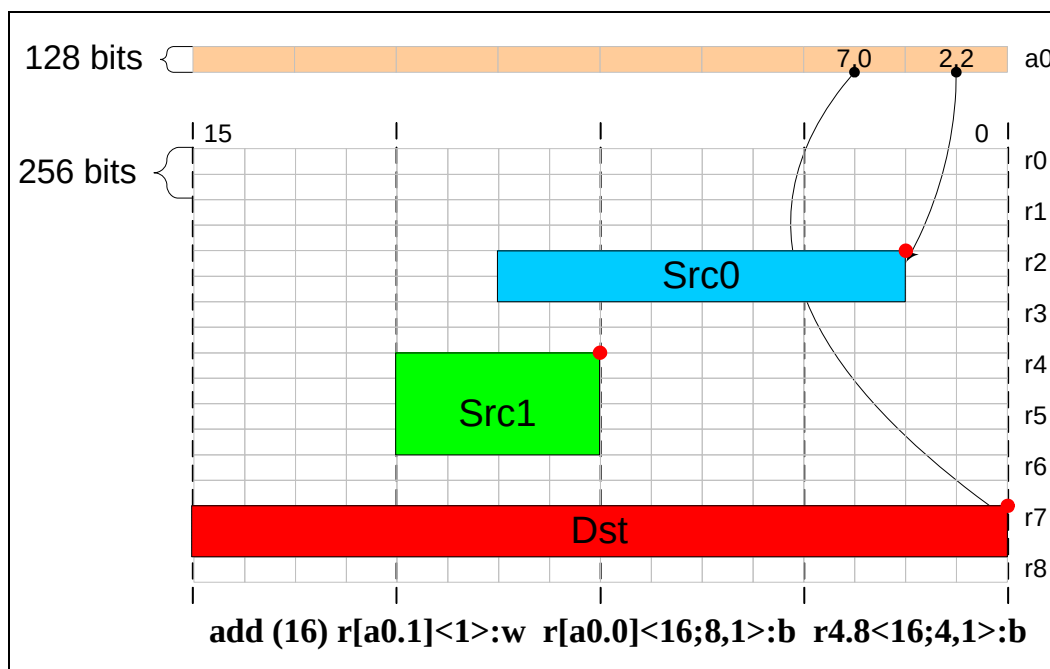
11.3.7.2 Register-indirect Register Addressing with a 1x1 Index Region

In the register-indirect register addressing mode with 1x1 index region, the region origin is provided by the content of the address register, the rest of region parameters are provided by the fields in the instruction word.

Figure 11-15 depicts an example for this addressing mode. For example, the present of full region description <16;8,1> for Src0 indicates that only the origin of the region is provided by the address register a0.0.



Figure 11-15. An example illustrating register-indirect register addressing mode with a 1x1 index region

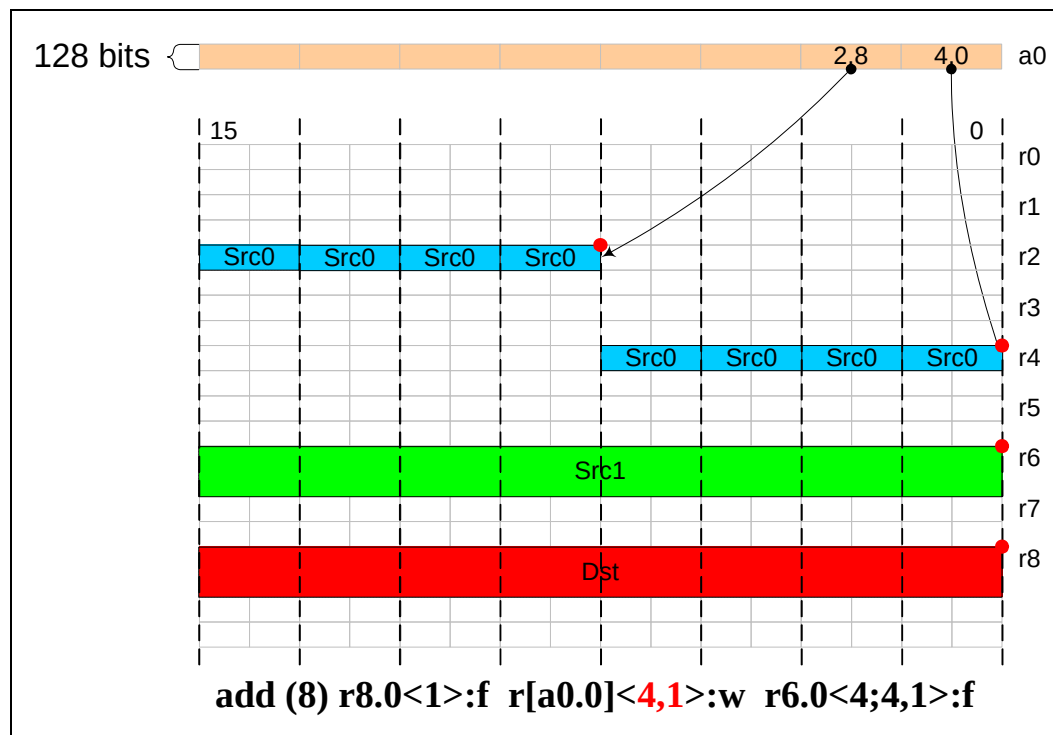


11.3.7.3 Register-indirect Register Addressing with a Vx1 Index Region

In the register-indirect register addressing mode with Vx1 index region, horizontal dimension is described by the fields in the instruction word and the vertical dimension is described by an address register region. Specifically, the origin of each row of the data region is provided by the contents of an address register region. The rows are described by the width and the horizontal stride. The first address register is provided, the following contiguous address registers are for the following rows. The total number of address registers used is inferred by parameters *ExecSize* and *Width*.

An example is provided in Figure 11-16. The assembly syntax notion of a register region without vertical stride, $\langle 4, 1 \rangle$, corresponding to the special encoding of vertical stride of 0xF in the instruction word, indicates the VxH or Vx1 mode of indirect register addressing. In this case, the origin for each row of Src0 is provided by the address register. As $\text{ExecSize}/\text{Width} = 2$, there are two address registers a0.0 and a0.1, each pointing to a row of 4 data elements.

Figure 11-16. An example illustrating register-indirect-register addressing mode with a Vx1 index region (Src0)



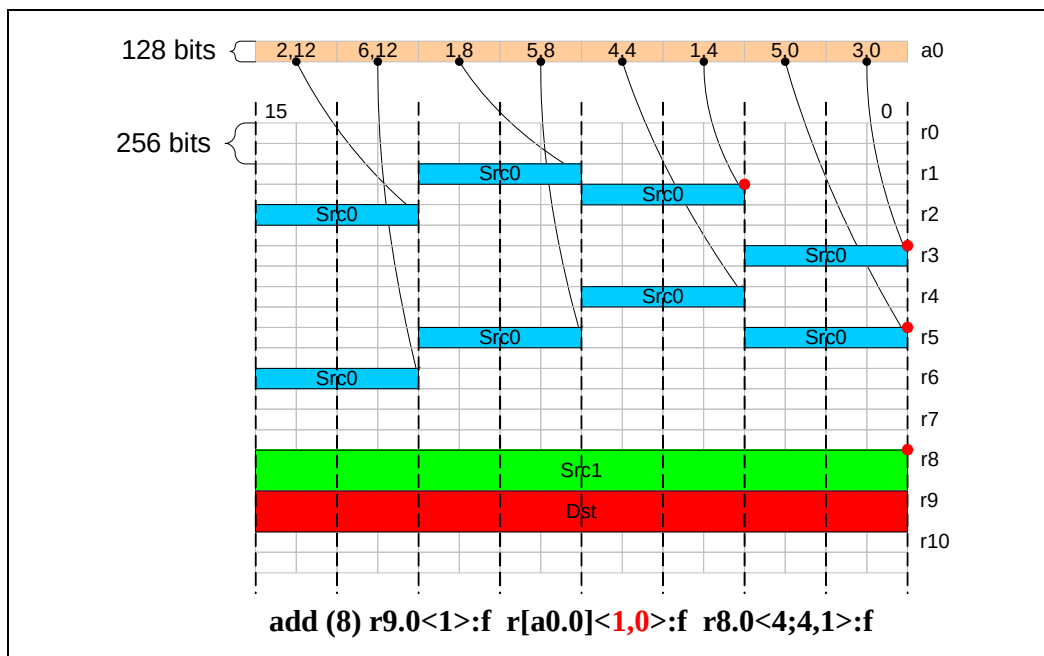
11.3.7.4 Register-indirect Register Addressing with a VxH Index Region

In the register-indirect register addressing mode with VxH index region, the position of each data element is provided by the contexts in an address register region. This mode has the identical syntax as the Vx1 index region mode, and in fact, can be viewed as a special case of the Vx1 mode. When *Width* of the region is 1, the number of address registers used equals *ExecSize*.

An example is provided in Figure 11-17. The absent of vertical stride in the region description <1,0> with width = 1 indicates that the origin for each row of 1 data element of Src0 is provided by the address register. As $\text{ExecSize}/\text{Width} = 8$, there are 8 address registers from a0.0 to a0.7, each pointing to a single data elements.



Figure 11-17. An example illustrating register-indirect register addressing mode with a VxH index region (Src0).



11.3.8 Access Modes

There are two basic GEN4 register access modes controlled by a single bit instruction subfield called Access Mode.

- 16-byte Aligned Access Mode (**align16**): In this mode, the origins of all operands (sources and destination), whether it is by direct addressing or register-indirect addressing, are 16-byte aligned. For example a row in the region description starts at 16-byte aligned and the width the row must be 4 and the 4 data elements within a row must span 16-bytes. In this access mode (and with other restrictions put forward later), full-channel swizzle for both source operands and per-channel mask for destination operand are supported on a 4-component basis. In other words, the control and setting of full source swizzle and destination mask are repeated for every 4 components up to total of *ExecSize* channels.
 - The **align16** access mode can be used for AOS operations. See examples provided in the Primary Usage Model section for SIMD4x2 and SIMD4x1 modes of operation to support 3D API Vertex Shader and Geometric Shader execution.
- 1-byte Aligned Access Mode (**align1**): In this mode, the origins of all operands may be aligned to their data type and could be 1-byte if the operand is of byte type. In this access mode, full region register descriptions are supported, however, source swizzle or destination mask are not supported.
 - The **align1** access mode can be used for SOA operations. See examples provided in the Primary Usage Model section for SIMD8 and SIMD16 modes of operation to support 3D API Pixel Shader. Many media applications also operate well in **align1** access mode.



11.3.9 Execution Data Type

GEN4 architecture supports instructions with mixed data types. The internal hardware computation is performed using the execution data type. When an instruction has only one source operand or has two source operands of the same data type, the execution data type is the same as that of the source. When an instruction has two source operands of different types, an execution data type is determined and one of the source operands will be converted to the execution type before the computation is performed. The execution type is independent of the destination data type. When the destination data type is different from the execution data type, a type conversion is performed on the intermediate compute results before the results are written into the destination register. Such a destination type conversion doesn't apply to accumulator registers, implicitly or explicitly. Therefore, accumulator type cannot differ from the execution data type.

Determination of the execution data type for two sources of different data types obeys the following rules

- If any source is a float, the execution data type is float (F)
- Else if any source is a dword, the execution data type is signed dword integer (D)
- Else execution data type is signed word integer (W)

Note that when the execution data type is an integer, it is always a signed integer. This doesn't affect the functional correctness of the instruction as extra precisions are carried within the hardware, including the accumulator. See Instruction Reference Chapter for detailed description for each instruction.

11.3.10 Register Region Restrictions

The following register region rules apply to the GEN4 implementation. Rules and restrictions for compressed instructions can be found in the Instruction Compression section.

1. *ExecSize* must be equal to or less than the maximum execution size supported for the operand type. As shown in Table 11-29, the maximum execution size is determined by the largest operand type of the sources and destination of the instruction.
2. The mapping of data elements within the region of a source operand is in row-major order and is determined by the region description of the source operand, plus *ExecSize* and destination region description.
3. *ExecSize* must be equal to or greater than *Width*.
4. If *ExecSize* = *Width* and *HorzStride* ≠ 0, *VertStride* must be set to *Width* * *HorzStride*.
5. If *ExecSize* = *Width* but *HorzStride* = 0, there is no restriction on *VertStride*.
6. If *Width* = 1, *HorzStride* must be 0 regardless of the values of *ExecSize* and *VertStride*.
7. If *ExecSize* = *Width* = 1, both *VertStride* and *HorzStride* must be set to zero.
8. If *VertStride* = *HorzStride* = 0, *Width* must be 1 regardless of the value of *ExecSize*.



9. Destination region cannot cross the 256-bit register boundary.
 - 9.1. Exception to this rule is for a compressed instruction where the destination region covers exactly **two adjacent** 256-bit physical registers.
10. Destination region alignment rule.
 - 10.1. With the exception on 'raw move' described in rule #10.3 and the exception on byte destination in rule #10.5, all destination data elements must be aligned to the size for the execution data type of the instruction. For example, if one of the source operands is in dword mode (a float, a signed or unsigned dword integer), the execution data type will be either float or signed dword integer. Therefore, the destination data elements must be dword aligned. This rule has the following two implications:
 - 10.1.1. The destination sub-register must be aligned to the size of the execution data type.
 - 10.1.2. If *ExecSize* is greater than 1, *dst.HorzStride* sizeof(dst.Type)* must be equal to or greater than the size of the execution data type.
 - 10.2. If *ExecSize* is 1, *dst.HorzStride* must not be 0. Note that this is relaxed from rule 10.1.2. Also note that this rule for destination horizontal stride is different from that for source as stated in rule #7.
 - 10.3. When destination type is byte (UB or B), only a 'raw move' using **mov** instruction supports packed byte destination register region: *dst.HorzStride* = 1 and *dst.type* = (UB or B). This packed byte destination region is not allowed for any other instructions, including a 'raw move' using **sel** instruction. This is because **sel** instruction is based on word or dword wide execution channels.
 - 10.4. When an instruction has a source region that spans two physical registers, one of the followings must be true:
 - 10.4.1. Destination region is entirely contained in the lower oword of a physical register,
 - 10.4.2. Destination region is entirely contained in the upper oword of a physical register, or
 - 10.4.3. Destination elements are evenly split between the two owords of a physical register.
 - 10.5. Relaxed alignment rule for byte destination. When destination type is byte (UB or B), destination data elements can be either aligned to the lowest byte or the second lowest byte of the execution channel. For example, if one of the source operands is in word mode (a signed or unsigned word integer), the execution data type will be signed word integer. In this case, the destination data bytes can be either all in the even byte locations or all in the odd byte locations. This rule has the following two implications:
 - 10.5.1. The destination sub-register must be either aligned to the size of the execution data type or one byte higher off the execution data type.
 - 10.5.2. If *ExecSize* is greater than 1, *dst.HorzStride* sizeof(dst.Type)* must be equal to or greater than the size of the execution data type. This is the same as that in #10.1.2.
11. In Align1 access mode, a source region must be within **two adjacent** 256-bit physical registers.
 - 11.1. It is further restricted that a single row cannot cross physical register boundary.
 - 11.2. It is further restricted that for source 1 only, when crossing physical register boundary, the vertical stride must be equivalent to 32-byte.



12. In Align16 access mode, a source region must be within **one** 256-bit physical registers.
13. Rules on register-indirect register access:
 - 13.1. Source 0, source 1 and destination can be indexed. Support for indirect register access for source operands is device dependent. See Table 11-30 for details.
 - 13.2. If supported, an indexed source 1 can only have a 1x1 indexed register region – only single index mode is allowed for a source 1.
 - 13.3. An indexed destination can only have a 1x1 indexed register region – only single index mode is allowed for a destination operand.
 - 13.4. Data elements referenced by a single index within a source region cannot cross 256-bit physical register boundary. This applies to register region with a single index or with multiple indices.
 - 13.4.1. A register region with multiple indices may access multiple physical registers as long as data elements associated with each index follow the above-mentioned rule. For example instruction “mov (16) r0.0:uw r[a0.0]<2,2>:uw” is allowed. This is a source gathering instruction whereas the source operand may potentially touch 8 different physical GRF registers.
 - 13.5. VxH index regioning using 2 index registers cannot start from a0.6. a0.6 can still be used as part of the VxH index regioning as long as the start index register is not itself, a0.6 can also be used in Vx1 index regioning by itself.
 - 13.6. **[DevCL]** When indirect addressing is used on a source or destination of an instruction, the following combination for the particular source or destination is not allowed.
 - 13.6.1. using indirect addressing
 - 13.6.2. the index register used is a0.7
 - 13.6.3. the index immediate is negative
 - 13.6.4. the datatype is byte
14. **Implementation Restriction:** Any non-compressed instruction with 2row regioning source(s) cannot be a jump/branch target or follows directly after a conditional branch instruction or a predicated jmpri instruction, unless the 2 rows are previously written from FPU, not from outside of EU. The workaround would be either break the instruction into 2 1row instruction, or insert a NOP.

Implementation Restriction: The relaxed alignment rule for byte destination (#10.5) is not supported.



Table 11-29. Execution size in device hardware

Device	Native GEN4 Instructions			Compressed GEN4 Instructions		
	DWORD	WORD	BYTE	DWORD	WORD	BYTE
[DevBW]	8	16	16	16	32	32
[DevCL]	8	16	16	16	32	32

Table 11-30. Indirect source addressing support available in device hardware

Device	Indirect Source 0	Indirect Source 1
[DevBW]	Yes	No
[DevCL]	Yes	Yes

Note: It is expected that some of the restrictions may be relaxed in future implementations of the GEN4 architecture.

11.3.10.1 Examples

Some examples are provided here to illustrate the cases when the register region restrictions are violated. It is provided as informative material to help understanding these restrictions.

Example 1: The following instructions are illegal as they violate rule #10.1, as the destination is not aligned to the execution data type.

```

mov (1) r0.1<1>:b r2.0:w // dst.SubReg must be even
mov (2) r0.0<1>:b r2.0:w // dst.HorzStride must be >= 2
mov (2) r0.0<2>:b r2.0:d // dst.HorzStride must be >= 4
mov (2) r0.0<2>:b r2.0:f // dst.HorzStride must be >= 4
mov (1) r0.2<1>:b r2.0:d // dst.SubReg must be dword aligned

```

Example 2: This instruction is illegal as it violates rule #10.1.2, as when *ExecSize* = 1, *dst.HorzStride* cannot be zero.

```

mov(1) r0.0<0>:b r0.0:d

```

Example 3: This instruction is illegal as it violates rule #11.1, as the source contains one row of 2 elements that spans physical register r2 and r3.

```

mov (2) r1.0:d r2.7<2;2;1>:d

```



Example 4: This instruction is illegal as it violates rule #14, as the jump target of the jmp instruction is using 2-row regioning.

```
send r4 null m1 0x0122005b <- write into both r4 and r5
...
(f0.0)jmp L0
...
L0: mov (8) r10 r4<8;4,1> <- source0 uses both r4 and r5
```

Example 5: This instruction is illegal as it violates rule #14, as the instruction immediately after the predicated jmp instruction is using 2-row regioning.

```
send r4 null m1 0x0122005b <- write into both r4 and r5
...
(f0.0)jmp L0
mov (8) r10 r4<8;4,1> <- source0 uses both r4 and r5
```

Example 6: This instruction is illegal as it violates rule #14, as the jump target of the iff instruction is using 2-row regioning.

```
send r4 null m1 0x0122005b <- write into both r4 and r5
...
(f0.0)iff L0
...
endif
L0: mov (8) r10 r4<8;4,1> <- source0 uses both r4 and r5
```

Example 7: This instruction is illegal as it violates rule #14, as the instruction immediately follows the if instruction is using 2-row regioning.

```
send r4 null m1 0x0122005b <- write into both r4 and r5
...
(f0.0)if L0
mov (8) r10 r4<8;4,1> <- source0 uses both r4 and r5
...
L0: else
```




11.3.10.2 Different Raw Moves

Definition of Raw Move: Raw move is an operation that moves data elements from source to destination without altering the bit fields of the data elements. It must use one of the move instructions such as *mov*, *sel*, *movi*. Arithmetic instruction that results in unaltered bit fields of the data elements are not treated as raw move. A raw move may subject to the execution channel enables by using prediction or being present in multi-channel branch code segment. Type conversion by definition cannot be used in a raw move. Therefore, source and destination operands must be of the identical data type. For example, if both source and destination are float, for an arithmetic instruction, denorm will be flushed to zero. However, for a raw move, denorm will be preserved.

Definition of Byte Raw Move: As the minimal execution channel type is word, when the destination stride is greater than one byte, each data element of the source can be mapped to one execution channel. This is referred to as Byte Raw Move. Byte Raw Move allows the destination to be byte aligned, in other words, allowing the destination to not align to execution channels. Byte Raw move subjects to execution channel enables.

Definition of Packed-Byte Raw Move: As the minimal execution channel type is word, when the destination stride is equal to one byte, two data elements of the source are mapped to one execution channel. This is referred to as Packed-Byte Raw Move. Packed-Byte Raw Move allows the destination to be byte aligned, in other words, allowing the destination to not align to execution channels. However, as the data elements are not mapped to execution channels, undefined results may occur if Packed-Byte Raw Move is mixed with execution channel enables. So for Packed-Byte Raw Move, **NoMask** should be used when there are un-enabled channels within the execution size of the instruction.

11.3.11 Destination Operand Description

11.3.11.1 Destination Region Parameters

Based on the above restrictions, a subset of register region parameters are sufficient to describe the destination operand:

- Destination Register Origin
 - Destination Register Number and Destination Subregister Number for direct register addressing mode
 - A Scalar Destination Register Index for register-indirect-register addressing mode
- Destination Register 'Region' – Note that destination register region does not have full region description parameters
 - Destination Horizontal Stride



11.4 SIMD Execution Control

11.4.1 Predication

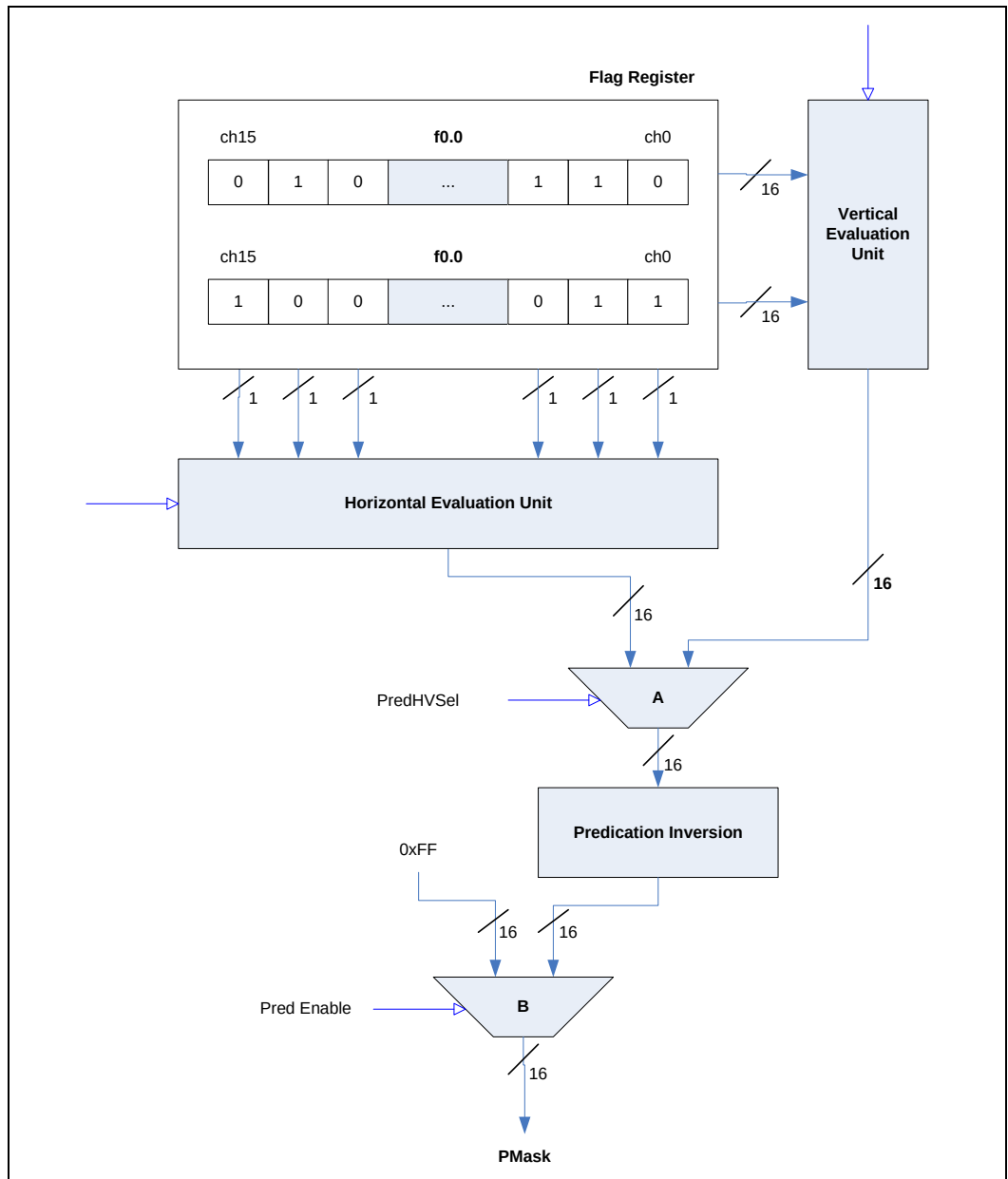
Predication is the conditional SIMD channel selection for execution on a per instruction basis. It is an efficient way of dynamic SIMD channel enabling without paying branch instruction overhead. When predication is enabled for an instruction, a Predicate Mask (PMask), which contains 16-bit channel enables, is generated internally in EU. Note that PMask is not a software visible register. It is provided here to explain how SIMD execution control works. PMask generation is based on the Predication Control (*PredCtrl*) field, Predication Inversion (*PredInv*) field and the flag source register in the instruction word. See Instruction Summary chapter for definition of these fields.

Figure 11-18 shows the block diagram of the hardware logic to generate PMask. PMask is generated based on combinatory logic operation of the bits in the flag register. Instruction field *PredCtrl* controls the horizontal evaluation unit and vertical evaluation unit. MUX A in the figure selects whether horizontally-evaluated results or vertically-evaluated results are sent to the Predication Inversion unit. The *PredInv* field controls the Prediction Inversion unit. Either one 16-bit flag subregister or the whole flag register may be selected to generate the PMask depending on the predication control modes. MUX B indicates that predication can be enabled and disabled. Predication can be grouped into the following three categories. Predication functionality also depends on the Access Mode of the instruction.

- No predication: Of course, predication can be disabled. This is the most commonly used case.
- Predication with horizontal combination: the predicate mask is generated based on combinatory logic operation of bits within a selected flag subregister.
- Predication with vertical combination: the predicate mask is generated based on combinatory logic operation of bits across flag multiple subregisters.



Figure 11-18. Generation of predication mask



11.4.2 No Predication

When PredCtrl field of a given instruction is set to 0 ("no predication"), it indicates that no predication is applied to this instruction. Effectively, the resulting PMask is all 1's. This is shown by the 2:1 multiplexer B controlled by the Pred Enable signal in Figure 11-18. Where predication is not enabled for an instruction, multiplexer B is selected to output 0xFF to PMask.



11.4.3 Predication with Horizontal Combination

Predication with horizontal combination inputs the 16 bits of a single flag subregister (f0.0:uw or f0.1:uw) and passes them through combinatory logic of the Horizontal Evaluation unit to create PMask.

The simplest combination is 'no combination' – the same 16 bits from selected flag subregister are output to MUX A. In this case, a bit in the selected flag subregister controls the conditional execution of the corresponding execution channel. Let the selected flag subregister be denoted as f0.#, the following pseudo code describes the predicate mask generation for predication with sequential flag channel mapping.

```

If (PredCtrl == "Sequential flag channel mapping") {
    For (ch=0; ch<16; ch++)
        PMask[ch] = (PredInv == TRUE) ? ~f0.#[ch] : f0.#[ch];
}

```

More complex horizontal evaluation is based on channel grouping. A group of adjacent channels (bits from flag subregister) are evaluated together and a single bit is replicated to the group. The size of groups is in power of 2. The supported combination depends on the Access Mode of an instruction.

In **Align16** access mode, horizontal combination is based on 4-channel groups.

- Channel replication: PredCtrl of '.x', '.y', '.z' and '.w' select a single channel from each 4-channel group and replicate it as the output for the group. For example, PredCtrl = '.x' means that channel 0 in each group is replicated.
- OR combination: PredCtrl of '.any4h' means that if **any** of the channel in a group is enabled, outputs for the 4 channels in the group are all enabled.
- AND combination: PredCtrl of '.all4h' means that only when **all** of the channels in a group are enabled, the output for the group is enabled.

These combinations in **Align16** mode can be described by the following pseudo-code.

```

If (Access Mode == Align16) {
    For (ch = 0; ch < 16; ch += 4)
        Switch (PredCtrl) {
            Case '.x': bTmp = f0.#[ch]; break;
            Case '.y': bTmp = f0.#[ch+1]; break;
            Case '.z': bTmp = f0.#[ch+2]; break;
            Case '.w': bTmp = f0.#[ch+3]; break;
            Case '.any4h': bTmp = f0.#[ch] | f0.#[ch+1] | f0.#[ch+2] | f0.#[ch+3]; break;
            Case '.all4h': bTmp = f0.#[ch] & f0.#[ch+1] & f0.#[ch+2] & f0.#[ch+3]; break;
        }
        bTmp = (PredInv == TRUE) ? ~bTmp : bTmp;
        PMask[ch] = PMask[ch+1] = PMask[ch+2] = PMask[ch+3] = bTmp;
    }
}

```

In **Align1** access mode, horizontal combination is based on AND combination '.any#h' and OR combination '.all#h' on channel groups with various sizes, where # is the number of channels in a group ranging from 2 to 16. This is described by the following pseudo-code.



```
If (Access Mode == Align1) {  
    Switch (PredCtrl) {  
        Case '.any2h': groupSize = 2; <op> = '!'; break;  
        Case '.all2h': groupSize = 2; <op> = '&'; break;  
        Case '.any4h': groupSize = 4; <op> = '!'; break;  
        Case '.all4h': groupSize = 4; <op> = '&'; break;  
        Case '.any8h': groupSize = 8; <op> = '!'; break;  
        Case '.all8h': groupSize = 8; <op> = '&'; break;  
        Case '.any16h': groupSize = 16; <op> = '!'; break;  
        Case '.all16h': groupSize = 16; <op> = '&'; break;  
    }  
    For (ch = 0; ch < 16; ch += groupSize) {  
        For (inc = 0, bTmp = FALSE; inc < groupSize; inc ++)  
            bTmp = bTmp <op> f0.#[ch+inc];  
        For (inc = 0; inc < groupSize; inc ++)  
            PMask[ch+inc] = bTmp;  
    }  
}
```

11.4.4 Predication with Vertical Combination

Predication with vertical combination uses both flag subregister as inputs. The AND or OR combination is across the subregisters on a channel by channel basis. This is shown by the following pseudo-code.

```
If (Access Mode == Align1) {  
    For (ch = 0; ch < 16; ch ++){  
        If (PredCtrl == 'any2v')  
            PMask[ch] = f0.0[ch] | f0.1[ch]  
        Else If (PredCtrl == 'any2h')  
            PMask[ch] = f0.0[ch] & f0.1[ch]  
    }  
}
```



11.5 Instruction Compression

11.5.1 Motivation and Expected Usage

When pixel processing is performed in the channel serial mode, each 3D graphics API Pixel Shader instruction (SIMD with 4 components) is normally translated into 4 GEN4 instructions with one for each component. It may result in fewer GEN4 instructions, when some components are not output. To process 16 pixels in parallel using the SIMD8 basic GEN4 instruction set adds another factor of 2. So a 32-line Pixel Shader program may become 256-line long. In order to reduce the burden to the instruction cache, GEN4 architecture employs a simple instruction compression technique available on a subset of instructions. For the Pixel Shader case, one GEN4 instruction in the instruction stream is assigned to process 16 pixels in parallel. When such kind of instruction is encountered, the execution unit automatically creates two instructions before issuing them into the instruction decoder.

For example, a compressed instruction may be in the following form:

```
add (16) r4.0<8;8,1>:d r2.0<8;8,1>:d r0.0<8;8,1>:d {Compr}
```

It will be split by hardware internally into two native GEN4 instructions as:

```
add (8) r4.0<8;8,1>:d r2.0<8;8,1>:d r0.0<8;8,1>:d
```

```
add (8) r5.0<8;8,1>:d r3.0<8;8,1>:d r1.0<8;8,1>:d {SecHalf}
```

The instruction compress technique is expected to be used for Pixel Shader program that runs in channel serial mode for 16 pixels in parallel. It is not clear whether 3D graphics API Vertex and Geometry Shader programs can take advantage of this hardware capability.

The instruction compress technique is also expected to be used for media kernels, where it is quite common to have an operation on a block of data. Instruction compression significantly reduces the burden to the instruction caches. It also allows media kernels to utilize both accumulator registers as implied operands.



11.5.2 Hardware Behavior

Upon encounter an instruction with the Compression Control field set to 10b (Compr), EU hardware treats it as a compressed instruction and converts it into two instructions before sending them down for instruction decode and dispatch. The two decompressed instructions share majority of instruction subfields with the original compressed instruction. The differences are fully described here.

When destination register is an MRF register, a special instruction compression control, **Compr4**, can be used. **Compr4** is the combined condition when instruction compression field is set to **Compr** and the destination register is a MRF register with RegNum[7] set to 1. With **Compr4**, the second decompressed instruction will have MRF register incremented by 4 registers.

- The first generated instruction
 - o Compression Control field is reset to 00b (normal)
 - o ExecSize is reduced by half
 - o RegNum[7] of the destination MRF register is reset to 0 if the compressed instruction is **Compr4**.
 - o The rests of the instruction word are unchanged
- The second generated instruction
 - o Compression Control field is set to 01b (**SecHalf**)
 - If ExecSize of the compressed instruction less than 32, the SecHalf flag determines that the generated instruction (e.g. with ExecSize of 8) uses the 8 MSB of the mask and flag registers. And if a flag register is a conditional destination operand, only the higher 8 bits of a flag register are updated.
 - If ExecSize of the compressed instruction is 32, the whole 16 bits of the mask and flag registers apply to the generated instruction (with ExecSize of 16). And if a flag register is a conditional destination operand, the whole 16 bits of a flag register may be updated. Effective, SecHalf doesn't affect the generation of execution mask in this case.
 - o Execution size is reduced by half
 - o The breakpoint field is reset (regardless of the value in the original instruction)
 - o For a source operand
 - Subject to the following exceptions, for a direct source register, the LSB of the operand's RegNum is set to 1. This effectively moves the register origin by a whole GRF physical register.
 - This rule applies to an accumulator register as an explicit source operand (and it must be acc0 in the compressed instruction). The LSB of the operand's RegNum is set to 1, effectively changing it to acc1. This rule is also consistent with that for an implicit accumulator source.



- Subject to the following exceptions, for an indirect source register, bit 1 of the address register is set to 1. This effectively moves the address register by one sub-field for the second instruction.
 - The subregister field of the address register must be even aligned, and must be in 1x1 mode.
- For an implicit accumulator source operand, the **SecHalf** flag determines that it is acc1.
 - This rule applies to a compressed instruction regardless of its ExecSize.
- If a source operand is a scalar (signified by a region with HStride = VStride = 0, also including an immediate operand regardless of it being a scalar or a vector), there is no change.
 - This rule only applies to a direct source but not a register-indirect source. If the intention is to use a single indexed scalar value for a compressed instruction, software must program two-adjacent address subregisters with the same value.
- If ExecSize of the compressed instruction is 16, the destination has a size of dword and stride of 1 (effectively with a 4-byte stride), and the source operand type is of word size with a horizontal stride of 1, the MSB of the operand's SubRegNum is set to 1. This effectively moves the register origin by half of a GRF physical register.
 - This rule is specifically for mixed data type operation (e.g. Pixel Shader usage).
- o For destination operand
 - For a direct destination GRF register, the LSB of the destination operand's RegNum is **set** to 1.
 - This rule applies to an accumulator register as an explicit destination (and it must be acc0 in the compressed instruction). The LSB of the operand's RegNum is set to 1, effectively changing it to acc1. This rule is consistent with that for an implicit accumulator destination.
 - For a destination MRF register (directly addressed only), the destination RegNum is **incremented** by 1, if the compressed instruction has **Compr**.
 - For a destination MRF register (directly addressed only), the destination RegNum[7] is reset to 0 and then RegNum is **incremented** by 4, if the compressed instruction has **Compr4**.
 - For an indirect destination GRF register, bit 1 of the address register is set to 1. This effectively moves the address register by a sub-field for the second instruction.
 - The subregister field of the address register must be even aligned, and must be in 1x1 mode.
 - For an implicit accumulator destination operand, the **SecHalf** flag determines that it is acc1.



As all 16 entries of the mask stack are pushed or popped together each time a mask stack is updated and the mask stack is implemented by counters, it is essential to not mix the execution channel select with the use of **SecHalf** when nesting branch/loop instruction blocks. In other words, mask values that are pushed onto a stack must all have been generated from instructions using the same Compression Control value (such as normal, **SecHalf** and **Compr**).

The Thread Control field is unchanged for both generated (decompressed) instructions. If this field is set to **Switch** in the compressed instruction, both decompressed instructions will cause forced thread switch.

Compr4 is not supported.

[DevBW, DevCL] Errata: If ExecSize is 16, the implicit accumulator is forced to acc0, even if **SecHalf** compression control is set. Normally, **SecHalf** determines that acc1 is the implicit accumulator. However, this is overruled by ExecSize of 16. This restriction implies that instructions with implicit accumulator as a source (such as mac, mach) cannot be used for a compressed instruction with ExecSize of 32.

[DevBW, DevCL] Errata: When **SecHalf** compression control is used, even with ExecSize of 16, only the second half of the flag register may be used (for example as conditional destination or as prediction source).

11.5.3 Rules and Restrictions

In order to reduce the hardware complexity, the following rules and restrictions apply to the compressed instruction:

- Instruction compression is on a per instruction basis. Compressed instruction and normal instruction can be intermixed in any program.
- A compressed instruction must be in Align1 access mode. Align16 mode instructions cannot be compressed.
- A compressed instruction with indirect addressed operands for any operand (source, destination) is allowed.
 - *Rational:* Commonly used in media kernels.
- Operand Alignment Rule: With the exceptions listed below, a source/destination operand in general should be aligned to **even** 256-bit physical register with a region size equal to two 256-bit physical registers.
 - *Rational:* This allows an insertion of 1 to the LSB of the RegNum to create the second instruction.
- Exception A to the Operand Alignment Rule: for compressed instructions operating on packed words (destination is not dword size with stride of one), an operand with a non-zero subregister field is also allowed, as long as its register number is an **even** number.
 - *Rational:* This allows an insertion of 1 to the LSB of the RegNum field to create the second instruction. And hardware does not modify the subregister field.
- Exception B to the Operand Alignment Rule: a source operand with scalar replication (with both HStride = 0 and VStride = 0) is allowed to have address unaligned to 256-bit.
 - *Rational:* There is no change to this source operand to create the second instruction.



