Intel® Iris® Plus Graphics and UHD Graphics Open Source

Programmer's Reference Manual

For the 2019 10th Generation Intel Core™ Processors based on the "Ice Lake" Platform

Volume 8: Command Stream Programming

January 2020, Revision 1.0

Creative Commons License

Notices and Disclaimers

Table of Contents

Command Stream Programming

Introduction

Command Streamer is the primary interface to the various engines that are part of the graphics hardware.

The graphics HW consists of multiple parallel engines that can execute different kinds of workloads. E.g Render engine for 3D and GPGPU tasks, Video Decode engine, Video Enhancement Engine and Blitter engine.

Some product SKU's have multiple instances of an engine (e.g 2 Video Decode engines).

As shown in figure 1, each of these engines have their own Command Streamer that is responsible for processing the commands in the workload and enabling execution of the task.

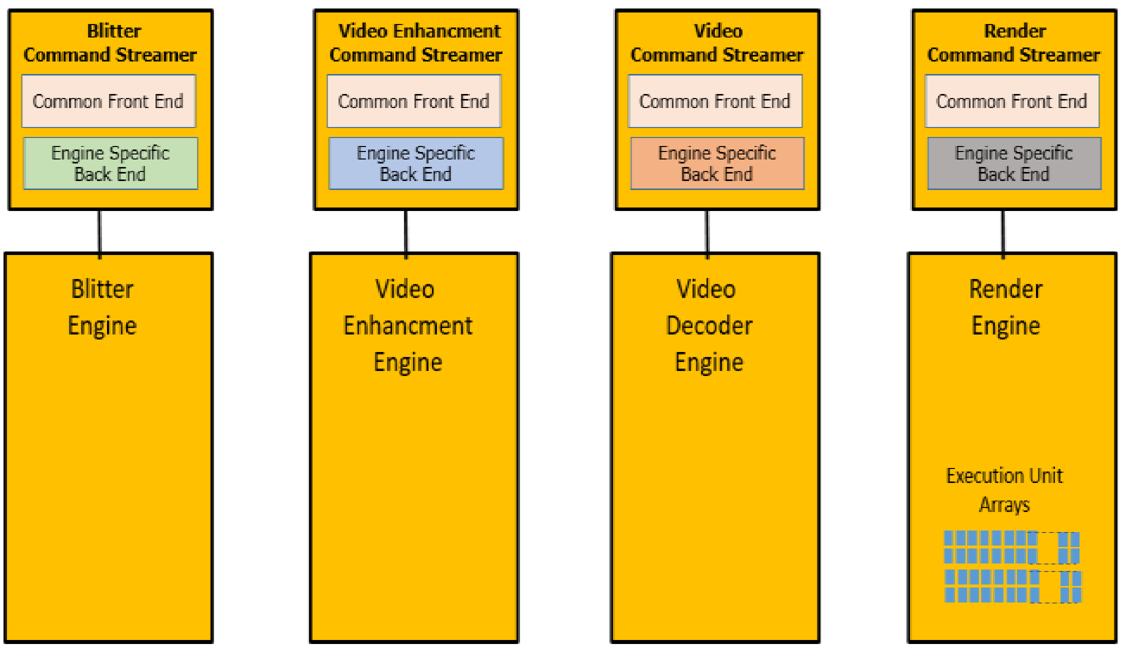


Figure 1: High level view of Command Streamer

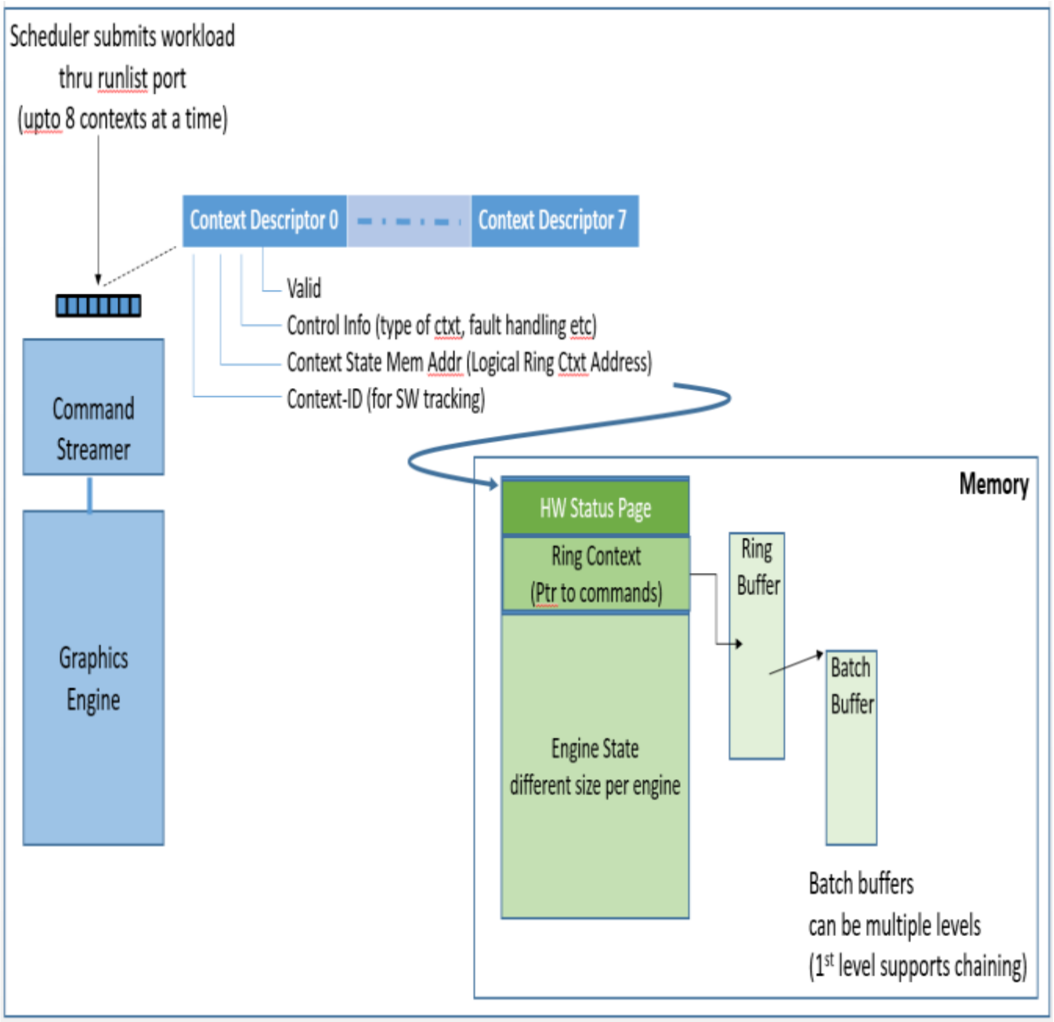
As shown in the figure, the command streamer is comprised of a Common Front end and an engine specific backend.

The common front end allows each engine to provide a uniform software interface (e.g infrastructure for submission of commands, sychronization, etc).

The back ends handle the engine specific commands and the protocols required to control the execution of the underlying engine.

Commands and Programming Interface

Each command streamer provides an 8 element runlist port to allow the scheduler to submit up to 8 contexts for execution.



Contents of an element are described in the Context Descriptor structure.

Context descriptor includes control information (required for the context execution and SW tracking) and a pointer to the Context State Memory Address (aka LRCA).

LRCA contains:

- State information required for context execution (Pointer to command buffers, pointers to per process page tables)
- Memory for saving engine execution state of a context

Note that commands are fed through a hierarchy of command buffers - starting with the ring buffer at the highest level and tiered batch buffers.

Command Buffers

Instructions to be executed by an engine are submitted to the hardware using command buffers.

Command Ring Buffers

Command 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.

Programming Note	
Context:	Command Ring Buffers in memory areas.
"Indirect" 3D primitive instructions (those that access vertex buffers) must reside in the same memory space as the vertex buffers.	

Command Batch Buffers

Command 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.

Programming Note	
Context:	Command batch buffers in memory objects
Batch buffers can be tagged with any memory type when produced by IA. If WB memory type is used, it should be tagged with "snoop required" for GPU consumption (to trigger snoop from CPU cache).	

Programming Note	
Context:	Command batch buffers in memory objects
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).	

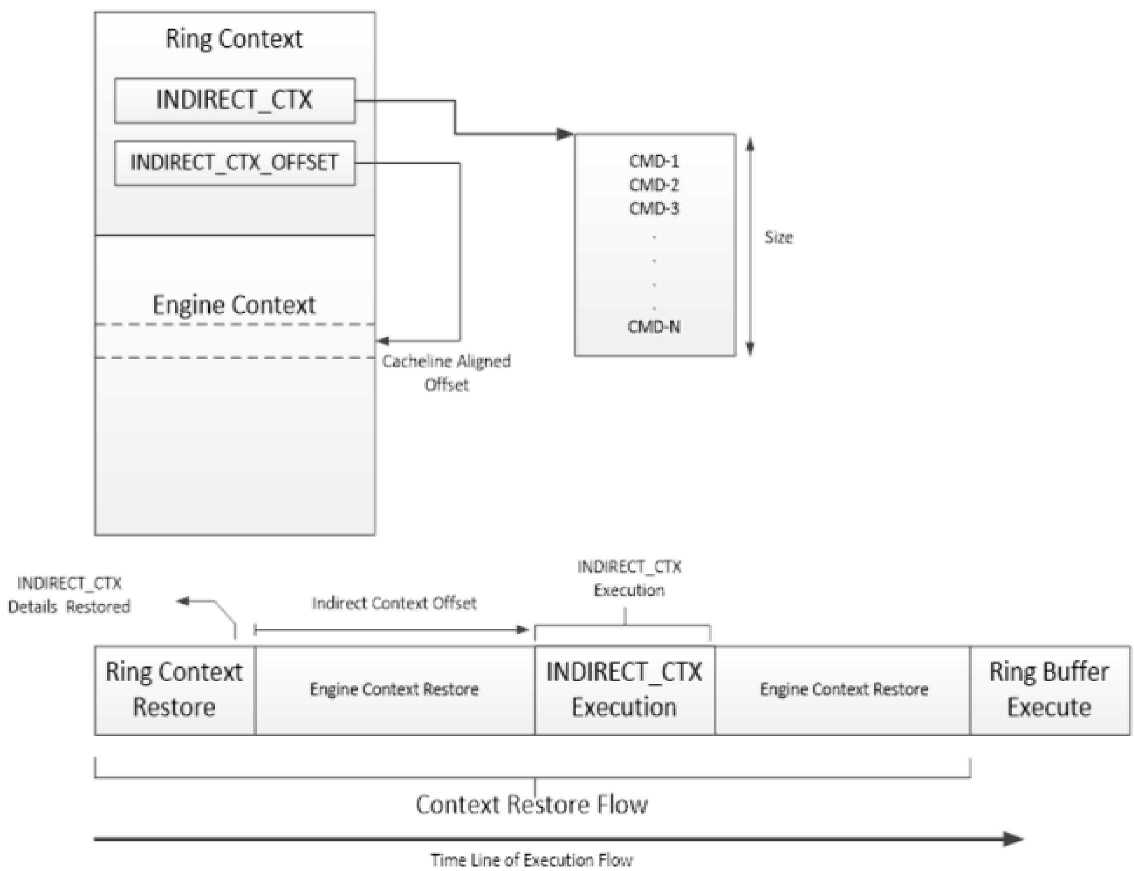
Workaround Batch Buffers

A Workaround batch buffer is a set of commands that is run by the hardware during context load time. i.e when Command Streamer hardware is restoring the state of the context that it is about to execute (before execution of any command in the ring buffer). The Workaround batch buffer uses pointers to command buffers that are setup by the Kernel Mode driver in the context image.

Two flavors of Workaround batch buffers are supported by the hardware. They differ in terms of exactly when the supplied workaround commands are executed in the context restore process. The mechanisms supported are:

Indirect Context Pointer (INDIRECT_CTX)

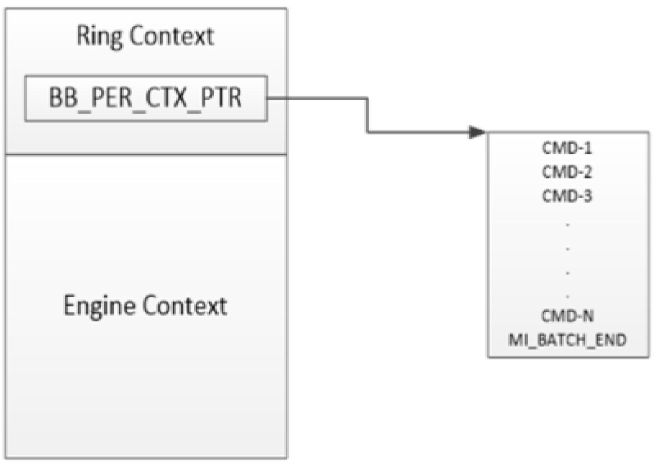
As shown in the figure below, this workaround buffer can be invoked at any cacheline aligned offset in the engine context.

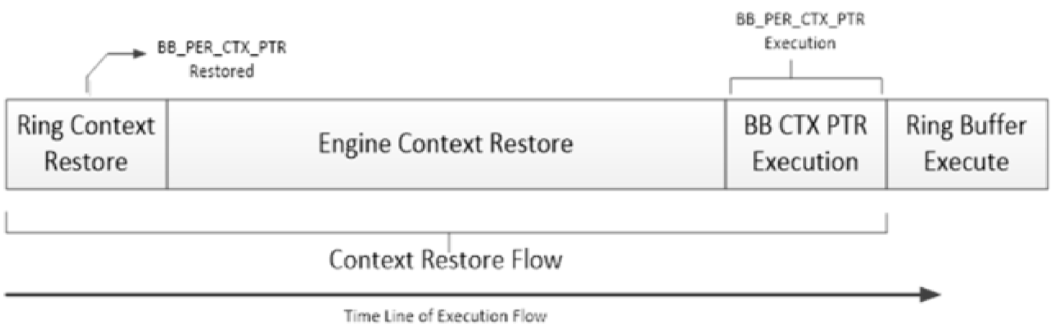


Command streamer, when enabled through "INDIRECT_CTX" provides a mechanism to pause executing context restore on a given cacheline aligned offset in the engine context image and execute a command sequence from a command buffer before resuming context restore flow. This command buffer execution during context restore is referred to as "Indirect Context Pointer" execution. The start address and the size of the command buffer to be executed is provided through "INDIRECT_CTX" register and the offset in the engine context restore is provided through "INDIRECT_CTX_OFFSET". "INDIRECT_CTX" and "INDIRECT_CTX_OFFSET" registers are part of the context image and gets restored as part of the given context's context restore flow, these registers are part of the ring context image which are prior to engine context restore and hence the requirement of the offset being in engine context restore. "Indirect Context Pointer" is always in the GGTT address space of the virtual function or physical function from which the context is submitted. "Indirect context pointer" can be programmed differently for each context providing flexibility to execute different command sequence as part of "Indirect Context Pointer" execution during context restore flows.

Post Context Restore Workaround Batch Buffer

As shown in the figure, this workaround buffer is invoked at the end of the context restore.





Command streamer, when enabled through "BB_PER_CTX_PTR" provides a mechanism to execute a command sequence from a batch buffer at the end of the context restore flow during context switch process. This batch buffer is referred to as "Context Restore Batch Buffer". The batch start address for the "Context Restore Batch Buffer" gets programmed through "BB_PER_CTX_PTR", which is part of the context image and gets restored as part of the given context's context restore flow. "Context Restore Batch Buffer" execution begins (like a regular batch buffer) after the completion of fetching and execution of all the commands for the context restore flow. "Context Restore Batch Buffer" execution ends on executing MI_BATCH_BUFFER_END in the command sequence. "Context Restore Batch Buffer" is always in the GGTT address space of the virtual function or physical function from which the context is submitted. "BB_PER_CTX_PTR" can be programmed differently for every context giving flexibility to execute different command sequence (batch buffers) as part of "Context Restore Batch Buffer" execution or can be programmed to disable execution of the "Context Restore Batch Buffer" for a given context.

This mechanism is especially helpful in programming a set of commands/state that has to be always executed prior to executing a workload from a context every time it is submitted to HW for execution. Limited capability is built for "Context Restore Batch Buffer" unlike a regular MI_BATCH_BUFFER_START due to envisioned usage model, refer BB_PER_CTX_PTR for detailed programming notes.

Graphics 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 processes 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.

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.

Command Header

Render Command Header Format

Type	Bits				
	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	

Type	Bits				
	31:29	28:24	23:19	18:16	15:0
Reserved	001, 010	Opcode – 11111	Sub Opcode 00h – 01h	Reserved	DWord Count

Type	Bits					
	31:29	28:27	26:24	23:16	15:8	7:0
Common	011	00	Opcode – 000	Sub Opcode	Data	DWord Count
Common (NP)[1]	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 (Pipelined)	011	11	Opcode – 000	Sub Opcode	Data	DWord Count
3DState (NP)[1]	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			
Reserved	011	11	Opcode - 101			
Reserved	011	11	Opcode – 110 – 111			
Reserved	100	XX				
Reserved	101	XX				
Reserved	110	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).

Video Command Header Format

Type	Bits				
	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	Bits				
	31:29	28:27	26:24	23:16	15:0
Reserved	011	00	XXX	XX	
MFX Single DW	011	01	000	Opcode: 0h	0
Reserved	011	01	1XX		
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	11X		
Reserved	011	11	XXX		

Type	Bits					
	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
MPEG2Dec	011	10	011	001	subopcode	DWord Count
Reserved (for MPEG2 Enc)	011	10	011	010	subopcode	DWord Count

Type	Bits					
Reserved	011	10	011	011-111	subopcode	DWord Count
Reserved	011	10	100-111	XXX		

Video Enhancement Command Header Format

Type	Bits				
	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, 010				

Type	Bits						
	31:29	28:27	26:24	23:21	20:16	15:12	11:0
VEBOX (Parallel Video Pipe)	011	10: Pipeline 00: Reserved 01: Reserved 11: Reserved	Command Opcode – 100	Sub Opcode A	Sub Opcode B	Reserved	Dword Count

Blitter Command Header Format

Type	Bits				
	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, 011				

Type	Bits			
	31:29	28:22	21:9	8:0
Blitter (2D)	010	Command Opcode	Command Dependent Data	Dword Count

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

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

Memory Interface Commands for RCP

Opcode (28:23)	Command	Pipes
1 DWord		
00h	MI_NOOP	All
01h	MI_SET_PREDICATE	Render
02h	MI_USER_INTERRUPT	All
03h	MI_WAIT_FOR_EVENT	All
05h	MI_ARB_CHECK	All
07h	MI_REPORT_HEAD	All
08h	MI_ARB_ON_OFF	All except Blitter
0Ah	MI_BATCH_BUFFER_END	All
0Bh	MI_SUSPEND_FLUSH	All
0Ch	MI_PREDICATE	Render
2+ DWord		
10h	Reserved	
12h	MI_LOAD_SCAN_LINES_INCL	Render and Blitter
13h	MI_LOAD_SCAN_LINES_EXCL	Render and Blitter
14h	MI_DISPLAY_FLIP	Render and Blitter
15h	Reserved	
17h	Reserved	
18h	MI_SET_CONTEXT	Render
1Ah	MI_MATH	All
1Bh	MI_SEMAPHORE_SIGNAL	All
1Ch	MI_SEMAPHORE_WAIT	All
1Dh	MI_FORCE_WAKEUP	All except Render

Opcode (28:23)	Command	Pipes
1Fh	Reserved	
Store Data		
20h	MI_STORE_DATA_IMM	All
21h	MI_STORE_DATA_INDEX	All
22h	MI_LOAD_REGISTER_IMM	All
23h	MI_UPDATE_GTT	All
24h	MI_STORE_REGISTER_MEM	All
26h	MI_FLUSH_DW	All except Render
27h	MI_CLFLUSH	Render
29h	MI_LOAD_REGISTER_MEM	All
2Ah	MI_LOAD_REGISTER_REG	All
2Bh	MI_RS_STORE_DATA_IMM	Render
2Eh	MI_MEM_TO_MEM	All
2Fh	MI_ATOMIC	All
Ring/Batch Buffer		
30h	Reserved	
31h	MI_BATCH_BUFFER_START	Render
32h-35h	Reserved	
36h	MI_CONDITIONAL_BATCH_BUFFER_END	All
37h-38h	Reserved	
39h	Reserved	All
39h-3Fh	Reserved	

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.

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

2D Command Map

Opcode (28:22)	Command
00h	Reserved
01h	XY_SETUP_BLT
02h	Reserved

Opcode (28:22)	Command
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	ReservedReserved
31h	XY_TEXT_IMMEDIATE_BLT
32h-3Fh	Reserved
40h	COLOR_BLT
42h	XY_FAST_COPY_BLT
43h	SRC_COPY_BLT
45h-47h	Reserved
49h-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_BL
76h	XY_PAT_CHROMA_BLT
77h	XY_PAT_CHROMA_BLT_IMMEDIATE
78h-7Fh	Reserved

3D Commands

The 3D commands are used to program the graphics pipelines for 3D 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.

For all commands listed in **3D Command Map**, the Pipeline Type (bits 28:27) is 3h, indicating the 3D Pipeline.