- Exception C to the Operand Alignment Rule: For a compressed instruction with dword destination with stride of one, for a source operand with packed word data elements (type = W or UW, and HStride = 1), the register origin must be aligned to 256-bit physical register (i.e. SubRegNum = 0).
 - If a source type is B or UB, HStride must be either 0 (scalar) or 2 (packed word).
- Exception D to the Operand Alignment Rule: a destination operand to MRF must be aligned to 256-bit physical register (either **even** or **odd**).
 - *Rational*: This allows more efficient use of the rare MRF resource, considering the fact that many messages have a single register message header field.
- ExecSize must be 16 or 32
 - *Rational*: Allows hardware to decrement by 1 to the encoded ExecSize field.
 - *Note*: Most commonly used mode for Pixel Shader is that ExecSize = 16. Most commonly used mode for media kernels is that ExecSize = 32 with packed word operations. There are also some media usage of compressed instruction with ExecSize = 16.
- Instruction compression does not apply to branch instructions such as jmp, send, do, while, wait, etc.
- A compression instruction in general cannot address (read and/or write) ARF registers.
 - Exception to this rule is the explicit or implicit access to the accumulator registers and implicit access to flag registers and mask registers, as well as indirect addressed GRF registers.

11.5.4 Usage Examples

Some examples are provided in this section to help visualizing the effect of compressed instruction on source/destination operands. It is for illustration purpose only; it is not intended as comprehensive usage coverage.

Figure 11-19 shows examples of directly-addressed vector operands in a compressed instruction. In these cases, hardware inserts one to the LSB of the RegNum field of a GRF operand (or increment one to RegNum of a MRF operand) for the second decompressed instruction.

- In Figure 11-19(a), the operand occupies two adjacent rows of register space with the first decompressed instruction operating on the first eight elements and the second decompressed instruction on the second eight elements. It may be a dword source or destination operand of a SIMD16 compressed instruction. The operand register address must be even aligned to a GRF register or register aligned to a MRF register.
- Figure 11-19(b) shows a result when the register address of the dword (source) operand of a SIMD16 compressed instruction is aligned to odd GRF register. As inserting one to RegNum field doesn't change a thing, the same eight elements are repeated for the two decompressed instructions.
- Figure 11-19(c) shows a word/byte source/destination operand in a SIMD16 compressed instruction. RegNum field is an even number; however, SubRegNum here is not zero. So the elements of a decompressed instruction only occupy a portion of a GRF register.

Figure 11-19. Direct addressed vector operands in a compressed instruction

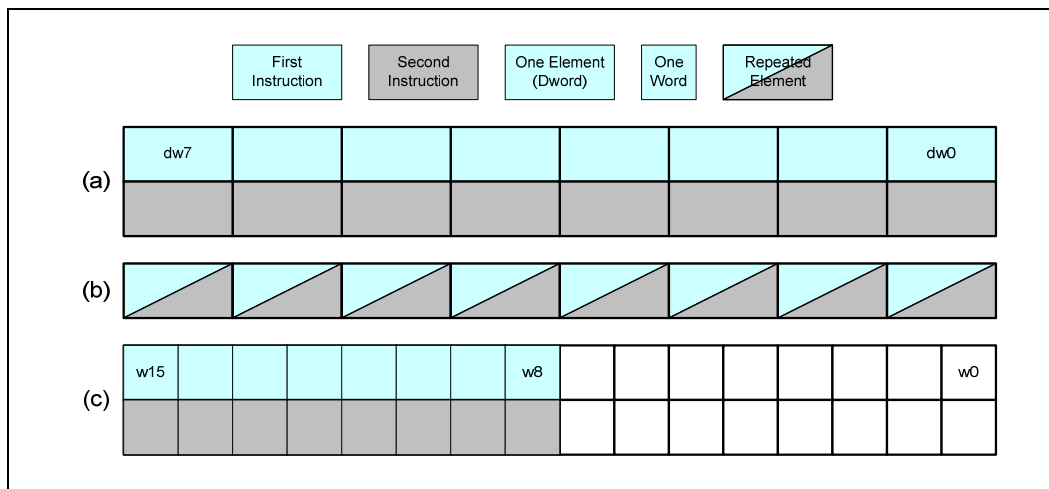


Figure 11-20 shows a case of a packed-word vector operand in a dword SIMD16 compressed instruction. When the destination stride is 4-byte, instead of inserting one to the LSB of RegNum field of src0, hardware will insert one to the MSB of SubRegNum field, as src0 is a packed-word vector (horizontal stride equals to 2-byte). This is specifically designed for Pixel Shader application.

Figure 11-20. A packed-word operand in a dword SIMD16 compressed instruction

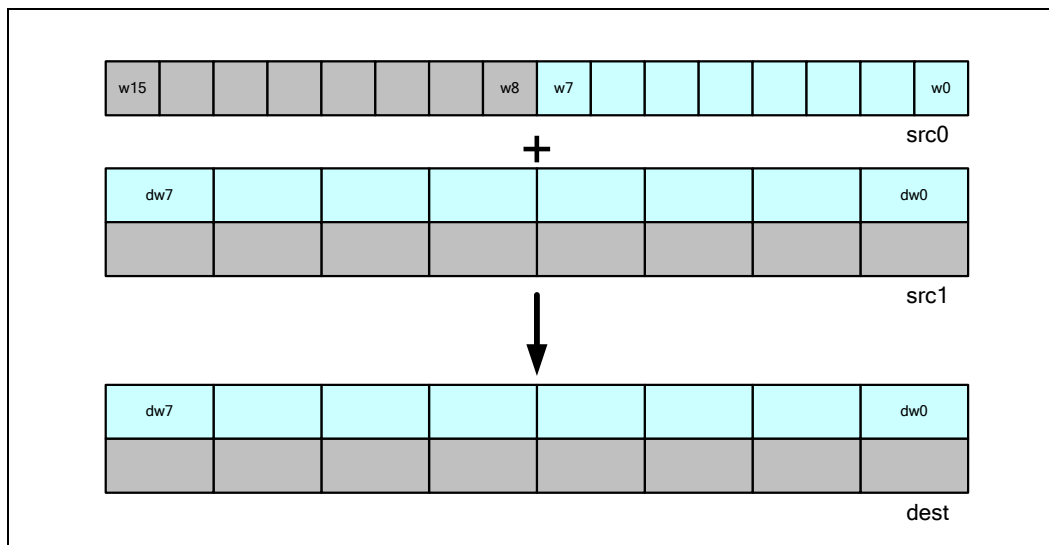


Figure 11-21 shows cases of scalar sources in a compressed instruction. There is no alignment restriction for a scalar source in a compressed instruction,

Figure 11-21. Dword and word (or byte) scalar source in a compressed instruction

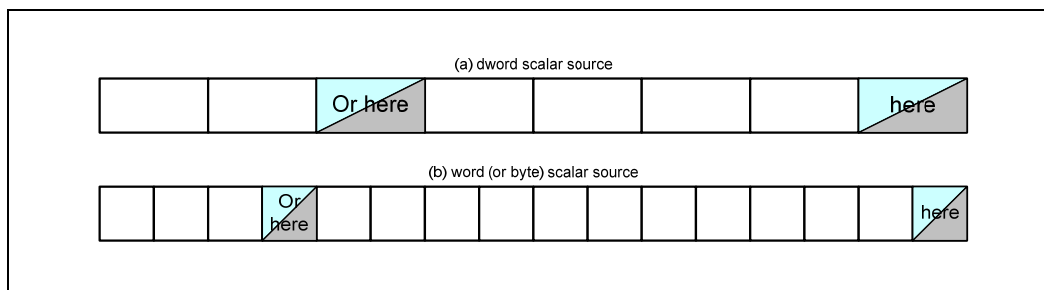


Figure 11-22 shows cases of indirectly-addressed operands in a compressed instruction. Hardware inserts one to bit 1 of SubRegNum field of the address register for the second decompressed instruction. This capability may be used for register regions that are not supported by the previously-mentioned cases. For example, source0 of the following eight instructions may be compressed with indirect-addressed.

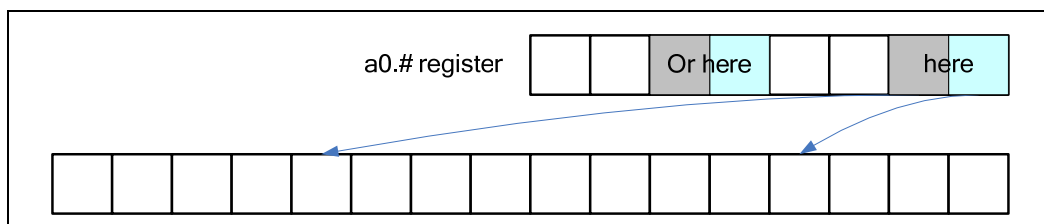
Direct-addressed instructions:

```
add.sat (8) r26.0<2>:ub r2.0<8;8,1>:w r26.0<16;8,2>:ub {Align1}
add.sat (8) r27.0<2>:ub r2.8<8;8,1>:w r27.0<16;8,2>:ub {Align1}
add.sat (8) r28.0<2>:ub r3.0<8;8,1>:w r28.0<16;8,2>:ub {Align1}
add.sat (8) r29.0<2>:ub r3.8<8;8,1>:w r29.0<16;8,2>:ub {Align1}
add.sat (8) r30.0<2>:ub r4.0<8;8,1>:w r30.0<16;8,2>:ub {Align1}
add.sat (8) r31.0<2>:ub r4.8<8;8,1>:w r31.0<16;8,2>:ub {Align1}
add.sat (8) r32.0<2>:ub r5.0<8;8,1>:w r32.0<16;8,2>:ub {Align1}
add.sat (8) r33.0<2>:ub r5.8<8;8,1>:w r33.0<16;8,2>:ub {Align1}
```

Compressed instructions with indirectly-addressing mode:

```
mov (1) a0.1<1>:d 0x00500040:d
add.sat (16) r26.0<2>:ub r[a0.2,0]<8;8,1>:w r26.0<16;8,2>:ub {Align1, Compr}
add.sat (16) r28.0<2>:ub r[a0.2,32]<8;8,1>:w r28.0<16;8,2>:ub {Align1, Compr}
add.sat (16) r30.0<2>:ub r[a0.2,64]<8;8,1>:w r30.0<16;8,2>:ub {Align1, Compr}
add.sat (16) r32.0<2>:ub r[a0.2,96]<8;8,1>:w r32.0<16;8,2>:ub {Align1, Compr}
```

Figure 11-22. Indirect-addressed source / destination operand in a compressed instruction





11.6 End of Thread

There is no special instruction opcode (such as an END instruction) to cause the thread to terminate execution. Instead, the end of thread is signified by a *send* instruction with the end-of-thread (EOT) sideband bit set. Upon executing a *send* instruction with EOT set, the EU stops on the thread. Upon observing an EOT signal on the output message bus, the Thread Dispatcher makes the thread's resource available. If a thread uses pre-allocated resource managed by a fixed function, such as URB handles and scratch memory, some fixed function protocol also requires the thread to terminate with the message header phase to carry the information in order for the fixed function to release the pre-allocated resource.

EU hardware guarantees that if a terminated thread has in-flight read messages or loads at the time of 'end' that their writebacks will not interfere with either other threads in the system or new threads loaded in the system in the future.

More details can be found in the *send* instruction description in Instruction Reference chapter.



11.7 Creating Conditional Flags

FPU will output 2 sets of conditional signals, 1 set will be generated from before the adder outputs clamping/re-normalizing/format conversion logic, we call this the pre conditional signals. 1 set will be generated from the final results after clamping and re-normalizing/format conversion logic, and we will call this the post conditional signals. The post conditional signals are used for fusing the compare instruction. *The flags generated from the post conditional signals should be equivalent to the flags generated by a separate CMP instruction after the current arithmetic instruction.*

The pre conditional signals will be used to generated flags for CMP/CMPN instructions only, this logically does the compare of the 2 input sources. The post conditional signals will be used to generated flags for all the other arithmetic instructions, this logically does the compare of the result with zero.

CMPN with both sources are NaN is a don't care case since this doesn't impact the MIN/MAX operations.

The pre conditional signals include the following:

- **pre_sign** bit: this bit reflects the sign of the computed result directly from the adders, without going through any kind clamping, normalizing, or format conversion logic.
- **pre_zero** bit: this bit reflects whether the value of the adder results are zero, again this should be obtained before any kind clamping, normalizing, or format conversion logic.

The post conditional signals include the following:

- **post_sign** bit: this bit reflects the sign of the final result after all the clamping, normalizing, or format conversion logic.
- **post_zero** bit: this bit reflects whether the value of the adder results are zero after all the clamping, normalizing, or format conversion logic.
- **OF** bit: this bit reflects whether an overflow occurred in any of the computation of the current instruction, including clamping, re-normalizing, and format conversion.
- **INC** bit: The increment bit is only used for RNDU, RNDE, and RNDZ instructions to convey the information whether an additional increment is needed for the execution channel for the given round instruction. It always returns 0 for RNDD and is undefined for other instructions.
- **NC** bit: The NaN computed bit indicates whether the computed result is not a number. It carries valid information for instructions operating on floating point values. For an operation on integer operands, this bit is always set to 0.
- **NS0** bit: The NaN bit indicates whether source 0 of an execution channel is not a number. It carries valid information for instructions operating on floating point values. For an operation on integer operands, this bit is always set to 0.
- **NS1** bit: The NaN bit indicates whether source 1 of an execution channel is not a number. It carries valid information for instructions operating on floating point values. For an operation on integer operands, this bit is always set to 0. For an operation with one source operand, this bit is also set to 0. This bit is only used for the comparison instruction CMPN, which is specifically provided to emulate MIN/MAX operations. For any other instructions, this bit is undefined.



Flag Generation for CMP instructions (The supported Conditional Modifiers are **.e**, **.ne**, **.g**, **.ge**, **.l**, and **.le**.)

Conditional Modifier	Meaning	Resulting Flag Value (for an execution channel)
`.e`	Equal-to	(pre_zero & !(NS0 NS1)) . This conditional modifier tests whether the 2 sources are equal. If either source is NaN (i.e. NC is true), the flag is force to false.
`.ne`	Not-Equal-to	!(pre_zero & !(NS0 NS1)) . This conditional modifier test whether the 2 sources are equal. It takes exactly the reverse polarity as modifier `.e` .
`.g`	Greater-than	(!pre_sign & !pre_zero & !(NS0 NS1)) . This conditional modifier tests whether source0 is greater than source1. If either source is a NaN (i.e. NC is true), the flag is forced to false.
`.ge`	Greater-than-or-equal-to	((!pre_sign pre_zero) & !(NS0 NS1)) . This conditional modifier tests whether source0 is greater than or equal to source1. If either source is a NaN (i.e. NC is true), the flag is forced to false.
`.l`	Less-than	(pre_sign & !pre_zero & !(NS0 NS1)) . This conditional modifier tests whether source0 is less than source1. If either source is a NaN (i.e. NC is true), the flag is forced to false.
`.le`	Less-than-or-equal-to	((pre_sign pre_zero) & !(NS0 NS1)) . This conditional modifier tests whether source0 is less than or equal to source1. If either source is a NaN (i.e. NC is true), the flag is forced to false.

Flag Generation for CMPN instructions (The supported Conditional Modifiers are **ge**, and **.l**)

Conditional Modifier	Meaning	Resulting Flag Value (for an execution channel)
`.ge`	Greater-than-or-equal-to	(!pre_sign pre_zero (NS1 & (Opcode==CMPN Opcode==SELwCMod))) & !(NS0 & (Opcode==CMPN)) . This conditional modifier tests whether source0 is greater than or equal to source1. If source-1 is a NaN (i.e. NS is true), the flag is forced to true.
`.l`	Less-than	((pre_sign & !pre_zero) (NS1 & (Opcode==CMPN Opcode==SELwCMod))) & !(NS0 & (Opcode==CMPN)) . This conditional modifier tests whether source0 is less than source1. If source-1 is a NaN (i.e. NS is true), the flag is forced to true.



Flag Generation for All Arithmetic Instructions other than CMP/CMPN instructions (The supported Conditional Modifiers are **.e**, **.ne**, **.g**, **.ge**, **.l**, **.le**, **.r**, **.o**, and **.u**.)

Conditional Modifier	Meaning	Resulting Flag Value (for an execution channel)
`.e`	Equal-to	(post_zero & !NC) . This conditional modifier tests whether the 2 sources are equal. If either source is NaN (i.e. NC is true), the flag is force to false.
`.ne`	Not-Equal-to	!(post_zero & !NC) . This conditional modifier test whether the 2 sources are equal. It takes exactly the reverse polarity as modifier `.e` .
`.g`	Greater-than	(!post_sign & !post_zero & !NC) . This conditional modifier tests whether source0 is greater than source1. If either source is a NaN (i.e. NC is true), the flag is forced to false.
`.ge`	Greater-than-or-equal-to	((!post_sign post_zero) & !NC) . This conditional modifier tests whether source0 is greater than or equal to source1. If either source is a NaN (i.e. NC is true), the flag is forced to false.
`.l`	Less-than	(post_sign & !post_zero & !NC) . This conditional modifier tests whether source0 is less than source1. If either source is a NaN (i.e. NC is true), the flag is forced to false.
`.le`	Less-than-or-equal-to	((post_sign post_zero) & !NC) . This conditional modifier tests whether source0 is less than or equal to source1. If either source is a NaN (i.e. NC is true), the flag is forced to false.
`.r`	Round-Increment	(IN & (!OF)) . This conditional modifier tests whether the rounding result (of a RNDx instruction) requires increment. Normally, the condition is true if IN is true. However, if overflow occurs for the execution channel (OF is true), the condition is force to false.
`.o`	Overflow	(OF) . This conditional modifier tests whether the computed result causes overflow – the computed result is outside the range of the destination data type. All other internal conditional signals are ignored.
`.u`	Unordered	(NC) . This conditional modifier tests whether the computed result is a NaN (unordered). All other internal conditional signals are ignored.

Programming Notes:

CMPN should be used ONLY to emulate MIN/MAX operations, and only the following macros should be used for MIN and MAX operations.

Macro for MIN:

```
CMPN.l.f0.0    null    s0 s1
(f0.0) SEL     dst s0 s1
```




Macro for MAX:

```
CMPN.ge.f0.0 null    s0 s1
```

```
(f0.0) SEL      dst    s0 s1
```

11.8 Destination Hazard

GEN4 architecture has built-in hardware to avoid destination hazard.

Destination Hazard stands for the risk condition when multiple operations are trying to write to the same destination and the result of the destination may be ambiguous. This may or may not happen on GEN4 for two instructions with the same destination, or with destinations that have overlapped register region, depending on the ordering of the arrival of destination results. Let's consider two instructions in a thread with potential destination hazard. There may be other instruction between them as long as there is no instruction sourcing the same destination. Using register scoreboards, GEN4 hardware automatically takes care of the destination hazard by not issuing the second instruction until the destination scoreboard is cleared. However, for certain cases, in fact for most cases, such destination hazard indicated by the register scoreboard is false, causing unnecessary delay of instruction issuing. This may result in lower performance. The destination dependency control field in the instruction word *{NoDDClr, NoDDchk}* allows software to selectively override such hardware destination dependency mechanism. Such performance optimization hooks must be used with extreme caution. When it is not 100% certainty that it is a false destination hazard, programmer should rely on hardware to result the dependency.

As the destination dependency control field does not apply to *send* instruction, there is only one condition that a programmer may use the *{NoDDClr, NoDDchk}* capability.

- If none of the two instructions is *send*, there CANNOT be any destination hazard. This is because instructions within a thread are dispatched in order (single-issued) and the execution pipeline is in-order and has a fixed latency.

11.9 Non-present Operands

Some instructions do not have two source operands and one destination operand. If an operand is not present for an instruction the operand field in the binary instruction must be filled with null. Otherwise, results are unpredictable.

Specifically, for instructions with a single source, it only uses the first source operand *<src0>*. In this case, the second source operand *<src1>* must be set to null and also with the same type as the first source operand *<src0>*. It is a special case when *<src0>* is an immediate, as an immediate *<src0>* uses DW3 of the instruction word, which is normally used by *<src1>*. In this case, *<src1>* must be programmed with register file ARF and the same data type as *<src0>*.



11.10 Instruction Prefetch

Due to prefetch of the instruction stream, the EUs may attempt to access up to 8 instructions (128 bytes) beyond the end of the kernel program – possibly into the next memory page. Although these instructions will not be executed, software must account for the prefetch in order to avoid invalid page access faults. One possible (though inefficient) solution would be to pad the end of all kernel programs with 8 NOOP instructions. A more efficient approach would be to ensure that the page after all kernel programs is at least valid (even if mapped to a dummy page). Note that the **General State Access Upper Bound** field of the STATE_BASE_ADDRESS command can be used to prevent memory accesses past the end of the General State heap (where kernel programs must reside).



12 Exceptions

12.1 Introduction

The Gen4 Architecture defines a basic exception handling mechanism for several exception cases. This mechanism supports both normal operations such as extensions of the mask-stack depth, as well as illegal conditions and debug features.

The following exception-types are supported:

Type	Trigger / Source	Sync / Async Recognition
MaskStack Overflow / Underflow	Hardware	Synchronous (w/ special case for 'do'; see 12.4.3)
Software Exception	Thread code	Synchronous
Breakpoint	A bit in the instruction word Triggering a debug snapshot	Synchronous
Illegal Opcode	Hardware	Synchronous
Halt	MMIO register write	Asynchronous

Threads may choose which exceptions to recognize and which to ignore. This mask information is specified on a per-kernel basis in fixed function state generated by the driver, and delivered to an EU as part of a new-thread dispatch. Upon arrival at the EU, the exception-mask information is used to initialize the exception enable fields of that thread's CR0.1 register, which controls exception recognition. This register is instantiated on a per-thread basis, allowing independent control of exception-type recognition across hardware threads. The exception enables in the CR0.1 register are r/w, and thus can be enabled/disabled via software at anytime during thread execution.

The exception handling mechanism relies on the "system routine", a single subroutine which provides common exception handling for all threads on all EUs in the system. This system routine is defined per-context and is identified via a 32b System-IP (SIP) register in context state. At the time of each context switch, the appropriate SIP for that context is loaded into each EU, allowing each context to have custom implementation of exception handling routines if so desired.

12.2 Exception-Related Architectural Registers

Exception-related registers are defined in architectural register CR0.0 through CR0.2. These registers are instantiated on a per-thread basis providing each hardware thread with unique control over exception recognition and handling. The registers provide the capability to mask exception types, determine the type of raised exception, provide storage the return address, and control exiting from the system routine back to the application thread.

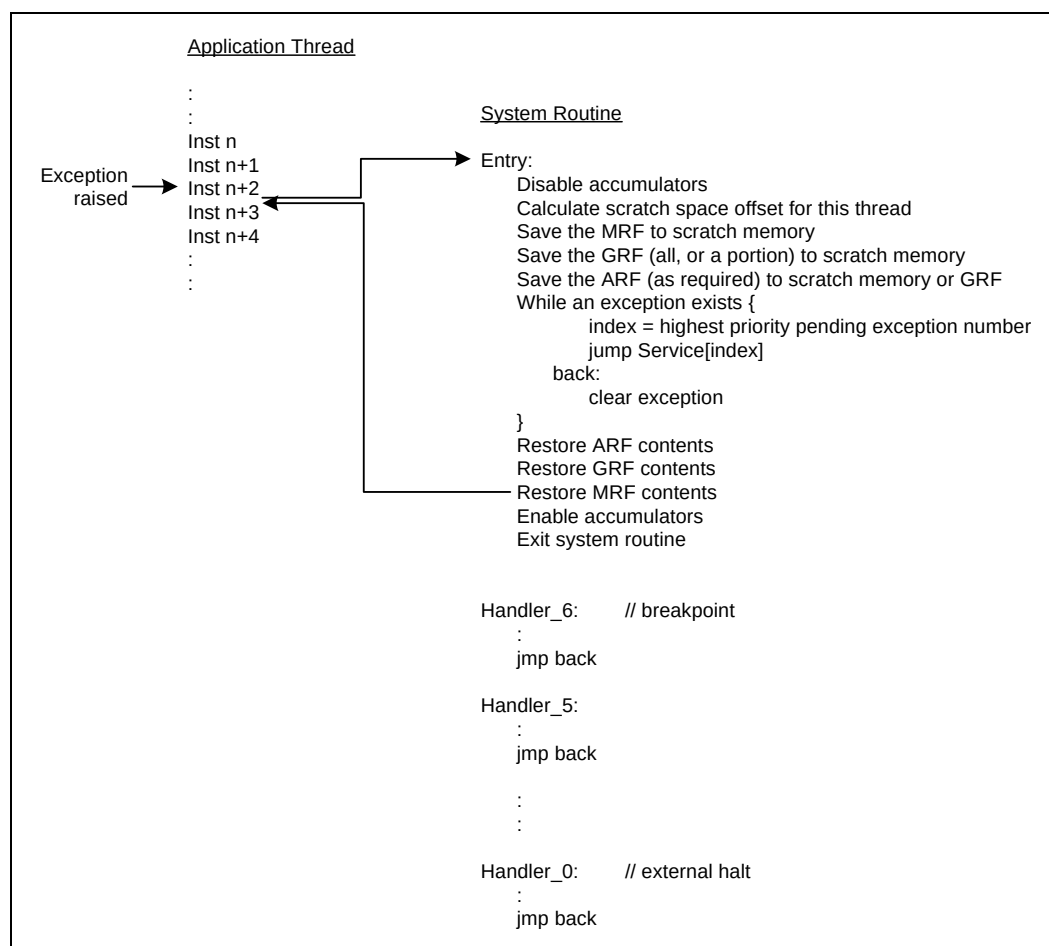


Many of the bits in these registers are manipulated by both hardware and software. In all cases, the read/write operations by hardware and software occur at exclusive times in a thread's lifetime, thus there is no need for an atomic R-M-W operation when accessing these registers.

12.3 System Routine

12.3.1 General Flow of the System Routine

The following diagram illustrates the basic flow of exception handling and structure of the system routine.





12.3.2 Invoking the System Routine

The system routine is invoked in response to a raised exception. Once an exception is raised, no further instructions from the application thread will be issued until the system routine has executed and returned control back to the application thread.

After an exception is recognized by hardware, the EU saves the thread's IP into its AIP register (CR0.2), and then moves the system routine offset, SIP, into the thread's IP register. At this point the next instruction to issue for that thread will be the first instruction of the system routine.

The system routine maintains the same execution priority, GRF and MRF register space, and thread state as that of the application thread from which it was invoked. Due to the assumption of the same priority, there may be significant absolute time between exception being raised and the actual invocation of the system routine, as other higher priority threads within the EU continue to execute. From a thread's perspective, once an exception is recognized, the next instruction to issue is from the system routine.

At the time of system routine invocation, there may still be outstanding registers in-flight from the application thread. Depending on the instruction sequence in the system routine, an in-flight register may be referenced by the system routine and cause a register-in-flight dependency. These dependencies are honored by the system routine and may cause the system routine to be suspended until such time that the register retires.

Exception processing is non-nested within a system routine. If a future exception is detected while executing the system routine, the exception is latched into CR0.1, but does not cause a nested re-invocation of the system routine. The exception recognition hardware recognizes only one outstanding exception of each type; i.e. once a specific exception type is detected and latched in CR0.1, and until the exception is cleared, any further exception of that type will be lost.

Accumulators are not natively preserved across the system routine. To make sure the accumulators are in the identical state once control is returned to the application thread, the system routine must either set the Accumulator Disable bit of CR0.0 prior to using any instruction which modifies an accumulator, or manually save/restore the accumulators (to GRF registers or system thread scratch memory) around the system routine. Saving/restoring accumulators, including their extended precision bits, can be accomplished by a short series of mov's and shifts of the accumulator register. Also note the state of the Accumulator Disable bit itself must be preserved unless, by convention, the driver software limits its manipulation to only the system routine.

Further, upon system routine entry, the execution-related masks (Continue, Loop, If, and Active masks, contained in the Mask Register) will remain set as they were in the application thread. Thus only a subset of channels may be active for execution. To enable execution on all channels, the system routine may choose to use the instruction option 'NoMask', or may choose to set the mask registers to the desired value so long as it saves/restores the original masks upon system routine entry/exit.

Similarly there is no hardware mechanism to preserve flags, mask-stacks, or other architectural registers across the system routine. The system routine must ensure that these values are preserved (see Section 12.3.7 for related discussion).



12.3.3 Returning to the Application Thread

Prior to returning control to the application thread, the system routine should clear the proper Exception Status and Control bit in CR1. Failure to do so will force the thread's execution to re-enter the system routine prior to any further instructions being executed from that application thread. (Note that single-stepping functionality is the one exception where the exception's Status and Control bit is not reset prior to exit.)

The system routine may choose to loop under a single invocation of the system routine until all pending exceptions are serviced, or may choose to service exceptions one at a time (a simpler solution, but less efficient).

The system routine is exited, and control returned to the application thread, via a write to the Master Exception State and Control bit of CR0.0. Upon clearing this bit, the value of the AIP architectural register (CR0.2) is restored to the thread's IP register and, with no further exceptions pending, execution resumes that address. The system routine must follow any write to Master Exception State and Control bit with at least one simd-16 'nop' instruction to allow control to transition. Throughout the system routine, the AIP register maintains its value at the time the exception was raised unless directly modified by the system routine. (See the AIP register definition for specifics on the IP value saved to AIP).

12.3.4 System-IP (SIP)

The System IP (SIP) is a 16B-aligned 32b offset of the first instruction of the system routine, relative to the General State Base Address. It is set via the `STATE_IP` command to the command streamer. The upper 28b of the 32b address is automatically delivered to all Gen4 EUs.

When the system routine is invoked, the application thread's current IP is first saved into the AIP field of the thread's architectural register CR0.2. The SIP address is then loaded into the thread's IP register and execution continues within the system routine. Thus each invocation of the system routine has a common entry point at the first instruction of the system routine. Upon system routine completion, the value held in AIP is returned to IP and execution continues on the application thread at the place where the exception was recognized.

12.3.5 System Routine Register Space

The system routine uses the same GRF and MRF space at the thread which invoked it. As such all of the calling thread's registers and their contents are visible to the system routine. Further, the system routine must only use r0..r15 of the GRF, as a minimal thread may have requested and been allocated this few. If the system routine requires more registers than this, the driver should establish a higher minimum allocation to all threads. It should also be noted that the system routine may encounter any residual register dependencies of the calling thread until such time that they clear by the return of in-flight writebacks.

Only one 32b GRF location, R0.4, is reserved for system routine usage. This is sufficient to allow the system routine to calculate the appropriate offset of its private scratch memory in the larger system-scratch memory space (as dictated by binding table entry 254). The offset is left as a driver convention, but likely based on a combination of Thread and EU IDs (see example system handler in section 12.3.6).



Other than the reserved R0.4 register field, there is no explicit GRF register space dedicated to the system routine, and any GRF needs must be accomplished via: (a) convention between the system routine and application thread, or (b) the system routine temporarily spilling the thread's GRF register contents to scratch memory, and restoration prior to system routine exit.

No persistent storage is natively allocated to the system routine, although a driver implementation may choose to carve out a piece of system scratch memory through its own convention.

Any parameter passing to the system routine (for use by s/w exceptions) is performed via the GRF based on a system-thread/application-thread convention.

12.3.6 System-Scratch Memory Space

There is a single unified system-scratch memory space per context shared by all EUs. It is anticipated that block is further partitioned into a unique scratch sub-space per-thread via convention implemented in the system routine, with each hardware thread having a uniform block size at a calculated offset from the base address. The block address for a thread is based on an offset derived from the thread's execution unit ID and thread ID made available through the TID and EUID field of architectural register SR0.0.

$$\text{Per\_Thread\_Block\_Size} = \text{System\_Scratch\_Block\_Size} / (\text{EU\_Count} * \text{Thread\_Per\_EU});$$
$$\text{Offset} = (\text{SR0.0.EID} * \text{Threads\_Per\_EU} + \text{SR0.0.TID}) * \text{Per\_Thread\_Block\_Size};$$

where in Gen4...

Threads_Per_EU = 4

EU_Count = 8

System_Scratch_Block_Size is a driver choice

Access to the system-scratch memory is performed through the Data Port via linear single-register or block-based read/write messages. The driver may choose to use any binding table index for system-scratch surface description. As a practical matter, the same index is expected to be used across all binding tables, as the index is typically hard coded in dataport messages used within the system routine coupled with the fact that a single system instance routine is used for all threads. Read/write messages to the Data Port contain the address of the binding table (provided in R0 of all threads) and an offset, from which the Data Port calculates the final target address.

The size of the overall system-scratch memory block is a function of the system routine's feature set, but for debug purposes should be, at a minimum, of size sufficient to cover storing the entire GRF, MRF, flags, and architectural register set for all hardware threads. Additional per-thread memory may also be required for mask-stack spill/fill, system routine constants, and any global persistent storage needs of the system routine.

It is expected that the system-memory block is allocated by the driver at context-create time and remains persistent at a constant memory address throughout the context's lifetime.



12.3.7 Conditional Instructions Within System Routines

It is expected that most, if not all, control flow within the system routine is scalar in nature. If so, the system routine should set SPF (Single Program Flow, CR0.0) to enable scalar branching. In this mode, conditional/loop instructions do not update the mask-stacks and therefore do not have restrictions on their use nor require the save/restore of hardware mask-stack registers.

If SIMD branching is desired within the system routine, special considerations must be taken. Upon entry to the system routine, the depth of the mask-stacks is unknown at that point, and may be near-full. If so, a subsequent conditional instruction and its associated mask 'push' may cause a stack overflow. This would generate an exception-within-the-system-routine, an unsupported occurrence. To prevent this, if the system routine uses SIMD conditional instructions, it must save the mask-stacks prior to the first SIMD conditional instruction, and restore them after the last SIMD conditional instruction. As a general solution, it may be easiest simply to implement the save/restore as part of the entry/exit code sequence, using an available GRF register-pair as storage location. Once saved, the stacks should be reset to their empty condition, namely depth = 0 and top-of-stack = 0xFFFFFFFF.

12.3.8 Messages in System Routines

The system routine uses the same MRF space as the thread on whose behalf the system routine was invoked. To allow the thread to resume with the same state as prior to the system routine invocation, the thread's MRF contents must be preserved across a system routine invocation. If the system routine requires MRF space for messages, it must manually save and restore the MRF locations which it uses.

Note that the MRF can only be used as an instruction's destination register, not a source. Therefore there is no option to save the MRF to the GRF. Thus the system routine should save the MRF contents to its dedicated scratch space. By convention it is recommended that MRF register m0 be reserved for system-thread use. This allows the system routine enough space to construct an initial Data Port write message starting at m0 without corrupting any MRF registers, facilitating a complete save/restore of the MRF by the system-thread.

12.3.9 Use of 'NoDDClr'

The Gen4 instruction word defines an instruction option 'NoDDClr' which overrides the native register dependency clearing mechanism of the typical instruction. When specified, 'NoDDClr' does not clear, at register writeback time, the dependency placed on the destination register of the instruction. Use of this mechanism may provide increased performance when the kernel can guarantee no dependency issues between instructions, but may cause issues with exception handling in some circumstances as discussed here.

Typically 'NoDDClr' is used in an instruction series to enable a sequence of writes to sub-fields of a GRF register without paying a dependency penalty on each instruction. In this case, 'NoDDClr' and 'NoDDChk' are used across an instruction sequence to allow the first instruction to set the destination dependency, interior instructions to write to the GRF register w/o dependency checks, and the last instruction clear the dependency. (This sequence is referred to as a 'NoDDClr' code block going forward).



By only allowing the last instruction to clear the dependency, program execution is prevented from going beyond a certain point until all writes of that sequence are known to retire.

The problem arises should an exception be raised within a 'NoDDClr' code block. In this case, there exists the potential for the system routine to hang while attempting to save/restore the code blocks destination register, as the outstanding dependency on that register will not clear until the final instruction of the block is executed – sometime after the system thread returns. Should the system routine attempt to use that register, the system routine will hang waiting on a dependency to clear from an instruction which has not yet been issued.

This is a known condition and will in some cases not allow the full GRF contents to be externally visible in system routine scratch space during a break or halt exception. To minimize the number of cases of such, guidelines are provided below for consideration. (Note that these are general guidelines, some of which can be alleviated through careful coding and register usage conventions and restrictions.)

- 'NoDDClr' code blocks should only be used where absolutely necessary.
- Instructions which may generate exceptions should not be placed within 'NoDDClr' blocks. This includes most conditional branch instructions (if, do, while, ...) as well as breakpoints explicitly in the instruction stream or triggered by the debug facilities in the IL1 instruction cache.
- To guarantee that r0 is visible in system scratch memory for debug purposes under any condition, (a) threads should not use 'NoDDClr' on r0, and (b) the system routine should move and send r0 to system scratch memory as a single register (as opposed to a block of 'n' registers). For practical reasons related to dataport message sizes, it may be beneficial to extend this restriction to r1 as well.
- If possible, use 'NoDDClr' on registers high in the thread's register allocation (e.g. r120), thus even if a system routine hang occurs, as much of the GRF is visible as possible. (Note this would also require the system routine to update the progress of the GRF dump, perhaps with each GRF block written, or to initialize the system routine's scratch space to a known value, to be able to distinguish valid/locations from unwritten locations).

Also a driver implementation may consider a "disable-NoDDClr" option in which jitted code does not use the 'NoDDClr' capability. In this case, there is no change to the code that is jitted other than removal of the 'NoDDClr' instruction option. The code executes as normal, but with a higher number of thread switches in what would have been a NoDDClr code block.



12.4 Exception Descriptions

12.4.1 'Illegal' opcode

The Gen4 ISA defines a single 'illegal' opcode. The byte value of the 'illegal' opcode is selected to be 0x00 due to it being a likely byte-value encountered by a wayward instruction pointer value. The 'illegal' instruction raises an exception prior to issue and operates as a 'nop' when issued down the execution pipeline. (Specifically, the opcode acts a 'nop', although other non-opcode instruction attributes still apply).

12.4.2 Undefined opcode

All undefined opcodes in the 8b opcode space are detected by hardware. If an undefined opcode is detected, the opcode is overridden by hardware, forcing it to the defined 'illegal' opcode. The offending instruction, should it eventually be issued down the execution unit's pipeline, generates an 'illegal opcode' exception as described in section 12.4.1. Note that the memory location of the offending opcode remains modified and may be queried if desired to determine its original value.

12.4.3 MaskStack Overflow / Underflow

Hardware-based mask-stacks are used for saving and restoring mask values for nested ifs and loops. These structure are limited to 16 levels of nesting. Thus, in the face of normal execution where nesting levels exceed 16, the hardware mask-stacks may occasionally overflow.

Upon reaching maximum capacity, the hardware sets the appropriate LStack/ISStack Overflow Exception Status and Control bits provided in the thread's architectural register CR0.1. This exception may be used by the system routine to extend the depths of mask-stacks by saving the stack contents and current depth to scratch memory prior to overflow. After saving, the stacks are reset and execution continues for another multiple of 16 nesting levels where a further exceptions are generated and the mask-stacks saved.

Likewise, as a thread exits if or loop nesting levels, masks values are restored automatically from the hardware mask-stack via 'pop' operations. A stack underflow may occur if the nesting level crosses a multiple of the hardware capacity. The LStack/ISStack Underflow exception if provided for restoring stack values from where they had been previously saved.

As an alternative implementation, the driver may choose to manage the mask-stacks manually, by maintaining nesting level counts during the thread's JIT process, and inserting code snippets to save/restore the stacks prior to overflow/underflow.



12.4.4 Software Exception

A mechanism is provided to allow an application thread to invoke an exception and is triggered through of the Software Exception Set and Clear bit of CR0.1. Sub-function determination and parameter passing into and out-of the exception handler is left to convention between the system-thread and application-thread. The thread's AIP instruction pointer is incremented prior to system-routine entry, therefore causing execution to resume at the subsequent application-thread instruction when the system routine is exited.

12.4.5 Breakpoint

A bit ('DebugCtrl') is defined in the instruction word which, when set, causes as breakpoint exception to be triggered prior to the associated instruction being executed. This bit can be set statically in the instruction word or may be dynamically set by hardware upon triggering a snapshot event in the Instruction L1 ("Instruction L1 Cache Breakpoint Address" registers in the Debug chapter). Note that the dynamic setting mechanism of this bit is on an instruction-fetch by instruction-fetch basis (i.e. is non-sticky in the L1). The system routine is expected to handle this exception by providing debug facilities, such as dumping/restoring the thread's state to programmer-visible memory for inspection.