3D Command Map

Opcode Bits 26:24	Sub Opcode Bits 23:16	Command	Definition Chapter
0h	01h	Reserved	3D Pipeline
0h	02h	Reserved	3D Pipeline
0h	03h	Reserved	
0h	04h	3DSTATE_CLEAR_PARAMS	3D Pipeline
0h	05h	3DSTATE_DEPTH_BUFFER	3D Pipeline
0h	06h	3DSTATE_STENCIL_BUFFER	3D Pipeline
0h	07h	3DSTATE_HIER_DEPTH_BUFFER	3D Pipeline
0h	08h	3DSTATE_VERTEX_BUFFERS	Vertex Fetch
0h	09h	3DSTATE_VERTEX_ELEMENTS	Vertex Fetch
0h	0Ah	3DSTATE_INDEX_BUFFER	Vertex Fetch
0h	0Bh	3DSTATE_VF_STATISTICS	Vertex Fetch
0h	0Ch	3DSTATE_VF	Vertex Fetch
0h	0Dh	3DSTATE_VIEWPORT_STATE_POINTERS	3D Pipeline
0h	0Eh	3DSTATE_CC_STATE_POINTERS	3D Pipeline
0h	10h	3DSTATE_VS	Vertex Shader
0h	11h	3DSTATE_GS	Geometry Shader
0h	12h	3DSTATE_CLIP	Clipper
0h	13h	3DSTATE_SF	Strips & Fans
0h	14h	3DSTATE_WM	Windower
0h	15h	3DSTATE_CONSTANT_VS	Vertex Shader
0h	16h	3DSTATE_CONSTANT_GS	Geometry Shader
0h	17h	3DSTATE_CONSTANT_PS	Windower
0h	18h	3DSTATE_SAMPLE_MASK	Windower
0h	19h	3DSTATE_CONSTANT_HS	Hull Shader
0h	1Ah	3DSTATE_CONSTANT_DS	Domain Shader
0h	1Bh	3DSTATE_HS	Hull Shader
0h	1Ch	3DSTATE_TE	Tesselator

Opcode Bits 26:24	Sub Opcode Bits 23:16	Command	Definition Chapter
0h	1Dh	3DSTATE_DS	Domain Shader
0h	1Eh	3DSTATE_STREAMOUT	HW Streamout
0h	1Fh	3DSTATE_SBE	Setup
0h	20h	3DSTATE_PS	Pixel Shader
0h	21h	3DSTATE_VIEWPORT_STATE_POINTERS_SF_CLIP	Strips & Fans
0h	22h	3DSTATE_CPS	Course Pixel Shader
0h	23h	3DSTATE_VIEWPORT_STATE_POINTERS_CC	Windower
0h	24h	3DSTATE_BLEND_STATE_POINTERS	Pixel Shader
0h	25h	3DSTATE_DEPTH_STENCIL_STATE_POINTERS	Pixel Shader
0h	26h	3DSTATE_BINDING_TABLE_POINTERS_VS	Vertex Shader
0h	27h	3DSTATE_BINDING_TABLE_POINTERS_HS	Hull Shader
0h	28h	3DSTATE_BINDING_TABLE_POINTERS_DS	Domain Shader
0h	29h	3DSTATE_BINDING_TABLE_POINTERS_GS	Geometry Shader
0h	2Ah	3DSTATE_BINDING_TABLE_POINTERS_PS	Pixel Shader
0h	2Bh	3DSTATE_SAMPLER_STATE_POINTERS_VS	Vertex Shader
0h	2Ch	3DSTATE_SAMPLER_STATE_POINTERS_HS	Hull Shader
0h	2Dh	3DSTATE_SAMPLER_STATE_POINTERS_DS	Domain Shader
0h	2Eh	3DSTATE_SAMPLER_STATE_POINTERS_GS	Geometry Shader
0h	2Fh	3DSTATE_SAMPLER_STATE_POINTERS_PS	Pixel Shader
0h	30h	3DSTATE_URB_VS	Vertex Shader
0h	31h	3DSTATE_URB_HS	Hull Shader
0h	32h	3DSTATE_URB_DS	Domain Shader
0h	33h	3DSTATE_URB_GS	Geometry Shader
0h	34h	3DSTATE_GATHER_CONSTANT_VS	Vertex Shader
0h	35h	3DSTATE_GATHER_CONSTANT_GS	Geometry Shader
0h	36h	3DSTATE_GATHER_CONSTANT_HS	Hull Shader
0h	37h	3DSTATE_GATHER_CONSTANT_DS	Domain Shader
0h	38h	3DSTATE_GATHER_CONSTANT_PS	Pixel Shader
0h	39h	3DSTATE_DX9_CONSTANTF_VS	Vertex Shader
0h	3Ah	3DSTATE_DX9_CONSTANTF_PS	Pixel Shader
0h	3Bh	3DSTATE_DX9_CONSTANTI_VS	Vertex Shader
0h	3Ch	3DSTATE_DX9_CONSTANTI_PS	Pixel Shader
0h	3Dh	3DSTATE_DX9_CONSTANTB_VS	Vertex Shader
0h	3Eh	3DSTATE_DX9_CONSTANTB_PS	Pixel Shader
0h	3Fh	3DSTATE_DX9_LOCAL_VALID_VS	Vertex Shader

Opcode Bits 26:24	Sub Opcode Bits 23:16	Command	Definition Chapter
0h	40h	3DSTATE_DX9_LOCAL_VALID_PS	Pixel Shader
0h	41h	3DSTATE_DX9_GENERATE_ACTIVE_VS	Vertex Shader
0h	42h	3DSTATE_DX9_GENERATE_ACTIVE_PS	Pixel Shader
0h	43h	3DSTATE_BINDING_TABLE_EDIT_VS	Vertex Shader
0h	44h	3DSTATE_BINDING_TABLE_EDIT_GS	Geometry Shader
0h	45h	3DSTATE_BINDING_TABLE_EDIT_HS	Hull Shader
0h	46h	3DSTATE_BINDING_TABLE_EDIT_DS	Domain Shader
0h	47h	3DSTATE_BINDING_TABLE_EDIT_PS	Pixel Shader
0h	48h	3DSTATE_VF_HASHING	Vertex Fetch
0h	49h	3DSTATE_VF_INSTANCING	Vertex Fetch
0h	4Ah	3DSTATE_VF_SGVS	Vertex Fetch
0h	4Bh	3DSTATE_VF_TOPOLOGY	Vertex Fetch
0h	4Ch	3DSTATE_WM_CHROMA_KEY	Windower
0h	4Dh	3DSTATE_PS_BLEND	Windower
0h	4Eh	3DSTATE_WM_DEPTH_STENCIL	Windower
0h	4Fh	3DSTATE_PS_EXTRA	Windower
0h	50h	3DSTATE_RASTER	Strips & Fans
0h	51h	3DSTATE_SBE_SWIZ	Strips & Fans
0h	52h	3DSTATE_WM_HZ_OP	Windower
0h	53h	3DSTATE_INT (internally generated state)	3D Pipeline
0h	54h	3DSTATE_RS_CONSTANT_POINTER	Resource Streamer
0h	55h	3DSTATE_VF_COMPONENT_PACKING	Vertex Fetch
0h	56h	3DSTATE_VF_SGVS_2	VertexFetch
0h	58h	Reserved	
0h	59h	Reserved	
0h	5Ah	Reserved	
0h	5Bh	Reserved	
0h	5Ch	Reserved	
0h	5Dh-5Fh	Reserved	
0h	60h-63h	Reserved	
0h	64h-69h	Reserved	
0h	6Ah	3DSTATE_PTBR_MARKER	3D Pipeline
0h	6Bh	3DSTATE_PTBR_TILE_SELECT	Vertex Fetch, Strips & Fans
0h	57h-59h	Reserved	
0h	60h-68h	Reserved	

Opcode Bits 26:24	Sub Opcode Bits 23:16	Command	Definition Chapter
0h	69h	Reserved	
0h	6Ch	Reserved	
0h	6Dh	Reserved	
0h	6Eh	Reserved	
0h	6Fh	Reserved	
0h	70h	Reserved	
0h	71h	Reserved	
0h	72h	Reserved	
0h	73h	Reserved	
0h	74h	Reserved	
0h	72h-73h	Reserved	
0h	75h	Reserved	
0h	76h	Reserved	
0h	77h-82h	Reserved	
0h	83h-FFh	Reserved	
1h	00h	3DSTATE_DRAWING_RECTANGLE	Strips & Fans
1h	02h	3DSTATE_SAMPLER_PALETTE_LOAD0	Sampling Engine
1h	03h	Reserved	
1h	04h	3DSTATE_CHROMA_KEY	Sampling Engine
1h	05h	Reserved	
1h	06h	3DSTATE_POLY_STIPPLE_OFFSET	Windower
1h	07h	3DSTATE_POLY_STIPPLE_PATTERN	Windower
1h	08h	3DSTATE_LINE_STIPPLE	Windower
1h	0Ah	3DSTATE_AA_LINE_PARAMS	Windower
1h	0Bh	3DSTATE_GS_SVB_INDEX	Geometry Shader
1h	0Ch	3DSTATE_SAMPLER_PALETTE_LOAD1	Sampling Engine
1h	0Dh	3DSTATE_MULTISAMPLE	Windower
1h	0Eh	3DSTATE_STENCIL_BUFFER	Windower
1h	0Fh	3DSTATE_HIER_DEPTH_BUFFER	Windower
1h	10h	3DSTATE_CLEAR_PARAMS	Windower
1h	11h	3DSTATE_MONOFILTER_SIZE	Sampling Engine
1h	12h	3DSTATE_PUSH_CONSTANT_ALLOC_VS	Vertex Shader
1h	13h	3DSTATE_PUSH_CONSTANT_ALLOC_HS	Hull Shader
1h	14h	3DSTATE_PUSH_CONSTANT_ALLOC_DS	Domain Shader
1h	15h	3DSTATE_PUSH_CONSTANT_ALLOC_GS	Geometry Shader

Opcode Bits 26:24	Sub Opcode Bits 23:16	Command	Definition Chapter
1h	16h	3DSTATE_PUSH_CONSTANT_ALLOC_PS	Pixel Shader
1h	17h	3DSTATE_SO_DECL_LIST	HW Streamout
1h	18h	3DSTATE_SO_BUFFER	HW Streamout
1h	19h	3DSTATE_BINDING_TABLE_POOL_ALLOC	Resource Streamer
1h	1Ah	3DSTATE_GATHER_POOL_ALLOC	Resource Streamer
1h	1Bh	3DSTATE_DX9_CONSTANT_BUFFER_POOL_ALLOC	Resource Streamer
1h	1Ch	3DSTATE_SAMPLE_PATTERN	Windower
1h	1Dh	3DSTATE_URB_CLEAR	3D Pipeline
1h	1Eh	3DSTATE_3D_MODE	3D Pipeline
1h	1Fh	3DSTATE_SUBSLICE_HASH_TABLE	3D Pipeline
1h	20h	3DSTATE_SLICE_TABLE_STATE_POINTERS	3D Pipeline
1h	21h	3DSTATE_PTBR_PAGE_POOL_BASE_ADDRESS	3D Pipeline
1h	22h	3DSTATE_PTBR_TILE_PASS_INFO	3D Pipeline
1h	23h	3DSTATE_PTBR_RENDER_LIST_BASE_ADDRESS	3D Pipeline
1h	24h	3DSTATE_PTBR_FREE_LIST_BASE_ADDRES	3D Pipeline
1h	25h-FFh	Reserved	
2h	00h	PIPE_CONTROL	3D Pipeline
2h	01h-FFh	Reserved	
3h	00h	3DPRIMITIVE	Vertex Fetch
3h	01h-FFh	Reserved	
4h-7h	00h-FFh	Reserved	

Pipeline Type (28:27)	Opcode	Sub Opcode	Command	Definition Chapter
Common (pipelined)	Bits 26:24	Bits 23:16		
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	Reserved	3D Pipeline
0h	1h	04h	GPGPU CSR BASE ADDRESS	Graphics Processing Engine
0h	1h	05h	Reserved	
0h	1h	06h	Reserved	
0h	1h	07h	Reserved	

Pipeline Type (28:27)	Opcode	Sub Opcode	Command	Definition Chapter
0h	1h	08h-FFh	Reserved	N/A
Reserved	Bits 26:24	Bits 23:16		
0h	2h-7h	XX	Reserved	N/A

VEBOX Commands

The VEBOX commands are used to program the Video Enhancement engine attached to the Video Enhancement Command Parser.

VEBOX Command Map

Pipeline Type (28:27)	Opcode (26:24)	SubopA (23:21)	SubopB (20:16)	Command
2h	4h	0h	0h	VEBOX_SURFACE_STATE
2h	4h	0h	2h	VEBOX_STATE
2h	4h	0h	3h	VEBOX_DI_IECP
2h	4h	0h	1h	VEBOX_TILING_CONVERT

MFX Commands

The MFX (MFD for decode and MFC for encode) commands are used to program the multi-format codec engine attached to the Video Codec Command Parser. See the *MFD* and *MFC* chapters for a description of these commands.

MFX state commands support direct state model and indirect state model. Recommended usage of indirect state model is provided here (as a software usage guideline).

Pipeline Type (28:27)	Opcode (26:24)	Subop A (23:21)	Subop B (20:16)	Command	Chapter	Recommended Indirect State Pointer Map	Interruptable?
				MFX Common (State)			
2h	0h	0h	0h	MFX_PIPE_MODE_SELECT	MFX	IMAGE	N/A
2h	0h	0h	1h	MFX_SURFACE_STATE	MFX	IMAGE	N/A
2h	0h	0h	2h	MFX_PIPE_BUF_ADDR_STATE	MFX	IMAGE	N/A
2h	0h	0h	3h	MFX_IND_OBJ_BASE_ADDR_STATE	MFX	IMAGE	N/A
2h	0h	0h	4h	MFX_BSP_BUF_BASE_ADDR_STATE	MFX	IMAGE	N/A
2h	0h	0h	6h	MFX_ STATE_POINTER	MFX	IMAGE	N/A
2h	0h	0h	7-8h	Reserved	N/A	N/A	N/A
				MFX Common (Object)			
2h	0h	1h	9h	MFD_ IT_OBJECT	MFX	N/A	Yes

Pipeline Type (28:27)	Opcode (26:24)	Subop A (23:21)	Subop B (20:16)	Command	Chapter	Recommended Indirect State Pointer Map	Interruptable?
2h	0h	0h	4-1Fh	Reserved	N/A	N/A	N/A
AVC Common (State)							
2h	1h	0h	0h	MFX_AVC_IMG_STATE	MFX	IMAGE	N/A
2h	1h	0h	1h	MFX_AVC_QM_STATE	MFX	IMAGE	N/A
2h	1h	0h	2h	MFX_AVC_DIRECTMODE_STATE	MFX	SLICE	N/A
2h	1h	0h	3h	MFX_AVC_SLICE_STATE	MFX	SLICE	N/A
2h	1h	0h	4h	MFX_AVC_REF_IDX_STATE	MFX	SLICE	N/A
2h	1h	0h	5h	MFX_AVC_WEIGHTOFFSET_STATE	MFX	SLICE	N/A
2h	1h	0h	6-1Fh	Reserved	N/A	N/A	N/A
AVC Dec							
2h	1h	1h	0-7h	Reserved	N/A	N/A	N/A
2h	1h	1h	8h	MFD_AVC_BSD_OBJECT	MFX	N/A	No
2h	1h	1h	9-1Fh	Reserved	N/A	N/A	N/A
AVC Enc							
2h	1h	2h	0-1h	Reserved	N/A	N/A	N/A
2h	1h	2h	2h	MFC_AVC_FQM_STATE	MFX	IMAGE	N/A
2h	1h	2h	3-7h	Reserved	N/A	N/A	N/A
2h	1h	2h	8h	MFC_AVC_PAK_INSERT_OBJECT	MFX	N/A	N/A
2h	1h	2h	9h	MFC_AVC_PAK_OBJECT	MFX	N/A	Yes
2h	1h	2h	A-1Fh	Reserved	N/A	N/A	N/A
2h	1h	2h	0-1Fh	Reserved	N/A	N/A	N/A
VC1 Common							
2h	2h	0h	0h	MFX_VC1_PIC_STATE	MFX	IMAGE	N/A
2h	2h	0h	1h	MFX_VC1_PRED_PIPE_STATE	MFX	IMAGE	N/A
2h	2h	0h	2h	MFX_VC1_DIRECTMODE_STATE	MFX	SLICE	N/A
2h	2h	0h	2-1Fh	Reserved	N/A	N/A	N/A
VC1 Dec							
2h	2h	1h	0-7h	Reserved	N/A	N/A	N/A
2h	2h	1h	8h	MFD_VC1_BSD_OBJECT	MFX	N/A	Yes
2h	2h	1h	9-1Fh	Reserved	N/A	N/A	N/A
VC1 Enc							
2h	2h	2h	0-1Fh	Reserved	N/A	N/A	N/A
MPEG2 Common							
2h	3h	0h	0h	MFX_MPEG2_PIC_STATE	MFX	IMAGE	N/A

Pipeline Type (28:27)	Opcode (26:24)	Subop A (23:21)	Subop B (20:16)	Command	Chapter	Recommended Indirect State Pointer Map	Interruptable?
2h	3h	0h	1h	MFX_MPEG2_QM_STATE	MFX	IMAGE	N/A
2h	3h	0h	2-1Fh	Reserved	N/A	N/A	N/A
MPEG2 Dec							
2h	3h	1h	1-7h	Reserved	N/A	N/A	N/A
2h	3h	1h	8h	MFD_MPEG2_BSD_OBJECT	MFX	N/A	Yes
2h	3h	1h	9-1Fh	Reserved	N/A	N/A	N/A
MPEG2 Enc							
2h	3h	2h	0-1Fh	Reserved	N/A	N/A	N/A
The Rest							
2h	4-5h, 7h	x	x	Reserved	N/A	N/A	N/A

Execution Control Infrastructure

This section describes the hardware infrastructure that can be used to control command execution.

Watchdog Timers

Watchdog Counter Control

The Watchdog Counter Control determines if the watchdog is enabled, disabled and count mode. The watchdog is enabled is when the value of the register [30:0] is equal to zero([30:0] = ’d0). If enabled, then the Watchdog Counter is allowed to increment. The watchdog is disabled is when the value of the register [30:0] is equal to one where only bit zero is a value of ‘1’([30:0] = 0x00000001). If disabled, then the value of Watchdog Counter is reset to a value of zero. Bit 31, specifies the counting mode. If bit 31 is zero, then we will count based timestamp toggle(refer to Reported Timestamp Count register for toggle time). If bit 31 is one, then we will count every ungated GPU clock.

Programming Notes: Watch dog timer will be disabled when there is no valid context. The watchdog will continue counting and cause an interrupt only when a valid context is active.

This register is context saved as part of engine context.

Watchdog Counter Threshold

If the Watchdog Counter Threshhold is equal to Watchdog Counter, then the interrupt bit is set in the IIR(bit 6) and the Watchdog Counter is reset to zero.

This register is context saved as part of engine context.

Watchdog Counter

The Watchdog Counter is the count value of the watchdog timer. The Counter can be reset due to the Watchdog Counter Control being disabled or being equal to the Watchdog Counter Threshhold. The increment of the Watchdog counter is enabled when the Watchdog Counter Control is enabled and the current context is valid and execlist is enabled which includes the time to execute, flush and save the context.

The increment of the Watchdog counter is under the following conditions:

- Watchdog timer is enabled.
- Context is valid

The increment granularity is based controlled by Watchdog Counter Control mode(bit 31).

This register is not context saved and restored.

Predication

This section is under development.

Predicate Render Registers

Register	Project
MI_PREDICATE_SRC0 - Predicate Rendering Temporary Register0	
MI_PREDICATE_SRC1 - Predicate Rendering Temporary Register1	
MI_PREDICATE_DATA - Predicate Rendering Data Storage	
MI_PREDICATE_RESULT - Predicate Rendering Data Result	
MI_PREDICATE_RESULT_1 - Predicate Rendering Data Result 1	
MI_PREDICATE_RESULT_2 - Predicate Rendering Data Result 2	

MI_SET_PREDICATE

MI_SET_PREDICATE is a command that allows the driver to conditionally execute or skip a command during execution time, as detailed in the instruction definition:

The following is a list of commands that can be programmed when the PREDICATE ENABLE field in MI_SET_PREDICATE allows predication. Commands not listed here will have undefined behavior when executed with predication enabled:

Command
3DSTATE_URB_VS
3DSTATE_URB_HS
3DSTATE_URB_DS
3DSTATE_URB_GS
3DSTATE_PUSH_CONSTANT_ALLOC_VS
3DSTATE_PUSH_CONSTANT_ALLOC_HS
3DSTATE_PUSH_CONSTANT_ALLOC_DS
3DSTATE_PUSH_CONSTANT_ALLOC_GS
3DSTATE_PUSH_CONSTANT_ALLOC_PS
MI_LOAD_REGISTER_IMM
MI_STORE_DATA_IMM
3DSTATE_WM_HZ_OP
MEDIA_VFE_STATE
MEDIA_OBJECT
MEDIA_OBJECT_WALKER
MEDIA_INTERFACE_DESCRIPTOR_LOAD

MI_PREDICATE

The MI_PREDICATE command is used to control the Predicate state bit, which in turn can be used to enable/disable the processing of 3DPRIMITIVE commands.