Upon return from the system routine, the instruction which contained the breakpoint is re-issued. To prevent a subsequent breakpoint exception from being generated by the same instruction, breakpoint recognition may be suppressed for one instruction via the Breakpoint Suppress field of CR0.0. This field is typically set by the system routine prior to exiting a breakpoint exception.

For normal breakpoint operations, the system routine must clear the "Breakpoint Exception Status and Control" field of CR0.1 prior to returning from the system routine.

A single-stepping capability may be implemented by leaving the "Breakpoint Exception Status and Control" set, and clearing the Breakpoint Suppress field prior to system routine exit. This combination causes the instruction associated with the breakpoint to be reissued, this time with the breakpoint suppressed, and then re-entry to the system routine prior to the subsequent instruction due to the lingering breakpoint exception that remained un-cleared.

12.4.6 External Halt

A 'halt' exception may occur upon satisfying a debug snapshot trigger or via direction manipulation of a MMIO bit by driver software (see the External Halt on Snapshot bit, or Force External Halt bit of the TD-Debug Control register - Debug chapter). The halt exception is sent to all EUs simultaneously (although no guarantee is made as to recognition in identical clocks). An EU recognizes this condition internally by generating an External Halt exception. A likely implementation of a handling routine would dump the thread's state to programmer-visible memory (such as the system routine's scratch space) for inspection and debugging purposes. Although generally recognized within a few clocks, there is no specification as to the latency between triggering the Halt condition and it being recognized by an EU.



12.5 Events Which Do Not Generate Exceptions

The following conditions are either not recognized or do not generate an exception.

Illegal Instruction Format

This includes malformed instructions in which the opcode is legal, but the source or destination operands, or instruction attributes are not compliant with the instruction specification. There is no direct hardware support to detect these cases and the outcome of issuing a malformed instruction is undefined. Note that Gen4 does not support self-modifying code, therefore the driver (perhaps in some form of debug mode) has an opportunity to detect such cases before the thread is placed in service.

Malformed Message

A messages contents, destination registers, lengths, and descriptors are not interpreted in anyway by the execution units. Errors in specifying any of these fields do not raise exceptions in the execution unit but may be detected and reported by the shared functions (see the Debug chapter for details).

GRF Register Out-of-Bounds

Unique GRF storage is allocated to each thread which, at a minimum, satisfies that the register requirements specified in the thread's declaration. References to GRF register numbers beyond that called for in the thread's declaration do not generate exceptions. Depending on implementation, out-of-bounds register numbers may be remapped to r0..r15, although this functionality should not be relied upon by the thread. The hardware guarantees the isolation of each threads register space, thus there is no possibility of direct register manipulation from an out-of-bounds register access.

MRF Register Out-of-Bounds

A fixed amount of MRF register space is allocated for each thread, namely m0 through m15. References to MRF registers beyond m15 do not generate exceptions. Depending on implementation details, out-of-bounds register numbers may alias to in-bounds register numbers, although this functionality should not be relied upon by the thread.

Hung Thread

There is no hardware mechanism in the execution units to detect a hung thread, and should it occur, the thread remains hung indefinitely. It is the expectation that one or more hung threads will eventually cause the driver to recognize a context timeout and take appropriate recovery action.

Instruction Fetch Out of Bounds

The Gen4 EUs implement a full 32b instruction address range (with the 4 lsb's don't care), making it possible for a thread to attempt to jump to any 16B aligned offset in the 32b address space. The EU itself does not provide any type of address checking on its instruction request stream sent to the memory/cache hierarchy, although various memory address related error conditions are reported through the Memory Interface Registers (specifically "Page Table Error Register").



FPU Math Errors

The EU's floating point units have defined behavior for traditional floating point errors and do not generate exceptions. Therefore there is no support for signaling FPU math errors as exceptions.

Destination Register Overflow

Depending on source operand contents, destination register size, and operation being performed, overflows may occur in the EU's pipeline. These are not flagged as exceptions and software must explicitly check the overflow bit in the thread's architectural register if overflow is a concern.

12.6 System Handler Example

The following code sequence illustrates some concepts of the system routine. It is intended to be just a shell, without getting into the specifics of each exception handler. The example frees enough MRF and GRF space to get the routine started, then jumps to the handler for the specific exception. Many other implementations are also valid, including single exception servicing (as opposed to looping) per invocation, and saving only the GRF or MRF space required by the exception being serviced.

```

#define ACC_DISABLE_MASK          0xFFFFFFFFD
#define MASTER_EXCP_MASK         0x7FFFFFFF
#define SYSROUTINE_SCRATCH_BLKSIZE 16384    //for example

// --- SharedFunc IDs ---
#define DPR                        0x04000000
#define DPW                        0x05000000

// --- message lengths ---
#define ML5                        0x00500000
#define ML9                        0x00900000

// --- response lengths ---
#define RL0                        0x00000000
#define RL4                        0x00040000
#define RL8                        0x00080000

// --- dataport block sizes ---
#define BS1_LOW                    0x0000
#define BS1_HIGH                   0x0100
#define BS2                        0x0200
#define BS4                        0x0300

// --- Scratch Layout ---
#define SCR_OFFSET_MRF             0
#define SCR_OFFSET_GRF             512      // + 16 reg
#define SCR_OFFSET_ARF             512 + 4096 // + 16 + 128
reg

// --- Write Dataport constants ---
// target=dcache, type= oword_block_wr, binding_tbl_offset=0
#define DPW                        0x000

// --- Read Dataport constants ---
// target=dcache, type= oword_block_rd, binding_tbl_offset=0
#define DPR                        0x000

Sys_Entry:
// --- disable accumulator for debug routine ---

```



```
and    (1) cr0.0 cr0.0 ACC_DISABLE_MASK      {NoMask}

// --- calc scratch offset for this thread into r0.4 ---
shr    (1) r0.4 sr0.0:uw 6                    {NoMask}
add    (1) r0.4 r0.4 sr0.0:ub                  {NoMask}
mul    (1) r0.4 r0.4 SYSROUTINE_SCRATCH_BLKSIZE {NoMask}

// --- setup m0 w/ block offset
mov    (8) m0 r0                              {NoMask}

// --- save mrf 7...0; (may choose to save the whole mrf)
add    (1) m0.2 r0.4 SCR_OFFSET_MRF           {NoMask}
send   (8) null m0 null DPW|ML9|RL0           {NoMask}

// --- save mrf 8...15; (optional; req'ed if sys-routine stays w/in
mrf7-0)
mov    (8) m7 r0                              {NoMask}
add    (1) m7.2 r0.4 (SCR_OFFSET_MRF + 256) {NoMask}
send   (8) null m7 null DPW|ML9|RL0           {NoMask}

// --- save r0..r1 to system scratch ---
// --- (Note: done as a single register to guarantee external
// --- visibility - see "Use of 'NoDDClr'" in Exceptions PRM chapter
mov    (16) m1 r0                             {NoMask}
send   (8) m0 null null DPW|ML2|RL0           {NoMask}

// --- save r2..r3 to free some room
mov    (16) m3 r2                             {NoMask}
add    (1) m0.2 r0.4 SCR_OFFSET_GRF + 64      {NoMask}
send   (8) m0 null null DPW|ML4|RL0           {NoMask}

// --- save r4..r7 to free some room (optional, depending on needs)
mov    (16) m8 r4                             {NoMask}
mov    (16) m10 r6                            {NoMask}
add    (1) m7.2 r0.4 (SCR_OFFSET_GRF + 128) {NoMask}
send   (8) m7 null null DPW|ML5|RL0           {NoMask}

// --- save r8..r11 to free some room (optional, depending on needs)
mov    (16) m1 r8                             {NoMask}
mov    (16) m3 r10                            {NoMask}
add    (1) m0.2 r0.4 (SCR_OFFSET_GRF + 256) {NoMask}
send   (8) m0 null null DPW|ML5|RL0           {NoMask}

// --- save r12..r15 to free some room (optional, depending on
needs)
mov    (16) m8 r12                             {NoMask}
mov    (16) m10 r14                            {NoMask}
add    (1) m7.2 r0.4 (SCR_OFFSET_GRF + 384) {NoMask}
send   (8) m7 null null DPW|ML5|RL0           {NoMask}

// --- save ARF registers (optional, depending on use) ---
// flags, maskstacks, others...

// --- save f0.0 ---
mov    (1) r1.0:uw f0.0                       {NoMask}

Next:  // --- exceptions pending? If not, exit ---
cmp.e  (1) f0.0 cr0.4:uw 0:uw                 {NoMask}
(f0.0) mov (1) IP EXIT                        {NoMask}

// --- find highest priority exception ---
lzd    (1) r1.1:uw cr0.4:uw                   {NoMask}

// --- jumtable to service routine ---
jmp    (1) r1.1:uw                            {NoMask}
mov    (1) IP CRService_0                     {NoMask}
mov    (1) IP CRService_1                     {NoMask}
mov    (1) IP CRService_2                     {NoMask}
// :
// :
```



```

// :
mov (1) IP CRService_15 {NoMask}

mov (1) IP Next
Service_0:
// clear exception from CR0.1
// perform service routine
// jump to exit (or if looping on exceptions, go to next loop)

// :
// :

Service_15:
// clear exception from CR0.1
// perform service routine
// jump to exit (or if looping on exceptions, go to next loop)

Exit:
// --- restore f0.0 ---

// --- restore ARF registers (as required) ---
// flags, maskstacks, others...

// --- restore r12..r15 ---
// --- restore r8..r11 ---
// --- restore r4..r7 ---
// --- restore r0..r3 ---

// --- restore m8..m15 ---
// --- restore m0..m7 ---

// --- clear Master Exception State bit in CR0.0
and (1) cr0.0 cr0.0 MASTER_EXCP_MASK
nop (16)

```

Below is a code sequence to programmatically clear the GRF scoreboard in the case of a timeout waiting on a register that may never return:

```

// At this point, all we know is we have a hung thread. We'd like to
copy the // GRF to scratch memory to make it visible for debug purposes, but there
may be // a register that is hung w. an outstanding dependency. To get around
mask // any hung dependency, we can walk the GRF using NoDDChk, using execution
// of f0 = 0 so we don't touch the register contents.

Clear_Dep:
mov f0 0x00
(f0) mov r0 0x00 {NoDDChk}
(f0) mov r1 0x00 {NoDDChk}
(f0) mov r2 0x00 {NoDDChk}
...
...
(f0) mov r127 0x00 {NoDDChk}
// GRF scoreboard now cleared.

```

§§





13 *Instruction Set Summary*

13.1 **Instruction Set Characteristics**

13.1.1 **SIMD Instructions and SIMD Width**

GEN4 instructions are SIMD (single instruction multiple data) instructions. The number of data elements per instruction, or the execution size, depends on the data type. For example, the execution size for GEN4 instructions operating on 256-bit wide vectors can be up to 8 for 32-bit data types, and be up to 16 for 16-bit data. The maximum execution size for GEN4 instructions for 8-bit data types is also limited to 16.

An instruction compression mode is supported for 32-bit instructions (including mixed 32-bit and 16-bit data computation). A compressed GEN4 instruction works on twice as many SIMD data as that for a non-compressed GEN4 instruction. Non-compressed instructions are also referred to as 'native' instructions. A compressed instruction is converted into two native instructions by the instruction dispatcher in the EU.

GEN4 instructions are executed on a narrower SIMD execution pipeline. Therefore, GEN4 native instructions take multiple execution cycles to complete. See Table 13-1 for parameters for difference device hardware.

Table 13-1. GRF instruction execution parameters in device hardware

Device	Execution Pipeline	Native Instruction		Compressed Instruction	
	Width (bits)	Max Width (bits)	Min Execution Cycles (clocks)	Width (bits)	Min Execution Cycles (clocks)
[DevBW]	128	256	2	512	4
[DevCL]	128	256	2	512	4

13.1.2 **Instruction Operands and Register Regions**

Majority of GEN4 instructions may have up to three operands, two sources and one destination. Each operand is able to address a register region. Source operands support negate and absolute modifier and channel swizzle, and the destination operand supports channel mask.

Dual destination instructions are also supported (four-operand instructions in a general sense): One case is for the implied destination – flag register, where the conditional modifiers and the predicate modifiers may apply. Another case is the message header creation (implied move or implied assembling of the header) in the *send* instruction.



Each execution channel contains an accumulator that is wider than the input data to support back-to-back accumulation operations with increased precision. The added precision (see accumulator register description in Execution Environment chapter) determines the maximum number of accumulations before possible overflow. The accumulator can be pre-loaded through the use of **mov**. It can also be pre-loaded by arithmetic instructions such as **add**, **mul**, since the result of these instructions can go to the accumulator. The accumulator registers are per thread and therefore safe for thread switching.

Register access can be direct or register-indirect. Register-indirect register access uses address registers plus an immediate offset term to compute the register addresses, and only applies to the first source operand (src0) and/or the destination operand.

There is one address register that contains 8 sub-registers. Each sub-register contains a 16-bit unsigned value. The leading two sub-registers form a special doubleword that can be used as the descriptor for the send instruction.

Source operand can also be immediate value (also referred to as inline constants). For instructions with two source operands, only the second operand <src1> is allowed to be immediate. For instructions with only one source operand, the source operand <src0> is used and it can be an immediate.

An immediate source operand can be a scalar value of specified type up to 32-bit wide, which is replicated to create a vector with length of Execution Size. An immediate operand can also be a special 32-bit vector with 8 elements each of 4-bit signed integer value, or a 32-bit vector with 4 elements each of 8-bit restricted float value.

13.1.3 Instruction Execution

It is implied that all instructions operate across all channels of data unless otherwise specified either via destination mask, predication, execution mask (caused by SIMD branch and loop instructions), or execution size.

Format conversion of the result is done through destination format specifier; normally, there is no need for specific format conversion instructions.

Mixed format operation, where the two source operands are in different formats, is supported through source type specifier.

Instruction execution size can be specified per instruction, from scalar (*ExecSize* = 1) up to the maximal execution size supported for the data type, with the restriction that execution size can only be in power of 2.



13.2 Instruction Machine Formats

This section shows the machine formats of the GEN4 instruction set. The instructions in GEN4 architecture have fixed length of 128 bits. Out of the 128 bits, there are 111 bits in use, and the remaining bits are reserved for future extensions. One instruction consists of instruction fields that control various stages of execution of the instruction. These fields are roughly groups into the 4 doublewords as the following.

- Instruction Operation Doubleword (DW0) contains the opcode and other general instruction control fields.
- Instruction Destination Doubleword (DW1) contains the destination operand (<dst>) and the register file and type of source operands.
- Instruction Source-0 Doubleword (DW2) contains the first source operand (<src0>) and flag register number
- Instruction Source-1 Doubleword (DW3) contains the second source operand (<src1>) and is used to hold the 32-bit immediate source (imm32 as <src0> or <src1>).

Table 13-2 depicts the details of the organization of fields in the 128-bit instruction word based on the Addressing Mode and Access Mode of an instruction. Definitions for individual instruction fields are provided in the following sections.

The **send** instruction is also shown in the table as it has some unique instruction fields. For example, the message descriptor (plus EOT) occupies the whole DW3, and the immediate destination register overlaps with the Conditional Modifier field. The rest of fields in DW0-3 follows the definition on the left, depending on Addressing Mode and Access Mode of the **send** instruction.

Not shown is for immediate operands. When an immediate source is present in an instruction, it always occupies the whole DW3 with a 32-bit value.

Support for indirect addressing for <src1>, as shown by the gray areas in Table 4-2 is device dependent. See Table 11-30 (Indirect source addressing support available in device hardware) in ISA Execution Environment for details.



Table 13-2. GEN4 Instruction Format

DW #	Instr Bits Alloc	High Bit	Low Bit	Instr Bits Used	AddrMode = Direct		AddrMode = Indirect		SEND			
					AccessMode = Align16	AccessMode = Align1	AccessMode = Align16	AccessMode = Align1	MsgDesc Imm	MsgDesc Reg		
3	1	127	127	0					EOT			
	6	126	121	0								
	4	120	117	4	Src1.VertStride		Src1.VertStride					
	1	116	116	1								
	2	115	114	2	Src1.Width		Src1.Width					
	2	113	112	2	Src1.ChanSel[7:4]	Src1.HorzStride	Src1.ChanSel[7:4]	Src1.HorzStride				
	1	111	111	1	Src1.AddrMode		Src1.AddrMode					
	2	110	109	2	Src1.SrcMod		Src1.SrcMod					
	3	108	106	3	Src1.RegNum [7:0]		Src1.AddrSubRegNum					
	5	105	101	5								
	1	100	100	1	Src1.SubRegNum [4]	Src1.SubRegNum [4:0]	Src1.AddrImm [9:4]	Src1.AddrImm [9:0]				
	4	99	96	4	Src1.ChanSel[3:0]		Src1.ChanSel[3:0]					
								Imm[30:0]	Reg32			
2	6	95	90	0								
	1	89	89	1	FlagSubRegNum							
	4	88	85	4	Src0.VertStride							
	1	84	84	1								
	2	83	82	2	Src0.Width		Src0.Width					
	2	81	80	2	Src0.ChanSel[7:4]	Src0.HorzStride	Src0.ChanSel[7:4]	Src0.HorzStride				
	1	79	79	1	Src0.AddrMode							
	2	78	77	2	Src0.SrcMod							
	3	76	74	3			Src0.AddrSubRegNum					
	5	73	69	5	Src0.RegNum [7:0]							
	1	68	68	1	Src0.SubRegNum [4]		Src0.AddrImm [9:4]	Src0.AddrImm [9:0]				
	4	67	64	4	Src0.ChanSel[3:0]	Src0.SubRegNum [4:0]	Src0.ChanSel[3:0]					
								Same				
1	1	63	63	1	Dst.AddrMode							
	2	62	61	2	Dst.HorzStride		Dst.HorzStride					
	3	60	58	3			Dst.AddrSubRegNum					
	5	57	53	5	Dst.RegNum [7:0]							
	1	52	52	1	Dst.SubRegNum [4]		Dst.AddrImm [9:4]	Dst.AddrImm [9:0]				
	4	51	48	4	Dst.ChanEn[3:0]	Dst.SubRegNum [4:0]	Dst.ChanEn[3:0]					
	1	47	47	0								
	3	46	44	3	Src1.SrcType		Src1.SrcType					
	2	43	42	2	Src1.RegFile		Src1.RegFile					
	3	41	39	3	Src0.SrcType							
	2	38	37	2	Src0.RegFile							
	3	36	34	3	Dst.DstType							
2	33	32	2	Dst.RegFile								
0	1	31	31	1	Saturate							Same
	1	30	30	1	DebugCtrl							
	2	29	28	0								
	4	27	24	4	CondModifier				MDst.RegNum[8:5]			
	3	23	21	3	ExecSize							
	1	20	20	1	PredInv							
	4	19	16	4	PredCtrl							
	2	15	14	2	ThreadCtrl							
	2	13	12	2	ComprCtrl							
	2	11	10	2	DepCtrl							
	1	9	9	1	MaskCtrl							
	1	8	8	1	AccessMode						Same	
1	7	7	0	(reserved for Opcode)								
7	6	0	7	Opcode				Same				
128		111		TOTAL								



13.2.1 Common Instruction Fields

As shown in Table 13-2, the meanings (encoding) of certain bit fields in the 128-bit instruction word varies depending on the values of other bit fields.

Table 13-3 provides the definition of common fields in the instruction word. The 'Width' column specifies the width of the field in bits. These common fields will be referred to later in describing the fields of different doublewords of the instruction. The definition for fields that have unique representation can be found in its corresponding doubleword of the instruction.

Table 13-3. Definitions of Common Instruction Fields

Field	Description	Width
CondModifier	Conditional Modifier. This field sets the flag register based on the internal conditional signals output from the execution pipe such as sign, zero, overflow, round-increment and NaNs, etc. If this field is set to 0000, no flag registers are updated. This field may also be referred to as the <i>flag destination control</i> field. This field applies to all instructions except <i>send</i>. 0000 = Do not modify the flag register (normal) 0001 = Zero or Equal (`.z` or `.e`) 0010 = Not Zero or Not Equal (`.nz` or `.ne`) 0011 = Greater-than (`.g`) 0100 = Greater-than-or-equal (`.ge`) 0101 = Less-than (`.l`) 0110 = Less-than-or-equal (`.le`) 0111 = Round-increment (`.r`) 1000 = Overflow (`.o`) 1001 = Unordered with Computed NaN (`.u`) 1010 - 1111 = Reserved	4
AddrMode	Addressing Mode. This field determines the addressing method of the operand. When it is cleared, the register address of the operand is directly provided by bits in the instruction word. It is called a direct register addressing mode. When it is set, the register address of the operand is computed based on the address register value and an address immediate field in the instruction word. This is referred to as a register-indirect register addressing mode. This field applies to the destination operand and the first source operand, <src0>. Support for <src1> is device dependent. See Table XX (Indirect source addressing support available in device hardware) in ISA Execution Environment for details. 0 = "Direct". Direct register addressing 1 = "Register-Indirect" (or in short "Indirect"). Register-indirect register addressing	1



Field	Description	Width
RegNum	Register Number. This field provides the register number for the operand. For GRF or MRF register operand, it provides the portion of register address aligning to 256-bit. For an ARF register operand, this field is encoded such that MSBs identify the architecture register type and LSBs provide its register number. This field together with the corresponding <i>SubRegNum</i> field provides the byte aligned address for the origin of the register region. Specifically, this field provides bits [12:5] of the byte address, while <i>SubRegNum</i> field provides bits [4:0]. This field applies to the destination operand and the source operands. It is ignored (or not present in the instruction word) for an immediate source operand. This field is present if the operand is in direct addressing mode; it is not present if the operand is register-indirect addressed. Format = U8, if <i>RegFile</i> = GRF. 0x00 to 0x7F = Register number in the range of [0, 127] 0x80 to 0xFF = Reserved Format = U8, if <i>RegFile</i> = MRF. 0x00 to 0x0F = Register number in the range of [0, 15] 0x10 to 0xFF = Reserved Format = 8-bit encoding, if <i>RegFile</i> = ARF. This field is used to encode the architecture register as well as providing the register number. See GEN4 Execution Environment chapter for details.	8
SubRegNum	Sub-Register Number. This field provides the sub-register number for the operand. For GRF or MRF register operand, it provides the byte address within a 256-bit register. For an ARF register operand, this field also provides the sub-register number according to special encoding for the given architecture register. This field together with the corresponding <i>RegNum</i> field provides the byte aligned address for the origin of the register region. Specifically, this field provides bits [4:0] of the byte address, while <i>RegNum</i> field provides bits [12:5]. This field applies to the destination operand and the source operands. It is ignored (or not present in the instruction word) for an immediate source operand. This field is present if the operand is in direct addressing mode; it is not present if the operand is register-indirect addressed. Format = U5, if <i>RegFile</i> = GRF or MRF 0x00 to 0x1F = Sub-Register number in the range of [0, 31] Format = 5-bit encoding, if <i>RegFile</i> = ARF. This field is used to encode the architecture register as well as providing the register number. See GEN4 Execution Environment chapter for details.	5



Field	Description	Width
AddrSubRegNum	Address Sub-Register Number. This field provides the sub-register number for the address register. The address register contains 8 sub-registers. The size of each sub-register is one word. The address register contains the register address of the operand, when the operand is in register-indirect addressing mode. This field applies to the destination operand and the source operands. It is ignored (or not present in the instruction word) for an immediate source operand. This field is present if the operand is in register-indirect addressing mode; it is not present if the operand is directly addressed. Format = U3 0x00 to 0x07 = Address Sub-Register number in the range of [0, 7]	3
AddrImm	Address Immediate. This field provides the immediate value in unit of byte to be added to the address register in order to compute the register address (byte-aligned region origin) for the operand. It is a 10-bit signed integer in 2's complement form. This field is present if the operand is in register-indirect addressing mode; it is not present if the operand is directly addressed. <i>Note: that the address immediate field may not be able to cover the whole GRF register range for a thread, as the maximum GRF register space for a thread is 4KB.</i> Format = S9 Valid range: [-512, 511]	10
SrcMod	Source Modifier. This field specifies the numerical modification to a source operand. The value of each data element of a source operand can optionally have its absolute value taken and/or its sign inverted prior to delivery to the execution pipe. The absolute value is prior to negate such that a guaranteed negative value can be produced. This field only applies to source operand. It does not apply to destination. This field is not present for an immediate source operand. 00 = No modification (normal) 01 = “(abs)”. Absolute 10 = “-”. Negate 11 = “-(abs)”. Negate of the absolute (forced negative value)	2



Field	Description	Width
VertStride	Vertical Stride. The field provides the vertical stride of the register region in unit of data elements for an operand. Encoding of this field provides values in power of 2, ranging from 0 to 32 elements. Larger values are not supported due to the restriction that a source operand must reside within two adjacent 256-bit registers (64 bytes total). Special encoding 1111b (0xF) is only valid when the operand is in register-indirect addressing mode (<i>AddrMode</i> = 1). If this field is set to 0xF, one or more sub-registers of the address registers may be used to compute the addresses. Each address sub-register provides the origin for a row of data element. The number of address sub-registers used is determined by the division of <i>ExecSize</i> of the instruction by the <i>Width</i> fields of the operand. This field only applies to source operand. It does not apply to destination. This field is not present for an immediate source operand. For Align16 access mode, only encodings of 0000 and 0011 are allowed. Other codes are reserved. <i>Note 1: Vertical Stride larger than 32 is not allowed due to the restriction that a source operand must reside within two adjacent 256-bit registers (64 bytes total).</i> <i>Note 2: In Align16 access mode, as encoding 0xF is reserved, only single-index indirect addressing is supported.</i> <i>Note 3: If indirect address is supported for <src1>, encoding 0xF is reserved for <src1> – only single-index indirect addressing is supported.</i> 0000 = 0 Elements 0001 = 1 Element 0010 = 2 Elements 0011 = 4 Elements 0100 = 8 Elements 0101 = 16 Elements (applies to byte or word operand only) 0110 = 32 Elements (applies to byte operand only) 0111-1110 = Reserved 1111 = VxH or Vx1 mode (only valid for register-indirect addressing in Align1 mode)	4
Width	Width. This field specifies the number of elements in the horizontal dimension of the region for a source operand. This field cannot exceed the <i>ExecSize</i> field of the instruction. This field only applies to source operand. It does not apply to destination. This field is not present for an immediate source operand. 000 = 1 Elements 001 = 2 Elements 010 = 4 Elements 011 = 8 Elements 100 = 16 Elements 101-111 = Reserved	3



Field	Description	Width																
HorzStride	Horizontal Stride. This field provides the distance in unit of data elements between two adjacent data elements within a row (horizontal) in the register region for the operand. This field applies to both destination and source operands. This field is not present for an immediate source operand. 00 = 0 Elements 01 = 1 Element 10 = 2 Elements 11 = 4 Elements	2																
Imm32	32-bit Immediate. The 32-bit immediate data field for the operand. It may contain any legal bit pattern for its associated type. Only one 32-bit immediate value may be present in an instruction, therefore binary operations only support <src1> as an immediate value. The low order bits are directly used when fewer than 32-bits are needed to describe the desired type; the 32-bits are not coerced into the designated type. For UW and W data types, programmer is required to replicate the lower word to the upper word of this field. This field only applies to the last source operand. Signed and unsigned byte integer data types are not supported for an immediate operand. Valid ranges according to data type: <table><tr><th>Immediate Data Type</th><th>Valid Range (inclusive)</th></tr><tr><td>F</td><td>[0...±1.0*2^{-128...127}]</td></tr><tr><td>UW</td><td>[0, 65535]</td></tr><tr><td>W</td><td>[-32768, 32767]</td></tr><tr><td>UD</td><td>[0, 2³²-1]</td></tr><tr><td>D</td><td>[-2³¹, 2³¹-1]</td></tr><tr><td>VF</td><td>[0, ±0.125...±31]</td></tr><tr><td>V</td><td>[-8, 7]</td></tr></table>	Immediate Data Type	Valid Range (inclusive)	F	[0...±1.0*2 ^{-128...127}]	UW	[0, 65535]	W	[-32768, 32767]	UD	[0, 2 ³² -1]	D	[-2 ³¹ , 2 ³¹ -1]	VF	[0, ±0.125...±31]	V	[-8, 7]	32
Immediate Data Type	Valid Range (inclusive)																	
F	[0...±1.0*2 ^{-128...127}]																	
UW	[0, 65535]																	
W	[-32768, 32767]																	
UD	[0, 2 ³² -1]																	
D	[-2 ³¹ , 2 ³¹ -1]																	
VF	[0, ±0.125...±31]																	
V	[-8, 7]																	



Field	Description	Width
ChanEn	Channel Enable. Four channel enables are defined for controlling which channels will be written into the destination region. These channel mask bits are applied in a modulo-four manner to all <i>ExecSize</i> channels. There is 1-bit Channel Enable for each channel within the group of 4. If the bit is cleared, the write for the corresponding channel is disabled. If the bit is set, the write is enabled. Mnemonic for the bit being set for the group of 4 is "x", "y", "z", and "w", respectively, where "x" corresponds to Channel 0 in the group and "w" corresponds to channel 3 in the group. This field only applies to destination operand. This field is only present in Align16 mode. 0 = Write Disabled 1 = Write Enabled (normal)	4
ChanSel	Channel Select. This field controls the channel swizzle for a source operand. The normally sequential channel assignment can be altered by explicitly identifying neighboring data elements for each channel. Out of the 8-bit field, 2 bits are assigned for each channel within the group of 4. ChanSel[1:0], [3:2], [5:4] and [7:6] are for channel 0 ("x"), 1 ("y"), 2 ("z"), and 3 ("w") in the group, respectively. For example with an execution size of 8, <i>r0.0<4>.zywz:f</i> would assign the channels as follows: Chan₀ = Data₂, Chan₁ = Data₁, Chan₂ = Data₃, Chan₃ = Data₂; Chan₄ = Data₆, Chan₅ = Data₅, Chan₆ = Data₇, Chan₇ = Data₆. This field only applies to source operand. This field is only present in Align16 mode. It is not present for an immediate source operand. The 2-bit Channel Selection field for each channel within the group of 4 is defined as the following. 00 = "x". Channel 0 is selected for the corresponding execution channel 01 = "y". Channel 1 is selected for the corresponding execution channel 10 = "z". Channel 2 is selected for the corresponding execution channel 11 = "w". Channel 3 is selected for the corresponding execution channel	8
MsgDscpt31	Message Description. This field, containing 31-bit immediate values, provides the description of the message to be sent. This field only applies to the <i>send</i> instruction. It is not present for other instructions. The meaning of the field depends on the type of message as well as the message shared function target. Format: U31	31
EOT	End of Thread. This field controls the termination of the thread. For a <i>send</i> instruction, if this field is set, EU will terminate the thread and also set the EOT bit in the message sideband. This field only applies to the <i>send</i> instruction. It is not present for other instructions. 0 = The thread is not terminated 1 = EOT	1



13.2.2 Instruction Operation Doubleword (DW0)

Most fields in Instruction Operation Doubleword (DW0) apply to all instructions. Bit field [27:24] is one exception. It is *CondModifier* for most instructions but is *CurrDest.RegNum* field for the *send* instruction.

Table 13-4. Definitions of Fields in Operation Doubleword (DW0)

Bits	Description																
31	Saturate. This field controls the destination saturation. When it is set, output data to the destination register are saturated. The saturation operation depends on the destination data type. Saturation is the operation that converts any data that is outside the <i>saturation target range</i> for the data type to the closest representable value with the target range. If destination type is float, saturation target range is [0, 1]. For example, any positive number greater than 1 (including +INF) is saturated to 1 and any negative number (including -INF) is saturated to 0. A NaN is saturated to 0. For integer data types, the maximum range for the given numerical data type is the saturation target range. When it is not set, output data to the destination register are not saturated. For example, a wrapped result (modular) is output to the destination for an overflowed integer data. More details can be found in the Data Types chapter. 0 = No destination modification (normal) 1 = “sat”. Saturate the output <table> <tr> <th>Destination Type</th><th>Saturation Target Range (inclusive)</th></tr> <tr> <td>Float (F)</td><td>[0.0, 1.0]</td></tr> <tr> <td>Byte (UB)</td><td>[0, 255]</td></tr> <tr> <td>Signed Byte (B)</td><td>[-128, 127]</td></tr> <tr> <td>Word (UW)</td><td>[0, 65535]</td></tr> <tr> <td>Signed Word (W)</td><td>[-32768, 32767]</td></tr> <tr> <td>Double Word (UD)</td><td>[0, $2^{32}-1$]</td></tr> <tr> <td>Signed Double (D)</td><td>$[-2^{31}, 2^{31}-1]$</td></tr> </table>	Destination Type	Saturation Target Range (inclusive)	Float (F)	[0.0, 1.0]	Byte (UB)	[0, 255]	Signed Byte (B)	[-128, 127]	Word (UW)	[0, 65535]	Signed Word (W)	[-32768, 32767]	Double Word (UD)	[0, $2^{32}-1$]	Signed Double (D)	$[-2^{31}, 2^{31}-1]$
Destination Type	Saturation Target Range (inclusive)																
Float (F)	[0.0, 1.0]																
Byte (UB)	[0, 255]																
Signed Byte (B)	[-128, 127]																
Word (UW)	[0, 65535]																
Signed Word (W)	[-32768, 32767]																
Double Word (UD)	[0, $2^{32}-1$]																
Signed Double (D)	$[-2^{31}, 2^{31}-1]$																
30	DebugCtrl – Debug Control. This field allows the insertion of a breakpoint at the current instruction. When the bit is set, hardware automatically stores the current IP in CR register and jumps to the System IP (SIP) BEFORE executing the current instruction. 0 = No breakpoint (normal) 1 = “Breakpoint”. Breakpoint is inserted before this instruction is executed.																
29	Reserved: MBZ																
28	Reserved: MBZ																



Bits	Description	
27:24	CondModifier or CurrDst.RegNum[3:0] Definition of this bit field depends on whether the instruction is a <i>send</i> or not.	
	Opcode != 'send'	Opcode = 'send'
	CondModifier: This field sets the flag register based on the internal conditional signals output from the execution pipe.	CurrDst.RegNum[3:0] This field sets the MRF register number for the current destination operand in the <i>send</i> instruction. No flag registers are updated for the <i>send</i> instruction. The 4-bit field provides full access of the 16 MRF registers. (See Instruction Reference chapter for <i>CurrDst</i> .)
23:21	ExecSize – Execution Size. This field determines the number of channels operating in parallel for this instruction. The size cannot exceed the maximum number of channels allowed for the given data type. 000 = 1 Channels (scalar operation) 001 = 2 Channels 010 = 4 Channels 011 = 8 Channels 100 = 16 Channels 101= 32 Channels 110-111 = Reserved	
20	PredInv – Predicate Inverse. This field, together with <i>PredCtrl</i> , enables and controls the generation of the predication mask for the instruction. When it is set, the predication uses the inverse of the predication bits generated according to setting of Predicate Control. In other words, effect of PredInv happens after PredCtrl. This field is ignored by hardware if Predicate Control is set to 0000 – there is no predication. 0 = “+”. Positive polarity of predication. 1 = “-”. Negative polarity of predication.	



Bits	Description
19:16	PredCtrl – Predicate Control. This field, together with <i>PredInv</i>, enables and controls the generation of the predication mask for the instruction. It allows per-channel conditional execution of the instruction based on the content of the selected flag register. Encoding depends on the access mode. In Align16 access mode, there are eight encodings (including no predication). All encodings are based on group-of-4 predicate bits, including channel sequential, replication swizzles and horizontal any/all operations. The same configuration is repeated for each group-of-4 execution channels. In Align1 access mode, there are twelve encodings (including no predication). The encodings applies to all execution channels with explicit channel grouping from single channel up to group of 16 channels. Predicate Control in Align16 access mode 0000 = No predication (normal) 0001 = Predication with sequential flag channel mapping 0010 = Predication with replication swizzle ‘.x’ 0011 = Predication with replication swizzle ‘.y’ 0100 = Predication with replication swizzle ‘.z’ 0101 = Predication with replication swizzle ‘.w’ 0110 = Predication with ‘.any4h’ 0111 = Predication with ‘.all4h’ 1000 -1111 = Reserved Predicate Control in Align1 access mode 0000 = No predication (normal) 0001 = Predication with sequential flag channel mapping 0010 = Predication with .anyv (any from f0.0-f0.1 on the same channel) 0011 = Predication with .allv (all of f0.0-f0.1 on the same channel) 0100 = Predication with .any2h (any in group of 2 channels) 0101 = Predication with .all2h (all in group of 2 channels) 0110 = Predication with .any4h (any in group of 4 channels) 0111 = Predication with .all4h (all in group of 4 channels) 1000 = Predication with .any8h (any in group of 8 channels) 1001 = Predication with .all8h (all in group of 8 channels) 1010 = Predication with .any16h (any in group of 16 channels) 1011 = Predication with .all16h (all in group of 16 channels) 1100 -1111 = Reserved



Bits	Description
15:14	ThreadCtrl – Thread Control. This field provides explicit control for thread switching. If this field is set to 00, it is up to the GEN4 execution units to manage thread switching. This is the normal operations mode. In this mode, for example, if the current instruction cannot proceed due to operand dependencies, EU switches to next available thread to fill the compute pipe. In another example, if the current instruction is ready to go, however, there is another thread with higher priority also has instruction ready, EU switches to that thread. If this field is set to Switch, a forced thread switch occurs after the current instruction is executed and before the next instruction. In addition, a long delay (longer than the execution pipe latency) for the current thread is introduced for the thread. Particularly, the instruction queue of the current thread is flushed after the current instruction is dispatched for execution. Switch is designed primarily as a safety feature in case there are race conditions for certain instructions. 00 = Normal Thread Control 10 = “Switch” 11 = Reserved
13:12	ComprCtrl – Compression Control. This field provides explicit control for instruction compression. If it is set to 00, the current instruction is a normal instruction (uncompressed) that can be directly dispatched for execution. A normal instruction most likely takes two instruction clock cycles to complete. If half of an architecture register (such as accumulator, mask registers and mask stack registers) is implicitly used by the instruction, the first half is used. If it is set to SecHalf, the current instruction is a normal instruction (uncompressed) that, if applicable, uses the second half of implicit architecture registers. If it is set to Compr, the current instruction is a compressed instruction that needs to be decompressed into two normal instructions. Hardware set the ComprCtrl field of the first decompressed instruction to 00 and that for the second decompressed instruction to SecHalf. The two native instructions are then dispatched for execution. A compressed instruction normally takes four instruction clock cycles to complete. 00 = Normal (uncompressed) instruction 01 = “SecHalf” 10 = “Compr” 11 = Reserved



Bits	Description
11:10	DepCtrl – Destination Dependency Control. This field selectively disables destination dependency check and clear for this instruction. When it is set to 00, normal destination dependency control is performed for the instruction – hardware checks for destination hazards to ensure data integrity. Specifically, destination register dependency check is conducted before the instruction is made ready for execution. After the instruction is executed, the destination register scoreboard will be cleared when the destination operands retire. When bit 10 is set (NoDDClr), the destination register scoreboard will NOT be cleared when the destination operands retire. When bit 11 is set (NoDDChk), hardware does not check for destination register dependency before the instruction is made ready for execution. NoDDClr and NoDDChk are not mutual exclusive. When this field is not all-zero, hardware does not protect against destination hazards for the instruction. This is typically used to assemble data in a fine grained fashion (e.g. matrix-vector compute with dot-product instructions), where the data integrity is guaranteed by software based on the intended usage of instruction sequences. 00 = Destination dependency checked and cleared (normal) 01 = “NoDDClr”. Destination dependency checked but not cleared 10 = “NoDDChk”. Destination dependency not checked but cleared 11 = “NoDDClr, NoDDChk”. Destination dependency not checked and not cleared
9	MaskCtrl – Mask Control. This field, together with MaskCtrlEx field, determines if conditional masks (A/B/L/CMasks) are used for creating the execution mask. Prediction control is still allowed. Encoding for the two-bit field [MaskCtrlEx, MaskCtrl] is as the following 00 = Use Masks (normal) 01 = “NoMask” – all masks are ignored for EMask creation for this instruction 10 = Reserved 11 = Reserved
8	AccessMode – Access Mode. This field determines the operand access for the instruction. It applies to all source and destination operands. When it is cleared (Align1), the instruction uses byte-aligned addressing for source and destination operands. Source swizzle control and destination mask control are not supported. When it is set (Align16), the instruction uses 16-byte-aligned addressing for all source and destination operands. Source swizzle control and destination mask control are supported in this mode. 0 = “Align1” 1 = “Align16”
7	Reserved: MBZ (for future opcode extension)
6:0	Opcode – Instruction Operation Code. This field contains the instruction operation code. Each opcode is given a unique mnemonic. For example, opcode 0x01 is for a move operation. Mnemonic for this opcode is <i>mov</i>. See Section 13.3 for details of opcode encoding.