MI_PREDICATE

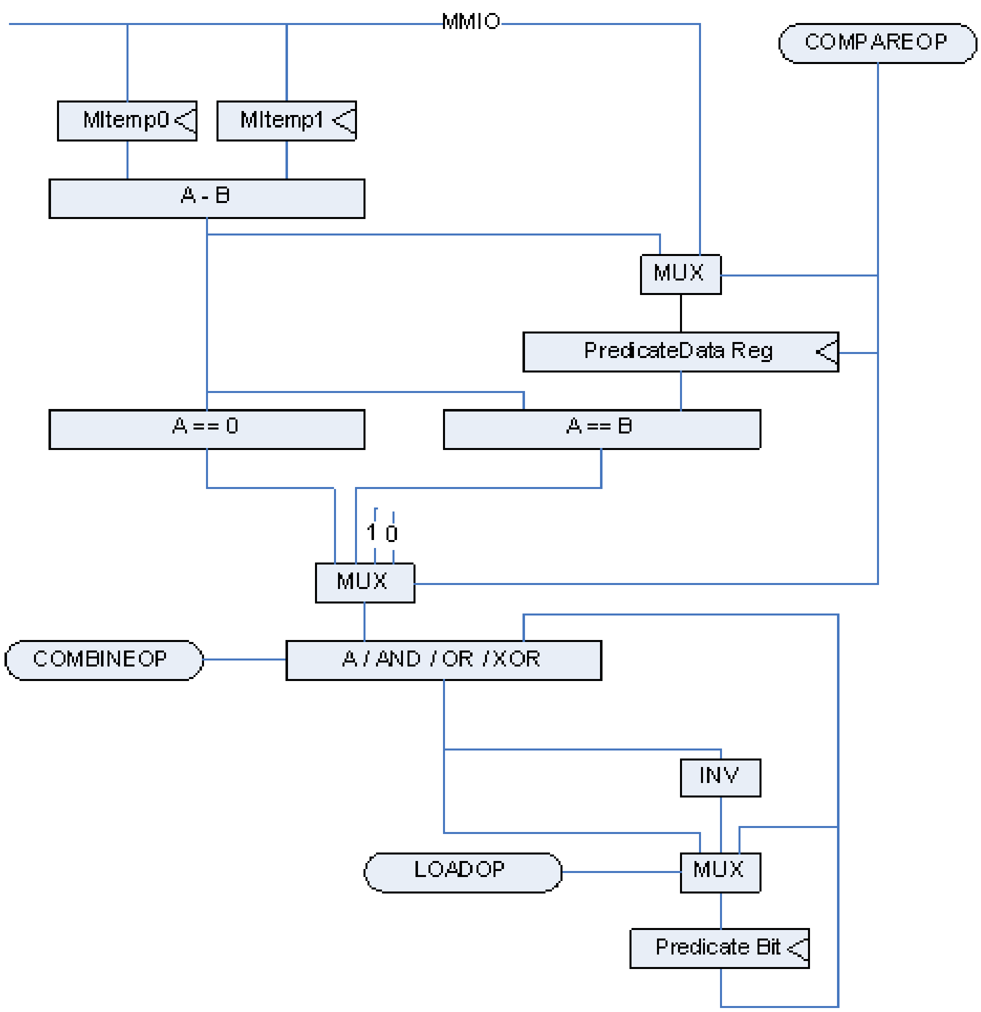
Predicated Rendering Support in HW

DX10 defines predicated rendering, where sequences of rendering commands can be discarded based on the result of a previous predicate test. A new state bit, Predicate, has been added to the command stream. In addition, a PredicateEnable bit is added to 3DPRIMITIVE. When the PredicateEnable bit is set, the command is ignored if the Predicate state bit is set.

A new command, MI_PREDICATE, is added. It contains several control fields which specify how the Predicate bit is generated.

Refer to the diagram below and the command description (linked above) for details.

MI_PREDICATE Function



MI_LOAD_REGISTER_MEM commands can be used to load the MItemp0, MItemp1, and PredicateData registers prior to MI_PREDICATE. To ensure the memory sources of the MI_LOAD_REGISTER_MEM commands are coherent with previous 3D_PIPECONTROL store-DWord operations, software can use the new **Pipe Control Flush Enable** bit in the PIPE_CONTROL command.

CS ALU Programming and Design

Command streamer implements a rudimentary ALU which supports basic Arithmetic (Addition and Subtraction) and logical operations (AND, OR, XOR) on two 64bit operands. ALU has two 64bit registers at the input SRCA and SRCB to which the operands should be loaded on which operations will be performed and outputted to a 64 bit Accumulator. Zero Flag and Carry Flag are set based on accumulator output.

CS_GPR - Command Streamer General Purpose Registers

Following are Command Streamer General Purpose Registers:

CS_GPR - General Purpose Register

Command Streamer (CS) ALU Programming

The command streamer implements a rudimentary Arithmetic Logic Unit (ALU) which supports basic arithmetic (Addition and Subtraction) and logical operations (AND, OR, XOR) on two 64-bit operands.

The ALU has two 64-bit registers at the input, SRCA and SRCB, to which source operands are loaded. The ALU result is written to a 64-bit accumulator. The Zero Flag and Carry Flag are assigned based on the accumulator output.

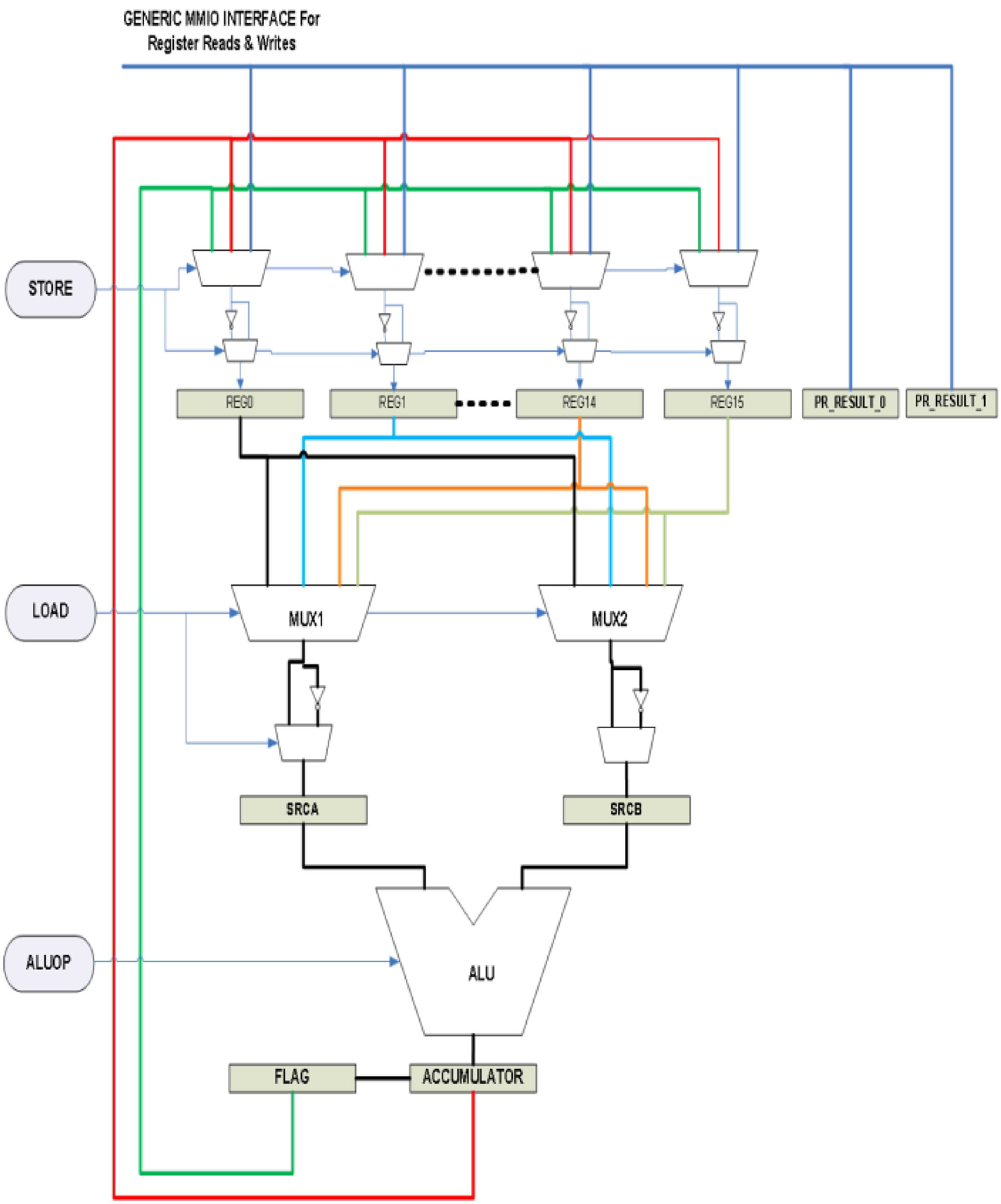
See the ALU Programming section in the Render Engine Command Streamer, for a description of the ALU programming model. Programming model is the same for all command streamers that support ALU, but each command streamer uses its own MMIO address range to address the registers. The following subsections describe the ALU registers and the programming details.

CS ALU Programming and Design

Generic Purpose Registers

Command streamer implements sixteen 64 bit General Purpose Registers which are MMIO mapped. These registers can be accessed similar to any other MMIO mapped registers through LRI, SRM, LRR, LRM or CPU access path for reads and writes. These registers will be labeled as R0, R1, … R15 throughout the discussion. Refer table in the B-spec update section mapping these registers to corresponding MMIO offset. A selected GPR register can be moved to SRCA or SRCB register using “LOAD” instruction. Outputs of the ALU, Accumulator, ZF and CF can be moved to any of the GPR using “STORE” instruction.

ALU BLOCK Diagram



Instruction Set

The instructions supported by the ALU can be broadly categorized into three groups:

- To move data from GPR to SRCA/SRCB – LOAD instruction.
- To move data from ACCUMULATOR/CF/ZF to GPR – STORE Instruction.
- To do arithmetic/Logical operations on SRCA and SRCB of ALU - ADD/SUB/AND/XOR/OR. Note: Accumulator is loaded with value of SRCA - SRCB on a subtraction.

Instruction Format

Each instruction is one Dword in size and consists of an ALU OPCODE, OPERAND1 and OPERAND2 in the format shown below.

ALU OPCODE	Operand-1	Operand-2
12 bits	10 bits	10 bits

LOAD Operation

The LOAD instruction moves the content of the destination register (Operand2) into the source register (Operand1). The destination register can be any of the GPR (R0, R1, ..., R15) and the source registers are SRCA and SRCB of the ALU. This is the only means SRCA and SRCB can be programmed.

LOAD has different flavors, wherein one can load the inverted version of the source register into the destination register or a hard coded value of all Zeros and All ones.

```
// Loads any of Reg0 to Reg15 into the SRCA or SRCB registers of ALU.

    LOAD <SRCA, SRCB>, <REG0..REG15>

// Loads inverted (bit wise) value of the mentioned Reg0 to 15 into SRCA or SRCB registers of
ALU.

    LOADINV <SRCA, SRCB>, <REG0..REG15>

// Loads "0" into SRCA or SRCB

    LOAD0 <SRCA, SRCB>

// Loads "1" into SRCA or SRCB

    LOAD1 <SRCA, SRCB>
```

31 20	19 10	9 0
Opcode	**Operand1**	**Operand2**
LOAD	SRCA/SRCB	R0,R1..R15
LOADINV	SRCA/SRCB	R0,R1..R15
LOAD0	SRCA/SRCB	N/A
LOAD1	SRCA/SRCB	N/A

Arithmetic/Logical Operations

ADD, SUB, AND, OR, and XOR are the Arithmetic and Logical operations supported by Arithmetic Logic Unit (ALU). When opcode corresponding to a logical operation is performed on SRCA and SRCB, the result is sent to ACCUMULATOR (ACCU), CF and ZF. Note that ACCU is 64-bit register. A NOOP when submitted to the ALU doesn't do anything, it is meant for creating bubble or kill cycles.

31 20	19 10	9 0
Opcode	**Operand1**	**Operand2**
ADD	N/A	N/A
SUB	N/A	N/A
AND	N/A	N/A
OR	N/A	N/A
XOR	N/A	N/A
NOOP	N/A	NA

STORE Operation

The STORE instruction moves the content of the destination register (Operand2) into the source register (Operand1). The source register can be accumulator (ACCU), CF or ZF. STORE has different flavors, wherein one can load the inverted version of the source register into destination register via STOREINV. When CF or ZF are stored, the same value is replicated on all 64 bits.

```
// Loads ACCMULATOR or Carry Flag or Zero Flag in to any of the generic registers
// Reg0 to Reg16. In case of CF and ZF same value is replicated on all the 64 bits.

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

// Loads inverted (ACCMULATOR or Carry Flag or Zero Flag) in to any of the
// generic registers Reg0 to Reg15.

    STOREINV <R0.. R15>, <ACCU, CF, ZF>
```

31 20	19 10	9 0
Opcode	**Operand1**	**Operand2**
STORE	R0,R1..R15	ACCU/ZF/CF
STOREINV	R0, R1.. R15	ACCU/ZF/CF

Summary for ALU

Total Opcodes Supported: 12

Total Addressable Registers as source or destination: 21

- 16 GPR (R0, R1 ...R15)
- 1 ACCU

- 1ZF
- 1CF
- SRCA, SRCB

Summary of Instructions Supported

31	20	19	10	9	0
Opcode		**Operand1**		**Operand2**	
LOAD		SRCA/SRCB		REG0..REG15	
LOADINV		SRCA/SRCB		REG0..REG15	
LOAD0		SRCA/SRCB		N/A	
LOAD1		SRCA/SRCB		N?A	
ADD		N/A		N/A	
SUB		N/A		N/A	
AND		N/A		N/A	
OR		N/A		N/A	
XOR		N/A		N/A	
NOOP		N/A		N/A	
STORE		REG0..REG15		ACCU/CF/ZF	
STOREINV		REG0..REG15		ACCU/CF/ZF	

Table for ALU OPCODE Encodings

ALU OPCODE	OPCODE ENCODING
NOOP	0x000
LOAD	0x080
LOADINV	0x480
LOAD0	0x081
LOAD1	0x481
ADD	0x100
SUB	0x101
AND	0x102
OR	0x103
XOR	0x104
STORE	0x180
STOREINV	0x580

In the above mentioned table, ALU Opcode Encodings look like random numbers. The rationale behind those encodings is because the ALU Opcode is further broken down into sub-sections for ease-of-design implementation.

PREFIX		OPCODE		SUBOPCODE	
11	10	9	7	6	0
PREFIX VALUE		**Description**			
0		Regular			
1		Invert			
OPCODE VALUE		**Description**			
0		NOOP			
1		LOAD			
2		ALU			
3		STORE			

ALU OPCODE	ENCODING	PREFIX		OPCODE		SUBOPCODE	
			10	9	7	6	0
NOOP	0x000	0		0		0	
LOAD	0x080	0		1		0	
LOADINV	0x480	1		1		0	
LOAD0	0x081	0		1		1	
LOAD1	0x481	1		1		1	
ADD	0x100	0		2		0	
SUB	0x101	0		2		1	
AND	0x102	0		2		2	
OR	0x103	0		2		3	
XOR	0x104	0		2		4	
STORE	0x180	0		3		0	
STOREINV	0x580	1		3		0	

Table for Register Encodings

Register	Register Encoding
R0	0x0
R1	0x1
R2	0x2
R3	0x3
R4	0x4
R5	0x5

Register	Register Encoding
R6	0x6
R7	0x7
R8	0x8
R9	0x9
R10	0xa
R11	0xb
R12	0xc
R13	0xd
R14	0xe
R15	0xf
SRCA	0x20
SRCB	0x21
ACCU	0x31
ZF	0x32
CF	0x33

MI Commands for Graphics Processing Engines

This chapter lists the MI Commands that are supported by Generic Command Streamer Front End implemented in the graphics processing engines (Render, Video, Blitter and Video Enhancement) from SKL+ products onwards.

Command
MI_NOOP
MI_ARB_CHECK
MI_ARB_ON_OFF
MI_BATCH_BUFFER_START
MI_CONDITIONAL_BATCH_BUFFER_END
MI_DISPLAY_FLIP
MI_LOAD_SCAN_LINES_EXCL
MI_LOAD_SCAN_LINES_INCL
MI_CLFLUSH
MI_MATH
MI_REPORT_HEAD
MI_STORE_DATA_IMM
MI_STORE_DATA_INDEX
MI_ATOMIC
MI_COPY_MEM_MEM
MI_LOAD_REGISTER_REG
MI_LOAD_REGISTER_MEM
MI_STORE_REGISTER_MEM
MI_USER_INTERRUPT
MI_WAIT_FOR_EVENT
MI_SEMAPHORE_SIGNAL
MI_SEMAPHORE_WAIT

Register Access and User Mode Privileges

This section describes access to the MMIO internal to the GPU and funny I/O and how to access the ranges. Command streamer limits accesses for commands that are executed out of a PPGTT batch buffer. This is also referred to a non-privilege command buffer.

Below are the Base Addresses of each command streamer and engine blocks. While this is not all the ranges, it is the ones used to reference which registers are accessible or restricted by command streamer.

Unit	MMIO Base Offset	Description
RCS	0x2000	Render Command Streamer
POCS	0x18000	Position Command Streamer
BCS	0x22000	Blitter Command Streamer
VCS/MFC	0x1C0000	Video Command Streamer 0
VCS1/MFC	0x1C4000	Video Command Streamer 1
VCS2/MFC	0x1D0000	Video Command Streamer 2
VCS3/MFC	0x1D4000	Video Command Streamer 3
VCS4/MFC	0x1E0000	Video Command Streamer 4
VCS5/MFC	0x1E4000	Video Command Streamer 5
VCS6/MFC	0x1F0000	Video Command Streamer 6
VCS7/MFC	0x1F4000	Video Command Streamer 7
VECS/MFC	0x1C8000	Video Enhancement Command Streamer 0
VECS1	0x1D8000	Video Enhancement Command Streamer 1
VECS2	0x1E8000	Video Enhancement Command Streamer 2
VECS3	0x1F8000	Video Enhancement Command Streamer 3
HEVC	0x1C2800	
HEVC1	0x1C6800	
HEVC2	0x1D2800	
HEVC3	0x1D6800	
HEVC4	0x1E2800	
HEVC5	0x1E6800	
HEVC6	0x1F2800	
HEVC7	0x1F6800	

User Mode Privileged Commands

A subset of the commands are privileged. These commands may be issued only from a privileged batch buffer or directly from a ring. Batch buffers in GGTT memory space are privileged and batch buffers in PPGTT memory space are non-privileged. On parsing privileged command from a non-privileged batch buffer, a Command Privilege Violation Error is flagged and the command is dropped. Command Privilege Violation Error is logged in Error identity register of command streamer which gets propagated as "Command Parser Master Error" interrupt to SW. Privilege access violation checks in HW can be disabled by setting "Privilege Check Disable" bit in GFX_MODE register. When privilege access checks are disabled HW executes the Privilege command as expected.

User Mode Privileged Commands

User Mode Privileged Command	Function in Non-Privileged Batch Buffers	Source
MI_UPDATE_GTT	Command is converted to NOOP.	*CS
MI_STORE_DATA_IMM	Command is converted to NOOP if **Use Global GTT** is enabled.	*CS
MI_STORE_DATA_INDEX	Command is converted to NOOP.	*CS
MI_STORE_REGISTER_MEM	Register read is always performed. Memory update is dropped if **Use Global GTT** is enabled.	*CS
MI_BATCH_BUFFER_START	Command when executed from a batch buffer can set its "Privileged" level to its parent batch buffer or lower. Chained or Second level batch buffer can be "Privileged" only if the parent or the initial batch buffer is "Privileged". This is HW enforced.	*CS
MI_LOAD_REGISTER_IMM	Command is converted to NOOP if the register accessed is privileged.	*CS
MI_LOAD_REGISTER_MEM	Command is converted to NOOP if **Use Global GTT** is enabled. Command is converted to NOOP if the register accessed is privileged.	*CS
MI_LOAD_REGISTER_REG	Register write to a **Privileged Register** is discarded.	*CS
MI_REPORT_PERF_COUNT	Command is converted to NOOP if **Use Global GTT** is enabled.	Render CS

User Mode Privileged Command	Function in Non-Privileged Batch Buffers	Source
PIPE_CONTROL	Still send flush down, Post-Sync Operation is NOOP if **Use Global GTT** or Use "Store Data Index" is enabled. Post-Sync Operation LRI to **Privileged Register** is discarded.	Render CS
MI_SET_CONTEXT	Command is converted to NOOP.	Render CS
MI_ATOMIC	Command is converted to NOOP if **Use Global GTT** is enabled.	Render CS
MI_COPY_MEM_MEM	Command is converted to NOOP if **Use Global GTT** is used for source or destination address.	*CS
MI_SEMAPHORE_WAIT	Command is converted to NOOP if **Use Global GTT** is enabled.	*CS
MI_ARB_ON_OFF	Command is converted to NOOP.	*CS
MI_DISPLAY_FLIP	Command is converted to NOOP.	*CS
MI_CONDITIONAL_BATCH_BUFFER_END	Command is converted to NOOP if **Use Global GTT** is enabled.	*CS
MI_FLUSH_DW	Still send flush down, Post-Sync Operation is converted to NOOP if **Use Global GTT** or Use "Store Data Index" is enabled.	Blitter CS, Video CS, Video Enhancement CS

Parsing one of the commands in the table above from a non-privileged batch buffer flags an error and converts the command to a NOOP.