13.2.3 Instruction Destination Doubleword (DW1)

Destination Doubleword (DW1) contains the register file and numeric type of all operands, as well as the register region parameters of the destination operand. Table 13-5 shows the field definition of the Instruction Destination Doubleword. Furthermore, the Destination Register Region is described in Table 13-6 through Table 13-9.

Table 13-5. Instruction Destination Doubleword

Bits	Description
31:16	Destination Register Region. This word contains the parameters describing the register region of the destination operand. Subfield definition depends on the AccessMode. Detailed descriptions can be found in Table 13-6 through Table 13-9.
15	Reserved: MBZ
14:12	Src1.SrcType – Source-1 Data Type. This field specifies the numerical data type of the source operand <src1>. The bits of a source operand are interpreted as the identified numerical data type, rather than coerced into a type implied by the operator. Depending on <i>RegFile</i> field of the source operand, there are two different encoding for this field. If a source is a register operand, this field follows the Source Register Type Encoding. If a source is an immediate operand, this field follows the Source Immediate Type Encoding. Source Register Type Encoding is identical to that for Destination Type. Source Immediate Type Encoding differs in two areas. First, it does not support byte and unsigned numerical data types. Secondly, it has two 32-bit vector types – halfbyte integer vector (V) type and exponent-only float vector (VF) type. <i>Implementation Note 1: Both source operands, <src0> and <src1>, support immediate types, but only one immediate is allowed for a given instruction and it must be the last operand.</i> <i>Implementation Note 2: Halfbyte integer vector (v) type can only be used in instructions in packed-word execution mode. Therefore, in a two-source instruction where <src1> is of type :v, <src0> must be of type :b, :ub, :w, or :uw.</i> Source Register Type Encoding 000 = “UD”. Unsigned Doubleword integer 001 = “D”. Signed Doubleword integer 010 = “UW”. Unsigned Word integer 011 = “W”. Signed Word integer 100 = “UB”. Unsigned Byte integer 101 = “B”. Signed Byte integer 110 = Reserved 111 = “F”. Single precision Float (32-bit) Source Immediate Type Encoding: 000 = “UD” 001 = “D” 010 = “UW” 011 = “W” 100 = Reserved 101 = “VF”. 32-bit restricted Vector Float 110 = “V”. 32-bit halfbyte integer Vector 111 = “F”



Bits	Description
11:10	Src1.RegFile – Source-1 Register File. This field identifies the register file of source operand <src1>. 00 = “ARF”. Architecture Register File (a#, acc#, f#, n#, null, ip, etc.) 01 = “GRF”. General Register File (r#) 10 = “MRF”. Message Register File (m#) 11 = “IMM”. Immediate
9:7	Src0.SrcType – Source-0 Data Type. This field is the <i>SrcType</i> for <src0> operand. It has the same definitions as <i>Src1.SrcType</i>.
6:5	Src0.RegFile – Source-0 Register File. This field is the <i>RegFile</i> for <src0> operand. It has the same definitions as <i>Src1.RegFile</i>.
4:2	Dst.DstType – Destination Data Type. This field specifies the numerical data type of the destination operand <dst>. The bits of the destination operand are interpreted as the identified numerical data type, rather than coerced into a type implied by the operator. For a <i>send</i> instruction, this field applies to the CurrDst – the current destination operand. Encoding: 000 = “UD”. Unsigned Doubleword integer 001 = “D”. Signed Doubleword integer 010 = “UW”. Unsigned Word integer 011 = “W”. Signed Word integer 100 = “UB”. Unsigned Byte integer 101 = “B”. Signed Byte integer 110 = Reserved 111 = “F”. Single precision Float (32-bit)
1:0	Dst.RegFile – Destination Register File. This field identifies the register file of the destination operand <dst>. Note that it is obvious that immediate cannot be a destination operand. For a <i>send</i> instruction, this field applies to the PostDst – the post destination operand. Encoding: 00 = “ARF”. Architecture Register File (a#, acc#, f#, n#, null, ip, etc.) 01 = “GRF”. General Register File (r#) 10 = “MRF”. Message Register File (m#) 11 = reserved

The following tables describe the Destination Register Region based on the access mode and addressing mode.



Table 13-6. Destination Register Region in Direct + Align16 mode

Bits	Description
15	Dst.AddrMode – Destination Address Mode. This field is the <i>AddrMode</i> for the destination operand. (See section 13.2.1 for definition of <i>AddrMode</i>.) For a <i>send</i> instruction, this field applies to PostDst – the post destination operand. Addressing mode for <i>CurrDst</i> (current destination operand) is fixed as Direct. (See Instruction Reference chapter for <i>CurrDst</i> and <i>PostDst</i>.)
14:13	Reserved: MBZ
12:5	Dst.RegNum – Destination Register Number. This field is the <i>RegNum</i> field for the destination operand. (See section 13.2.1 for definitions of <i>RegNum</i>.) For a <i>send</i> instruction, this field applies to PostDst.
4	Dst.SubRegNum[4]. This is the 16-byte aligned sub-register address. (See section 13.2.1 for definitions of <i>SubRegNum</i>) For a <i>send</i> instruction, this field applies to CurrDst.
3:0	Dst.ChanEn – Destination Channel Enable. The channel enable field for the destination operand. (See section 13.2.1 for definitions of <i>ChanEn</i>) For a <i>send</i> instruction, this field applies to the CurrDst.

Table 13-7. Destination Register Region in Direct+Align1 mode

Bits	Description
15	Dst.AddrMode – Destination Address Mode. This field is the <i>AddrMode</i> for the destination operand. For a <i>send</i> instruction, it applies to PostDst. Addressing mode for CurrDst is fixed as Direct.
14:13	Dst.HorzStride – Destination Horizontal Stride. This field is the <i>HorzStride</i> for the destination operand. For a <i>send</i> instruction, this field applies to CurrDst. PostDst only uses the register number.
12:5	Dst.RegNum – Destination Register Number. This field is the <i>RegNum</i> field for the destination operand. For a <i>send</i> instruction, this field applies to PostDst.
4:0	Dst.SubRegNum – Destination Sub-Register Number. This field is the SubRegNum for the destination operand. (See section 13.2.1 for definition of <i>SubRegNum</i>) For a <i>send</i> instruction, this field applies to CurrDst.



Table 13-8. Destination Register Region in Indirect+Align16 mode

Bits	Description
15	Dst.AddrMode – Destination Address Mode. This field is the <i>AddrMode</i> for the destination operand. For a <i>send</i> instruction, this field applies to PostDst . Addressing mode for CurrDst is fixed as Direct.
14:13	Reserved: MBZ
12:10	Dst.AddrSubRegNum – Destination Address Sub-Register Number. This field is the <i>AddrSubRegNum</i> for the destination operand. (See section 13.2.1 for definition of <i>AddrSubRegNum</i> .) For a <i>send</i> instruction, this field applies to PostDst .
9:4	Dst.AddrImm[9:4] This is the half-register aligned <i>AddrImm</i> field for the destination operand. (See section 13.2.1 for definition of <i>AddrImm</i>) For a <i>send</i> instruction, this field applies to PostDst .
3:0	Dst.ChanEn – Destination Channel Enable. The channel enable field for the destination operand. For a <i>send</i> instruction, this field applies to the CurrDst .

Table 13-9. Destination Register Region in Indirect+Align1 mode

Bits	Description
15	Dst.AddrMode – Destination Address Mode. This field is the <i>AddrMode</i> for the destination operand. For a <i>send</i> instruction, this field applies to PostDst . Addressing mode for CurrDst is fixed as Direct.
14:13	Dst.HorzStride – Destination Horizontal Stride This field is the <i>HorzStride</i> for the destination operand. For a <i>send</i> instruction, this field applies to CurrDst . PostDst only uses the register number.
12:10	Dst.AddrSubRegNum – Destination Address Sub-Register Number. This field is the <i>AddrSubRegNum</i> for the destination operand. For a <i>send</i> instruction, this field applies to PostDst .
9:0	Dst.AddrImm – Destination Address Immediate. This field is the byte-aligned <i>AddrImm</i> for the destination operand. For a <i>send</i> instruction, this field applies to PostDst .

13.2.4 Instruction Source-0 Doubleword (DW2)

Instruction Source-0 Doubleword (DW2) contains the first source operand and also flag register number.

- Table 13-10 shows the field definition for Direct Addressing with Align16.
- Table 13-11 shows the field definition for Direct Addressing with Align1.
- Table 13-12 shows the field definition for Indirect Addressing with Align16.
- Table 13-13 shows the field definition for Indirect Addressing with Align1.

Table 13-10. Instruction Source-0 Doubleword in Direct+Align16 mode

Bits	Description
31:26	Reserved: MBZ
25	FlagSubRegNum – Flag Sub-Register Number. This field specifies the sub-register number for a flag register operand. There are two sub-registers in the flag register. Each sub-register contains 16 flag bits. The selected flag sub-register is the source for predication if predication is enabled for the instruction. It is the destination to store conditional flag bits if conditional modifier is enabled for the instruction. The same flag sub-register can be both the predication source and conditional destination, if both predication and conditional modifier are enabled.
24:21	Src0.VertStride – Source-0 Vertical Stride. This field is the <i>VertStride</i> for <src0> operand. (See section 13.2.1 for definition of <i>VertStride</i>) It is ignored if <src0> is an immediate operand.
20	Reserved: MBZ
19:16	Src0.ChanSel[7:4] This is bits [7:4] of the <i>ChanSel</i> field for <src0> operand. (See section 13.2.1 for definition of <i>ChanSel</i>). It is ignored if <src0> is an immediate operand.
15	Src0.AddrMode – Source-0 Address Mode. This field is the <i>AddrMode</i> for <src0> operand. (See section 13.2.1 for definition of <i>AddrMode</i>) It is ignored if <src0> is an immediate operand.
14:13	Src0.SrcMod – Source-0 Source Modifier. This field is the <i>SrcMod</i> for source operand <src0>. (See section 13.2.1 for definition of <i>SrcMod</i>) It is ignored if <src0> is an immediate operand.
12:5	Src0.RegNum – Source-0 Register Number This is the <i>RegNum</i> field for source operand <src0>. (See section 13.2.1 for definition of <i>RegNum</i> .) It is ignored if <src0> is an immediate operand.
4	Src0.SubRegNum[4] This is the 16-byte aligned sub-register address for source operand <src0>. (See section 13.2.1 for definition of <i>SubRegNum</i>) It is ignored if <src0> is an immediate operand.
3:0	Src0.ChanEn – Source-0 Channel Enable This is the <i>ChanEn</i> field for source operand <src0>. (See section 13.2.1 for definitions of <i>ChanEn</i>) It is ignored if <src0> is an immediate operand.



Table 13-11. Instruction Source-0 Doubleword in Direct+Align1 mode

Bits	Description
31:26	Reserved: MBZ
25	FlagSubRegNum – Flag Sub-Register Number. This field specifies the sub-register number for a flag register operand.
24:21	Src0.VertStride – Source-0 Vertical Stride This is the <i>VertStride</i> field for <src0> operand. (See section 13.2.1 for definition of <i>VertStride</i>) It is ignored if <src0> is an immediate operand.
20:18	Src0.Width. This is the <i>Width</i> field for source operand <src0>. (See section 13.2.1 for definition of <i>Width</i>) It is ignored if <src0> is an immediate operand.
17:16	Src0.HorzStride. This is the <i>HorzStride</i> field for source operand <src0>. (See section 13.2.1 for definition of <i>HorzStride</i>) It is ignored if <src0> is an immediate operand.
15	Src0.AddrMode – Source-0 Address Mode. This is the <i>AddrMode</i> for source operand <src0>. (See section 13.2.1 for definition of <i>AddrMode</i>) It is ignored if <src0> is an immediate operand.
14:13	Src0.SrcMod – Source-0 Source Modifier. This is the <i>SrcMod</i> field for source operand <src0>. (See section 13.2.1 for definition of <i>SrcMod</i>) It is ignored if <src0> is an immediate operand.
12:5	Src0.RegNum – Source-0 Register Number. This is the <i>RegNum</i> field for source operand <src0>. (See section 13.2.1 for definition of <i>RegNum</i> .) It is ignored if <src0> is an immediate operand.
4:0	Src0.SubRegNum – Source-0 Sub-Register Number. This is the <i>SubRegNum</i> field for source operand <src0>. (See section 13.2.1 for definition of <i>SubRegNum</i>) It is ignored if <src0> is an immediate operand.



Table 13-12. Instruction Source-0 Doubleword in Indirect+Align16 mode

Bits	Description
31:26	Reserved: MBZ
25	FlagSubRegNum – Flag Sub-Register Number. This field specifies the sub-register number for a flag register operand.
24:21	Src0.VertStride – Source-0 Vertical Stride. This is the <i>VertStride</i> field for <src0> operand. (See section 13.2.1 for definition of <i>VertStride</i>) It is ignored if <src0> is an immediate operand.
20	Reserved: MBZ
19:16	Src0.ChanSel[7:4] – Source-0 Channel Select. This is bits [7:4] of the <i>ChanSel</i> field for <src0> operand. (See section 13.2.1 for definition of <i>ChanSel</i>). It is ignored if <src0> is an immediate operand.
15	Src0.AddrMode – Source-0 Address Mode. This is the <i>AddrMode</i> for source operand <src0>. (See section 13.2.1 for definition of <i>AddrMode</i>) It is ignored if <src0> is an immediate operand.
14:13	Src0.SrcMod – Source-0 Source Modifier. This is the <i>SrcMod</i> field for source operand <src0>. (See section 13.2.1 for definition of <i>SrcMod</i>) It is ignored if <src0> is an immediate operand.
12:10	Src0.AddrSubRegNum – Source-0 Address Sub-Register Number. This is the <i>AddrSubRegNum</i> field for source operand <src0>. (See section 13.2.1 for definition of <i>AddrSubRegNum</i> .) It is ignored if <src0> is an immediate operand.
9:4	Src0.AddrImm[9:4] – Source-0 Address Immediate. This contains the half-register aligned <i>AddrImm</i> field ((bits [9:4]) for <src0>. (See section 13.2.1 for definition of <i>AddrImm</i>) It is ignored if <src0> is an immediate operand.
3:0	Src0.ChanEn – Source-0 Channel Enable . This is the <i>ChanEn</i> field for source operand <src0>. (See section 13.2.1 for definitions of <i>ChanEn</i>) It is ignored if <src0> is an immediate operand.



Table 13-13. Instruction Source-0 Doubleword in Indirect+Align1 mode

Bits	Description
31:26	Reserved: MBZ
25	FlagSubRegNum – Flag Sub-Register Number. This field specifies the sub-register number for a flag register operand.
24:21	Src0.VertStride – Source-0 Vertical Stride. This is the <i>VertStride</i> field for <src0> operand. (See section 13.2.1 for definition of <i>VertStride</i>) It is ignored if <src0> is an immediate operand.
20:18	Src0.Width. This is the <i>Width</i> field for source operand <src0>. (See section 13.2.1 for definition of <i>Width</i>) It is ignored if <src0> is an immediate operand.
17:16	Src0.HorzStride. This is the <i>HorzStride</i> field for source operand <src0>. (See section 13.2.1 for definition of <i>HorzStride</i>) It is ignored if <src0> is an immediate operand.
15	Src0.AddrMode – Source-0 Address Mode. This is the <i>AddrMode</i> for source operand <src0>. (See section 13.2.1 for definition of <i>AddrMode</i>) It is ignored if <src0> is an immediate operand.
14:13	Src0.SrcMod – Source-0 Source Modifier. This is the <i>SrcMod</i> field for source operand <src0>. (See section 13.2.1 for definition of <i>SrcMod</i>) It is ignored if <src0> is an immediate operand.
12:10	Src0.AddrSubRegNum – Source-0 Address Sub-Register Number. This is the <i>AddrSubRegNum</i> field for source operand <src0>. (See section 13.2.1 for definition of <i>AddrSubRegNum</i> .) It is ignored if <src0> is an immediate operand.
9:0	Src0.AddrImm – Source-0 Address Immediate. This is the byte aligned <i>AddrImm</i> field for <src0>. (See section 13.2.1 for definition of <i>AddrImm</i>) It is ignored if <src0> is an immediate operand.



13.2.5 Instruction Source-1 Doubleword (DW3)

Source-1 Doubleword (DW3) contains the second source operand (<src1>) and is used to hold the 32-bit immediate source (imm32 as <src0> or <src1>). Table 13-14 and Table 13-15 define the fields in this doubleword with the following exceptions:

- If <src0> is an immediate operand, this doubleword contains **imm32** for <src0>.
- If <src1> is an immediate operand, this doubleword contains **imm32** for <src1>.
- If the instruction is a send, bit 31 of this doubleword contains **EOT** field.
 - If <src1> is immediate, the remaining 31 bits in this doubleword is **MsgDescpt31**.
 - If <src1> is a register, <src1> must be a0.0. The rest of this doubleword will be configured accordingly.
- If indirect address is supported for <src1>, Table 13-16 and Table 13-17 define the fields in DW3 for indirectly addressed <src1> in Align16 and Align1 modes.

Table 13-14. Instruction Source-1 Doubleword in Direct + Align16 mode

Bits	Description
31:25	Reserved: MBZ
24:21	Src1.VertStride – Source-1 Vertical Stride. This field is the <i>VertStride</i> for <src1> operand. (See section 13.2.1 for definition of <i>VertStride</i>) It is ignored if <src1> is an immediate operand.
20	Reserved: MBZ
19:16	Src1.ChanSel[7:4] This contains bits [7:6] of the <i>ChanSel</i> field for <src1> operand. (See section 13.2.1 for definition of <i>ChanSel</i>) It is ignored if <src1> is an immediate operand.
15	Reserved: MBZ
14:13	Src1.SrcMod – Source-1 Source Modifier. This field is the <i>SrcMod</i> for <src1> operand. (See section 13.2.1 for definition of <i>SrcMod</i>) It is ignored if <src1> is an immediate operand.
12:5	Src1.RegNum. This field is the <i>RegNum</i> field for <src1> operand. (See section 13.2.1 for definition of <i>RegNum</i> .) It is ignored if <src1> is an immediate operand.
4	Src1.SubRegNum[4]. This field is bit [4] of the <i>SubRegNum</i> field for <src1>. (See section 13.2.1 for definition of <i>SubRegNum</i>) It is ignored if <src1> is an immediate operand.
3:0	Src1.ChanEn – Source-1 Channel Enable. It is the channel enable field for <src1>. (See section 13.2.1 for definitions of <i>ChanEn</i>) It is ignored if <src1> is an immediate operand.



Table 13-15. Instruction Source-1 Doubleword in Direct + Align1 mode

Bits	Description
31:25	Reserved: MBZ
24:21	Src1.VertStride – Source-1 Vertical Stride. This field is the <i>VertStride</i> for <src1> operand. (See section 13.2.1 for definition of <i>VertStride</i>) It is ignored if <src1> is an immediate operand.
20:18	Src1.Width. This is the <i>Width</i> field for source operand <src1>. (See section 13.2.1 for definition of <i>Width</i>) It is ignored if <src1> is an immediate operand.
17:16	Src1.HorzStride. This is the <i>HorzStride</i> field for source operand <src1>. (See section 13.2.1 for definition of <i>HorzStride</i>) It is ignored if <src1> is an immediate operand.
15	Reserved: MBZ
14:13	Src1.SrcMod – Source-1 Source Modifier. This field is the <i>SrcMod</i> for <src1> operand. (See section 13.2.1 for definition of <i>SrcMod</i>) It is ignored if <src1> is an immediate operand.
12:5	Src1.RegNum – Source-1 Register Number. This is the <i>RegNum</i> field for source operand <src1>. (See section 13.2.1 for definition of <i>RegNum</i> .) It is ignored if <src1> is an immediate operand.
4:0	Src1.SubRegNum – Source-1 Sub-Register Number. This is the <i>SubRegNum</i> field for source operand <src1>. (See section 13.2.1 for definition of <i>SubRegNum</i>) It is ignored if <src1> is an immediate operand.



Table 13-16. Instruction Source-1 Doubleword in Indirect+Align16 mode

Bits	Description
31:25	Reserved: MBZ
24:21	Src1.VertStride – Source-1 Vertical Stride This is the <i>VertStride</i> field for <src1> operand. (See section 13.2.1 for definition of <i>VertStride</i>) It is ignored if <src1> is an immediate operand.
20	Reserved: MBZ
19:16	Src1.ChanSel[7:4] – Source-1 Channel Select This is bits [7:4] of the <i>ChanSel</i> field for <src1> operand. (See section 13.2.1 for definition of <i>ChanSel</i>). It is ignored if <src1> is an immediate operand.
15	Src1.AddrMode – Source-1 Address Mode This is the <i>AddrMode</i> for source operand <src1>. (See section 13.2.1 for definition of <i>AddrMode</i>) It is ignored if <src1> is an immediate operand.
14:13	Src1.SrcMod – Source-1 Source Modifier This is the <i>SrcMod</i> field for source operand <src1>. (See section 13.2.1 for definition of <i>SrcMod</i>) It is ignored if <src1> is an immediate operand.
12:10	Src1.AddrSubRegNum – Source-1 Address Sub-Register Number This is the <i>AddrSubRegNum</i> field for source operand <src1>. (See section 13.2.1 for definition of <i>AddrSubRegNum</i> .) It is ignored if <src1> is an immediate operand.
9:4	Src1.AddrImm[9:4] – Source-1 Address Immediate This contains the half-register aligned <i>AddrImm</i> field ((bits [9:4]) for <src1>. (See section 13.2.1 for definition of <i>AddrImm</i>) It is ignored if <src1> is an immediate operand.
3:0	Src1.ChanEn – Source-1 Channel Enable This is the <i>ChanEn</i> field for source operand <src1>. (See section 13.2.1 for definitions of <i>ChanEn</i>) It is ignored if <src1> is an immediate operand.



Table 13-17. Instruction Source-1 Doubleword in Indirect+Align1 mode

Bits	Description
31:25	Reserved: MBZ
24:21	Src1.VertStride – Source-1 Vertical Stride This is the <i>VertStride</i> field for <src1> operand. (See section 13.2.1 for definition of <i>VertStride</i>) It is ignored if <src1> is an immediate operand.
20:18	Src1.Width This is the <i>Width</i> field for source operand <src1>. (See section 13.2.1 for definition of <i>Width</i>) It is ignored if <src1> is an immediate operand.
17:16	Src1.HorzStride This is the <i>HorzStride</i> field for source operand <src1>. (See section 13.2.1 for definition of <i>HorzStride</i>) It is ignored if <src1> is an immediate operand.
15	Src1.AddrMode – Source-1 Address Mode This is the <i>AddrMode</i> for source operand <src1>. (See section 13.2.1 for definition of <i>AddrMode</i>) It is ignored if <src1> is an immediate operand.
14:13	Src1.SrcMod – Source-1 Source Modifier This is the <i>SrcMod</i> field for source operand <src1>. (See section 13.2.1 for definition of <i>SrcMod</i>) It is ignored if <src1> is an immediate operand.
12:10	Src1.AddrSubRegNum – Source-1 Address Sub-Register Number This is the <i>AddrSubRegNum</i> field for source operand <src1>. (See section 13.2.1 for definition of <i>AddrSubRegNum</i> .) It is ignored if <src1> is an immediate operand.
9:0	Src1.AddrImm – Source-1 Address Immediate This is the byte aligned <i>AddrImm</i> field for <src1>. (See section 13.2.1 for definition of <i>AddrImm</i>) It is ignored if <src1> is an immediate operand.



Table 13-18. GEN4 Compacted Instruction Format

DW #	Instr Bits Alloc	High Bit	Low Bit	Instr Bits Used	Description
1	8	63	56	8	Bits[108:101] Source1 register number
	8	55	48	8	Bits [76:69] Source0 register number
	8	47	40	8	Bits [60:53] Destination register number
	5	39	35	5	Src1 Index[4:0]
	3	34	32	3	Src0 Index[4:2]
0	2	31	30	2	Src0 Index[1:0]
	1	29	29	1	CompactCtrl – Compaction Control
	1	28	28	1	Bit [89] FlagSubRegNum
	4	27	24	4	Bits [27:24] CondModifier
	1	23	23	1	MaskCtrlEx - Mask Control Extension
	5	22	18	5	Subregister Index[4:0]
	3	17	13	5	Datatype Index[4:0]
	3	12	8	5	Control Index[4:0]
	5	7	7	1	Bit[30] - Debug Control
	7	6	0	7	Bits[6:0] - Opcode

Table 13-19. Definitions of Fields in the Compact Instruction

Bits	Description
63:56	Bits [108:101] Source1 register number forms bits [108:101], the source 1 register number field.
55:48	Bits [76:69] Source0 register number This field, after unpacking, forms bits [76:69], the source 0 register number field, of the 128-bit instruction word.
47:40	Bits [60:53] Destination register number This field, after unpacking, forms bits [60:53], the destination register number field, of the 128-bit instruction word.
39:35	Src1Index The 5-bit index for source 1. The 12-bit table-look-up result forms bits [120:109], the source 1 register region fields, of the 128-bit instruction word
34:30	Src0Index The 5-bit index for source 0. The 12-bit table-look-up result forms bits [88:77], the source 0 register region fields, of the 128-bit instruction word.



Bits	Description
29	CompactCtrl – Compaction Control This field indicates whether the instruction is in the 64-bit compaction form. When this bit is set (bit 29 of DW0), the instruction length is only 64-bit.. The bit location is fixed in both 128-bit and 64-bit instruction forms. 0 = 128-bit form (normal) 1 = 64-bit compaction form
28	Bit [89] – FlagSubRegNum This field, after unpacking, is bit [89] of the 128-bit instruction word.
27:24	Bits [27:24] – CondModifier This field, after unpacking, is bits [27:24] of the 128-bit instruction word. The bit location is fixed in both 128-bit and 64-bit instruction forms.
23	MaskCtrlEx – Mask Control Extension. This field is an extension of the MaskCtrl field, see MaskCtrl field for detailed definition. This field, after unpacking, is bit[28] of the 128-bit instruction word.
22:18	SubRegIndex The 5-bit index for sub-register fields. The 15-bit table-look-up result forms bits [100:96], [68,64] and [52,48] of the 128-bit instruction word.
17:13	DataTypeIndex The 5-bit index for data type fields. The 18-bit table-look-up result forms bits [63:61] and [46, 32] of the 128-bit instruction word.
12:8	ControlIndex The 5-bit index for data type fields. The 17-bit table-look-up result forms bits[31], and [23, 8] of the 128-bit instruction word.
7	Bits [30] – DebugCtrl This field, after unpacking, is bit [30] of the 128-bit instruction word.
6:0	Bits [6:0] – Opcode This field, after unpacking, is bits [6:0] of the 128-bit instruction word. The bit location is fixed in both 128-bit and 64-bit instruction forms.



13.3 Opcode Encoding

Byte 0 of the 128-bit instruction word contains the opcode. The opcode uses 7 bits. Bit location 7 in byte 0 is reserved for future opcode extension.

There are total of 48 opcodes defined. These opcodes are encoded and organized into five groups based on the type of operations: Special instructions, move/logic instructions (opcode=00xxxxb), flow control instructions (opcode=010xxxxb), miscellaneous instructions (opcode=011xxxxb), parallel arithmetic instructions (opcode=100xxxxb), and vector arithmetic instructions (opcode=101xxxxb). Opcodes 110xxxxb are reserved.

13.3.1 Move and Logic Instructions

This instruction group has an opcode format of 00xxxxb.

- The opcodes for move instructions (**mov**, **sel** and **movi**) share the common 5 MSBs in the form of 00000xb.
- The opcodes for logic instructions (**not**, **and**, **or**, and **xor**) share the common 5 MSBs in the form of 00001xb.
- The opcodes for shift instructions (**shr**, **shl**, and **asr**) share the common 4 MSBs in the form of 0001xxxxb. Bit 2 indicates arithmetic or logic shift (0 = logic, 1 = arithmetic). Bit 1 is always 0 (which is reserved for future extension to support rotation shift as 0 = shift, 1 = rotate). Bit 0 indicates the shift direction (0 = right, 1 = left).
- The opcodes for compare instructions (**cmp** and **cmpn**) share the common 6 MSBs in the form of 001000xb. Bit 0 indicates whether it is a normal compare, *cmp*, or a special compare-NaN, *cmpn*.
- This group of instructions does not implicitly update the accumulators.
- Instruction compression applies to this group.



Table 13-20. Move and Logic Instructions

Opcode		Instruction	Description	#src	#dst
dec	hex				
1	0x01	mov	Component-wise move	1	1
2	0x02	sel	Component-wise selective move based on predication	2	1
3	0x03	<i>Reserved</i>			
4	0x04	not	Component-wise one's compliment (bitwise not)	1	1
5	0x05	and	Component-wise logical AND (bitwise and)	2	1
6	0x06	or	Component-wise logical OR (bitwise or)	2	1
7	0x07	xor	Component-wise logical XOR (bitwise xor)	2	1
8	0x08	shr	Component-wise logical shift right	2	1
9	0x09	shl	Component-wise logical shift left	2	1
10-11	0x0A-0x0B	<i>Reserved</i>			
12	0x0C	asr	Component-wise arithmetic shift right	2	1
13-15	0x0D-0x0F	<i>Reserved</i>			
16	0x10	cmp	Component-wise compare, store condition code in destination	2	1
17	0x11	cmpn	Component-wise compare-NaN, store condition code in destination	2	1
20-31	0x12-0x1F	<i>Reserved</i>			

13.3.2 Flow Control Instructions

This instruction group has an opcode format of 010xxxxb.

- This group of instructions does not implicitly update the accumulators.
- Instruction compression is not allowed for this group.

Table 13-21. Flow Control Instructions

Opcode		Instruction	Description	#src	#dst
dec	hex				
32	0x20	<i>jmp</i>	Jump indexed	1	0
33	0x21	<i>Reserved</i>			
34	0x22	<i>if</i>	If	0/2	0
35	0x23	<i>iff</i>	Fast if	1	0
36	0x24	<i>else</i>	Else	1	0
37	0x25	<i>endif</i>	End if	0	0
38	0x26	<i>do</i>	Do	0	0
39	0x27	<i>while</i>	While	1	0
40	0x28	<i>break</i>	Break	1	0
41	0x29	<i>cont</i>	Continue	1	0
42	0x2A	<i>halt</i>	Halt	1	0
43	0x2B	<i>Reserved</i>			
44	0x2C	<i>msave</i>	Push mask to stack and update mask	1	1
45	0x2D	<i>mrest</i>	Pop mask stack and restore mask	1	1
46	0x2E	<i>push</i>	Push mask to stack without updating mask	1	1
46	0x2E	<i>Reserved</i>			
47	0x2F	<i>pop</i>	Pop mask stack without updating mask	2	0



13.3.3 Miscellaneous Instructions

This instruction group has an opcode format of 011xxxxb.

- This group of instructions does not implicitly update the accumulators.
- Instruction compression is not allowed for this group.

Table 13-22. Miscellaneous Instructions

Opcode		Instruction	Description	#src	#dst
dec	hex				
48	0x30	<i>wait</i>	Wait for (external) notification	1	0
49	0x31	<i>send</i>	Send	1	1
51-55	0x33-0x37	<i>Reserved</i>			
57-63	0x39-0x3F	<i>Reserved</i>			



13.3.4 Parallel Arithmetic Instructions

This instruction group has an opcode format of 100xxxxb.

- The opcode for round instructions (***rndu***, ***rndd***, ***rnde***, and ***rndz***) share the common 5 MSBs in the form of 10001xxb, with the lower 2 bits indicate the type of round.
- These instructions implicitly update the accumulators if the Accumulator Disable bit in control register cr0.0 not set.
 - Some instructions such as ***frc***, ***lzd***, etc., perform the operation after the accumulator. Therefore, when the accumulator is implicitly updated, the content is undefined. Details can be found in ISA Reference Chapter.
- Instruction compression applies to this group.

Table 13-23. Parallel Arithmetic Instructions

Opcode		Instruction	Description	#src	#dst
dec	hex				
64	0x40	<i>add</i>	Component-wise addition	2	1
65	0x41	<i>mul</i>	Component-wise multiply	2	1
66	0x42	<i>avg</i>	Component-wise average of the two source operands	2	1
67	0x43	<i>frc</i>	Component-wise floating point truncate-to-minus-infinity fraction	1	1
68	0x44	<i>rndu</i>	Component-wise floating point rounding up (ceiling)	1	1
69	0x45	<i>rndd</i>	Component-wise floating point rounding down (floor)	1	1
70	0x46	<i>rnde</i>	Component-wise floating point rounding toward nearest even	1	1
71	0x47	<i>rndz</i>	Component-wise floating point rounding toward zero	1	1
72	0x48	<i>mac</i>	Component-wise multiply accumulate	2	1
73	0x49	<i>mach</i>	multiply accumulate high	2	1
74	0x4A	<i>lzd</i>	leading zero detection	1	1
75-79	0x4B-0x4F	<i>Reserved</i>			



13.3.5 Vector Arithmetic Instructions

- This instruction group has an opcode format of 101xxxxb.
- These instructions implicitly update the accumulators if the Accumulator Disable bit in control register cr0.0 not set.
 - Some instructions such as **dp4-dp2**, etc., perform the operation after the accumulator. Therefore, when the accumulator is implicitly updated, the content is undefined. Details can be found in ISA Reference Chapter.
- Instruction compression applies to this group.

Table 13-24. Vector Arithmetic Instructions

Opcode		Instruction	Description	#src	#dst
dec	hex				
80	0x50	sad2	2-wide sum of absolute difference	2	1
81	0x51	sada2	2-wide sad accumulate	2	1
82-83	0x52-0x53	<i>reserved</i>			
84	0x54	dp4	4-wide dot product for 4-vector	2	1
85	0x55	dph	4-wide homogenous dot product for 4-vector	2	1
86	0x56	dp3	3-wide dot product for 4-vector	2	1
87	0x57	dp2	2-wide dot product for 4-vector	2	1
88	0x58	<i>reserved</i>			
89	0x59	line	Component-wise line equation computation (a multiply-add)	2	1
90	0x5A	<i>reserved</i>			
91	0x5B	<i>reserved</i>			
92-95	0x5C-0x5F	<i>reserved</i>			



13.3.6 Special Instructions

There are two special instructions, namely, *nop* (opcode = 0x7E) and *illegal* (opcode = 0x00).

- *Nop* instruction may be used for instruction padding in memory between two normal instructions to force alignment or to introduce instruction execution delay. Currently, there is no need for between-instruction padding.
- *Illegal* instruction may be used for instruction padding in memory outside the normal instruction sequence such as before or after the kernel program as well as between subroutines. It is useful for debugging purpose to signal when a thread execution hits a branch that erroneously causes the IP to reach an instruction memory location outside the program.
- *Nop* and *illegal* instructions do not have source operands or destination operand. Therefore, they do not implicitly update the accumulator register. They cannot be compressed.

Table 13-25. Special Instructions

Opcode		Instruction	Description	#src	#dst
dec	hex				
0	0x00	<i>illegal</i>	Illegal instruction	0	0
96-124	0x60-0x7C	<i>Reserved</i>			
125	0x7D	<i>reserved</i>			
126	0x7E	<i>nop</i>	No-op	0	0
127	0x7F	<i>Reserved</i>	(may be used as an extension code)		



13.4 Native Instruction BNF

The Backus-Naur Form (BNF) grammar identifies the assembly language syntax, which is native to the hardware. It does not include intelligent defaults, assembler pragmas, etc.