The tables below list the non-privileged registers that can be written to from a non-privileged batch buffer executed from various command streamers.

User Mode Non-Privileged Registers for Render Command Streamer (RCS) and POSH Command Streamer (POCS)

MMIO Name	MMIO Offset	Size in DWords
Cache_Mode_0	0x7000	1
Cache_Mode_1	0x7004	1
GT_MODE	0x7008	1
L3_Config	0x7034	1
HDC_MODE	0xE5F4	1
NOPID	0x2094	1
NOPID (POCS)	0x18094	1
INSTPM	0x20C0	1

MMIO Name	MMIO Offset	Size in DWords
INSTPM (POCS)	0x180C0	1
IA_VERTICES_COUNT	0x2310	2
IA_VERTICES_COUNT (POSH)	0x18310	2
IA_PRIMIVTIVES_COUNT	0x2318	2
IA_PRIMITIVES_COUNT (POSH)	0x18318	2
VS_INVOCATION_COUNT	0x2320	2
VS_INVOCATION_COUNT (POSH)	0x18320	2
HS_INVOCATION_COUNT	0x2300	2
DS_INVOCATION_COUNT	0x2308	2
GS_INVOCATION_COUNT	0x2328	2
GS_PRIMITIVES_COUNT	0x2330	2
SO_NUM_PRIMS_WRITTEN0	0x5200	2
SO_NUM_PRIMS_WRITTEN1	0x5208	2
SO_NUM_PRIMS_WRITTEN2	0x5210	2
SO_NUM_PRIMS_WRITTEN3	0x5218	2
SO_PRIM_STORAGE_NEEDED0	0x5240	2
SO_PRIM_STORAGE_NEEDED1	0x5248	2
SO_PRIM_STORAGE_NEEDED2	0x5250	2
SO_PRIM_STORAGE_NEEDED3	0x5258	2
SO_WRITE_OFFSET0	0x5280	1
SO_WRITE_OFFSET1	0x5284	1
SO_WRITE_OFFSET2	0x5288	1
SO_WRITE_OFFSET3	0x528C	1
CL_INVOCATION_COUNT	0x2338	2
CL_INVOCATION_COUNT (POSH)	0x18338	2
CL_PRIMITIVES_COUNT	0x2340	2
CL_PRIMITIVES_COUNT (POSH)	0x18340	2
PS_INVOCATION_COUNT_0	0x22C8	2
PS_DEPTH_COUNT _0	0x22D8	2
PS_INVOCATION_COUNT_1	0x22F0	2
PS_DEPTH_COUNT _1	0x22F8	2
PS_INVOCATION_COUNT_2	0x2448	2
PS_DEPTH_COUNT_2	0x2450	2
PS_INVOCATION_COUNT_3	0x2458	2
PS_DEPTH_COUNT_3	0x2460	2
PS_INVOCATION_COUNT_4	0x2468	2

MMIO Name	MMIO Offset	Size in DWords
PS_DEPTH_COUNT_4	0x2470	2
PS_INVOCATION_COUNT_5	0x24A0	2
PS_DEPTH_COUNT_5	0x24A8	2
PS_INVOCATION_COUNT_6	0x25D0	2
PS_DEPTH_COUNT_6	0x25B0	2
PS_INVOCATION_COUNT_7	0x25D8	2
PS_DEPTH_COUNT_7	0x25B8	2
CPS_INVOCATION_COUNT	0x2478	2
GPUGPU_DISPATCHDIMX	0x2500	1
GPUGPU_DISPATCHDIMY	0x2504	1
GPUGPU_DISPATCHDIMZ	0x2508	1
MI_PREDICATE_SRC0	0x2400	1
MI_PREDICATE_SRC0 (POSH)	0x18400	1
MI_PREDICATE_SRC0	0x2404	1
MI_PREDICATE_SRC0 (POSH)	0x18404	
MI_PREDICATE_SRC1	0x2408	1
MI_PREDICATE_SRC1 (POSH)	0x18408	
MI_PREDICATE_SRC1	0x240C	1
MI_PREDICATE_SRC1 (POSH)	0x1840C	
MI_PREDICATE_DATA	0x2410	1
MI_PREDICATE_DATA (POSH)	0x18410	
MI_PREDICATE_DATA	0x2414	1
MI_PREDICATE_DATA (POSH)	0x18414	
MI_PREDICATE_RESULT	0x2418	1
MI_PREDICATE_RESULT (POSH)	0x18418	
MI_PREDICATE_RESULT_1	0x241C	1
MI_PREDICATE_RESULT_1 (POSH)	0x1841C	
MI_PREDICATE_RESULT_2	0x23BC	1
MI_PREDICATE_RESULT_2 (POSH)	0x183BC	
3DPRIM_END_OFFSET	0x2420	1
3DPRIM_END_OFFSET (POSH)	0x18420	1
3DPRIM_START_VERTEX	0x2430	1
3DPRIM_START_VERTEX (POSH)	0x18430	1
3DPRIM_VERTEX_COUNT	0x2434	1
3DPRIM_VERTEX_COUNT (POSH)	0x18434	1
3DPRIM_INSTANCE_COUNT	0x2438	1

MMIO Name	MMIO Offset	Size in DWords
3DPRIM_INSTANCE_COUNT (POSH)	0x18438	1
3DPRIM_START_INSTANCE	0x243C	1
3DPRIM_START_INSTANCE (POSH)	0x1843C	1
3DPRIM_BASE_VERTEX	0x2440	1
3DPRIM_BASE_VERTEX (POSH)	0x18440	1
3DPRIM_XP0	0x2690	1
3DPRIM_XP0 (POSH)	0x18690	1
3DPRIM_XP1	0x2694	1
3DPRIM_XP1 (POSH)	0x18694	1
3DPRIM_XP2	0x2698	1
3DPRIM_XP2 (POSH)	0x18698	1
GPGPU_THREADS_DISPATCHED	0x2290	2
BB_OFFSET	0x2158	1
BB_OFFSET (POCS)	0x18158	1
CS_GPR (1-16)	0x2600	32
CS_GPR (1-16) (POSH)	0x18600	32
OA_CTX_CONTROL	0x2360	1
OACTXID	0x2364	1
OA CONTROL	0x2B00	1
PERF_CNT_1_DW0	0x91b8	1
PERF_CNT_1_DW1	0x91bc	1
PERF_CNT_2_DW0	0x91c0	1
PERF_CNT_2_DW1	0x91c4	1
PR_CTR_CTL_RCSUNIT	0x2178	1
PR_CTR_THRSH_RCSUNIT	0x217C	1
VSR_PUSH_CONSTANT_BASE	0xE518	1
PTBR_PAGE_POOL_SIZE_REGISTER	0x18590	1
CMD_BUFF_CTL	0x2084	1
TCCNTLREG	0xB0A4	1
Z_DISCARD_EN	0x7040	1

MMIO Name	MMIO Offset	Size in DWords
BCS_GPR	0x22600	32
BCS_SWCTRL	0x22200	1
PR_CTR_CTL_BCSUNIT	0x22178	1
PR_CTR_THRSH_BCSUNIT	0x2217C	1

Refer to **Register Access and User Mode Privileges** section for Base address for the below offsets.

User Mode Non-Privileged Registers for Video Enhancement Command Streamer (VECS)

MMIO Name	MMIO Base	MMIO Offset	Size in DWords
VECS_GPR	VECS	0x600	32
PR_CTR_CTL_VECSUNIT	VECS	0x178	1
PR_CTR_THRSH_VECSUNIT	VECS	0x17C	1

* These registers are not at a standard offset from their corresponding CS MMIO base address and hence are stated individually per CS in a separate table below.

User Mode Non-Privileged Registers for Video Command Streamer (ALL VCS)

MMIO Name	Unit Base	MMIO Range	Size in DWords
VCS_GPR	VCS	0x600	32
PR_CTR_CTL_VCSUNIT	VCS	0x178	1
PR_CTR_THRSH_VCSUNIT	VCS	0x17C	1
MFC_VDBOX1	VCS	0x800	512
HEVC	HEVC	0x00	64

* These registers are not at a standard offset from their corresponding CS MMIO base address and hence are stated individually per CS in a separate table below.

Workload Submission and Execution Status

This section describes the interface to submit work and obtain status

Scheduling

RINGBUF — Ring Buffer Registers

See the "Device Programming Environment" chapter for detailed information on these registers.

Register
RING_BUFFER_TAIL - Ring Buffer Tail
RING_BUFFER_HEAD - Ring Buffer Head
RING_BUFFER_START - Ring Buffer Start
RING_BUFFER_CTL - Ring Buffer Control

Command Stream Virtual Memory Control

Per-Process GTT (PPGTT) is setup for an engine (Render, Blitter, Video and Video Enhancement) by programming corresponding Page Directory Pointer (PDP) registers listed below. Refer "Graphics Translation Tables" in "Memory Overview" for more details on Per-Process page table entries and related translations.

Enhanced Execlists

Execution-List provides a HW-SW interface mechanism to schedule context as a fundamental unit of submission to GFX-device for execution. GFX-device has multiple engines (Render, Blitter, Video, Video Enhancement) with each of them having an execution list for context submission. At any given time, all engines could be concurrently running different contexts.

A context is identified with a unique identifier called Context ID. Each context is associated with an address space for memory accesses and is assigned a unique ring buffer for command submission.

SW submits workload for a context by programming commands in to its assigned ring buffer prior to submitting context to HW (engine) for execution.

Context State:

Each context programs the engine state according to its workload requirements. All the hardware state variables of an engine required to execute a context is called context state. Each context has its own context state. Context state gets programmed on execution of commands from the context ring buffer. All the contexts designated to run on an engine have the same context format, however the values may differ based on the individual state programming.

Logical Context Address:

Each context is assigned a Logical Context Address to which the context state is saved by the engine on a context getting switched out from execution. Similarly, engine restores the context state from the logical context address of a context on getting switched in for execution.

Logical context address is an absolute graphics virtual address in global virtual memory. Context state save/restore mechanism by the engine avoids SW from re-programming the state across context switches.

Each engine has its own hardware state variables and hence they have different context sate formats. A context run on a Render engine can't be submitted to Blitter engine and vice-versa and holds true for any other engines.