13.4.1 Instruction Groups

<Instruction>	::=	<UnaryInstruction> <BinaryAccInstruction> <BinaryInstruction> <TriInstruction> <JumpInstruction> <BranchLoopInstruction> <ElseInstruction> <BreakInstruction> <MaskControlInstruction> <SyncInstruction> <SpecialInstruction>
<UnaryInstruction>	::=	<Predicate> <UnaryInst> <ExecSize> <Dst> <SrcAcclmm> <InstOptions>
<UnaryInst>	::=	<UnaryOp> <ConditionalModifier> <Saturate>
<UnaryOp>	::=	"mov" "frc" "rdu" "rdd" "rnde" "rndz" "not" "lzd"
<BinaryInstruction>	::=	<Predicate> <BinaryInst> <ExecSize> <Dst> <Src> <SrcLmm> <InstOptions>
<BinaryInst>	::=	<BinaryOp> <ConditionalModifier> <Saturate>
<BinaryOp>	::=	"mul" "mac" "mach" "line" "sad2" "sada2" "dp4" "dph" "dp3" "dp2"
<BinaryAccInstruction>	::=	<Predicate> <BinaryAccInst> <ExecSize> <Dst> <SrcAcc> <SrcLmm> <InstrOptions>
<BinaryAccInst>	::=	<BinaryAccOp> <ConditionalModifier> <Saturate>
<BinaryAccOp>	::=	"avg" "add" "sel" "and" "or" "xor" "shr" "shl" "asr" "cmp" "cmpn"
<TriInstruction>	::=	<Predicate> <TriInst> <ExecSize> <PostDst> <CurrDst> <TriSrc> <MsgDesc> <InstOptions>
<TriInst>	::=	<TriOp> <ConditionalModifier> <Saturate>
<TriOp>	::=	"send"
<JumpInstruction>	::=	<JumpOp> <RelativeLocation2>
<JumpOp>	::=	"jmp"



<BranchLoopInstruction>	::=	<Predicate> <BranchLoopOp> <RelativeLocation>
<BranchLoopOp>	::=	"if" "iff" "while"
<ElseInstruction>	::=	<ElseOp> <RelativeLocation>
<ElseOp>	::=	"else"
<BreakInstruction>	::=	<Predicate> <BreakOp> <LocationStackCtrl>
<BreakOp>	::=	"break" "cont" "halt"
<MaskControllInstruction>	::=	<MaskPushOp> <MaskStackReg> <MaskReg> <MaskPopOp> <MaskReg> <MaskStackReg> <MaskPopOpEx> <MaskStackReg> <Imm32>
<MaskPushOp>	::=	"msave" "push"
<MaskPopOp>	::=	"mrest"
<MaskPopOpEx>	::=	"pop"
<SyncInstruction>	::=	<Predicate> <SyncOp> <NotifyReg>
<SyncOp>	::=	"wait"
<SpecialInstruction>	::=	"do" "endif" "nop" "illegal"

13.4.2 Destination Register

<Dst>	::=	<DstOperand> <DstOperandEx>
<DstOperand>	::=	<DstReg> <DstRegion> <WriteMask> <DstType>
<DstOperandEx>	::=	<AccReg> <DstRegion> <DstType> <FlagReg> <DstRegion> <DstType> <AddrReg> <DstRegion> <DstType> <MaskReg> <DstRegion> <DstType> <MaskStackReg> <ControlReg> <IPReg> <NullReg>
<DstReg>	::=	<DirectGenReg> <IndirectGenReg> <DirectMsgReg> <IndirectMsgReg>
<PostDst>	::=	<PostDstReg> <DstRegion> <WriteMask> <DstType> <NullReg>
<PostDstReg>	::=	<DirectGenReg> <IndirectGenReg>
<CurrDst>	::=	<DirectAlignedMsgReg>



13.4.3 Source Register

Source with Accumulator Access and with Immediate

<SrcAccImm>	::=	<SrcAcc> <Imm32> <SrcImmType>
<SrcAcc>	::=	<DirectSrcAccOperand> <IndirectSrcOperand>
<DirectSrcAccOperand>	::=	<DirectSrcOperand> <SrcArcOperandEx> <AccReg> <SrcType>
<SrcArcOperandEx>	::=	<FlagReg> <Region> <SrcType> <AddrReg> <Region> <SrcType> <MaskReg> <Region> <SrcType> <MaskStackReg> <ControlReg> <StateReg> <NotifyReg> <IPReg> <NullReg>
<IndirectSrcOperand>	::=	<SrcModifier> <IndirectGenReg> <IndirectRegion> <Swizzle> <SrcType>

Source without Accumulator Access

<Src>	::=	<DirectSrcOperand> <IndirectSrcOperand>
< DirectSrcOperand >	::=	<SrcModifier> <DirectGenReg> <Region> <Swizzle> <SrcType> <SrcArcOperandEx>
<TriSrc>	::=	<SrcModifier> <DirectGenReg> <Region> <Swizzle> <SrcType> <NullReg>
<MsgDesc>	::=	<ImmDesc> <Reg32>
<Reg32>	::=	<DirectGenReg> <Region> <SrcType>

Source without Accumulator Access or IP Access

<SrcImm>	::=	<DirectSrcOperand> <Imm32> <SrcImmType>



13.4.4 Address Registers

<AddrParam>	::=	<AddrReg> <ImmAddrOffset>
<ImmAddrOffset>	::=	""
		"<ImmAddrNum>"

13.4.5 Register Files and Register Numbers

<DirectGenReg>	::=	<GenRegFile> <GenRegNum> <GenSubRegNum>
<IndirectGenReg>	::=	<GenRegFile> "[" <AddrParam> "]"
<GenRegFile>	::=	"r"
<GenRegNum>	::=	"0" ... "127"
<GenSubRegNum>	::=	"
		".0" ... ".7"
		".0" ... ".15"
		".0" ... ".31"

<DirectMsgReg>	::=	<DirectAlignedMsgReg> <MsgSubRegNum>
<DirectAlignedMsgReg>	::=	<MsgRegFile> <MsgRegNum>
<IndirectMsgReg>	::=	<MsgRegFile> "[" <AddrParam> "]"
<MsgRegFile>	::=	"m"
<MsgRegNum>	::=	"0" ... "15"
<MsgSubRegNum>	::=	<GenSubRegNum>

<AddrReg>	::=	<AddrRegFile> <AddrSubRegNum>
<AddrRegFile>	::=	"a0"
<AddrSubRegNum>	::=	"
		".0" ... ".7"

<AccReg>	::=	"acc" <AccRegNum> <AccSubRegNum>
<AccRegNum>	::=	"0" "1"
<AccSubRegNum>	::=	<GenSubRegNum>

<FlagReg>	::=	"f0" <FlagSubRegNum>
<FlagSubRegNum>	::=	"
		".0" ... ".1"

<MaskReg>	::=	"Mask0" <MaskSubRegNum>
		"AMask" "IMask" "LMask" "CMask"
<MaskSubRegNum>	::=	"
		".0" ... ".3"



<MaskStackReg>	::=	"ms0" <MaskStackSubRegNum> "ims" "lms"
<MaskStackSubRegNum>	::=	"" ".0" ".16"
<MaskStackDepthReg>	::=	"MSD0" <MaskStackDepthSubRegNum> "IMSD" "LMSD"
<MaskStackDepthSubRegNum>	::=	"" ".0" ... ".1"
<NotifyReg>	::=	"n" <NotifyRegNum>
<NotifyRegNum>	::=	"0" ... "1"
<StateReg>	::=	"sr0" <StateSubRegNum>
<StateSubRegNum>	::=	".0" ... ".1"
<ControlReg>	::=	"cr0" <ControlSubRegNum>
<ControlSubRegNum>	::=	".0" ... ".2"
<IPReg>	::=	"ip"
<NullReg>	::=	"null"

13.4.6 Relative Location and Stack Control

<RelativeLocation>	::=	<imm16>
<RelativeLocation2>	::=	<imm32> <reg32>
<LocationStackCtrl>	::=	<imm32>

13.4.7 Regions

<DstRegion>	::=	"<" <HorzStride> ">"
<IndirectRegion>	::=	<Region> <RegionWH> <RegionV>
<Region>	::=	"<" <VertStride> ";" <Width> ";" <HorzStride> ">"
<RegionWH>	::=	"<" <Width> ";" <HorzStride> ">"
<RegionV>	::=	"<" <VertStride> ">"
<VertStride>	::=	"0" "1" "2" "4" "8" "16" "32"
<Width>	::=	"1" "2" "4" "8" "16"
<HorzStride>	::=	"0" "1" "2" "4"



13.4.8 Types

<SrcType>	::=	“:f” “:ud” “:d” “:uw” “:w” “:ub” “:b”
<SrcImmType>	::=	<SrcType> “:v” “:vf”
<DstType>	::=	<SrcType>

13.4.9 Write Mask

<WriteMask>	::=	“” “.” “x” “.” “y” “.” “z” “.” “w” “.” “xy” “.” “xz” “.” “xw” “.” “yz” “.” “yw” “.” “zw” “.” “xyz” “.” “xyw” “.” “xzw” “.” “yzw” “.” “xyzw”

13.4.10 Swizzle Control

<Swizzle>	::=	“” “.” <ChanSel> “.” <ChanSel> <ChanSel> <ChanSel> <ChanSel>
<ChanSel>	::=	“x” “y” “z” “w”

13.4.11 Immediate Values

<ImmAddrNum>	::=	“_512” ... “511”
<Imm32>	::=	“0.0” ... “ $\pm 1.0 \cdot 2^{-128} \dots 2^{127}$ ” “0” ... “ $2^{32}-1$ ” “ -2^{31} ” ... “ $2^{31}-1$ ”
<Imm16>	::=	“0” ... “ $2^{16}-1$ ” “ -2^{15} ” ... “ $2^{15}-1$ ”
<ImmDesc>	::=	“0” ... “ $2^{32}-1$ ”



13.4.12 Predication and Modifiers

Instruction Predication

```

<Predicate> ::= ""
              | "(" <PredState> <FlagReg> <PredCntrl> ")"

<PredState> ::= "+"
              | "-"

<PredCntrl> ::= ""
              | ".x" | ".y" | ".z" | ".w"
              | ".any2h" | ".all2h"
              | ".any4h" | ".all4h"
              | ".any8h" | ".all8h"
              | ".any16h" | ".all16h"
              | ".anyv" | ".allv"

```

Source Modification

```

<SrcModifier> ::= ""
               | "-"
               | "(abs)"
               | "-" "(abs)"

```

Instruction Modification

```

<ConditionalModifier> ::= ""
                       | <CondMod> "." <FlagReg>

<CondMod> ::= ".z" | ".e" | ".nz" | ".ne" | ".g" | ".ge" | ".l" | ".le" | ".o" | ".r" | ".u"

<Saturate> ::= ""
            | ".sat"

```

Execution Size

```

<ExecSize> ::= "(" <NumChannels> ")"
<NumChannels> ::= "1" | "2" | "4" | "8" | "16" | "32"

```



13.4.13 Instruction Options

<InstOptions>	::=	""
		"{" <InstOption> "}"
		"{" <InstOption> <InstOptionEx> "}"
<InstOptionEx>	::=	""
		"," <InstOption> <InstOptionEx>
<InstOption>	::=	<AccessMode>
		<ComprCtrl>
		<ThreadCtrl>
		<DependencyCtrl>
		<MaskCtrl>
		<DebugCtrl>
		<SendCtrl>
<AccessMode>	::=	"Align1" "Align16"
<ComprCtrl>	::=	"SecHalf" "Compr" "Compr4"
<ThreadCtrl>	::=	"Switch" "Atomic"
<DependencyCtrl>	::=	"NoDDChk" "NoDDClr"
<MaskCtrl>	::=	"NoMask" "NoCMask"
<DebugCtrl>	::=	"Breakpoint"
<SendCtrl>	::=	"EOT"

Note for Assembler: Compression control "Compr" has a direct map to the binary instruction word. It may be omitted as long as the Assembler is able to determine whether an instruction is in compressed mode or not based on the execution size and the mode of operation. However, "Compr4" requires Assembler to modify both compression control field and the MSB of the MRF destination register, so it must be present in the instruction if used.

Compr4 and **NoCMask** are not supported.



13.5 Deprecated Features

13.5.1 Defeatured Instructions

The following instructions are removed from Gen4 implementation mainly due to implementation cost/schedule reasons. They are candidates for future generations.

- Sum of Absolute Difference 4 (sad4)
- Sum of Absolute Difference Accumulate 4 (sada4)
- Add Accumulate (aac)
- Min (min)
- Max (max)
- Next (next)
- Swizzle (swz)
- Dot Product Accumulate 2 (dpa2)
- Rotation Shift Left (rsl)
- Rotation Shift Right (rsr)

13.5.2 Others

The following features are also deprecated from GEN4 implementation.

- Restricted 16-bit Half Floating-Point Numbers

§§





14 *Instruction Set Reference*

This chapter describes the functions of GEN4 instructions. Each GEN4 instruction is given a different page and the pages are sorted in alphabetical order according to assembly language mnemonic.

14.1 Conventions

14.1.1 Pseudo Code Format

The instructions are explained in the following pseudo-code format that resembles the GEN4 assembly instruction format.

```
[(<pred>)] opcode (<exec_size>) <dst> <src0> [<src1>]
```

Square brackets “[]” are used to signify that the field is optional. Saturation modifier and instruction options are omitted for simplicity.

14.1.2 General Macros and Definitions

INST_BYTE_COUNT is defined as a constant of 16 bytes.

```
#define INST_BYTE_COUNT 16 // byte count of instruction  
word
```

Function *floor()* converts a floating point value to an integral floating point value. For a given floating point value, from its closest two integral float values, function *floor()* returns the one that is closer to the negative infinity. For example, *floor*(1.3f) = 1.0f, and *floor*(-1.3f) = -2.0f.

```
float floor(float g)  
{  
    return maximum( any integral float f: f <= g)  
}
```

Function *Condition()* takes the conditional signals {SN, ZR, OF, IN, NC} of *result*, generates a Boolean data according to a conditional evaluation controlled by the conditional modifier *cmod*, and returns the Boolean data.

```
Bool Condition(result, cmod)  
{  
}
```



Function *ConditionNaN()* takes the conditional signals {SN, ZR, OF, IN, NC, NS} of *result*, generates a Boolean data according to a conditional evaluation controlled by the conditional modifier *cmod*, and returns the Boolean data. The only difference between *Condition()* and *ConditionNaN()* is that *ConditionNaN()* uses the NS (NaN of the second source) signal.

```
Bool ConditionNaN(result, cmod)
{
}
```

Function *Jump()* jumps the instruction sequence from the current instruction location by *InstCount* number of instructions. If *InstCount* is a positive number, it jumps forward; if *InstCount* is a negative number, it jumps backward; if *InstCount* is zero, it is effectively an infinite loop on the current instruction.

```
void Jump(int InstCount)
{
    IP = IP + (InstCount * INST_BYTE_COUNT)
}
```

14.1.3 Mask Stack Operations

The following operations manipulate mask and mask stack. They are used in the instruction pseudocodes.

Function *Reset()* resets both the depth count and stack adder of mask stack *DstStack*.

```
void DstStack.Reset()
{
    DstStack.DepthCount = 0;
    for (n = 0; n <= 15; n++) {
        DstStack.adder[n] == 0;
    }
}
```




Function *Push()* pushes mask *SrcMask* into mask stack *DstStack*. It also increments the depth count. If the depth count overflows (wrapped to 0), it signals the overflow exception for the ask stack. Mask *SrcMask* is unchanged. When SPF is set, nothing is done in this function.

```
void DstStack.Push(Mask SrcMask)
{
    if (!SPF) {
        DstStack.DepthCount++;
        if (DstStack.DepthCount == 0) {
            DstStack.DepthOverflow_exception = TRUE;
        } else {
            DstStack.DepthOverflow_exception = FALSE;
        }
        for (n = 0; n <= 15; n++) {
            if (SrcMask.BitPos(n) == 0) {
                DstStack.adder[n]++;
            }
        }
    }
}
```

Function *Pop()* pops mask stack *DstStack* by *m* levels. It subtracts *m* from each stack counter. Any negative results are saturated to zero. The depth count is decremented by *m*. If *m* exceeds the current depth count, a stack underflow exception is signaled. When underflow occurs, depth count contains the modular value – amount of underflow. When SPF is set, nothing is done in this function.

```
void DstStack.Pop(unsigned int m)
{
    if (!SPF) {
        DstStack.DepthUnderflow_exception = (m >
        DepthCount);
        DstStack.DepthCount -= m;
        for (n = 0; n <= 15; n++) {
            if (DstStack.adder[n] > m) {
                DstStack.adder[n] -= m;
            } else {
                DstStack.adder[n] = 0;
            }
        }
    }
}
```



Function *TopOfStack()* returns the top entry of mask stack *SrcStack*.

```
Mask* SrcStack.TopOfStack()
{
    Mask* DestMask[15:0];
    for (n = 0; n <= 15; n++) {
        *DestMask.BitPos[n] = (SrcStack.adder[n] == 0);
    }
    Return DestMask;
}
```

Function *Evaluate()* the mask *Mask* according to control parameters in *Ctrl*. *Mask* may be the intermediate predication mask *PMask*, the intermediate branch mask *BMask*, or the internal execution mask *EMask*.

```
void Evaluate(Mask* Mask, int Ctrl)
{
    if (Mask == PMask) {
        if (Predication is off) {
            PMask[15:0] = 0xFFFF;
        } else {
            PMask[15:0] is set on predication control
            Details are omitted
        }
    }
    else if (Mask == BMask) {
        BMask[15:0] = AMask[15:0];
        BMask[15:0] &= LMask[15:0];
        BMask[15:0] &= CMask[15:0];
    }
    else if (Mask == EMask) {
        EMask[15:0] = AMask[15:0];
        EMask[15:0] &= IMask[15:0];
        EMask[15:0] &= LMask[15:0];
        If (Ctrl != NoCMask) { // used for while
            EMask[15:0] &= CMask[15:0];
        }
        If (MaskCtrl == NoMask) {
            EMask[15:0] = 0xFFFF;
        }
        If (Ctrl != NoPMask) { // used for 'sel'
            Evaluate(PMask);
            EMask[15:0] &= PMask[15:0];
        }
        If (CompCtrl == SecHalf) {
            EMask[7:0] = EMask[15:8];
        }
        EMask[15:0] &= ExecMask(ExecSize);
        EMask[15:0] &= DestMask(ExecSize);
    }
}
```

Function *ExecMask()* returns a mask value for the execution size *ExecSize*. The mask bits of the first *ExecSize* channels are set to 1 and the rest bits are set to zero. This function is used by function *Evaluate()*.



```
Mask ExecMask(int ExecSize)
{
    Mask TmpMask[15:0] = 0xFFFF;
    for (int i=0; i < ExecSize; i++) {
        TmpMask[i] = 1;
    }
    Return TmpMask[15:0];
}
```

Function *DeskMask()* returns a mask value based on the destination channel select – part of the instruction word in **Align16** mode. It returns a 0xFFFF mask value if the instruction is in **Align1** mode. This function is used by function *Evaluate()*.

```
Mask DestMask()
{
    Mask TmpMask[15:0] = 0x0000;
    If (Dest.ChannelSelect == TRUE) {
        If (Dest.ChannelSelect[x] == TRUE) {
            TmpMask[0] = TmpMask[4] = 1;
            TmpMask[8] = TmpMask[12] = 1;
        }
        If (Dest.ChannelSelect[y] == TRUE) {
            TmpMask[1] = TmpMask[5] = 1;
            TmpMask[9] = TmpMask[13] = 1;
        }
        If (Dest.ChannelSelect[z] == TRUE) {
            TmpMask[2] = TmpMask[6] = 1;
            TmpMask[10] = TmpMask[14] = 1;
        }
        If (Dest.ChannelSelect[w] == TRUE) {
            TmpMask[3] = TmpMask[7] = 1;
            TmpMask[11] = TmpMask[15] = 1;
        }
    } else {
        TmpMask[15:0] = 0xFFFF;
    }
    Return TmpMask[15:0];
}
```

14.2 Instruction Description

The rest of the chapter contains the description of GEN4 instructions.



14.2.1 add – Addition

Opcode	Instruction	Description
64 (0x40)	add <dst> <src0> <src1>	Component-wise addition of <src0> and <src1> and storing the results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	•	•	[FLT] [INT]	[FLT] [INT]

Format:

`[(<pred>)] add[.<cmode>] (<exec_size>) <dst> <src0> <src1>`

Syntax:

`[(<pred>)] add[.<cmode>] (<exec_size>) reg reg reg`
`[(<pred>)] add[.<cmode>] (<exec_size>) reg reg imm32`

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
    if (WrEn.chan[n] == 1) {
        dst.chan[n] = src0.chan[n] + src1.chan[n];
    }
}
```

Description:

The *add* instruction performs component-wise addition of <src0> and <src1> and stores the results in <dst>.

Addition of two floating point numbers follows rules in Table 14-1 (or Table 14-2), if the current floating point mode is IEEE mode (or ALT mode).



Table 14-1. Floating point addition of A (column) and B (row) in IEEE mode

	–inf	–finite	–denorm	–0	+0	+denorm	+finite	+inf	NaN
–inf	–inf	–inf	–inf	–inf	–inf	–inf	–inf	NaN	NaN
–finite	–inf	*	A	A	A	A	**	+inf	NaN
–denorm	–inf	B	–0	–0	+0	+0	B	+inf	NaN
–0	–inf	B	–0	–0	+0	+0	B	+inf	NaN
+0	–inf	B	+0	+0	+0	+0	B	+inf	NaN
+denorm	–inf	B	+0	+0	+0	+0	B	+inf	NaN
+finite	–inf	**	A	A	A	A	***	+inf	NaN
+inf	NaN	+inf	+inf	+inf	+inf	+inf	+inf	+inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Notes:

- * Result can be { –finite}
- ** Result can be {–finite, –0, +0, +finite}
- *** Result can be { +finite}

Table 14-2. Floating point addition of A (column) and B (row) in ALT mode

	– fmax	–finite	–denorm	–0	+0	+denorm	+finite	+ fmax	****
–fmax	–fmax	–fmax	–fmax	–fmax	–fmax	–fmax	–finite	+0	
–finite	–fmax	*	A	A	A	A	**	+fmax	
–denorm	–fmax	B	–0	–0	+0	+0	B	+fmax	
–0	–fmax	B	–0	–0	+0	+0	B	+fmax	
+0	–fmax	B	+0	+0	+0	+0	B	+fmax	
+denorm	–fmax	B	+0	+0	+0	+0	B	+fmax	
+finite	–finite	**	A	A	A	A	***	+fmax	
+fmax	+0	+fmax	+fmax	+fmax	+fmax	+fmax	+fmax	+fmax	

Notes:

- * Result can be { –fmax, –finite}
- ** Result can be {–finite, –0, +0, +finite}
- *** Result can be { +fmax, +finite}
- **** Result is undefined If any of A and/or is {–inf, +inf, NaN}

Restrictions:

Dword integer source is not allowed for this instruction in float execution mode. In other words, if one source is of type float (:f, :vf), the other source cannot be of type dword integer (:ud or :d).



14.2.2 and – Logical And

Opcode	Instruction	Description
5 (0x05)	and <dst> <src0> <src1>	Performing component-wise logic AND of <src0> and <src1> and storing the results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•		•		•	•	[INT]	[INT]

Format:

[(<pred>)] and[.<cmod>] (<exec_size>) <dst> <src0> <src1>

Syntax:

[(<pred>)] and[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] and[.<cmod>] (<exec_size>) reg reg imm32

Pseudocode:

```
Evaluate(WrEn);  
for (n = 0; n < exec_size; n++) {  
    if (WrEn.chan[n] == 1) {  
        dst.chan[n] = src0.chan[n] & src1.chan[n];  
    }  
}
```

Description:

The *and* instruction performs component-wise logic AND operation between <src0> and <src1> and stores the results in <dst>. Source modifiers are allowed.

Accumulator register is allowed to be the destination of this instruction with the restrictions listed below.

Restrictions:

Sign (SN) and Overflow (OF) conditions are undefined for this logic instruction. Consequently, saturation modifier (.sat) is not allowed.

This instruction only applies to integer data types. The behavior is undefined if any operand is float.



When accumulator is the destination of this instruction, only the low bits corresponding to the data type (16 bits for word or 32 bits for dword integer instruction) in the accumulator contain the correct results. The internal extra-precision bits as well as the sign bit of the accumulator are undefined. Consequently, there are restrictions for subsequent instructions that use the data in the accumulator register created from the previous logical instruction.

- Only logical and data move instructions are allowed to source the accumulator. Results of other instructions (e.g. arithmetic or shift) are undefined.
- When the accumulator is the source of a data move (mov or sel) instruction, the destination operand must be of integer type (e.g. no conversion to float) and this instruction cannot have saturation instruction modifier.



14.2.3 asr – Arithmetic Shift Right

Opcode	Instruction	Description
12 (0x0C)	asr <dst> <src0> <src1>	Performing component-wise arithmetic right shift of <src0> and storing the results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•		•	•	•	•	[INT]	[INT]

Format:

`[(<pred>)] asr[.<cmod>] (<exec_size>) <dst> <src0> <src1>`

Syntax:

`[(<pred>)] asr[.<cmod>] (<exec_size>) reg reg reg`
`[(<pred>)] asr[.<cmod>] (<exec_size>) reg reg imm32`

Pseudocode:

```

Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
    if (WrEn.channel[n] == 1) {
        if (src0.chan[n] >= 0) {
            dst.chan[n] = src0.chan[n] >> src1.chan[n];
        } else {
            int maskLSB = pow(2, src1.chan[n]) - 1;
            if (maskLSB & src0.chan[n] == 0) {
                dst.chan[n] = sign(src0.chan[n]) *
                    ((abs)src0.chan[n] >>
                     src1.chan[n]);
            } else {
                dst.chan[n] = sign(src0.chan[n]) *
                    ((abs)src0.chan[n] >>
                     src1.chan[n]) - 1;
            }
        }
    }
}

```


**Description:**

The *asr* instruction performs component-wise arithmetic right shift of <src0> and storing the results in <dst>. Arithmetic right shift performs sign-extension by repeating the MSB of each data channel of <src0>. The amount of bit shift is provided by <src1>. Only the 5 LSBs of each channel of <src1> are used as an unsigned integer value. The rest of MSBs of <src1> data channels are ignored.

Operands for this instruction can be signed or unsigned integers, but cannot be floating point type. 5-bit shifting applies to packed-dword mode and packed-word mode. For packed word mode, the accumulators have 33 bits per channel.

This instruction is effectively a power-of-2 integer divide with truncate in 2's complement form. Truncate in 2's complement form is also known as downward rounding – closest integer that is smaller than or equal to the result. For example, regardless of the bit shift amount in <src1>, the result of arithmetic right-shift of -1 (<src0>) is always -1.

Restrictions:

This instruction does not work with float type operands.

The results are not stored in the accumulator register and therefore this instruction cannot have accumulator as destination.



14.2.4 avg – Average

Opcode	Instruction	Description
66 (0x42)	avg <dst> <src0> <src1>	Component-wise averaging of <src0> and <src1> and storing the results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
.	.	.	.	.	.	[INT]	[INT]

Format:

`[(<pred>)] avg[.<cmod>] (<exec_size>) <dst> <src0> <src1>`

Syntax:

`[(<pred>)] avg[.<cmod>] (<exec_size>) reg reg reg`
`[(<pred>)] avg[.<cmod>] (<exec_size>) reg reg imm32`

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
    if (WrEn.chan[n] == 1) {
        dst.chan[n] = (src0.chan[n] + src1.chan[n] + 1) >> 1;
    }
}
```

Description:

The `avg` instruction performs component-wise integer average of <src0> and <src1> and stores the results in <dst>. An integer average uses integer upward rounding. It is equivalent to increment one to the addition of <src0> and <src1> and then apply an arithmetic right shift to this intermediate value.

Restrictions:

This instruction only applies to integer data types. The behavior is undefined if any operand is float.



14.2.5 break – Break

Opcode	Instruction	Description
40 (0x28)	Break <exitcode>	Terminating enabled execution channels and conditionally breaking out from the inner most loop.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
		•					

Format:

`[(<pred>)] break (<exec_size>) <exitcode>`

Syntax:

`[(<pred>)] break (<exec_size>) imm32`

Pseudocode:

```

Evaluate(EMask);
LMask = LMask & !EMask;
CMask = CMask & !EMask;           // *** (1)
TempMask = LMask & AMask;         // *** (2)
if (TempMask == 0) {
    CMask = Lstack.TopOfStack;
    Lstack.pop(1);
    LMask = Lstack.TopOfStack;
    Lstack.pop(1);
    if (<exitcode.ISPopCount> != 0) {

        IStack.pop(<exitcode.ISPopCount> - 1);
        IMask = IStack.TopOfStack;
        IStack.pop(1);
    }
    Jump(<exitcode.IPCount>);
}

```

Description:

The *break* instruction is used to early-out from the inner most loop. When executed, the *break* instruction terminates the loop for all enabled execution channels. This is performed by clearing the associated bit positions in LMask. The



loop is complete when all bit positions of LMask are cleared. If the loop completes on a *break* instruction, the specified number of entries (If-nesting depth) is popped from the IStack, 2 levels of pop of the LStack, and a jump taken to the address specified in the <exitcode> field. These pops are intended to restore the mask-stacks and LMask/CMask/IMask to their values prior to entering the loop.

The following table describes the 32-bit exit code <exitcode>. The InstCount field of <exitcode> is a signed 16-bit number, added to IP pre-increment, and should point to the first instruction following loop block. In GEN4 binary, <exitcode> is at location <src1> and must be of type D (signed doubleword integer).

Bit	Description
31:20	Reserved: MBZ
19:16	ISPopCount (If-Stack Pop Count) This field specifies the number of times to pop the IStack. Format = U4. Unsigned integer [0, 15]
15:0	InstCount (Jump Instruction Count) This field specifies the jump distance in Instruction Count if a jump is taken for the instruction. Format = S15. Signed integer in 2's complement

This instruction executes regardless of the calculated EMask at the time of issue. It invokes a thread switch after issue to allow any masks and/or IP to be resolved if necessary.

This instruction performs a mask-stack push/pop operation. Mask-stack push/pop operations are always done in 16-bit width regardless of execution size. Nesting depths must be tracked to ensure that a mask-stack under/overflow does not occur, or that an appropriate mask-stack exception handler is in place.

If SPF is set, this instruction does not update any mask stack. When the condition is true, this instruction restores LMask and CMask. If "ISPopCount" is not zero, it also restores IMask.

If a jump is invoked by this instruction, IMask will be fully restored, same as LMask and CMask. ISPopCount = 0 is a trivial case that IMask is not changed.

Notes to the pseudo code:

(1) CMask is updated by a break instruction to force it to always be a 1's-subset of the LMask. This allows for the sharing of a mask-stack and the reliance on a fixed order of pushing LMask first and then CMask for embedded loops. If we did not update CMask, the order of pushing would be data dependent and would take significant code to do properly or we would not be able to share a stack.

(2) It should not take CMask as the subset of channels disabled by CMask are temporally terminated but still in the do-while loop.



Restrictions:

Instruction compression is not allowed.

IP register must be put (for example, by the assembler) at <dst> and <src0> locations.



14.2.6 cmp – Compare

Opcode	Instruction	Description
16 (0x10)	cmp.<cmo> <dst> <src0> <src1>	Component-wise comparison of <src0> and <src1> according to conditional modifier in <cmo> and storing the results in flag register in <cmo> and <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•		•		•	•	[FLT] [INT]	[FLT] [INT]

Format :

[(<pred>)] cmp[.<cmo>] (<exec_size>) <dst> <src0> <src1>

Syntax:

[(<pred>)] cmp[.<cmo>] (<exec_size>) reg reg reg
[(<pred>)] cmp[.<cmo>] (<exec_size>) reg reg imm32

Pseudocode :

```

Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
    bitMask[n] = 0;
    if (WrEn.chan[n] == 1) {
        results[n] = src0.chan[n] - src1.chan[n];
        bitMask[n] = Condition(results[n]);
        dst.chan[n][0] = bitMask[n];
    }
}
flag# = bitMask;

```

Description:

The *cmp* instruction performs component-wise comparison of <src0> and <src1> and stores the results in the selected flag register and in <dst>. It takes component-wise subtraction of <src0> and <src1>, evaluating the conditional code (excluding NS signal) based on the conditional modifier, and storing the conditional bits in bit-packed form in the destination flag register and, optionally, in vector form in the LSB of the channels in <dst>. Conditional modifier field cannot be 0000b, i.e., it must be one of the defined conditional modifier codes. Destination operand can be a GRF, an MRF or a null register. If it is not null, for the enabled channels, the LSB of the result in the destination channel contains the



flag value for the channel. The other bits are undefined. When the instruction operates on packed word format, one GRF register may store up to 16 such comparison results. In dword format, one GRF may store up to 8 results. When the register is used later as a vector of Booleans, as only LSB at each channel contains meaning data, software should make sure all higher bits are masked out (e.g. by 'and-ing' an 0x01 constant).

If `<exec_size>` is equal or less than 8, when 'SecHalf' option flag is not set, only the lower 8 bits of the selected flag register is updated; otherwise, the higher 8 bits are updated.

When at least one of the source operands is float, the *cmp* instruction obeys the floating point rules detailed in the tables in the Floating Point Mode section of Data Type chapter.

Restrictions:

Destination operand cannot be an ARF register, including accumulator.

Saturation modifier cannot be set in this instruction.

Dword integer source is not allowed for this instruction in float execution mode. In other words, if one source is of type float (:f, :vf), the other source cannot be of type dword integer (:ud or :d).



14.2.7 cmpn – Compare NaN

Opcode	Instruction	Description
17 (0x11)	cmpn.<cmode> <dst> <src0> <src1>	Performing component-wise special NaN comparison of <src0> and <src1> according to conditional modifier in <cmode> and storing the results in flag register in <cmode> and <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•		•		•	•	[FLT] [INT]	[FLT] [INT]

Format:

[(<pred>)] cmpn[.<cmode>] (<exec_size>) <dst> <src0> <src1>

Syntax:

[(<pred>)] cmpn[.<cmode>] (<exec_size>) reg reg reg
[(<pred>)] cmpn[.<cmode>] (<exec_size>) reg reg imm32

Pseudocode:

```

Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
    bitMask[n] = 0;
    if (EMask.chan[n] == 1) {
        results[n] = src0.chan[n] - src1.chan[n];
        bitMask[n] = ConditionNaN(results[n]);
        dst.chan[n][0] = bitMask[n];
    }
}
flag# = bitMask;

```

Description:

The *cmpn* instruction performs component-wise special-NaN comparison of <src0> and <src1> and stores the results in the selected flag register and in <dst>. It takes component-wise subtraction of <src0> and <src1>, evaluating the conditional signals including NS based on the conditional modifier, and storing the conditional flag bits in bit-packed form in the destination flag register and, optionally, in vector form in the LSB of the channels in <dst>. Conditional modifier field cannot be 0000b, i.e., it must be one of the defined conditional modifier codes. Destination operand can be a GRF, an MRF or a null register. If it



is not null, for the enabled channels, the LSB of the result in the destination channel contains the flag value for the channel. The other bits are undefined. When the instruction operates on packed word format, one GRF register may store up to 16 such comparison results. In dword format, one GRF may store up to 8 results. When the register is used later as a vector of Booleans, as only LSB at each channel contains meaning data, software should make sure all higher bits are masked out (e.g. by 'and-ing' an 0x01 constant).

If `<exec_size>` is equal or less than 8, when '*SecHalf*' option flag is not set, only the lower 8 bits of the selected flag register is updated; otherwise, the higher 8 bits are updated.

When at least one of the source operands is float, the *cmpn* instruction obeys the floating point rules detailed in the tables in the Floating Point Mode section of Data Type chapter.

This instruction is similar to *cmp*. The only difference is that if the second source operand `<src1>` is a NaN, the result for any conditional modifier except **.nz** is false.

For integer operands, *cmpn* and *cmp* are identical.

Restrictions:

Destination operand cannot be an ARF register, including accumulator.

Saturation modifier cannot be set in this instruction.

Dword integer source is not allowed for this instruction in float execution mode. In other words, if one source is of type float (*:f, :vf*), the other source cannot be of type dword integer (*:ud or :d*).



14.2.8 cont – Continue

Opcode	Instruction	Description
41 (0x29)	cont <exitcode>	Temporally disabling enabled execution channels for the remainder of the inner most loop and conditionally jumping to the last instruction (while) of the loop.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
		.					

Format:

[(<pred>)] cont (<exec_size>) <exitcode>

Syntax:

[(<pred>)] cont (<exec_size>) imm32

Pseudocode:

```
Evaluate(EMask);
CMask = CMask & !EMask;
TempMask = CMask & AMask;          // *** (1)
if (TempMask == 0) {
    if (<exitcode>.ISPopCount != 0) {
        IStack.pop(<exitcode>.ISPopCount - 1);
        IMask = IStack.TopOfStack;
        IStack.pop(1);
    }
    Jump(<exitcode>.IPCount);
}
```

Description:

The *cont* instruction disables execution for the subset of channels for the remainder of the current loop iteration. Channels remain disabled until explicitly re-enabled by the thread through writing to the CMask register. If CMask = 0 after evaluating this instruction, a jump is made a distance of <InstCount> where execution continues. It is expected that this is the location of the associated 'while'. For proper loop operation, CMask must be re-initialized to EMask on each loop pass.

The following table describes the 32-bit exit code <exitcode>. The InstCount field of <exitcode> is a signed 16-bit number, added to IP pre-increment, and should



point to the loop's associated 'while' instruction. In GEN4 binary, <exitcode> is at location <src1> and must be of type D (signed doubleword integer).

Bit	Description
31:20	Reserved: MBZ
19:16	ISPopCount (If-Stack Pop Count) This field specifies the number of times to pop the IStack. Format = U4. Unsigned integer [0, 15]
15:0	InstCount (Jump Instruction Count) This field specifies the jump distance in Instruction Count if a jump is taken for the instruction. Format = S15. Signed integer in 2's complement

This instruction executes regardless of the calculated EMask at the time of issue. It invokes a thread switch after issue to allow any masks and/or IP to be resolved if necessary.

This instruction performs a mask-stack push/pop operation. Mask-stack push/pop operations are always done in 16-bit width regardless of execution size. Nesting depths must be tracked to ensure that a mask-stack under/overflow does not occur, or that an appropriate mask-stack exception handler is in place.

If SPF is set, this instruction does not update any mask stack. When the condition is true and "ISPopCount" is not zero, this instruction restores IMask.

If a jump is invoked by this instruction, IMask will be fully restored, same as LMask and CMask. ISPopCount = 0 is a trivial case that IMask is not changed.

Upon completion of this instruction, B/LMask will only be valid if popping 1 level. If more than one level needs to be popped from any mask-stack, n-1 levels should be popped at the instruction and the final level popped by an explicit *pop* instruction.

Note that the 'break' instruction also updates CMask. This allows CMask and LMask to share the same mask-stack.

Restrictions:

Instruction compression is not allowed.

IP register must be put (for example, by the assembler) at <dst> and <src0> locations.



14.2.9 do – Do

Opcode	Instruction	Description
38 (0x26)	do	Updating mask and mask stack to enter a do-while loop.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types

Format :

do

Syntax:

do

Pseudocode :

```
Evaluate(EMask);  
LStack.push(LMask);  
LStack.push(CMask);  
LMask = EMask;  
CMask = EMask;
```

Description:

The *do* instruction indicates the start of a do-while block. Each *do* must have a matching *while* instruction. Execution of the *do* instruction causes the LMask and CMask (in that order) to be saved to the LStack for preservation and eventual restoration upon completion of the do-while block.

This instruction is equivalent to two *msave* instructions (in the order of “*msave lstack lmask*” and “*msave lstack cmask*”). It is an efficient construct for a do-while block.

This instruction performs a mask-stack push/pop operation. Mask-stack push/pop operations are always done in 16-bit width regardless of execution size. Nesting depths must be tracked to ensure that a mask-stack under/overflow does not occur, or that an appropriate mask-stack exception handler is in place.

SPF effectively turns this instruction into a nop, as LMask and CMask should be coherent with EMask. It may be used as instruction filler for code readability keeping in mind that a nop wastes an instruction cycle.

Restrictions:

Predication is not allowed. Instruction compression does not apply to this instruction.

Execution size is ignored for this instruction.



14.2.10 dp2 – Dot Product 2

Opcode	Instruction	Description
87 (0x57)	Dp2 <dst> <src0> <src1>	Performing two-wide dot product in four-tuples of <src0> and <src1> and storing the replicated results in four-tuples in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	•	•	[FLT] [INT]	[FLT] [INT]

Format:

`[(<pred>)] dp2[.<cmod>] (<exec_size>) <dst> <src0> <src1>`

Syntax:

`[(<pred>)] dp2[.<cmod>] (<exec_size>) reg reg reg`
`[(<pred>)] dp2[.<cmod>] (<exec_size>) reg reg imm32`

Pseudocode:

```

Evaluate(WrEn);
for (n = 0; n < exec_size; n+=4) {
    fTmp = src0.chan[n] * src1.chan[n]
          + src0.chan[n+1] * src1.chan[n+1];
    if (WrEn.chan[n] == 1) dst.chan[n] = fTmp;
    if (WrEn.chan[n+1] == 1) dst.chan[n+1] = fTmp;
    if (WrEn.chan[n+2] == 1) dst.chan[n+2] = fTmp;
    if (WrEn.chan[n+3] == 1) dst.chan[n+3] = fTmp;
}

```

Description:

The *dp2* instruction performs a two-wide dot-product on four-tuple vector basis and storing the same scalar result per four-tuple to all four channels in <dst>. This instruction is similar to *dp4* except that every third and fourth elements of <src0> (post-source-swizzle if present) are not involved in the computation.

Special care has been taken in the hardware such that if the resulting value for a given group of four channels is 0.0f, the sign of the result correctly reflects the input data of the first two channels of the group of four.

**Restrictions:**

Source operands cannot be an accumulator register.

Execution size cannot be less than 4.

This instruction does not support integer operation. Furthermore, both source operands must be float. Destination can be float or integer.

Horizontal stride must be 1.

The results are NOT stored in the accumulator register. This instruction does implicitly update accumulator register, however with undefined values.



14.2.11 dp3 – Dot Product 3

Opcode	Instruction	Description
86 (0x56)	dp3 <dst> <src0> <src1>	Performing three-wide dot product in four-tuples of <src0> and <src1> and storing the replicated results in four-tuples in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	•	•	[FLT] [INT]	[FLT] [INT]

Format:

`[(<pred>)] dp3[.<cmod>] (<exec_size>) <dst> <src0> <src1>`

Syntax:

`[(<pred>)] dp3[.<cmod>] (<exec_size>) reg reg reg`
`[(<pred>)] dp3[.<cmod>] (<exec_size>) reg reg imm32`

Pseudocode:

```

Evaluate(WrEn);
for (n = 0; n < exec_size; n+=4) {
    fTmp = src0.chan[n] * src1.chan[n]
          + src0.chan[n+1] * src1.chan[n+1]
          + src0.chan[n+2] * src1.chan[n+2];
    if (WrEn.chan[n] == 1) dst.chan[n] = fTmp;
    if (WrEn.chan[n+1] == 1) dst.chan[n+1] = fTmp;
    if (WrEn.chan[n+2] == 1) dst.chan[n+2] = fTmp;
    if (WrEn.chan[n+3] == 1) dst.chan[n+3] = fTmp;
}

```

Description:

The *dp3* instruction performs a three-wide dot-product on four-tuple vector basis and storing the same scalar result per four-tuple to all four channels in <dst>. This instruction is similar to *dp4* except that every fourth element of <src0> (post-source-swizzle if present) is not involved in the computation.

Special care has been taken in the hardware such that if the resulting value for a given group of 4 channels is 0.0f, the sign of the result correctly reflects the input data of the first three channels of the group of four.

**Restrictions:**

Source operands cannot be an accumulator register.

Execution size cannot be less than 4.

This instruction does not support integer operation. Furthermore, both source operands must be float. Destination can be float or integer.

Horizontal stride must be 1.

The results are NOT stored in the accumulator register. This instruction does implicitly update accumulator register, however with undefined values.



14.2.12 dp4 – Dot Product 4

Opcode	Instruction	Description
84 (0x54)	dp4 <dst> <src0> <src1>	Performing four-wide dot product of <src0> and <src1> and storing the four-wide replicated results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	•	•	[FLT] [INT]	[FLT] [INT]

Format:

`[(<pred>)] dp4[.<cmod>] (<exec_size>) <dst> <src0> <src1>`

Syntax:

`[(<pred>)] dp4[.<cmod>] (<exec_size>) reg reg reg`
`[(<pred>)] dp4[.<cmod>] (<exec_size>) reg reg imm32`

Pseudocode:

```

Evaluate(WrEn);
for (n = 0; n < exec_size; n+=4) {
    fTmp = src0.chan[n] * src1.chan[n]
          + src0.chan[n+1] * src1.chan[n+1]
          + src0.chan[n+2] * src1.chan[n+2]
          + src0.chan[n+3] * src1.chan[n+3];
    if (WrEn.chan[n] == 1) dst.chan[n] = fTmp;
    if (WrEn.chan[n+1] == 1) dst.chan[n+1] = fTmp;
    if (WrEn.chan[n+2] == 1) dst.chan[n+2] = fTmp;
    if (WrEn.chan[n+3] == 1) dst.chan[n+3] = fTmp;
}

```

Description:

The *dp4* instruction performs a four-wide dot-product on four-tuple vector basis and storing the same scalar result per four-tuple to all four channels in <dst>.

**Restrictions:**

Source operands cannot be an accumulator register.

Execution size cannot be less than 4.

This instruction does not support integer operation. Furthermore, both source operands must be float. Destination can be float or integer.

Horizontal stride must be 1.

The results are NOT stored in the accumulator register. This instruction does implicitly update accumulator register, however with undefined values.



14.2.13 dph –Dot Product Homogeneous

Opcode	Instruction	Description
85 (0x55)	dph <dst> <src0> <src1>	Performing four-wide homogeneous dot product of <src0> and <src1> and storing the four-wide replicated results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	•	•	[FLT] [INT]	[FLT] [INT]

Format:

`[(<pred>)] dph[.<cmod>] (<exec_size>) <dst> <src0> <src1>`

Syntax:

`[(<pred>)] dph[.<cmod>] (<exec_size>) reg reg reg`
`[(<pred>)] dph[.<cmod>] (<exec_size>) reg reg imm32`

Pseudocode:

```

Evaluate(WrEn);
for (n = 0; n < exec_size; n+=4) {
    fTmp = src0.chan[n] * src1.chan[n]
          + src0.chan[n+1] * src1.chan[n+1]
          + src0.chan[n+2] * src1.chan[n+2]
          + src1.chan[n+3];
    if (WrEn.chan[n] == 1) dst.chan[n] = fTmp;
    if (WrEn.chan[n+1] == 1) dst.chan[n+1] = fTmp;
    if (WrEn.chan[n+2] == 1) dst.chan[n+2] = fTmp;
    if (WrEn.chan[n+3] == 1) dst.chan[n+3] = fTmp;
}

```

Description:

The *dph* instruction performs a four-wide homogeneous dot-product on four-tuple vector basis and storing the same scalar result per four-tuple to all four channels in <dst>. This instruction is similar to *dp4* except that every fourth element of <src0> (post-source-swizzle if present) is forced to 1.0f.

**Restrictions:**

Source operands cannot be an accumulator register.

Execution size cannot be less than 4.

This instruction does not support integer operation. Furthermore, both source operands must be float. Destination can be float or integer.

Horizontal stride must be 1.

The results are NOT stored in the accumulator register. This instruction does implicitly update accumulator register, however with undefined values.



14.2.14 else – Else

Opcode	Instruction	Description
36 (0x24)	else <exitcode>	An optional statement within an if/else/endif block of code.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types

Format:

else (<exec_size>) <exitcode>

Syntax:

else (<exec_size>) imm32

Pseudocode:

```

Evaluate(EMask);
IMask = !IMask & IStack.TopOfStack;
Evaluate(EMask);          // *** (1)
if (EMask == 0) {
    IMask = IStack.TopOfStack;
    IStack.pop(ISPopCount);
    Jump(<exitcode.IPCount>);
}

```

Description:

The *else* instruction is an optional statement within an **if/else/endif** block of code. It restricts execution within the else/endif portion to the opposite set of channels enabled under the if/else portion. Channels which were inactive prior to entering the **if/endif** block remain inactive throughout the entire block.

The IMask, which maintains the bit-mask of enabled channels inside the conditional block, is updated to reflect the new subset of active channels within the **else/endif** portion. If all channels are inactive (IMask = 000...), a relative jump is performed to the location specified in <exitcode.IPCount>. The jump target should be the instruction immediately following the matching **endif** instruction for that conditional block.

The following table describes the 32-bit exit code <exitcode>. In GEN4 binary, <exitcode> is at location <src1> and must be of type D (signed doubleword integer). <exitcode> must be an immediate operand, whereas IPCount is a signed



16-bit number and is intended to be forward referencing. This value is added to IP pre-increment. The ISPopCount field must be 1 as only one pop is performed if a jump occurs and hardware uses this field to pop the IStack. Hardware behavior is undefined if any other values are used in this field.

Bit	Description
31:20	Reserved: MBZ
19:16	ISPopCount (If-Stack Pop Count) This field specifies the number of times to pop the IStack. Format = U4. Must be set to 1.
15:0	InstCount (Jump Instruction Count) This field specifies the jump distance in Instruction Count if a jump is taken for the instruction. Format = S15. Signed integer in 2's complement

This instruction executes regardless of the calculated EMask at the time of issue. It invokes a thread switch after issue to allow any masks and/or IP to be resolved if necessary.

This instruction performs a mask-stack push/pop operation. Mask-stack push/pop operations are always done in 16-bit widths regardless of execution size. Nesting depths must be tracked to ensure that a mask-stack under/overflow does not occur, or that an appropriate mask-stack exception handler is in place.

If SPF is set, this instruction does not update mask stack. SPF can be used for the case when a scalar condition is evaluated.

Each **else** instruction must be preceded with a matching **if** instruction and followed by a matching **endif** instruction.

Pseudocode notes:

(1) Uses the updated IMask calculated during instruction execution, not the IMask prior to instruction dispatch.

(2) The EMask calculation already has <exec_size> incorporated; therefore the full 16-bit EMask value can be used in the comparison.

Restrictions:

Instruction compression is not allowed.

Predication is not allowed.

IP register must be put (for example, by the assembler) at <dst> and <src0> locations.



14.2.15 **endif – End-If**

Opcode	Instruction	Description
37 (0x24)	endif <exitcode>	Restoring execution to those data channels that were active prior to the if/else/endif block.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types

Format :

endif <exitcode>

Syntax:

endif imm32

Pseudocode:

```
IMask = IStack.TopOfStack;
IStack.pop(ISPopCount);
```

Description:

The *endif* instruction terminates an if/else/endif block of code. It restores the execution to these data channels that were active prior to the if/else/endif block by popping the IMask once. IMask is updated to reflect the popped value from the IStack.

The following table describes the 32-bit exit code <exitcode>. In GEN4 binary, <exitcode> is at location <src1> and must be of type D (signed doubleword integer). <exitcode> must be an immediate operand, whereas ISPopCount field must be 1 as only one pop is performed if a jump occurs and hardware uses this field to pop the IStack.

Software must put a 1 in unsigned integer format in this field as this instruction only pop the IStack once. Hardware behavior is undefined if any other values are used in this field.

<exitcode> must be an immediate operand, whereas ISPopCount field must be 1 as only one pop is performed if a jump occurs and hardware uses this field to pop the IStack.



Bit	Description
31:20	Reserved: MBZ
19:16	ISPopCount (If-Stack Pop Count) . This field specifies the number of times to pop the IStack. Format = U4. Must be set to 1.
15:0	Reserved: MBZ

This instruction updates IMask, therefore, a thread switch is always invoked after the issuance of an 'endif' instruction.

If SPF is set, this instruction restores IMask from IStack's Top-Of-Stack, which is supposed to be static for code with single program flow.

Execution size is ignored for this instruction.

Restrictions:

Instruction compression does not apply to this instruction.

Predication is not allowed.



14.2.16 frc – Fraction

Opcode	Instruction	Description
67 (0x43)	frc <dst> <src0>	Taking component-wise truncate-to-minus-infinity fraction operation of <src0> and storing the results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•		•	•	[FLT] [INT]	[FLT] [INT]

Format:

[(<pred>)] frc[.<cmod>] (<exec_size>) <dst> <src0>

Syntax:

[(<pred>)] frc[.<cmod>] (<exec_size>) reg reg
[(<pred>)] frc[.<cmod>] (<exec_size>) reg imm32

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
    if (WrEn.chan[n] == 1) {
        dst.chan[n] = src0.chan[n] - floor(src0.chan[n]);
    }
}
```

Description:

The *frc* instruction computes, component-wise, the truncate-to-minus-infinity fractional values of <src0> and stores the results in <dst>. The results, in the range of [0.0, 1.0], are the fractional portion of the source data.

Source operand for this instruction must be of floating point type. This instruction can only operate on normalized floating source and therefore cannot take accumulator as source or destination operand.

This instruction only applies to floating point operands.

Floating point fraction computation follows rules in Table 14-3 (or Table 14-4), if the current floating point mode is IEEE mode (or ALT mode).



Table 14-3. Floating point fraction computation in IEEE mode

<src0>	–inf	–finite	–denorm	–0	+0	+denorm	+finite	+inf	NaN
<dst>	NaN	*	+0	+0	+0	+0	*	NaN	NaN
Notes: * Result is in the range of [+0, 1) – not including 1.									

Table 14-4. Floating point fraction computation in ALT mode

<src0>	– fmax	–finite	–denorm	–0	+0	+denorm	+finite	+ fmax	**
<dst>	+0	*	+0	+0	+0	+0	*	+0	
Notes: * Result is in the range of [+0, 1) – not including 1. ** Result is undefined if <src0> is {–inf, +inf, NaN}.									

Restrictions:

Saturation modifier does not apply to this instruction.

This instruction cannot take accumulator as source or destination operand as it can only operate on normalized floating source.

This instruction does implicitly update accumulator register when enabled, however with undefined values.



14.2.17 halt – Halt

Opcode	Instruction	Description
42 (0x2A)	halt <exitcode>	Temporarily suspending execution for all enabled execution channels.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
		•					

Format:

`[(<pred>)] halt (<exec_size>) <exitcode>`

Syntax:

`[(<pred>)] halt (<exec_size>) imm32`

Pseudocode:

```

Evaluate(EMask);
AMask = AMask & !EMask;
if (AMask == 0) {
    Jump(<exitcode>.IPCount);
}

```

Description:

The *halt* instruction temporarily suspends execution for all enabled compute channels. The value of AMask is updated, with bits in positions of enabled channels set to '0'. If all the bits of the resultant AMask are cleared, a jump is made <inst_count> instructions away.

The *halt* instruction is also used inside subroutines as a 'return', utilizing AMask to keep track of which execution channels have returned and which to continue execution. Since there is no hardware mask stack for AMask, software must manually preserve the value of AMask around a subroutine call.

The following table describes the 32-bit exit code <exitcode>. In GEN4 binary, <exitcode> is at location <src1> and must be of type D (signed doubleword integer). The InstCount field of <exitcode> is a signed 16-bit number, added to IP pre-increment, and should typically point to code which restores AMask.



Bit	Description
31:16	Reserved: MBZ
15:0	InstCount (Jump Instruction Count) This field specifies the jump distance in Instruction Count if a jump is taken for the instruction. Format = S15. Signed integer in 2's complement

This instruction executes regardless of the calculated EMask at the time of issue. This instruction invokes a thread switch after issue to allow any masks and/or IP to be resolved if necessary.

The jump target used in the *halt* instruction typically crosses loops or 'if's, leaving the associated mask-stacks stale. It is the responsibility of program to manually restore these stacks to their values at the nesting level of the jump target.

Restrictions:

Instruction compression is not allowed.

IP register must be put (for example, by the assembler) at <dst> and <src0> locations.

As DMask is not automatically reloaded into AMask upon completion of this instruction, software has to manually restore AMask upon completion.



14.2.18 if – If

Opcode	Instruction	Description
34 (0x22)	if <exitcode>	Signifying the start of an if/else/endif block of code.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
		•					

Format:

[(<pred>)] if (<exec_size>) <exitcode>

Syntax:

[(<pred>)] if (<exec_size>) imm32

Pseudocode:

```

Evaluate(EMask);
IStack.push(IMask);
IMask = EMask;
if (EMask == 0) {
    Jump(<exitcode.IPCount>);
}

```

Description:

The *if* instruction starts an if/endif or an if/else/endif block of code. It restricts execution within the conditional block to only those channels that were enabled via EMask calculation in this instruction.

Each *if* instruction must have a matching *endif* instruction and may have up to one matching *else* instruction before *endif*.

The *if* instruction causes the value of the IMask prior to the *if* instruction to be saved to the IStack for retrieval at the conclusion of the conditional block. The *if* instruction evaluates the EMask. If all channels are inactive (for the if/endif or if/else block), a jump is performed of the relative distance as specified in the instruction. This jump must be to the matching *else* instruction when present, or otherwise to the matching *endif* instruction of that conditional block.

The following table describes the 32-bit exit code <exitcode>. <exitcode> must be an immediate operand, whereas IPCount is a signed 16-bit number. When a jump occurs, this value is added to IP pre-increment. In GEN4 instruction binary,



<exitcode> is at location <src1> and must be of type D (signed doubleword integer). IP register must be put (for example, by the assembler) at both <dst> and <src0> locations.

Bit	Description
31:16	Reserved: MBZ
15:0	IPCount (Jump Instruction Count) . This field specifies the jump distance in number of instructions if a jump is taken for the instruction. Format = S15. Signed integer in 2's complement

This instruction executes regardless of the calculated EMask at the time of issue. This instruction invokes a thread switch after issue to allow any masks and/or IP to be resolved if necessary.

This instruction performs a mask-stack push/pop operation. Mask-stack push/pop operations are always done in 16-bit widths regardless of execution size. Nesting depths must be tracked to ensure that a mask-stack under/overflow does not occur, or that an appropriate mask-stack exception handler is in place.

If SPF is set, this instruction does not update mask stack. SPF can be used for the case when a scalar condition is evaluated.

Restrictions:

Instruction compression is not allowed.



14.2.19 iff – Fast-If

Opcode	Instruction	Description
35 (0x23)	iff <exitcode>	Signifying the start of a fast if/endif block of code.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
		•					

Format:

[(<pred>)] iff (<exec_size>) <exitcode>

Syntax:

[(<pred>)] iff (<exec_size>) imm32

Pseudocode:

```

Evaluate(EMask);
if (EMask != 0) {
    IStack.push(IMask);
    IMask = EMask;
} else {
    Jump(<exitcode.IPCount>);
}

```

Description:

The *iff* instruction starts a high-performance (fast-if) if/endif block of code. It restricts execution within the conditional block to only those channels that were enabled via EMask calculation by this instruction.

Each *iff* instruction must have a matching *endif* instruction, but must NOT have a matching *else* instruction.

This fast-if has only one execution cycle of overhead comparing 2-3 for the general purpose *if* instruction. Unlike the *if* instruction, this instruction only pushes the IMask when at least one execution channels is active (i.e. EMask != 0) and thus requires processing within the if-endif block. When all execution channels are inactive (i.e. EMask = 0), the mask stack push is not performed, and execution continues at a relative distance specified by the <exitcode.IPCount> field. In this case, since the mask stack push was not performed, the mask stack pop associated with an *endif* must not be executed. Thus it is intended that the



instruction pointed to by <exitcode.IPCount> is one instruction beyond the block's *endif* instruction.

The following table describes the 32-bit exit code <exitcode>. <exitcode> must be an immediate operand, whereas IPCount is a signed 16-bit number. When a jump occurs, this value is added to IP pre-increment. In GEN4 instruction binary, <exitcode> is at location <src1> and must be of type D (signed doubleword integer). IP register must be put (for example, by the assembler) at both <dst> and <src0> locations.

Bit	Description
31:16	Reserved: MBZ
15:0	IPCount (Jump Instruction Count) . This field specifies the jump distance in number of instructions if a jump is taken for the instruction. Format = S15. Signed integer in 2's complement

This instruction executes regardless of the calculated EMask at the time of issue. This instruction invokes a thread switch after issue to allow any masks and/or IP to be resolved if necessary.

This instruction may perform a mask-stack push/pop operation. If it is performed, mask-stack push/pop operations are always done in 16-bit widths regardless of execution size. Nesting depths must be tracked to ensure that a mask-stack under/overflow does not occur, or that an appropriate mask-stack exception handler is in place.

If SPF is set, this instruction does not update mask stack. SPF can be used for the case when a scalar condition is evaluated.

Restrictions:

Instruction compression is not allowed.

Each *iff* must have one matching *endif*, but cannot have a matching *else* instruction.



14.2.20 jmp - Jump Indexed

Opcode	Instruction	Description
32 (0x20)	jmp <index>	Redirecting program execution to <index> instructions forward of the current post-incremented instruction pointer.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
		.					

Format :

[(<pred>)] jmp (1) <exitcode> {NoMask}

Syntax:

[(<pred>)] jmp (1) reg32 {NoMask}
 [(<pred>)] jmp (1) imm32 {NoMask}

Pseudocode :

```
Evaluate(WrEn);
if (WrEn != 0) {
    Jump(<exitcode.index> + 1);
}
```

Description:

The *jmp* instruction redirects program execution to <exitcode.index> instructions forward of the current **post-incremented** instruction pointer. <exitcode.index> is treated a signed integer value, with positive integers or zero generating forward jumps, and negative integers generating backward jumps. An <exitcode.index> value of 0 means execution continues at the instruction immediately following the *jmp* instruction, while an index value of -1 would imply an infinite loop.

Bit	Description
31:16	Reserved: MBZ
15:0	index (Jump Index) This field specifies the jump distance in number of instructions if a jump is taken for the instruction. Format = S15. Signed integer in 2's complement



<exitcode> may be a scalar register or an immediate. The data type of <exitcode> must be D (signed doubleword integer). However, hardware only uses lower 16 bits of <exitcode>. The valid range of <exitcode.index> is [–32768, 32767]. Behavior for <exitcode.index> outside that range is undefined.

This instruction executes regardless of the calculated WrEn at the time of issue. – To reduce hardware complexity, instruction optional control {NoMask} must be set for this instruction. This instruction invokes a thread switch after issue to allow any masks and/or IP to be resolved if necessary.

Execution size must be 1.

Predication is allowed to provide conditional jump with a scalar condition. As the execution size is 1, the first channel of PMASK (flags post prediction control and negate) is used to determine whether the jump is taken or not. If the condition is false, the jump is not taken and the IP immediately following will be executed next.

In GEN4 binary, <exitcode.index> is at location <src1>. IP register must be put (for example, by the assembler) at <dst> and <src0> locations.

Restrictions:

Instruction compression is not allowed.



14.2.21 line – Line

Opcode	Instruction	Description
89 (0x59)	Line <dst> <src0> <src1>	Computing a component-wise line equation ($v = p*u+q$) of <src0> and <src1> and storing the results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	•	•	[FLT] [INT]	[FLT] [INT]

Format:

[(<pred>)] line[.<cmod>] (<exec_size>) <dst> <src0> <src1>

Syntax:

[(<pred>)] line[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] line[.<cmod>] (<exec_size>) reg reg imm32

Pseudocode:

```

Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
  for (n = 0; n < exec_size; n++) {
    dwP = src0.RegNum.SubRegNum[bits4:2]    // a DW aligned
    scalar
    dwQ = src0.RegNum.(SubRegNum[bit4]|0x8)  // 4-th
    component
    if (WrEn.chan[n] == 1) {
      dst.chan[n] = dwP * src1.chan[n] + dwQ
    }
  }
}

```

Description:

The *line* instruction computes a component-wise line equation ($v = p*u+q$ where u/v are vectors and p/q are scalars) of <src0> and <src1> and storing the results in <dst>. <src1> is the input vector u . <src0> provides input scalars p and q , where p is the scalar value based on the region description of <src0> and q is the scalar value implied from <src0> region. Specifically, q is the fourth component of the 4-tuple (128-bit aligned) that p belongs to.

**Restrictions:**

This is a specialized instruction that only support execution size of 8 or 16.

<src0> region must be a replicated scalar (with HorzStride = VertStride = 0).

Dword integer source is not allowed for this instruction in float execution mode. In other words, if one source is of type float (:f, :vf), the other source cannot be of type dword integer (:ud or :d).

In particular, <src0> must be float. <src1> may be float, byte or word integer. <src1> cannot be dword integer. <dst> may be float or integer of any size.

Source operands cannot be an accumulator register.

<src0> for *line* instruction has to have .0 or .4 as the subregister number.



14.2.22 lzd – Leading Zero Detection

Opcode	Instruction	Description
74 (0x4A)	lzd <dst> <src0>	Performing component-wise leading zero detection of <src0> and storing the results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
.	.	.	.	.		[INT]	[INT]

Format:

`[(<pred>)] lzd[.<cm0d>] (<exec_size>) <dst> <src0>`

Syntax:

`[(<pred>)] lzd[.<cm0d>] (<exec_size>) reg reg`
`[(<pred>)] lzd[.<cm0d>] (<exec_size>) reg reg`

Pseudocode:

```

Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
    if (WrEn.chan[n] == 1) {
        UD udScalar = src0.chan[n];
        UD cnt = 0;
        while ( (udScalar & (1<<31)) == 0 && cnt != 32) {
            cnt ++;
            udScalar = udScalar << 1;
        }
        dst.chan[n] = cnt;
    }
}

```

Description:

The *lzd* instruction counts component-wise the leading zeros from <src0> and storing the resulting counts in <dst>.

This instruction only work on unsigned dword source. Source operand may be a signed or unsigned. If it is a signed integer, source modifier (abs) must be used to convert the source into an unsigned integer type.

The destination operand must also be of unsigned dword type.

**Restrictions:**

The destination operand cannot be the accumulator.

This instruction does implicitly update accumulator register when enabled, however with undefined values.



14.2.23 mac – Multiply Accumulate

Opcode	Instruction	Description
72 (0x48)	mac <dst> <src0> <src1>	Performing component-wise multiply accumulate of <src0> and <src1> and storing the results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	•	•	[FLT] [INT]	[FLT] [INT]

Format:

`[(<pred>)] mac[.<cmode>] (<exec_size>) <dst> <src0> <src1>`

Syntax:

`[(<pred>)] mac[.<cmode>] (<exec_size>) reg reg reg`
`[(<pred>)] mac[.<cmode>] (<exec_size>) reg reg imm32`

Pseudocode:

```

Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
    if (WrEn.chan[n] == 1) {
        dst.chan[n] = src0.chan[n] * src1.chan[n] +
        acc0.chan[n]
    }
}

```

Description:

The *mac* instruction takes component-wise multiplication of <src0> and <src1>, adds the results with the corresponding accumulator values, and then stores the final results in <dst>.

Restrictions:

Accumulator is an implied source to the addition portion of the computation. Explicit source operands cannot be accumulator.

This instruction doesn't support dword integers (D or UD).



14.2.24 mach – Multiply Accumulate High

Opcode	Instruction	Description
73 (0x49)	mach <dst> <src0> <src1>	Performing component-wise multiply accumulation of <src0>, <src1> and accumulator register, and returning the high dword of results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•		•	•	[INT]	[INT]

Format:

`[(<pred>)] mach[.<cmode>] (<exec_size>) <dst> <src0> <src1>`

Syntax:

`[(<pred>)] mach[.<cmode>] (<exec_size>) reg reg reg`
`[(<pred>)] mach[.<cmode>] (<exec_size>) reg reg imm32`

Pseudocode:

```

Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
    acc0.chan[n][63:0]
        = (src0.chan[n][31:16] *
           src1.chan[n][31:0]) << 16
        + acc0.chan[n][63:0];
    if (WrEn.channel[n] == 1) {
        dst.channel[n][31:0] = acc0.chan[n][63:32]
    }
}

```

Description:

The *mach* instruction performs dword integer multiply-accumulate operation and outputs the high dword (bits [63:32]). On a component by component basis, this instruction multiplies dwords in <src1> with the high words of dwords in <src0>, left-shifts the results by 16 bits, adds them with the corresponding accumulator values, and keeps the whole 64-bit results in the accumulator. It then stores the high dword (bits [63:32]) of the results in <dst>.

This instruction is intended to be used to emulate 32-bit dword integer multiplication by utilizing the large number of bits available in the accumulator. For example, the following three instructions perform vector multiplication of two



32-bit signed integer source from r2 and r3 and store the resulting vectors with high 32-bit in r4 and low 32-bit in r5.

```
mul    (8) acc0:d r2.0<8;8,1>d r3.0<8;8,1>:d
mach   (8) r4.0<1>:d r2.0<8;8,1>d r3.0<8;8,1>:d
mov    (8) r5.0<1>:d acc0:d
```

As *mach* is used to generate part of 64-bit dword integer results, saturation modifier should not be used. In fact, saturation modifier should not be used for any of these three instructions.

Source and destination operands must be dword integers. Source and destination must be of the same type, signed integer or unsigned integer.

- If <dst> is UD, <src0> and <src1> may be UD and/or D. However, if any of <src0> and <src1> is D, source modifier, (abs), must be present to convert it to match with <dst>.
- If <dst> is D, <src0> and <src1> must also be D. They cannot be UD as it may cause unexpected overflow because the computed results are limited to 64 bits.

Restrictions:

Accumulator is an implied source to the addition portion of the computation. Therefore, explicit source operands cannot be accumulator.



14.2.25 mov – Move

Opcode	Instruction	Description
1 (0x01)	mov <dst> <src0>	Component-wise move from <src0> to <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•		•	•	•	•	[FLT] [INT]	[FLT] [INT]

Format:

[(<pred>)] mov[.<cmod>] (<exec_size>) <dst> <src0>

Syntax:

[(<pred>)] mov[.<cmod>] (<exec_size>) reg reg
[(<pred>)] mov[.<cmod>] (<exec_size>) reg imm32

Pseudocode:

```
Evaluate(WrEn);  
for (n = 0; n < exec_size; n++) {  
    if (WrEn.chan[n] == 1) {  
        dst.chan[n] = src0.chan[n]  
    }  
}
```

Description:

The *mov* instruction moves the components in <src0> into the channels of <dst>. If <src0> and <dst> are of different types, format conversion is performed. If <src0> is a scalar immediate, the immediate value is loaded into all channels of <dst>.

Restrictions:

The accumulator can be either <src0> or <dst> but not both.

This instruction does not implicitly update accumulator register.



14.2.26 mrest – Mask Restore

Opcode	Instruction	Description
45 (0x2D)	mrest <mstack> <mask>	Restoring <mask> from <mstack> and also updating <mstack>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
						[INT]	[INT]

Format :

mrest (1) <mask> <mstack>

Syntax:

mrest (1) <mask> <mstack>

Pseudocode:

```
<mask> = <mstack>.TopOfStack;
<mstack>.pop(1);
```

Description:

The *mrest* instruction restores the value of a mask register from the specified mask-stack.

This instruction performs a mask-stack push/pop operation. Nesting depths must be tracked to ensure that a mask-stack under/overflow does not occur, or that an appropriate mask-stack exception handler is in place.

Nesting depths must be tracked to ensure that a mask-stack under/overflow does not occur, or that an appropriate mask-stack exception handler is in place.

This instruction is scalar operation only; it can only update one mask stack at a time.

If SPF is set, this instruction restores <mask> from the top of stack of <mstack>, but it does not pop the mask stack.

Restrictions:

Predication is not allowed.

<exec_size> must be set to 1 (by Assembler for example).

Instruction compression is not allowed. Both source and destination operands must be unsigned integer. Source modifier is not allowed.



14.2.27 msave – Mask Save

Opcode	Instruction	Description
44 (0x2C)	msave <mstack> <mask>	Pushing <mask> onto <mstack> and updating <mask> with the current EMask.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
						[INT]	[INT]

Format :

msave (1) <mstack> <mask>

Syntax:

msave (1) <mstack> <mask>

Pseudocode:

```
<mstack>.push(<mask>);  
<mask> = EMask;
```

Description:

The *msave* instruction pushes the value held in the specified mask register to the specified mask-stack, and then updates the mask value to the current execution mask (EMask). This is typically used just prior to entering a nested loop.

This instruction performs a mask-stack push/pop operation. Nesting depths must be tracked to ensure that a mask-stack under/overflow does not occur, or that an appropriate mask-stack exception handler is in place.

This instruction is scalar operation only; it can only update one mask stack at a time.

If SPF is set, this instruction updates <mask>, but does not update any mask stack.

Restrictions:

Predication is not allowed.

<exec_size> must be set to 1 (by Assembler for example).

Instruction compression is not allowed. Both source and destination operands must be unsigned integer. Source modifier is not allowed.



14.2.28 mul – Multiply

Opcode	Instruction	Description
65 (0x41)	mul <dst> <src0> <src1>	Performing component-wise multiplication of <src0> and <src1> and storing the results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	•	•	[FLT] [INT]	[FLT] [INT]

Format:

`[(<pred>)] mul[.<cmode>] (<exec_size>) <dst> <src0> <src1>`

Syntax:

`[(<pred>)] mul[.<cmode>] (<exec_size>) reg reg reg`
`[(<pred>)] mul[.<cmode>] (<exec_size>) reg reg imm32`

Pseudocode:

```

Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
    if (WrEn.chan[n] == 1) {
        dst.chan[n] = src0.chan[n] * src1.chan[n];
    }
}

```

Description:

The *mul* instruction performs component-wise multiplication of <src0> and <src1> and stores the results in <dst>.

When both <src0> and <src1> are of type D or UD, only the lower 16 bits of each element of <src0> are used. Accumulator maintains full 48-bit precision. Together with *mach* and *mov* instructions, full precision 64 bits multiplication results can be produced. (For that purpose, the type of <dst> must UD for the last *mov* instruction to get the lower dwords of partial results out.)

Multiplication of two floating point numbers follows rules in Table 14-5 (or Table 14-6), if the current floating point mode is IEEE mode (or ALT mode).



Table 14-5. Floating point multiplication of A (column) and B (row) in IEEE mode

	–inf	–finite	–1.0	–denorm	–0	+0	+denorm	+1.0	+finite	+inf	NaN
–inf	+inf	+inf	+inf	NaN	NaN	NaN	NaN	–inf	–inf	–inf	NaN
–finite	+inf	*	–A	+0	+0	–0	–0	A	**	–inf	NaN
–1.0	+inf	–B	+1.0	+0	+0	–0	–0	–1.0	–B	–inf	NaN
–denorm	NaN	+0	+0	+0	+0	–0	–0	–0	–0	NaN	NaN
–0	NaN	+0	+0	+0	+0	–0	–0	–0	–0	NaN	NaN
+0	NaN	–0	–0	–0	–0	+0	+0	+0	+0	NaN	NaN
+denorm	NaN	–0	–0	–0	–0	+0	+0	+0	+0	NaN	NaN
+1.0	–inf	B	–1.0	–0	–0	+0	+0	+1.0	B	+inf	NaN
+finite	–inf	**	–A	–0	–0	+0	+0	A	*	+inf	NaN
+inf	–inf	–inf	–inf	NaN	NaN	NaN	NaN	+inf	+inf	+inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Note:

- * Result may be {+finite, +inf (overflow)}
- ** Result may be {–inf (overflow), –finite}

Table 14-6. Floating point multiplication of A (column) and B (row) in ALT mode

	– fmax	–finite	–1.0	–denorm	–0	+0	+denorm	+1.0	+finite	+fmax	***
– fmax	+fmax	+fmax	+fma x	NaN	NaN	NaN	NaN	–fmax	–fmax	–fmax	
–finite	+fmax	*	–A	+0	+0	–0	–0	A	**	–fmax	
–1.0	+fmax	–B	+1.0	+0	+0	–0	–0	–1.0	–B	–fmax	
–denorm	+0	+0	+0	+0	+0	–0	–0	–0	–0	–0	
–0	+0	+0	+0	+0	+0	–0	–0	–0	–0	–0	
+0	–0	–0	–0	–0	–0	+0	+0	+0	+0	+0	
+denorm	–0	–0	–0	–0	–0	+0	+0	+0	+0	+0	
+1.0	–fmax	B	–1.0	–0	–0	+0	+0	+1.0	B	+fmax	
+finite	–fmax	**	–A	–0	–0	+0	+0	A	*	+fmax	
+fmax	–fmax	–fmax	– fmax	–0	–0	+0	+0	+fmax	+fmax	+fmax	

Note:

- * Result may be {+finite, +fmax (overflow)}
- ** Result may be {–fmax (overflow), –finite}
- *** Result is undefined If any of A and/or is {–inf, +inf, NaN}

Restrictions:

Source operands cannot be an accumulator register.

When operating on integers with at least one of the source being a dword type (signed or unsigned), the destination cannot be a float (implementation note: the data converter only looks at the lower 34 bits of the results).



Dword integer source is not allowed for this instruction in float execution mode. In other words, if one source is of type float (:f, :vf), the other source cannot be of type dword integer (:ud or :d).

When operating on integers with at least one of the source being a dword type (signed or unsigned), the Overflow and Sign flags are undefined. Therefore, conditional modifier and instruction operation ``.sat`` cannot be used.



14.2.29 `nop` – No Operation

Opcode	Instruction	Description
126 (0x7E)	<code>nop</code>	Issuing an dummy instruction and performing no operation.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types

Format :

`nop`

Syntax:

`nop`

Pseudocode:

n/a

Description:

The `nop` instruction takes an instruction dispatch but performs no operation. It may be used for assembly patching in memory, or be used to insert an instruction delay in the program sequence.

The `nop` instruction takes no operands, no instruction modifier, no conditional modifier and no predication.



14.2.30 not – Logic Not

Opcode	Instruction	Description
4 (0x04)	not <dst> <src0>	Performing component-wise logic NOT of <src0> and storing the results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•		•		•	•	[INT]	[INT]

Format:

[(<pred>)] not[.<cmod>] (<exec_size>) <dst> <src0>

Syntax:

[(<pred>)] not[.<cmod>] (<exec_size>) reg reg
[(<pred>)] not[.<cmod>] (<exec_size>) reg imm32

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
    if (WrEn.chan[n] == 1) {
        dst.chan[n] = !src0.chan[n]
    }
}
```

Description:

The *not* instruction performs logical NOT operation (or one's compliment) of <src0> and storing the results in <dst>.

Source modifiers are allowed.

Accumulator register is allowed to be the destination of this instruction with the restrictions listed below.

Restrictions:

This instruction does not work with float type operands.

Sign (SN) and Overflow (OF) conditions are undefined for this logic instruction. Consequently, saturation modifier (.sat) is not allowed.



This instruction does not implicitly update accumulator register.

When accumulator is the destination of this instruction, only the low bits corresponding to the data type (16 bits for word or 32 bits for dword integer instruction) in the accumulator contain the correct results. The internal extra-precision bits as well as the sign bit of the accumulator are undefined. Consequently, there are restrictions for subsequent instructions that use the data in the accumulator register created from the previous logical instruction.

- Only logical and data move instructions are allowed to source the accumulator. Results of other instructions (e.g. arithmetic or shift) are undefined.
- When the accumulator is the source of a data move (mov or sel) instruction, the destination operand must be of integer type (e.g. no conversion to float) and this instruction cannot have saturation instruction modifier.



14.2.31 or – Logic Or

Opcode	Instruction	Description
6 (0x06)	or <dst> <src0> <src1>	Performing component-wise logic OR of <src0> and <src1> and storing the results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•		•		•	•	[INT]	[INT]

Format:

[(<pred>)] or[.<cmod>] (<exec_size>) <dst> <src0> <src1>

Syntax:

[(<pred>)] or[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] or[.<cmod>] (<exec_size>) reg reg imm32

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
    if (WrEn.chan[n] == 1) {
        dst.chan[n] = src0.chan[n] | src1.chan[n];
    }
}
```

Description:

The *or* instruction performs component-wise logic OR operation between <src0> and <src1> and stores the results in <dst>.

Source modifiers are allowed.

Accumulator register is allowed to be the destination of this instruction with the restrictions listed below.

Restrictions:

Sign (SN) and Overflow (OF) conditions are undefined for this logic instruction. Consequently, saturation modifier (.sat) is not allowed.

This instruction does not work with float type operands.



This instruction does not implicitly update accumulator register.

When accumulator is the destination of this instruction, only the low bits corresponding to the data type (16 bits for word or 32 bits for dword integer instruction) in the accumulator contain the correct results. The internal extra-precision bits as well as the sign bit of the accumulator are undefined. Consequently, there are restrictions for subsequent instructions that use the data in the accumulator register created from the previous logical instruction.

- Only logical and data move instructions are allowed to source the accumulator. Results of other instructions (e.g. arithmetic or shift) are undefined.
- When the accumulator is the source of a data move (mov or sel) instruction, the destination operand must be of integer type (e.g. no conversion to float) and this instruction cannot have saturation instruction modifier.



14.2.32 pop – Mask Stack Pop

Opcode	Instruction	Description
47 (0x2F)	pop <mstack> <depth>	Popping <depth> level from <mstack> and updating <mstack>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
						[INT]	[INT]

Format:

pop (1) <mstack> <depth>

Syntax:

pop (1) <mask> reg32
pop (1) <mask> imm32

Pseudocode:

<mstack>.pop(1);

Description:

The *pop* instruction pops 'n' levels off the specific mask-stack structure. The <depth> field specified the number of levels to discard from the stack, and is always treated as an unsigned integer value.

This instruction performs a mask-stack push/pop operation. Nesting depths must be tracked to ensure that a mask-stack under/overflow does not occur, or that an appropriate mask-stack exception handler is in place.

Nesting depths must be tracked to ensure that a mask-stack under/overflow does not occur, or that an appropriate mask-stack exception handler is in place.

This instruction is scalar operation only; it can only update one mask stack at a time.

If SPF is set, this instruction does not update the mask stack.

Restrictions:

Predication is not allowed.

<exec_size> must be set to 1 (by Assembler for example).

Instruction compression is not allowed. Both source and destination operands must be unsigned integer. Source modifier is not allowed.

As the mask stacks are 16 deep, <depth> must be an unsigned dword integer with bits [31:4] set to zero. Effectively, it is a 4-bit unsigned integer.



14.2.33 push – Mask Stack Push

Opcode	Instruction	Description
46 (0x2E)	push <mstack> <mask>	Pushing <mask> onto <mstack>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
						[INT]	[INT]

Format :

push (1) <mstack> <mask>

Syntax:

push (1) <mstack> <mask>

Pseudocode:

<mstack>.push(<mask>);

Description:

The *push* instruction pushes the value held in the specified mask register to the specified mask-stack, and leaves the mask unchanged.

This instruction performs a mask-stack push/pop operation. Nesting depths must be tracked to ensure that a mask-stack under/overflow does not occur, or that an appropriate mask-stack exception handler is in place.

This instruction is scalar operation only; it can only update one mask stack at a time. <exec_size> must be set to 1 (for example by Assembler).

If SPF is set, this instruction does not update the mask stack.

Restrictions:

Predication is not allowed.

Instruction compression is not allowed. Both source and destination operands must be unsigned integer. Source modifier is not allowed.



14.2.34 *rndd* – Round Down

Opcode	Instruction	Description
69 (0x45)	<i>rndd</i> <dst> <src0>	Taking component-wise floating point downward rounding of <src0> and storing the results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	•	•	[FLT] [INT]	[FLT] [INT]

Format:

`[(<pred>)] rndd[.<cmode>] (<exec_size>) <dst> <src0>`

Syntax:

`[(<pred>)] rndd[.<cmode>] (<exec_size>) reg reg`
`[(<pred>)] rndd[.<cmode>] (<exec_size>) reg imm32`

Pseudocode:

```

Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
    if (WrEn.chan[n] == 1) {
        dst.chan[n] = floor(src0.chan[n]);
    }
}

```

Description:

The *rndd* instruction takes component-wise floating point downward rounding (to the integral float number closer to negative infinity) of <src0> and storing the rounded integral float results in <dst>. This is commonly referred to as the *floor()* function.

This instruction only applies to floating point source operands. Destination may be of float type or of integer type.

This is the only single instruction floating point rounding operation. The other three rounding modes (up, to-even, to-zero) require two instructions.

Output data <dst> and conditional flag Increment (IN) for floating point rounding-down follow rules in Table 14-7 (or Table 14-8), if the current floating point mode is IEEE mode (or ALT mode). Note again that conditional flag IN is 0 for all cases.



Table 14-7. Floating point round-down in IEEE mode

<src0>	–inf	–finite	–denorm	–0	+0	+denorm	+finite	+inf	NaN
<dst>	–inf	*	–0	–0	+0	+0	**	+inf	NaN
IN	0	0	0	0	0	0	0	0	0
Notes: * Result may be {–finite, –0}. ** Result may be {+finite, +0}.									

Table 14-8. Floating point round-down in ALT mode

<src0>	– fmax	–finite	–denorm	–0	+0	+denorm	+finite	+ fmax	***
<dst>	–fmax	*	–0	–0	+0	+0	**	+fmax	
IN	0	0	0	0	0	0	0	0	
Notes: * Result may be {–finite, –0}. ** Result may be {+finite, +0}. *** Result is undefined if <src0> is {–inf, +inf, NaN}.									

Restrictions:

This instruction cannot take accumulator as source or destination operand. However, when the accumulator is implicitly updated by this instruction, the results in the accumulator are undefined.



14.2.35 rndu – Round Up

Opcode	Instruction	Description
68 (0x44)	<code>rndu <dst> <src0></code>	Taking component-wise floating point upward rounding of <src0> and storing the results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	•	•	[FLT] [INT]	[FLT] [INT]

Format:

`[(<pred>)] rndu[.<cmod>] (<exec_size>) <dst> <src0>`

Syntax:

`[(<pred>)] rndu[.<cmod>] (<exec_size>) reg reg`
`[(<pred>)] rndu[.<cmod>] (<exec_size>) reg imm32`

Pseudocode:

```

Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
    if (WrEn.chan[n] == 1) {
        dst.chan[n] = floor(src0.chan[n]);
        if (src0.chan[n] - floor(src0.chan[n]) > 0.0f)
            dst.increment[n] = 1;
        else
            dst.increment[n] = 0;
    }
}

```

Description:

The *rndu* instruction takes component-wise floating point upward rounding (to the integral float number closer to positive infinity) of <src0> with results in two pieces – a downward rounded integral float results stored in <dst> and the round-up rounding increment stored in the increment bits. The rounding increments must be added to the results in <dst> to create the final round-up values to emulate the round-up operation, commonly known as the *ceiling()* function.

This instruction only applies to floating point source operands. Destination operand may be a floating point or an integer type operand.



The *ceiling()* function can be emulated in two instructions using *rndu*, for example, as

```
[(<pred>)] rndu.r.f# (<exec_size>) <dst> <src0>
(f#) add (<exec_size>) <dst> <dst> 1.0:f
```

The first instruction stores the round up increments in a flag register, *f#*, by turning on the conditional modifier. The second instruction increments *<dst>* by 1.0 predicated by *f#*. This works even when predication is on for the *rndu* instruction, as the set bits in the resulting flag register, *f#*, is a subset of the enabled channels of the previous instruction. Output data *<dst>* and conditional flag Increment (IN) for floating point rounding-up follow rules in Table 14-9 (or Table 14-10), if the current floating point mode is IEEE mode (or ALT mode). Note that conditional flag IN is 1 only if source data is a finite non-integral float.

Table 14-9. Floating point round-up in IEEE mode

<src0>	–inf	–finite	–denorm	–0	+0	+denorm	+finite	+inf	NaN
<dst>	–inf	*	–0	–0	+0	+0	**	+inf	NaN
IN	0	***	0	0	0	0	***	0	0
Notes: * Result may be {–finite, –0}. ** Result may be {+finite, +0}. *** Increment may be {0, 1}. It is 0 if source data is an integral float, and is 1 otherwise.									

Table 14-10. Floating point round-up in ALT mode

<src0>	– fmax	–finite	–denorm	–0	+0	+denorm	+finite	+ fmax	****
<dst>	–fmax	*	–0	–0	+0	+0	**	+fmax	
IN	0	***	0	0	0	0	***	0	
Notes: * Result may be {–finite, –0}. ** Result may be {+finite, +0}. *** Increment may be {0, 1}. It is 0 if source data is an integral float, and is 1 otherwise. **** Result is undefined if <src0> is {–inf, +inf, NaN}.									

Restrictions:

This instruction cannot take accumulator as source or destination operand. This instruction does change the content of accumulator, however, the results in the accumulator are undefined.



14.2.36 rnde – Round to Even

Opcode	Instruction	Description
70 (0x46)	Rnde <dst> <src0>	Taking component-wise floating point round-to-even operations of <src0> and storing the results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	•	•	[FLT] [INT]	[FLT] [INT]

Format:

`[(<pred>)] rnde[.<cmod>] (<exec_size>) <dst> <src0>`

Syntax:

`[(<pred>)] rnde[.<cmod>] (<exec_size>) reg reg`
`[(<pred>)] rnde[.<cmod>] (<exec_size>) reg imm32`

Pseudocode:

```

Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
    if (WrEn.chan[n] == 1) {
        dst.chan[n] = floor(src0.chan[n]);
        if (src0.chan[n] - floor(src0.chan[n]) > 0.5f) {
            dst.increment[n] = 1;
        } else if (src0.chan[n] - floor(src0.chan[n]) < 0.5f) {
            dst.carry[n] = 0;
        } else {
            if (dst.chan[n] is odd) {
                dst.increment[n] = 1;
            } else {
                dst.increment[n] = 0;
            }
        }
    }
}

```



Description:

The *rnde* instruction takes component-wise floating point round-to-even operation of <src0> with results in two pieces – a downward rounded integral float results stored in <dst> and the round-to-even increments stored in the rounding increment bits. The round-to-even increment must be added to the results in <dst> to create the final round-to-even values to emulate the round-to-even operation, commonly known as the *round()* function. The final results are the one of the two integral float values that is nearer to the input values. If the neither possibility is nearer, the even alternative is chosen.

This instruction only applies to floating point operands. Similar to *ceiling()*, the *round()* function can be emulated in two instructions using *rnde*, for example, as

```
[(<pred>)] rnde.c.f# (<exec_size>) <dst> <src0>
(f#) add (<exec_size>) <dst> <dst> 1.0:f
```

Output data <dst> and conditional flag Increment (IN) for floating point rounding-to-even follow rules in Table 14-11 (or Table 14-12), if the current floating point mode is IEEE mode (or ALT mode). Note that conditional flag IN may be 0 or 1 if source data is a finite non-integral float.

Table 14-11. Floating point round-to-even in IEEE mode

<src0>	–inf	–finite	–denorm	–0	+0	+denorm	+finite	+inf	NaN
<dst>	–inf	*	–0	–0	+0	+0	**	+inf	NaN
IN	0	***	0	0	0	0	***	0	0
Notes: * Result may be {–finite, –0}. ** Result may be {+finite, +0}. *** Increment may be {0, 1}. It is 0 if source data is an integral float. It may be 0 or 1 otherwise.									

Table 14-12. Floating point round-to-even in ALT mode

<src0>	– fmax	–finite	–denorm	–0	+0	+denorm	+finite	+ fmax	****
<dst>	–fmax	*	–0	–0	+0	+0	**	+fmax	
IN	0	***	0	0	0	0	***	0	
Notes: * Result may be {–finite, –0}. ** Result may be {+finite, +0}. *** Increment may be {0, 1}. It is 0 if source data is an integral float. It may be 0 or 1 otherwise. **** Result is undefined if <src0> is {–inf, +inf, NaN}.									

Restrictions:

This instruction cannot take accumulator as source or destination operand. However, when the accumulator is implicitly updated by this instruction, the results in the accumulator are undefined.



14.2.37 rndz – Round to Zero

Opcode	Instruction	Description
71 (0x47)	rndz <dst> <src0>	Taking component-wise floating point round-to-zero operations of <src0> and storing the results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•	•	•	•	•	•	[FLT] [INT]	[FLT] [INT]

Format:

`[(<pred>)] rndz[.<cmod>] (<exec_size>) <dst> <src0>`

Syntax:

`[(<pred>)] rndz[.<cmod>] (<exec_size>) reg reg`
`[(<pred>)] rndz[.<cmod>] (<exec_size>) reg imm32`

Pseudocode:

```

Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
    if (WrEn.chan[n] == 1) {
        dst.chan[n] = floor(src0.chan[n]);
        if (abs(src0.chann[n]) < abs(dst.chan[n])) {
            dst.increment[n] = 1;
        } else {
            dst.increment[n] = 0;
        }
    }
}

```

Description:

The *rndz* instruction takes component-wise floating point round-to-zero operation of <src0> with results in two pieces – a downward rounded integral float results stored in <dst> and the round-to-zero increments stored in the rounding increment bits. The round-to-zero increment must be added to the results in <dst> to create the final round-to-zero values to emulate the round-to-zero operation, commonly known as the *truncate()* function. The final results are the one of the two closest integral float values to the input values that is nearer to zero.



This instruction only applies to floating point operands. Similar to *ceiling()* and *round()*, the *truncate()* function can be emulated in two instructions using *rndz* instruction. For example, as

```
[(<pred>)] rndz.c.f# (<exec_size>) <dst> <src0>
(f#) add (<exec_size>) <dst> <dst> 1.0:f
```

Output data <dst> and conditional flag Increment (IN) for floating point rounding-to-zero follow rules in Table 14-13 (or Table 14-14), if the current floating point mode is IEEE mode (or ALT mode). Note that conditional flag IN is 0 for a source that is a positive float, and is 1 only for negative non-integral float.

Table 14-13. Floating point round-to-zero in IEEE mode

<src0>	–inf	–finite	–denorm	–0	+0	+denorm	+finite	+inf	NaN
<dst>	–inf	*	–0	–0	+0	+0	**	+inf	NaN
IN	0	***	0	0	0	0	0	0	0
Notes: * Result may be {–finite, –0}. ** Result may be {+finite, +0}. *** Increment may be {0, 1}. It is 0 if source data is an integral float, and is 1 otherwise.									

Table 14-14. Floating point round-to-zero in ALT mode

<src0>	– fmax	–finite	–denorm	–0	+0	+denorm	+finite	+ fmax	****
<dst>	–fmax	*	–0	–0	+0	+0	**	+fmax	
IN	0	***	0	0	0	0	0	0	
Notes: * Result may be {–finite, –0}. ** Result may be {+finite, +0}. *** Increment may be {0, 1}. It is 0 if source data is an integral float, and is 1 otherwise. **** Result is undefined if <src0> is {–inf, +inf, NaN}.									

Restrictions:

This instruction cannot take accumulator as source or destination operand. However, when the accumulator is implicitly updated by this instruction, the results in the accumulator are undefined.



14.2.38 sad2 – Sum of Absolute Difference 2

Opcode	Instruction	Description
80 (0x50)	sad2 <dst> <src0> <src1>	Performing a two-wide sum-of-absolute-difference operation on a 2-tuple basis of <src0> and <src1>, and storing the scalar result to the first channel per 2-tuple in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
	•	•	•	•	•	[INT]	[INT]

Format:

`[(<pred>)] sad2[.<cmod>] (<exec_size>) <dst> <src0> <src1>`

Syntax:

`[(<pred>)] sad2[.<cmod>] (<exec_size>) reg reg reg`
`[(<pred>)] sad2[.<cmod>] (<exec_size>) reg reg imm32`

Pseudocode:

```

Evaluate(WrEn);
for (n = 0; n < exec_size; n+=2) {
    if (WrEn.chan[n] == 1) {
        dst.chan[n] = abs(src0.chan[n] - src1.chan[n])
                     + abs(src0.chan[n+1] - src1.chan[n+1]);
    }
}

```

Description:

The *sad2* instruction takes source data channels from <src0> and <src1> in groups of 2-tuples. For each 2-tuple, it computes the sum-of-absolute-difference (SAD) between <src0> and <src1> and stores the scalar result in the first channel of the 2-tuple in <dst>.

This instruction only applies to integer operands. In particular, source operands must be unsigned bytes and/or signed bytes and destination operand must be of word type. Source modifiers are allowed.

The results are also stored in the accumulator register. Destination operand and accumulator maintain 16-bit per channel precision.

Destination register must have a stride of 2 bytes and must be aligned to even word. The even words in destination region will contain the correct data. The odd words are also written but with undefined values



Restrictions:

Source operands cannot be an accumulator register.

Execution size cannot be 1 as the computation requires at least two data channels.



14.2.39 sada2 – Sum of Absolute Difference Accumulate 2

Opcode	Instruction	Description
81 (0x51)	sada2 <dst> <src0> <src1>	Performing a two-wide sum-of-absolute-difference operation on a 2-tuple basis of <src0> and <src1>, added to that from the accumulator, and storing the scalar result to the first channel per 2-tuple in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
	•	•	•	•	•	[INT]	[INT]

Format:

[(<pred>)] sada2[.<cmode>] (<exec_size>) <dst> <src0> <src1>

Syntax:

[(<pred>)] sada2[.<cmode>] (<exec_size>) reg reg reg
[(<pred>)] sada2[.<cmode>] (<exec_size>) reg reg imm32

Pseudocode:

```

Evaluate(WrEn);
for (n = 0; n < exec_size; n+=2) {
    uwTmp = abs(src0.channel[n] - src1.channel[n])
           + abs(src0.channel[n+1] - src1.channel[n+1])
    if (WrEn.channel[n] == 1) {
        dst.channel[n] = uwTmp + acc[n]
    }
}

```

Description:

The *sada2* instruction takes source data channels from <src0> and <src1> in groups of 2-tuples. For each 2-tuple, it computes the sum-of-absolute-difference (SAD) between <src0> and <src1>, adds the intermediate result with the accumulator value corresponding to the first channel, and stores the scalar result in the first channel of the 2-tuple in <dst>.

This instruction only applies to integer operands. In particular, source operands must be unsigned bytes and/or signed bytes and destination operand must be of word type. Source modifiers are allowed.

The results are also stored in the accumulator register. Destination operand and accumulator maintain 16-bit per channel precision. Higher precision (guide bits)



stored in the accumulator allows multiple rounds (64 rounds) of sada2 instructions to be issued back to back without overflow the accumulator.

Destination register must have a stride of 2 bytes and must be aligned to even word. The even words in destination region will contain the correct data. The odd words are also written but with undefined values

Restrictions:

Source operands cannot be an accumulator register.

Execution size cannot be 1 as the computation requires at least two data channels.



14.2.40 sel – Select

Opcode	Instruction	Description
2 (0x02)	(pred) sel <dst> <src0> <src1>	Component-wise selective move from <src0> or <src1> to <dst> based on predication.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•		•	•		•	[FLT] [INT]	[FLT] [INT]

Format:

(<pred>) sel (<exec_size>) <dst> <src0> <src1>

Syntax:

(<pred>) sel (<exec_size>) reg reg reg
 (<pred>) sel (<exec_size>) reg reg imm32

Pseudocode:

```

Evaluate(WrEn, NoPMask);
Evaluate(PMask);
for (n = 0; n < exec_size; n++) {
    if (r.channel[n] == 1) {
        if (PMask.channel[n] == 1) {
            dst.channel[n] = src0.channel[n]
        } else {
            dst.channel[n] = src1.channel[n]
        }
    }
}

```

Description:

The *sel* instruction selectively moves the components in <src0> or <src1> into the channels of <dst> based on the predication. On a channel by channel basis, if the channel condition is true, data in <src0> is moved into <dst>; Otherwise, data in <src1> is moved into <dst>.

As the predication is used to select the two sources, it is not included in the evaluation of WrEn. <pred> is mandatory. If it is <omitted>, the results are unpredictable.



If <src0>, <src1> and <dst> are of different types, format conversion is performed.

Restrictions:

Destination channels cannot be on odd-byte sub-register locations. In other words, when destination is of byte type, destination horizontal stride cannot be 1. If destination horizontal stride is not 1, destination register region origin cannot be on an odd byte location. This is because that the conditional flag for execution channels that have minimal granularity of word are used by this instruction.

This instruction does not implicitly update accumulator registers.



14.2.41 send – Send Message

Opcode	Instruction	Description
49 (0x31)	send <post_dest> <curr_dest> <src0> <desc>	Performing an implied move from <src0> to <curr_dest>, sending a message stored in MRF starting at <curr_dest> to a shared function identified by <desc> with a GRF writeback location at <post_dest>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
		.	.	.	.	[FLT] [INT]	[FLT] [INT]

Format:

[(<pred>)] send (<exec_size>) <pdst> <cdst> <src0> <desc>

Syntax:

[(<pred>)] send (<exec_size>) reg reg reg reg32a
[(<pred>)] send (<exec_size>) reg reg reg imm32

Pseudocode:

```

Evaluate(EMask);
if (src0 != null) {
    MRFREG <mdest> = <curr_dest>.RegNum with
    <post_dest>.region;
    for (n = 0; n < exec_size; n++) {
        mdest.chan[n] = src0.chan[n];
    }
}
if (post_dest != null) {
    <ResponseReg> = <post_dest>.RegNum;
}
<MsgChEnable> = EMask;
<SourceReg> = <curr_dest>.RegNum;
MessageEnqueue(<MsgChEnable>, <ResponseReg>, <SourceReg>,
<desc>);

```



Description:

The *send* instruction performs data communication between a thread and external function units, including shared functions (Extended Math unit, Sampler, Data Port Read, Data Port Write and URB) and some fixed functions (e.g. Thread Spawner, who also have an unique Shared Function ID). The *send* instruction adds an entry to the EU's message request queue. The request message is stored in a block of contiguous MRF registers. The response message, if present, will be returned to a block of contiguous GRF registers. The return GRF writes may be in any order depending on the external function units. The current destination operand, `<curr_dest>`, is the lead MRF register for request. The posted destination operand, `<post_dest>`, is the lead GRF register for response. The message descriptor field `<desc>` contains the Message Length (the number of consecutive MRF registers) and the Response Length (the number of consecutive GRF registers). It also contains the target function unit ID and function control signals. The bit field definition of `<desc>` can be found in Table 14-15. EMask is forwarded to the target function in the message sideband.

In addition to enqueueing message request, the *send* instruction also performs an implied move from source operand `<src0>` to the current MRF destination `<curr_dest>`. The implied move is not subject to EMask. The *send* instruction is the only way to terminate a thread. When the EOT (End of Thread) bit of `<desc>` is set, it indicates the end of thread to the EU, the Thread Dispatcher and, in most cases, the parent fixed function.

Message descriptor field `<desc>` can be a 32-bit immediate, *imm32*, or a 32-bit scalar register, `<reg32a>`. The MSB of the message descriptor, the EOT field, always comes from bit 127 of the instruction word, which is the MSB of *imm32*. GEN4 restricts that the 32-bit scalar register `<reg32a>` must be the leading dword of the address register. It should be in the form of `a0.0<0;1,0>:ud`. When `<desc>` is a register operand, only the lower 31 bits of `<reg32a>` are used. A thread must terminate with a *send* instruction with EOT turned on.

`<curr_dest>` is a 256-bit aligned physical MRF register. Only the register number is present in the instruction word. It serves for two purposes: as the leading MRF register of the request message and as the destination of the implied move.

`<src0>` is used to signal whether there is an implied move or not. It can be either a null register or a GRF register. For other cases, hardware behavior is undefined. `<src0>` cannot be any ARF and cannot be register-indirect-addressed, as address register may be used as the operand for `<desc>`. If `<src0>` is a null register, there is no implied move. If it is a GRF register, there is an implied move from GRF to MRF. The implied move is fully described based on the parameters of the 'send' instruction. The destination operand of the implied move, `<move_dest>`, assumes the register number of `<curr_dest>` and the rest parameters from `<post_dest>`. The source operand of the implied move is `<src0>`. The implied move also assumes the same `<exec_size>`. The implied move turns on all the write enables for MRF destination, therefore, the whole MRF register of `<curr_dest>` is updated regardless of the content of EMask. The subfields belonging to the region of `<move_dest>` contain valid data, other subfields are undefined. The implied move may be used to save an instruction of moving the message header from GRF to MRF. It may also be used for header-less message going to the Extended Math unit by taking advantage of source swizzle specifications. Specifically, the 'send' instruction with implied move makes the



computation on Extended Math unit just like other arithmetic operations performed by the EU. In that case, the full MRF register can be viewed as pipeline storage to hold intermediate data post source swizzle and before the Extended Math unit performs the actual computation.

<post_dest> serves for three purposes: to provide the leading GRF register location for the response message if present, to provide parameters to form the channel enable sideband signals, and to provide the subregister parameters for the implied move.

<post_dest> signals whether there is a response to the message request. It can be either a null register, a direct-addressed GRF register or a register-indirect GRF register. Otherwise, hardware behavior is undefined.

If <post_dest> is null, there is no response to the request. Meanwhile, the Response Length field in <desc> must be 0. Certain types of message requests, such as memory write (store) through the Data Port, do not want response data from the function unit. If so, the posted destination operand can be null.

If <post_dest> is a GRF register, the register number is forwarded to the shared function. In this case, the target function unit must send one or more response message phases back to the requesting thread. The number of response message phases must match the Response Length field in <desc>, which of course cannot be zero. For some cases, it could be an empty return message. An empty return message is defined as a single phase message with all channel enables turned off.

The subregister number, horizontal stride, destination mask and type fields of <post_dest> are always valid and are used in part to generate the EMask. This is true even if <post_dest> is a null register (this is an exception for null as for most cases these fields are ignored by hardware). These parameters of <post_dest> follow the same restriction as that of normal destination operand – destination region cannot cross the 256-bit register boundary.

If the implied move is performed, the subregister number, horizontal stride and type fields of <post_dest> are used to describe the implied MRF destination register region.

The EMask corresponds to the 256-bit aligned data fields of the type of <post_dest>. EMask can have up to 16 bits. Therefore, fundamental type of <post_dest> can be either word or dword, but not byte. If <post_dest> is of dword, lower 8 bits of EMask (as dword enables) are generated based on the instruction parameters, and the higher 8 bits are forced to zero by hardware. If <post_dest> is of word, all 16 bits are generated. In other words, if execution size is 16, <post_dest> must be 256-bit aligned and must be of fundamental type of word. If execution size is 8 or less (yes, smaller execution size is allowed), the fundamental type of <post_dest> can be word or dword. The generation of EMask is subject to the following parameters: predication control if present, execution size, {subregister number, mask if present, horizontal stride if present, type} of <post_dest>, composite mask (from A/L/I/Cmask) if NoMask is not present, SecHalf if present. Only the 16-bit EMask is forwarded to the target function as the channel enable sideband signals; none of the input parameters are sent out. Therefore, these parameters may be manipulated to form a desired EMask according to the specifications of the target function. For example, in order to forward the 16-bit dword enables to the Data Port Write unit for a scattered dword store message, <post_dest> may be set to a null register of 'uw' type, even though the actual data stored in MRF are floating point values.



The 16-bit channel enables of the message sideband are formed based on the above mentioned EMask. Interpretation of the channel enable sideband signals is subject to the target external function. In general for a 'send' instruction with return messages, they are used as the destination dword write mask for the GRF registers starting at <post_dest>. For a message that has multiple return phases, the same set of channel enable signals applies to all the return phases.

Instruction compression is not allowed for this instruction. The hardware behavior is undefined if this instruction is set as compressed. However, compress control can be set to "SecHalf" to affect the EMask generation.

Thread managed memory coherency: A special usage of using non-null <post_dest> is to support write-commit signaling for memory write service by the Data Port Write unit. If <post_dest> is not null for a memory write request, the Data Port along with the Data Cache or Render Cache will wait until all the posted writes for the request have reached the coherent domain before sending back to the requesting thread an empty message to <post_dest> register. A memory write reaching the coherent domain, also referred to as reaching the global observable state, means that subsequent read to the same memory location, no matter which thread issues the read, must return the data of the write.

The destination dependency control, {NoDDClr}, can be used in this instruction. This allows software to control the destination dependencies for multiple 'read'-type messages similar to that for multiple instructions using EU execution pipeline. As *send* does not check register dependencies for the post destination, {NoDDChk} should not be used for this instruction.



Table 14-15. Message Descriptor Definition

Bit	Description
31	End Of Thread This field, if set, indicates that this is the final message of the thread and the thread's resources can be reclaimed. This field is also valid when the Message Descriptor is from a register. However, in that case, this field comes from the instruction word instead of the register.
30:28	Reserved : MBZ
27:24	Target Function ID This field indicates the function unit for which the message is intended. 0000 = Null 0001 = Extended Math 0010 = Sampling Engine 0011 = Message Gateway 0100 = Data Port Read 0101 = Data Port Write 0110 = URB 0111 = Thread Spawner 1000--1111: Reserved
23:20	Message Length. This field specifies the number of 256-bit MRF registers starting from <curr_dest> to be sent out on the request message payload. Valid value ranges from 1 to 15. A value of 0 is considered erroneous. Format = U4 Range = [1,15]
19:16	Response Length. This field indicates the number of 256-bit registers expected in the message response. The valid value ranges from 0 to 8. A value 0 indicates that the request message does not expect any response. The largest response supported by GEN4 is 8 GRF registers. Format = U4 Range = [0,8]
15:0	Function Control This field is intended to control the target function unit. Refer to the section on the specific target function unit for details on the contents of this field.

Table 14-16 provides a summary of the signals associated with each message that is sent to the shared function. It contains fields from the send instruction as well as fields from control register *cr0* and state register *sr0*.

Table 14-16. Sideband Signals Associated with Each Message Sent to the Shared Function

Signal	Bits	Source
EOT	1	End of Thread: Sourced from the EOT bit in <i>send</i> instruction word
SFID	3	Shared Function Identifier: Sourced from the target function ID field in <desc> of <i>send</i>
MLEN	4	Message Length: Sourced from the message length field in <desc> of <i>send</i>
RLEN	4	Response Length: Sourced from the response length field in <desc> of <i>send</i>
FC	16	Function Control: Sourced from the function control field in <desc> of <i>send</i>
REG	7	Destination Register: Sourced from the 256-bit register aligned register number of the <post_dest> field of <i>send</i>
CE	16	Channel Enable: Sourced from the execution mask of <i>send</i>
CLEAR	1	Destination Register Clear: Source from the Destination Dependency Control field (inverse of NoDDClr) in <i>send</i> instruction word
FFID	4	Fixed Function Identifier: Sourced from the Fixed Function ID field in <i>sr0</i>
EUID	4	Execution Unit Identifier: Sourced from the EUID field in <i>sr0</i>
TID	2	Thread Identifier: Sourced from the TID field in <i>sr0</i>
FPMODE	1	Floating Point Mode: Sourced from the floating point mode field in <i>cr0</i>

Restrictions:

Software must obey the following rules in signaling the end of thread using the *send* instruction:

- The posted destination operand must be null.
 - No acknowledgement is allowed for the *send* instruction that signifies the end of thread. This is to avoid deadlock as the EU is expecting to free up the terminated thread's resource.
- A thread must terminate with a *send* instruction with message to a shared function on the output message bus; therefore, it cannot terminate with a *send* instruction with message to the following shared functions: Sampler unit, Extended Math unit, NULL function
 - For example, a thread may terminate with a URB write message or a render cache write message.
- A root thread originated from the media (generic) pipeline must terminate with a *send* instruction with message to the Thread Spawner unit. A child thread should also terminate with a *send* to TS. Please refer to the Media Chapter for more detailed description.

This instruction does not implicitly update accumulator registers.



Implementation Restrictions: As the hardware does not check for post destination dependencies on this instruction, software must ensure that there is no destination hazard for the case of 'write followed by a posted write' shown in the following example.

1. `mov r3 0`
2. `send r3.xy <rest of send instruction>`
3. `mov r2 r3`

Due to no post-destination dependency check on the 'send', the above code sequence could have two instructions (1 and 2) in flight at the same time that both consider 'r3' as the target of their final writes. The first write to complete will release the dependency on 'r3' for subsequent instructions, thus allowing instruction 3 to proceed, even though one of the two prior instructions is still in-flight and will eventually modify 'r3'. Thus the proper instruction-order contents of 'r3' cannot be guaranteed for such a code sequence, and such sequence should not be used.

A proper code sequence which does guarantee program-order writes to 'r3' in this example is shown below. The solution is to use {NoDDClr} to the first instruction. Due to in-order instruction dispatch and known/fixed execution pipeline latencies, the posted write from the second instruction cannot bypass the first instruction, r3 update order from instructions 1 and 2 are ensured. So any dependencies on 'r3' will only be cleared by the posted write from the second instruction. Therefore, the third instruction will access the correct data from r3.

1. `mov r3 0 {NoDDClr}`
2. `send r3.xy <rest of send instruction>`
3. `mov r2 r3`

Another coding option is to insert a dummy move before the send instruction to stall the thread's execution until the first write to 'r3' has retired, this avoiding the potential destination hazard. However, this approach will consume additional instruction cycles waiting for 'r3' to retire.

1. `mov r3 0`
2. `mov null r3 // This instruction clears the scoreboard for r3`
3. `send r3.xy <rest of send instruction>`
4. `mov r2 r3`

Yet another coding option is to use {switch} instruction option on the instruction preceding the send instruction. The 'switch' instruction option will flush the execution pipe, ensuring all writes in the execution pipe to be retired.

1. `mov r3 0`
2. `<any instruction> {switch} // This instruction flushes the execution pipe`
3. `send r3.xy <rest of send instruction>`
4. `mov r2 r3`

This coding option is particularly useful for send instruction with indirect-addressed post destination, where the post destination registers are not known to the programmer.



As the retirement order for post-destinations of different send instructions may be unknown (e.g. from different shared functions), a posted write overlapping with a previously-issued posted write should be avoided in general, particularly for send instructions with indirect post-destinations.

[DevBW, DevCL] Errata: A destination register from a send cannot be used as a destination register until after it has been sourced by an instruction with a different destination register.



14.2.42 shl – Shift Left

Opcode	Instruction	Description
9 (0x09)	shl <dst> <src0> <src1>	Performing component-wise logic left shift of <src0> and storing the results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•		•	•	•	•	[INT]	[INT]

Format:

[(<pred>)] shl[.<cmod>] (<exec_size>) <dst> <src0> <src1>

Syntax:

[(<pred>)] shl[.<cmod>] (<exec_size>) reg reg reg
[(<pred>)] shl[.<cmod>] (<exec_size>) reg reg imm32

Pseudocode:

```
Evaluate(WrEn);  
for (n = 0; n < exec_size; n++) {  
    if (WrEn.chan[n] == 1) {  
        dst.chan[n] = src0.chan[n] << src1.chan[n]  
    }  
}
```

Description:

The *shl* instruction performs component-wise logical left shift of <src0> with zero-insertion and storing the results in <dst>. The amount of bit shift is provided by <src1>, where only the 5 LSBs of each channel of <src1> are used as an unsigned integer value. The MSBs of <src1> data channels are ignored. The results are NOT stored in the accumulator register.

5-bit shifting applies to packed-dword mode and packed-word mode. For packed word mode, the accumulators have 33 bits per channel. <src0> and <dst> can be signed or unsigned integers and can be of different types. This instruction does not work with float type operands. Saturation modifier is only allowed when this instruction is in packed-word mode. Hardware detects overflow properly and use it to perform saturation operation on the output, as long as the shifted result is within 33 bits. Otherwise, the result is undefined.

Results of saturation in packed-dword mode are unpredictable.

Restrictions:

This instruction does not work with float type operands.

This instruction does not implicitly update accumulator register.



14.2.43 shr – Shift Right

Opcode	Instruction	Description
8 (0x08)	shr <dst> <src0> <src1>	Performing component-wise logic right shift of <src0> and storing the results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•		•	•	•	•	[INT]	[INT]

Format:

`[(<pred>)] shr[.<cmode>] (<exec_size>) <dst> <src0> <src1>`

Syntax:

`[(<pred>)] shr[.<cmode>] (<exec_size>) reg reg reg`
`[(<pred>)] shr[.<cmode>] (<exec_size>) reg reg imm32`

Pseudocode:

```
Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
    if (WrEn.chan[n] == 1) {
        dst.chan[n] = src0.chan[n] >> src1.chan[n]
    }
}
```

Description:

The *shr* instruction performs component-wise logical right shift of <src0> with zero-insertion and storing the results in <dst>. The amount of bit shift is provided by <src1> where only the 5 LSBs of each channel of <src1> are used as an unsigned integer value. The MSBs of <src1> data channels are ignored.

5-bit shifting applies to packed-dword mode and packed-word mode. For packed word mode, the accumulators have 33 bits per channel.

This instruction only takes on unsigned sources. When <src0> contains unsigned integers, no source modifier is allowed. <src0> is only allowed to be signed integer if source modifier (abs) is used. *Note: for unsigned sources, the behavior of shr and asr are effectively the same.*

Restrictions:

This instruction does not work with float type operands.

This instruction does not implicitly update accumulator register.



14.2.44 wait – Wait Notification

Opcode	Instruction	Description
48 (0x30)	wait <nreg>	Waiting for notification on the notification register <nreg>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types

Format :

wait (<exec_size>) <nreg>

Syntax:

wait (1) n#

Pseudocode:

n/a

Description:

The *wait* instruction evaluates the value of the notification count register <nreg>. If <nreg> is zero, the execution of the thread is stalled and the thread is put in 'wait_for_notification' state. If <nreg> is not zero (i.e., one or more notifications have been received), <nreg> is decremented by one and the thread continues executing on the next instruction. If a thread is in the 'wait_for_notification' state, when a notification arrives, the notification count register is incremented by one. As the notification count register becomes non-zero, the thread wakes up to continue execution and at the same time the notification register is decremented by one. If there was only one notification arrived, the notification register value becomes zero. However, during the above mentioned time period, it is possible that more notifications may arrive, making the notification register non-zero again.

When multiple notifications are received, software must use 'wait' instruction to decrement notification count register for each of the notifications.

Notification register n0:ud is for thread to thread communication (through message gateway shared function) and n1:ud for host to thread communication (through MMIO registers). See Message Gateway chapter for thread-thread communication and Debug chapter for host-to-thread communication.

**Restrictions:**

Only one source operand.

<src0> and <dst> must be n0 or n1, <src1> must be *null*.

Execution size must be 1 as the notification registers are scalar.

Predication is not allowed.

Implementation restriction: Two back-to-back *wait* instructions in a program (without any instruction in between) are not allowed. As a minimal, a *nop* has to be inserted between two *wait* instructions.



14.2.45 while – While

Opcode	Instruction	Description
39 (0x27)	while <exitcode>	Marking the end of a do-while block of code.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
		•					

Format:

[[<pred>]] while (<exec_size>) <exitcode>

Syntax:

[[<pred>]] while (<exec_size>) imm32

Pseudocode:

```

Evaluate(EMask, NoCMask);
if (EMask != 0) {
    CMask = EMask;
    LMask = EMask;
    Jump(<exitcode.IPCount>);
} else {
    CMask = LStack.TopOfStack;
    LStack.Pop(1);
    LMask = LStack.TopOfStack;
    LStack.Pop(1);
}

```

Description:

The *while* instruction marks the end of a do-while block. The instruction first evaluates the loop termination condition for each channel as determined by the bit-wise AND of the AMask, LMask, and the predication flag specified in the instruction. Both CMask and LMask are updated to this value for any further passes through the loop. If any channel has not terminated as indicated by its bit position = 1, a branch is taken to a destination address based specified in the instruction, and the loop continued for those channels.

Once all channels have been suspended (as indicated by LMask = ...000b), the loop is terminated by popping CMask and then LMask from the LStack, enabling those data channels active prior to entering the loop. The instruction sequence



then falls through and execution continues with the instruction immediately following the 'while'. Note that this instruction also updates CMask to ensure that CMask is always a subset of LMask.

Also note that the EMask evaluation does not consider CMask as the channels temporally turned off by CMask in the current loop need to be considered in the conditional evaluation in the *while* instruction.

The following table describes the 32-bit exit code <exitcode>. <InstCount> is a signed 16-bit number, added to IP pre-increment, and should point to the first instruction after the *do* instruction of the do-while block of code. It should be a negative number for the backward referencing.

In GEN4 binary, <exitcode> is at location <src1> and must be of type D (signed doubleword integer).

Bit	Description
31:16	Reserved: MBZ
15:0	InstCount (Jump Instruction Count) . This field specifies the jump distance in Instruction Count if a jump is taken for the instruction. Format = S15. Signed integer in 2's complement

This instruction executes regardless of the calculated EMask at the time of issue. It invokes a thread switch after issue to allow any masks and/or IP to be resolved if necessary.

This instruction pops both CMask and LMask from LStack upon falling through.

This instruction performs a mask-stack push/pop operation. Mask-stack push/pop operations are always done in 16-bit width regardless of execution size. Nesting depths must be tracked to ensure that a mask-stack under/overflow does not occur, or that an appropriate mask-stack exception handler is in place.

If SPF is set, this instruction does not update any mask stack. When this instruction falls through, it restores the CMask and LMask from LStack.TopOfStack, which supposes to be static for code with single program flow.

If a jump is invoked by this instruction, IMask will be fully restored, same as LMask and CMask. ISPopCount = 0 is a trivial case that IMask is not changed.

Restrictions:

Instruction compression is not allowed.

IP register must be put (for example, by the assembler) at <dst> and <src0> locations.



14.2.46 xor – Logic Xor

Opcode	Instruction	Description
7 (0x07)	xor <dst> <src0> <src1>	Performing component-wise logic XOR of <src0> and <src1> and storing the results in <dst>.

Instr Comp	Imp'd Accu	Pred	Sat	Cond Mod	Src Mod	Src Types	Dst Types
•		•		•	•	[INT]	[INT]

Format:

[(<pred>)] xor[.<cmod>] (<exec_size>) <dst> <src0> <src1>

Syntax:

[(<pred>)] xor[.<cmod>] (<exec_size>) reg reg reg
 [(<pred>)] xor[.<cmod>] (<exec_size>) reg reg imm32

Pseudocode:

```

Evaluate(WrEn);
for (n = 0; n < exec_size; n++) {
    if (WrEn.chan[n] == 1) {
        dst.chan[n] = src0.chan[n] ^ src1.chan[n];
    }
}

```

Description:

The *xor* instruction performs component-wise logic XOR operation between <src0> and <src1> and stores the results in <dst>.

Source modifiers are allowed.

Accumulator register is allowed to be the destination of this instruction with the restrictions listed below.

Restrictions:

Sign (SN) and Overflow (OF) conditions are undefined for this logic instruction. Consequently, saturation modifier (.sat) is not allowed.

This instruction does not work with float type operands.



The results are NOT stored in the accumulator register.

When accumulator is the destination of this instruction, only the low bits corresponding to the data type (16 bits for word or 32 bits for dword integer instruction) in the accumulator contain the correct results. The internal extra-precision bits as well as the sign bit of the accumulator are undefined. Consequently, there are restrictions for subsequent instructions that use the data in the accumulator register created from the previous logical instruction.

- o Only logical and data move instructions are allowed to source the accumulator. Results of other instructions (e.g. arithmetic or shift) are undefined.
- o When the accumulator is the source of a data move (mov or sel) instruction, the destination operand must be of integer type (e.g. no conversion to float) and this instruction cannot have saturation instruction modifier.



15 EU Programming Guide

15.1 Assembler Pragmas

15.1.1 Declarations

A register or a register region can be declared as a symbol using the following form

```
.declare <symbol>      Base=RegFile RegBase {.SubRegBase}  
ElementSize=ElementSize {SrcRegion=DefaultSrcRegion}  
{DstRegion=DefaultDstRegion} {Type=DefaultType}
```

The register file, the base of the register origin and the element size (in unit of bytes) are the mandatory parameters for a declared register region. Optionally, the base of the sub-register address, the default source region, the default destination region and the default type can be provided in the declaration for the symbol.

For immediate register addressing mode, the declared symbol can be used in the following Cartesian form

$$\text{<symbol> (RegOff, SubRegOff)} \leftarrow \text{RegNum} = \text{RegBase} + \text{RegOff}; \text{SubRegNum} = \text{SubRegBase} + \text{SubRegOff}$$

or in the following simplified row-aligned form

$$\text{<symbol> (RegOff)} \leftarrow \text{RegNum} = \text{RegBase} + \text{RegOff}; \text{SubRegNum} = \text{SubRegBase}$$

For register-indirect-register-addressing mode, the declared symbol can be used to provide immediate address term in the following Cartesian form

$$\text{<symbol> [IdxReg, RegOff, SubRegOff]} \leftarrow \text{RegNum (byte-aligned)} = [\text{IdxReg}] + (\text{RegBase} + \text{RegOff}) * 32 + (\text{SubRegBase} + \text{SubRegOff}) * \text{ElementSize}$$

or in the following simplified row-aligned form

$$\text{<symbol> [IdxReg, RegOff]} \leftarrow \text{RegNum (byte-aligned)} = [\text{IdxReg}] + (\text{RegBase} + \text{RegOff}) * 32$$

or in the form without the immediate address term

$$\text{<symbol> [IdxReg]} \leftarrow \text{RegNum (byte-aligned)} = [\text{IdxReg}] + \text{RegBase}$$



15.1.2 Defaults and Defines

The default execution size is set according to the destination register type as the following

Destination Register Type	Default Execution Size
UB B	(16)
UW W	(16)
F UD D	(8)

The default execution size can be overwritten globally for all instructions using

.default_execution_size (<i>Execution_Size</i>)

or be set according the **destination** register type using

.default_execution_size_Type (<i>Execution_Size</i>)

The default register type can be set for all register files using

.default_register_type <i>Type</i>

or be set per register file using

.default_register_type_RegFile <i>Type</i>

The default **source** register region for all symbols can be set using

.default_source_register_region < <i>VirtStride; Width, HorzStride</i> >

or be set per register type using

.default_source_register_region_type < <i>VirtStride; Width, HorzStride</i> >

The default **destination** register region for all symbols can be set using

.default_destination_register_region < <i>HorzStride</i> >

or be set per register type using

.default_destination_register_region_type < <i>HorzStride</i> >



Finally, the precompiler supports the string replacement statement of `.define` in the following form

.define <symbol>	Expression

Notes:

- **.declare** does not support nesting. In other words, each symbol in `.declare` must be self defined. This would allow the pre-processor to expand all symbols in one pass.
- **.define** does support nesting. Only string substitution is supported (currently).
- White space within square, angle and round brackets are allowed for easy source code alignment.

15.1.3 Example Pragma Usages

Example 1: Declaration for 8x4=32-Byte Regions:

The following symbol *Block* can be used to address any 8x4 byte region within the Cartesian system of a 16x8 byte GRF register area starting from r0.

Declaration	// 32x4 Byte Array .declare Block Base=r0 ElementSize=1 Region=<32;8,1> Type=b		
Fully-Expressed Instr	mov(32) ?:b r0.16<32;8,1>:b	//	r0
	XXXXXXXXXXXXXXXXXXXXXXXXXXXX	//	r1
	XXXXXXXXXXXXXXXXXXXXXXXXXXXX	//	r2
	XXXXXXXXXXXXXXXXXXXXXXXXXXXX	//	r3
Short-handed Instr	Mov ?:b Block(0,16)	// (0,16):	RegNum=0,
	SubRegNum=16		

Example 2: Declaration for 8x1 Float Regions:

The following symbol *Trans* can be used to address any 8x1 float region within the Cartesian system of a 8x4 float GRF register area starting from r5.

Declaration	// 8x4 float Array starting at r5 .declare Trans Base=r5 ElementSize=4 Region=<0;8,1> Type=f		
Fully-Expressed Instr	mov(8) ?:f r6.0<0;8,1>:f	// 2 nd 16x1 Row of	
	Trans. Matrix		
		// r5	FFFFFFF
		// r6	00000000
		// r7	FFFFFFF
		// r8	FFFFFFF



Short-handed Instr	<code>mov ?:f Trans(1) // RegNum = 5+1 = 6</code>

Example 3: Declaration for 8x1 Float Regions with 1x1 Indirect Addressing:
Trans region defined (same as in the previous example) is used in conjunction with the address register.

Declaration	<code>//8x4 float data array and 16x1 word address array .declare Trans Base=r5 ElementSize=4 Region=<0;8,1> Type=f</code>
Fully-Expressed Instr	<code>mov(8) ?:f r[a0.0,224]<0;8,1>:f</code>
Short-handed Instr	<code>mov ?:f Trans[a0.0,2] // [a0.0 + 5*32 + 2*32]</code>

Example 4: Declaration with VxH Indirect Addressing:
The VxH register-indirect-register-addressing for *Trans* can be provided in the following short-hand form.

Declaration	<code>//8x4 float data array and word indices .declare Trans Base=r5 ElementSize=4 Region=<0;8,1> Type=f</code>
Fully-Expressed Instr	<code>mov(8) ?:f r[a0.0,224]<1,0>:f</code>
Short-handed Instr	<code>mov ?:f Trans[a0.0,2]<1,0> // [a0.0+224] [a0.1+224] ... [a0.7+224]</code>

Example 5: Declaration with Vx1 Indirect Addressing:
As width (4) is smaller than the execution region size (8), multiple indexed registers are used.

Declaration	<code>//8x4 float data array and word address array .declare Trans Base=r5 ElementSize=4 Region=<0;8,1> Type=f</code>
Fully-Expressed Instr	<code>mov(8) ?:f r[a0.0,244]<4,1>:f</code>
Short-handed Instr	<code>mov ?:f Trans[a0.0,2]<4,1> //[a0.0+224] [a0.1+224]</code>

15.1.4 Assembly Programming Guideline

The following program skeleton illustrates the basic structure of a typical assembly program.

```
// single line comment

/*
    block comment
*/
```




```

<preproc_directive> // macros, include, etc. Are global - handled by the pre-
processor
<preproc_directive> // applies to all code that follows in sequence

// ----- some kernel -----
.kernel <kernel_name_string> // [REQUIRED]
                                // ----- Register requirements -----
.reg_count_total <uint> // [REQUIRED] a more direct way to specify the exact
parameters require
.reg_count_payload <uint> // [REQUIRED] rather than to have to indirectly do
that by adding the
                                // the payload and temps together to get the total
(as is the case now)
                                // Note: no more "reg-count-temp"

                                // ----- Defaults -----
<default...> // these should be specified per-kernel and have
only kernel-scope
<default...> // Same defaults as those already defined in the ISA
doc, but just
<default...> // moved within the kernel to make each kernel
completely self-sufficient
                                // and not impacted defaults of earlier kernels

                                // ----- Memory Requirements -----
                                // [optional] memory block info (just a placeholder
for now...)
<MBDa> // memory block descriptor a (TBD)
<MBDb> // memory block descriptor b (TBD)
<MBDc> // memory block descriptor c (TBD)
<MBDd> // memory block descriptor d (TBD)

                                // ----- Code -----
.code // [REQUIRED]
    <instruction>
    <instruction>
    <instruction>
    <LabelLine> // labels are code-block scope
    <instruction>
    <instruction>

.end_code // [REQUIRED]

.end_kernel // [REQUIRED]

// ----- next kernel -----

// ----- next kernel -----

// ...

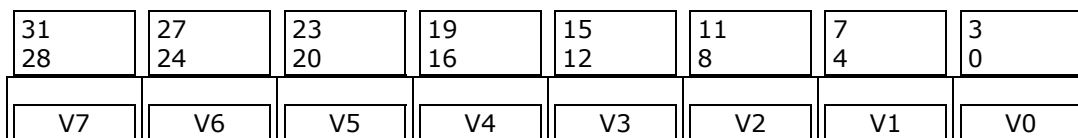
```



15.2 Usage Examples

15.2.1 Vector Immediate

The immediate form of vector allows a constant vector to be in-lined in the instruction stream. An immediate vector is denoted by type *v* as *imm32:v*, where the 32-bit immediate field is partitioned into 8 4-bit subfields. Each 4-bit subfield contains a signed integer value in 2's complement form. Therefore each 4-bit subfield has a range of [-8, +7]. This is depicted in the following figure.



15.2.1.1 Supporting Pixel Shader Indexing

When a pixel shader program is converted to run on Gen4 in channel-serial mode at 16 pixels in parallel, the per-pixel index must be translated into 16 indices with per channel offset. The creation of the per-channel offset can be achieved using the vector immediate.

Consider a generic pixel shader instruction in the form of

```
op r4 r[ind] r2
```

and assume that r0-r1 contain the 16 indices packed every other words, and r2-r3 contains source 1 and r4-r5 contain the destination. This instruction can be converted into the following Gen4 instructions. The corresponding operations are illustrated in Figure 15-1.

```
mov (16) r11.0<1>:w 0x01234567:v // assigning a ramp vector, repeated once
```

```
mul (16) acc0:w r11.0<0;16,1>:w 4:w // expand ramp range to 4 bytes per step
```

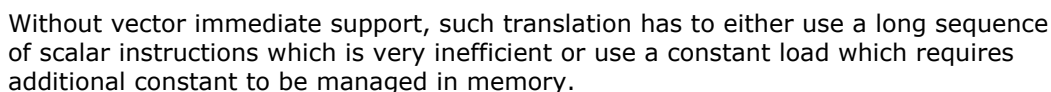
```
mac (16) r10.0<1>:w r0.0<16;8,2>:w 32:w // r10 = index*32 + 0|4|...|28|0|4...|28
```

```
mov (8) a0.0<1>:w r10.0<0;8,1>:w
```

```
op (8) r4.0<1>:f r[a0.0]<1,0>:f r2.0<0;8,1>:w // Operate on the first half
```

```
mov (8) a0.0<1>:w r10.8<0;8,1>:w // Index values are off by a reg (32b)
```

```
op (8) r5.0<1>:f r[a0.0+32]<1,0>:f r3.0<0;8,1>:w // Operate on the second half.
```



When an OpenGL Vertex Shader program is converted to run on Gen4 in Vertex Pair, i.e. two 4-wide vectors in parallel, the special OpenGL Shader instruction SWZ (Swizzle) needs to be emulated. OpenGL SWZ instruction uses an extended swizzle control field that, in addition to the 4-wide full swizzle control, also includes constant 0 and 1 replacement as well as per channel sign reversal. The later two are not supported by the Gen4 native instruction. The vector immediate can significantly reduce the overhead of emulating such OpenGL instruction.

```
SWZ    r1, r0.0-zx-1 // Expected results: r1.x = 0; r1.y = -r0.z; r1.z = r0.x; r1.w = -1
```

```
mul(8) r1.0<1>:f r0.xzxx 0x1F111F11:v // Constant vector of (1 -1 1 1 1 -1 1 1)
```

```
(f0.0)mov(8) r1.0<1>:f    0x000F000F:v           // Constant vector of (0 0 0 -1 0 0  
0 -1)
```

OpenGL:

510



Gen4:

```
mov (1)    f0.08b'01100110          // Set flag and masked out channels x and w
(f0.0)sel (8) r1.0<1>:f r0.yzxy 0x000F000F:v // Constant vector of (0 0 0 -1 0 0 0 -1)
```

15.2.2 Destination Mask for DP4 and Destination Dependency Control

The following example demonstrates the use of destination mask mode of floating point dot-product instruction as well as the use of destination dependency control to improve performance (i.e., avoiding unnecessary thread switch due to possible false dependencies).

Consider a generic vertex shader of matrix-vector product that is implemented on Gen4 in the pair of 4-component vector mode. The equivalent Shader instructions are as the following.

```
dp4 r5.x r0 r4
dp4 r5.y r1 r4
dp4 r5.z r2 r4
dp4 r5.w r3 r4
```

With destination dependency control, the Gen4 instructions are as the following. The first instruction in the sequence checks for the destination dependency, but does not clear the dependency bit. The subsequent two instructions would do neither of them. The last instruction avoids checking the destination dependency, but at completion, it clears the destination scoreboard. It ensures that the content of the destination register is coherent, if any of the following instructions uses the same register as source.

```
dp4 (8) r5.0<1>.x:f r0.0<4;4,1>:f r4.0<4;4,1>:f {NoDDClr}
dp4 (8) r5.0<1>.y:f r1.0<4;4,1>:f r4.0<4;4,1>:f {NoDDClr, NoDDCChk}
dp4 (8) r5.0<1>.z:f r2.0<4;4,1>:f r4.0<4;4,1>:f {NoDDClr, NoDDCChk}
dp4 (8) r5.0<1>.w:f r3.0<4;4,1>:f r4.0<4;4,1>:f {NoDDCChk}
```



Just as a comparison, IF Gen4 DP4 implies reduction at the destination; additional shifted moves are required to achieve the same results. The corresponding codes are as the following. The lower performance due to the additional three move instruction as well as added back-to-back dependencies shows that why we choose to implement the destination channel replication for floating point DP4.

```
dp4 (8) r5.0<1>.y:f    r1.0<4;4,1>:f r4.0<4;4,1>:f
mov (1) r5.1<1>:f r8.0<1;1,1>:f

dp4 (8) r5.0<1>.z:f    r2.0<4;4,1>:f r4.0<4;4,1>:f
mov (1) r5.2<1>:f r8.0<1;1,1>:f

dp4 (8) r5.0<1>.w:f    r3.0<4;4,1>:f r4.0<4;4,1>:f
mov (1) r5.3<1>:f r8.0<1;1,1>:f

dp4 (8) r5.0<1>.x:f    r0.0<4;4,1>:f r4.0<4;4,1>:f
```

15.2.3 Null Register as the Destination

Null register can be used as the destination for most of the instructions. Here are some example usages.

- Null as destination for regular ALU instructions: As all ALU instructions can be configured to update the flag registers using the conditional modifiers, it is not necessary to have a destination register if the programmer only cares about the conditionals of the operation. In that case, a null in the destination operand field saves register space as well as one less dependency checking.
- Null as the destination for SEND/STOR instructions: for the send instruction that only send messages out to an external unit and does not require any return data or feedback, a null in the destination register field signifies the case.
 - One extension of such case is that even though the operation does not have any return values, a return phase with no payload but simply updating the scoreboard flag for a non-null register can provide a signaling mechanism between the thread and the target external unit. One application of this usage is to allow software to manage the coherency of shared memory resources such like the many caches in the system (particularly, valuable for read/write caches). This is not currently the POR for Gen4 though.



15.2.4 Use of LINE Instruction

LINE instruction is specifically designed to speed up floating point vector/matrix computation when a program operates in channel serial.

The following example demonstrates how to use LINE instruction to compute line equations for a pixel shader. In this example, 2 sets of (Cx#, Cy#, Don't Care, C0#) 4-tuple coefficient vectors are stored in registers R1.

R1: Cx0 Cy0 DC Co0 Cx1 Cy1 DC Co1

8 sets of coordinate 2-D vectors (X, Y) are stored in R2 and R3 in the channel serial mode as

R2: X0 X1 ... X7
R3: Y0 Y1 ... Y7

The objective is to compute the following two line equations for each set of 2D coordinate and store the results in R4 and R5 as

R4: $(X0 \cdot Cx0 + Y0 \cdot Cy0 + Co0) \dots (X7 \cdot Cx0 + Y7 \cdot Cy0 + Co0)$

R5: $(X0 \cdot Cx1 + Y0 \cdot Cy1 + Co1) \dots (X7 \cdot Cx1 + Y7 \cdot Cy1 + Co1)$

Example 15-1. LINE Equations

```
//-----
// Example compute LINE equation in channel serial scenario
//-----

line (8) acc:f r1<0;1,0>:f r2<0;8,1>:f // does acc = X# * Cx0 + Co0
mac (8) r4<1>:f r1.1<0;1,0>:f r3<0;8,1>:f // does r4.# = Y# * Cy0 + acc.#

line (8) acc:f r1<0;1,0>:f r2<0;8,1>:f // does acc = X# * Cx0 + Co0
mac (8) r4<1>:f r1.1<0;1,0>:f r3<0;8,1>:f // does r4.# = Y# * Cy0 + acc.#
```

The next example is to compute homogeneous dot product for OpenGL pixel shader running in Channel Serial. In this example, an original OpenGL PS instruction is like

dph R2.x R0 R1

With register remapping, we can store the input coefficient vector R0 in original format in r0, but 8 sets of input coordinate vectors in channel serial format in r2, r3, r4 and r5, and the destination R2.x component in r6.

r0: Cx0 Cy0 Cz0 Co0 DC DC DC DC
r2: X0 X1 ... X7
r3: Y0 Y1 ... Y7
r4: Z0 Z1 ... Z7
r5: W0 W1 ... W7



The objective is to compute the following DPH equations and store the results in r6 as
R6: $(X0 \cdot Cx0 + Y0 \cdot Cy0 + Z0 \cdot Cz0 + Co0) \dots (X7 \cdot Cx0 + Y7 \cdot Cy0 + Z7 \cdot Cz0 + Co0)$

Example 15-2. Homogeneous Dot Product in Channel Serial

```
//-----  
// Example compute homogeneous dot product in channel serial scenario  
//-----  
  
line (8) acc:f  r0<0;1,0>:f  r2<0;8,1>:f      // does acc = X# * Cx0 + Co0  
mac  (8) acc:f  r0.1<0;1,0>:f r3<0;8,1>:f      // does acc.# = Y# * Cy0 + acc.#  
mac  (8) r6<1>:f r0.2<0;1,0>:f r4<0;8,1>:f      // does r6.# = Z# * Cz0 + acc.#
```



15.2.5 Mask for SEND Instruction

Execution mask (upto 16 bits) for the SEND instruction is transferred to the Shared Function. This provides optimized implementation of shader instructions.

15.2.5.1 Channel Enables for Extended Math Unit

The following example demonstrates how to use the SEND instruction to get service from the Extended Math unit.

Let's consider COS instruction in the following form:

```
[(!]p0.{select|any|all})] cos[_sat] dest[.mask], [-]src0[_abs][.swizzle]
```

For a SIMD4x2 VS implementation with the following register mappings:

```
p0      →  f0.0
src0     →  r0
dest     →  r1
```

The equivalent Gen4 instruction is as the following:

```
[(!]f0.0.{select|any4h|all4h})] SEND (8) r1[.mask]:f m0 [-][abs]r0[.swizzle]:f
MATHBOX|COS[|SAT]
```

If the source swizzle is replication, the message description field can be modified to MATHBOX|COS|SCALAR to take advantage of the fast mode (scalar mode) supported by the Extended Math. The implied move of the SEND instruction is equivalent to the following instruction:

```
MOV (8) m0[.mask]:f [-][abs]r0.0[.swizzle]:f {NoMask}
```

For a SIMD16 PS implementation, the register mappings are as the followings

```
p0      →  f0...f3           // in order of R, G, B, A
src0     →  r0,r1; r2,r3; r4,r5; r6,r7
dest     →  r8,r9; r10,r11; r12,r13; r14,r15
```

There are several ways to translate the instruction, depending on the operand/instruction modifiers present in the instruction. If predicate is not present and the source swizzle is replication, say, src0.y, which is r2-r3, the translation could be as the following instructions:

```
send (8) r8:f m0  -(abs)r2:f MATHBOX|COS
```




```
send (8) r9:f m1  -(abs)r3:f MATHBOX|COS {SecHalf}  // use the second half of 8
flag bits
```

```
mov (16) r10:f r8:f          // All destination color chan's are same
```

```
mov (16) r12:f r8:f          // MOV is faster than most MathBox func's
```

```
mov (16) r14:f r8:f          // These MOV's are compressed instructions
```

Notice that instead of issuing Extended Math messages with the same input data, destination color channel replication is performed by the MOV instructions. This is faster for the thread for most cases as many Extended Math functions consume multiple cycles. This also conserves message bus bandwidth as well as the usage of the shared resource – Extended Math. The destination mask in the instruction indicates which of the r8 to r15 registers are updated. If the source swizzle is not replication, there will be 8 SEND instructions.

With predication on, if the predication modifier is p0.select, translation is to take the selected flag register f#. The other predication modifiers '.any' and '.all' are translated into '.any4v' and '.all4v', respectively. Notice that with predication on, it is not required to run all 4 pixels in a subspan in the same way, so no need to enforce .any4h/.any4v. The following example shows the instruction with predication (but without .select modifier).

```
(f0[.any4v|.all4v]) send (8) r8:f m0  -(abs)r2:f MATHBOX|COS
```

```
(f0[.any4v|.all4v]) send (8) r9:f m1  -(abs)r3:f MATHBOX|COS {SecHalf}
```

```
(f1[.any4v|.all4v]) mov (16) r10:f r8:f          // All destination color chan's are same
```

```
(f2[.any4v|.all4v]) mov (16) r12:f r8:f          // MOV is faster than most MathBox
func's
```

```
(f3[.any4v|.all4v]) mov (16) r14:f r8:f          // These MOV's are compressed
instructions
```

The same instructions works also for predication with select component modifier. We simply replase f0 to f3 above by the selected flag register, say, f1. The modifier of any4h/all4v would also work.



15.2.5.2 Channel Enables for Scratch Memory

The following example demonstrates how to use the SEND instruction to get service from the Data Port for scratch memory access.

Let's consider general instruction that uses scratch memory as a source operand

```
[[(!]p0.{select|any|all})] add dest[.mask], [-]src0[_abs][.swizzle], [-]src1[_abs][.swizzle]
```

For a SIMD4x2 VS implementation with the following register mappings

```
p0      →  f0
src0     →  r0
src1     →  s2 / r10
dest     →  r1
```

In this example, the scratch memory offset is provided by an immediate and a GRF register r10 is used as the intermediate GRF location for spill/fill of scratch buffer accesses. This arithmetic instruction is converted into a Data Port read followed by an arithmetic instruction.

```
mov (8) r3:d r0:d {NoMask} // move scratch base address to be assembled with offset values
```

```
mov (1) r3.0:d 2*32 {NoMask} // s2 for vertex 0
```

```
mov (1) r3.1:d 2*32+16 {NoMask} // s2 for vertex 1
```

```
send (8) r10 m0 r3 DATAPORT|RC|READ_SIMD2
```

```
[[(!]f0.{sel|any4h|all4h})] add (8) r1[.mask]:f [-][abs)r0[.swizzle]:f [-][abs)r10[.swizzle]:f
```

So if scratch register is the source, there is no need to use the channel enable side band. This is also true for channel-serial PS cases.

Now, let's consider the case when a scratch register is the destination of an instruction.

```
p0      →  f0
src0     →  r0
src1     →  r1
dest     →  s2 / r10
```

We have

```
add (8) m1:f [-][abs)r0[.swizzle]:f [-][abs)r1[.swizzle]:f
```



```
mov (8) r3:d r0:d {NoMask} // move scratch base address to be assembled with
offset values
```

```
mov (1) r3.0:d 2*32 {NoMask} // s2 for vertex 0
```

```
mov (1) r3.1:d 2*32+16 {NoMask} // s2 for vertex 1
[[!f0.{sel|any4h|all4h}]] send (8) null [.mask] m0 r3 DATAPORT|RC|WRITE_SIMD2
```

Notice that with a null as the posted destination register, we are able to transfer the [.mask] over the message channel enables. In many cases for scratch memory access, a write-with-commit is required, therefore, the posted destination register could be r10.

Now, let's consider the PS case when a scratch register is the destination of an instruction.

```
p0      → f0-f4
src0     → r0-r7
src1     → r8-r15
dest     → s16-s23 / r16-r23
```

When predication is not on (or predication with swizzle control on), we have

```
add (16) m4:f [-][abs]r0/2/4/6_BasedOnSwizzle:f [-][abs]
r8/10/12/14_BasedOnSwizzle:f

add (16) m6:f [-][abs]r0/2/4/6_BasedOnSwizzle:f [-][abs]
r8/10/12/14_BasedOnSwizzle:f

add (16) m8:f [-][abs]r0/2/4/6_BasedOnSwizzle:f [-][abs]
r8/10/12/14_BasedOnSwizzle:f

add (16) m10:f [-][abs]r0/2/4/6_BasedOnSwizzle:f [-][abs]
r8/10/12/14_BasedOnSwizzle:f

mov (8) r3:d 0x76543210:v {NoMask} // ramp function

mul (16) acc0:d r3:d 16 {NoMask} // ramp function

add (8) acc0:d acc0:d 64 {NoMask,SecHalf} // ramp function

add (16) m2:d acc0:d 2*256 {NoMask} // ramp function
send (16) null m1 r3 DATAPORT|RC|WRITE_SIMD16
```

As there is no bit left from the unit specified descriptor field, the 4-bit mask must be put into the header field in m1, which requires at least two more instructions.

Alternatively, or for the case that predication without modifier is on, we can do a read-modify-write.

```
mov (8) r3:d 0x76543210:v {NoMask} // ramp function

mul (16) acc0:d r3:d 16 {NoMask} // ramp function
```



```
add (8) acc0:d acc0:d 64 {NoMask,SecHalf} // ramp function

add (16) m2:d acc0:d 2*256 {NoMask} // ramp function
send (16) r16 m1 r3 DATAPORT|RC|READ_SIMD16 // read from scratch

// some of the following four instructions may be omitted based on [.mask] field

[[(!]f0.{sel|any4v|all4v})] add (16) r16:f [-][!(abs)]r0/2/4/6_BasedOnSwizzle:f [-][!(abs)] r8/10/12/14_BasedOnSwizzle:f

[[(!]f0.{sel|any4v|all4v})] add (16) r18:f [-][!(abs)]r0/2/4/6_BasedOnSwizzle:f [-][!(abs)] r8/10/12/14_BasedOnSwizzle:f

[[(!]f0.{sel|any4v|all4v})] add (16) r20:f [-][!(abs)]r0/2/4/6_BasedOnSwizzle:f [-][!(abs)] r8/10/12/14_BasedOnSwizzle:f

[[(!]f0.{sel|any4v|all4v})] add (16) r22:f [-][!(abs)]r0/2/4/6_BasedOnSwizzle:f [-][!(abs)] r8/10/12/14_BasedOnSwizzle:f

mov (16) m4:f r16:f {NoMask}

mov (16) m6:f r18:f {NoMask}

mov (16) m8:f r20:f {NoMask}

mov (16) m10:f r22:f {NoMask}
send (16) null m1 null DATAPORT|RC|WRITE_SIMD16 {NoMask} // write back to scratch
```



15.2.6 Flow Control Instructions

Unconditional branches are performed through direct manipulation of the 32-bit IP architectural register. For example:

```
mov (1) IP <memory_address>      // jump absolute
add (1) IP IP <byte_count>        // jump relative
```

Note that jump distances are specified in terms of bytes, as opposed to instruction counts in the case of *break*, *halt*, etc. To minimize confusion, an assembler-only instruction `'jmp <inst_count>'`, where `<inst_count>` is an immediate term, may be defined which takes an instruction count for a distance. The *jmp* pseudo-opcode can be mapped to an `"add (1) ip ip <inst_count> * 16"` instruction.

Also note that IP is always an instruction-sized aligned address (16 bytes), thus the 4 LSB's are not maintained in the IP architectural register and should not be relied upon by software.

IP, when used as a source operand, reflects the memory address of the instruction in which it is used. The following are examples illustrating the use of IP:

```
add (1) IP 4*16      // jumps to HERE_1
add (1) IP 0x35      // jumps to HERE_1 (4 lsb's
don't-care)
<instruction>
<instruction>
HERE_1: <instruction>
HERE_2: <instruction>
<instruction>
add (1) IP -2*16     // jumps to HERE_2
...
add (1) IP 0         // infinite loop
add (1) IP 0xF       // infinite loop
...
```

Note for Assembler: The *if/iff/else/while/break* instructions identify relative addresses as the targets of an implicit jump associated with the instruction. These are optional in the assembly syntax as the jitter can determine the location of the matching instruction (e.g. matching *endif* instruction for a given *if* instruction).



15.2.7 Execution Masking

15.2.7.1 Branching

Example 15-3. If / Else / EndIf

```
//-----
// Example if/else/endif scenario
//  "if (r5==r4) ...else ... end-if"
//-----
...
cmp.e.f0 (8) null r5 r4 // does r5 == r4?
(f0) if (8)  HERE_1      // "if" part - save then update IMASK;
                        //  or goto the 'else' if all false
...
...
HERE_1:                // now do the 'else' part
else (8) HERE_2         // "else" part - invert IMASK
                        //  or goto the 'endif' if all false
...
...
HERE_2:
endif                  // "end-if" part - restore IMASK
...                   // and continue...
```

If it is known that the code has no nested conditionals, a predicate can be used for a lower overhead, more efficient if/else/endif. (One must consider the probability of all channels taking the same branch, and the number of instructions under the if/else blocks as to which conditional method, predicate or mask, is most efficient).

15.2.7.2 Fast-If

Below is an example of a fast-if instruction. For the 'iff' instruction, only and iff-endif construct is allowed, as opposed to a if-else-endif. Note that the target address for branching if all enabled channels fail is one instruction beyond the endif, as the 'iff' does not push and update the IMask unless the branch is taken for at least one execution channel.

Example 15-4. Fast If

```
//-----
// Example - Fast If
//  One instruction overhead conditional
//-----
...
cmp.e.f0 (8) null r5 r4 // any flag update
...
(f0) iff (8) HERE_1      // "fast-if" - only pushes IMask;
                        //  if execution falls through,
                        //  else go to HERE_1
...
...
endif                  // "end-if" part - restores IMask
HERE_1:                ...
                        // and continue...
```



15.2.7.3 Cascade Branching

As there is no 'elseif' instruction, a C-like cascade branching such as if / elseif / else / endif, can be realized using the basic building blocks of if / else / endif as shown in the following example. Notice that two 'endif's' are required in order to pop the IStack correctly.

Example 15-5 If / Elseif / Else / EndIf

```
//-----
// Example if/elseif/else/endif scenario
//  "if (r5==r4) ...elseif (r6>r7) else ... end-if"
//-----
...
    cmp.e.f0 (8) null r5 r4 // does r5 == r4?
(f0) if (8) HERE_1          // "if" part - save then update IMask;
                           // or go to the 'else' part if all false
...
    HERE_1:                // now do the 'else' part
        else (8) HERE_2    // "else if" part - invert IMask
                           // or go to the 'else' part if all false
        cmp.g.f0 (8) null r6 r7 // is r6 > r7?
(f0) if (8) HERE_3          // "if" part - save then update IMask;
                           // or go to the 'else' part if all false
...
    HERE_3:                // now do the 'else' part
        else (8) HERE_4    // "else" part - invert IMask
                           // or go to the 'end-if' part if all false
...
    HERE_4:                // "end-if" part - restore IMask for elseif
    endif
HERE_2:                    // "end-if" part - restore IMask for if
    endif
...

```

15.2.7.4 Compound Branches

Compound branches are supported through the ability logically combine flag registers for each intermediate result.

Example 15-6 Compound Branch

```
//-----
// Example: "if (r0 > r1) OR (r2 <= r3)"
//-----
...
    cmp.g.f0 (8) null r0:d r1:d // r0 > r1?
    cmp.le.f1 (8) null r2:d r3:d // r2 <= r3?
    or (1) f0:w f0:w f1:w // combine f0 and f1
(f0) if (8) HERE_1          // Can now do normal if/else
...
    HERE_1:                endif
...

```



Example 15-7. Compound Branch Using 'Any' or 'All'

```
//-----
// Example: assuming we're doing a channel-serial vector in r0-r3
// We want to know if all components of the vector are > 0x80
//-----
...
cmp.g.f0 (16) null r0 0x80      // r0 > 0x80?
cmp.g.f1 (16) null r1 0x80      // r1 > 0x80?
cmp.g.f2 (16) null r2 0x80      // r0 > 0x80?
cmp.g.f3 (16) null r3 0x80      // r1 > 0x80?
(f0.all4v) if (16) HERE_1
...
...      // code executed only for those channels
...      // where per-channel r0,r1,r2,r3 all > 0x80
...
HERE_1:   endif
...      // and continue...
```

15.2.7.5 Looping

Due to Gen4's SIMD-16 architecture, it must support the case of up to 16 loops running in parallel. These must be handled as independent loops, each with its own loop-exit condition which could occur after a different number of loop iterations. To account for each channel's progress, a 16b loop-mask 'LMask' is defined with 1b associated to each execution channel. This mask keeps track of which channels remain active inside a loop block.

Basic Do-While Loop

Example 15-8 illustrates the most basic loop. Two operations must be accomplished before loop entry. (1) Prior to loop entry, there is some subset of enabled channels as dictated by the code sequence prior. In general, the active status of each channel is indicated in the virtual EMask any point in time. These active channels will become the channels over which the loop is run, and LMask must be initialized with the EMask value. (2) Since a given loop may be nested within another loop, the previous LMask & CMask must be saved to the LStack for later restoration upon loop completion. The 'msave' instruction performs both the save and update in a single instruction, and thus all loop-blocks should be fronted with a "msave LStack LMask" and "msave LStack CMask" operation.

Note that the LMask and CMask share the same mask-stack. Thus, CMask must always be a 1's-subset of the LMask for proper stack operation. This is the case if CMask is updated to LMask each pass through the loop (see Example 15-8) and through the 'break' instruction updating both masks.

Each pass through the loop, a loop terminating operation must be evaluated and stored in a flag register. This condition must be evaluated on a channel-by-channel basis as exemplified:

```
cmp.z.f0    (8) null r2 d3      // any operation that updates a flag
```

The result of this operation sets a bit per channel in the specified flag register, which is then used in the 'while' instruction. As loops are performed, channels may become disabled as their termination condition is met.



'While' termination is determined on a channel-by-channel basis by the logical AND of corresponding bit positions of AMask, CMask and the specified flag. If the result is '1' the channel remains enabled for the next pass of the loop; if '0' the channel is disabled until loop fall-through. The 'while' instruction causes the LMask to be updated with the latest result of enabled channels. If any channel remains enabled (LMask != ...000b), an additional pass through the loop is made. Once a channel is terminated for the loop operation, it remains terminated until the loop is complete for all channels.

Upon fall through, the 'while' instruction causes the previously saved LMask & CMask to be popped from the LStack, enabling execution on the same subset of channels enabled prior to loop entry (unless a channel had been otherwise terminate inside the loop via 'halt').

Example 15-8. Basic Loop Construct

```
//-----
//  Example: Basic do-while loop structure
//-----
...
do                // save L/CMask & update
BEGIN_LOOP:
    mov (1) CMask LMask {NoMask} // update CMask for this pass
    ...
    ...
    <some flag update>
    (<p>) while (8) BEGIN_LOOP      // cond. branch
                                   //   + restores LMask on fall-thru
    ...
```

Do-While Loop with Break

A loop may also be terminated for any channel via the 'break' instruction. The 'break' instruction causes the corresponding bit positions of enabled channels to be cleared in the LMask. If the updated LMask = ...000b, a branch is made to the specified instruction location. An example is shown below in which the 'break' is at the same conditional-nesting level as the terminating 'while'. Its primary value may simply be to support a "do...break.. while (true)" -type structure for a more direct 1:1 translation from higher-level source code.

Example 15-9. Loop Construct With Non-Nested 'Break'

```
//-----
//  Example: While-true loop
//-----
#define BrkCode(i,d)  (i << 16) + d

do                // save L/CMask & update
BEGIN_LOOP:
    mov (1) CMask LMask {NoMask} // update CMask for this pass
    ...
    <some flag update>
    (<p>) break (8) BrkCode(0,HERE_1) // Restores LMask when all
                                   //   channels complete loop.
    ...
    ...
```



```

                                while (8) BEGIN_LOOP    // while true
HERE_1:
    ...

```

A break condition may occur from various levels of nested-ifs. This gives rise to the possibility that a the loop may terminate from within nested 'if's, and due to the jump inherent in the 'break' instruction, the associated 'endif's' are not encountered to clean-up the IStack as nesting levels are exited.

Example 15-10 Loop Construct With 'Break' From Within Nested If's

```

//-----
// Example: General Loop Structure w/ break inside if's
//-----
#define BrkCode(i,d)  (i << 16) + d

                                do
                                // save L/CMask & update
BEGIN_LOOP:
    mov (1) CMask LMask {NoMask} // update CMask for this pass
    ...
    if ...
    if ...
    if ...
    ...
    (<p>) break (8) BrkCode(3,HERE_1) // we're 3 levels deep, so
    ...
    endif
    endif
    endif
    ...
    (<p>) break (8) BrkCode(0,HERE_1)
    ...
    while (8) <flag_spec> BEGIN_LOOP // cond. branch
                                // + restores C/LMask on fall-thru
HERE_1:

```

Do-While Loop with Continue

A continue instruction 'cont' is provided skip to the next iteration of the loop. Because not all channels participating in the loop may be enabled at the time this instruction is executed, some channels may require continuation of the loop. A special mask 'CMask' is defined which accounts for channels temporarily disabled for the current loop pass.

Since loops may nested, the CMask must be saved and restored around a loop similar to LMask. Since the CMask value within a properly constructed loop is always a subset of the LMask, it can share the LStack for storage, so long as it is pushed after LMask as shown in Example 15-11. This save/restore operations are not required if the loop being entered does not have any occurrence of a continue instruction.



Example 15-11. Do-While with Continue

```
//-----  
// Example: General Loop Structure w/ basic break and cont.  
//-----  
#define ContCode(i,d) (i << 16) + d  
  
do                // save L/CMask & update  
BEGIN_LOOP:  
    mov (1) CMask EMask    // re-initialize CMask for this pass  
    ...  
    ...  
    (<p>) cont (8) ContCode(0,HERE_1)  
    ...  
HERE_1:  
    (<p>) while (8) BEGIN_LOOP    // cond. branch  
    // + restores C/LMask on fall-thru  
    ...
```

15.2.7.6 Indexed Jump

Example 15-12. Indexed Jump

```
//-----  
// Code example shows the use of jmpri to perform a case statement  
// of any number of options in 3 jumps  
//-----  
.default_execution_size    8  
...  
jmpri r0<0,1,0>           // jump relative, based on r0.a.x  
// ----- Jump Table -----  
jmp HERE_0                // redirect for case 0  
jmp HERE_1                // redirect for case 1  
jmp HERE_2                // redirect for case 2  
jmp HERE_3                // redirect for case 3  
...  
HERE_0:                   // ... case 0 ...  
    ...  
    jmp DONE  
HERE_1:                   // ... case 1 ...  
    ...  
    jmp DONE  
HERE_2:                   // ... case 2 ...  
    ...  
    jmp DONE  
HERE_3:                   // ... case 3 ...  
    ...  
DONE:                     // and continue...  
    ...
```