Context Submission:

A context is submitted to an engine for execution by writing the context descriptor to the Execlist Submit Port (ELSP). Refer ELSP for more details. Context descriptor provides the Context ID, Address space, Logical Context Address and context valid. Refer context descriptor for more details.

Logical context address points to the context state in global virtual memory which has ring buffer details, address space setup details and other important hardware state initialization for the corresponding context. Refer Logical Context Format for more details.

Note that this mechanism cannot be used when the **Execlist Enable** bit in the corresponding engines MODE register is not set, i.e GFX_MODE register for Render Engine, BLT_MODE register for Blitter Engine, VCS_MODE register for Video Engine, or VECS_MODE register for Video Enhancement Engine.

Context Descriptor Format

Context Descriptor Format

Before submitting a context for the first time, the context image must be properly initialized. Proper initialization includes the ring context registers (ring location, head/tail pointers, etc.) and the page directory.

Render CS Only: Render state need not be initialized; the **Render Context Restore Inhibit** bit in the Context/Save image in memory should be set to prevent restoring garbage render context. See the Logical Ring Context Format section for details.

Programming Note on Context ID field in the Context Descriptor

This section describes the current usage by SW.

General Layout:

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
Eng. ID				**SW Counter**					**HW Use**		**SW Context ID**																				

Eng. ID = Engine ID (a software defined enum to identify RCS, BCS etc..)

SW Counter = Submission Counter. (SW generates an unique counter value on every submission to ensure GroupID + PASID is unique to avoid ambiguity in fault reporting & handling)

Bit 20 = Is Proxy submission. If Set to true, SW Context ID[19:0] = LRCA [31:20], else it is an index into the Context Pool.

Direct Submission

Every application gets one context ID of their own.

SW Context ID + Engine ID + SW Counter forms the unique number

The Engine ID is used to identify which engine of a given context needs to be put into wait or ready state based on Semphore/Page Fault ID value in Semaphore/Page fault FIFO.

This method allows the context to submit work to other engines while its blocked on one.

Proxy Submission

KMD creates one context for submitting work on behalf of various user mode contexts (user mode application is not using direct submission model).

This method has certain key restrictions and behaviors:

- Work (LRCA) submitted will be scheduled on the CS in the order it was received.
- KMD uses its SW Context ID in [63:32] but uses the LRCA of the user mode context.
    - KMD’s LRCA is not used for any work submission.
- If a workload hits a wait event, it does not lose its position in the schedule queue.
    - Enforces “in order” ness.
- Due to in order execution, same engine – different context semaphore synchonization is not possible.
    - Therefore, cross engine sync is simple because it clears the semaphore of the head.
- Due to in order execution, page fault on a context cannot allow a different context on same engine to execute (may preempt to idle as a power optimization).

This method allows a clean SW architecture to have KMD submissions and Ring 3 submissions to co exist.

Logical Ring Context Format

Context descriptor has the graphics virtual address pointing to the logical context in memory. Logical context has all the details required for an engine to execute a context. This is the only means through which software can pass on all the required information to hardware for executing a context. Engine on selecting a context for execution will restore (fetch-context restore) the logical context from memory to setup the appropriate state in the hardware. Engine on switching out the context from execution saves (store- context save) the latest updated state to logical context in memory, the updated state is result of the command buffer execution.

The Logical Context of each engine (Render, Video, Blitter, Video Enhancement ..etc) primarily consists of the following sections:

- Per-Process HW Status Page (4K)
- Ring Context (Ring Buffer Control Registers, Page Directory Pointers, etc.)
- Engine Context (PipelineState, Non-pipelineState, Statistics, MMIO)

Per-Process of HW status Page (PPHWSP)

This is a 4KB scratch space memory allocated for each of the context in global address space. First few cachelines are used by the engine for implicit reports like auto-report of head pointer, timestamp statistics associated with a context execution, rest of the space is available for software as scratch space for reporting fences through MI commands. Context descriptor points to the base of Per-Process HW status page. See the PPHWP format in **PPHWSP_LAYOUT**.

Logical Ring Context

Logical Ring Context starts immediately following the PPHWSP in memory. Logical ring context is five cachelines in size. This is the minimal set of hardware state required to be programmed by SW for setting up memory access and the ring buffer for a context to be executed on an engine. Memory setup is required for appropriate address translation in the memory interface. Ring buffer details the location of the ring buffer in global graphics virtual address space with its corresponding head pointer and the tail pointer. Ring context also has "Context Save/Restore Control Register-CTXT_SR_CTL" which details the engine context save/restore format. Engine first restores the Logical Ring Context and upon processing CTXT_SR_CTL it further decides the due course of Engine Context restore. Logical Ring Context is mostly identical across all engines. Logical ring context is saved to memory with the latest up to date state when a context is switched out.

Engine Context

Engine context starts immediately following the logical ring context in memory. This state is very specific to an engine and differs from engine to engine. This part of the context consists of the state from all the units in the engine that needs to be save/restored across context switches. Engine restores the engine context following the logical ring context restore. It is tedious for software to populate the engine context as per the requirements, it is recommended to implicitly use engine to populate this portion of the context. Below method can be followed to achieve the same:

- When a context is submitted for the first time for execution, SW can inhibit engine from restoring engine context by setting the “Engine Context Restore Inhibit” bit in CTXT_SR_CTL register of the logical ring context. This will avoid software from populating the Engine Context. Software must program all the state required to initialize the engine in the ring buffer which would initialize the hardware state. On a subsequent context save engine will populate the engine context with appropriate values.
- Above method can be used to create a complete logical context with engine context populated by the hardware. This Logical context can be used as a Golden Context Image or template for subsequently created contexts.

Engine saves the engine context following the logical ring context on switching out a context.

The detailed format of the logical ring context for Blitter, Video, and VideoEnhancement is documented in the chapter.

The detailed formats of the Render Logical Ring and Engine Context, including their size, is mentioned in the topic for each product.

Context Status

Hardware reports the change in state of context execution to software (scheduler) through Context Status Dword. Soft-Ware can read the context status dword from time to time to track the state of context execution in hardware. A context switch reason (Context Switch Status) quad-word (64bits) is reported to the Soft-Ware (scheduler) on a valid context getting switched out. Context switch could be a synchronous context switch (from one valid element to the other valid element in the EQ) or asynchronous context switch (Load-switching from the current executing context to the very first valid element of the newly updated EQ or on Preempt to Idle). Context switch reason is also reported on HW executing the very first valid element from EQ coming out of idle indicating hardware has gone busy from idle state (Idle to Active). Context ID reported in Context Status Dword on Idle-to-Active context switch is undefined and note that there aren't any active contexts running in hardware coming out of reset, power-on or idle.

A context switch reason reported is always followed by generation of a context switch interrupt to notify the Soft-Ware about the context switch. Soft-Ware can selectively mask the context switch status being reported and the corresponding interrupt due to a specific context switch reason. Refer Context Status Report controls section for more details.

- A status QW for the context that was just switched away from will be written to the Context Status Buffer in the Global Hardware Status Page. Context Status Buffer in Global Hardware Status Page is exercised when IA based scheduling is done. The status contains the context ID and the reason for the context switch.

Format of Context Status QWord

Context Status

Context Status should be inferred as described in the tables below. In the table below only one of the context switch types will be set and it's quite possible multiple context switch reasons are set. A "Y" in a cell indicates the possibility of the context switch type for the corresponding context switch reason.

Inference of Context Status

Ctx Switch Type Ctx Switch Reason	IDLE to Active	Preempted/ Execlist Switch	Element Switch	ACTIVE to IDLE
Context Complete	X	Y	Y	Y
Wait on Sync Flip	X	Y	Y	Y
Wait on V-Blank	X	Y	Y	Y
Wait on ScanLine	X	Y	Y	Y
Wait on Semaphore	X	Y	Y	Y
Preempt To Idle*	Y**	Y***	N	Y***

"*" - Preempt To Idle is treated as special case of execlist submission with no valid contexts, causing preemption of any ongoing context being executed followed by engine going IDLE. Context getting preempted due to "Preempt To Idle" could be in a state of context complete or Wait on Sync Flip/V-Blank/Scanline/Semaphore.

"**"-Prempt To Idle occurred when hardware is idle.

"***"- Preempt to Idle occurred when hardware is actively executing a context.

"***"- Preempt to Idle occurred when hardware is actively executing a context. Both "Preempted" and "ACTIVE to IDLE" bit are set in the Context Status.

Context Status Buffer in Global Hardware Status Page

Status QWords are written to the Context Status Buffer in Global Hardware Status Page at incrementing locations starting from DWORD offset of 28h. The Context Status Buffer has a limited size (see Table Number of Context Status Entries) and simply wraps around to the beginning when the end is reached. The status QWs can be examined to determine the contexts executed by the hardware and the reason for switching out. The most recent location updated in the Context Status Buffer is indicated by the **Last Written Status Offset** in Global Hardware Status page at DWORD offset 47h.

Refer Global Hardware Status Page Layout at **Hardware Status Page Layout**.

Number of Context Status Entries

Number of Status Entries
12 (QW) Entries

Format of the Context Status Buffer starting at DWORD offset 28h in Global Hardware Status page

QW	Description
15	**Last Written Status Offset.** The lower byte of this QWord is written on every context switch with the (pre-increment) value of the **Context Status Buffer Write Pointer**. The lower 4 bits increment for every status Qword write; bits[7:4] are reserved and must be '0'. The lowest 4 bits indicate which of the Context Status Qwords was just written. The rest of the bits [63:8] are reserved.
14:12	Reserved: MBZ.
11:0	**Context Status QWords.** A circular buffer of context status QWs. As each context is switched away from, its status is written here at ascending QWs as indicated by the **Last Written Status Offset**. Once QW11 has been written, the pointer wraps around so that the next status is written at QW0. Format = ContextStatusDW

Controls for Context Switch Status Reporting

This section describes various configuration bits available which control the hardware reporting mechanism of Context Switch Status.

Hardware reports context switch reason through context switch status report mechanism on every context switch. "Context Status Buffer Interrupt Mask" register provides mechanism to selectively mask/un-mask the context switch interrupt and the context switch status report for a given context switch reason. Hardware will not generate a context switch interrupt and context switch status report on a context switch reason that is masked in "Context Status Buffer Interrupt Mask" register. Every context switch reason reported by hardware may not be of interest to the scheduler. Scheduler may selectively mas/un-mask the context switch reasons of its interest to get notified.

Context Status Buffer Interrupt Mask Register

Preemption

Preemption is a means by which HW is instructed to stop executing an ongoing workload and switch to the new workload submitted. Preemption flows are different based on the mode of scheduling.

ExecList Scheduling

In ExecList mode of scheduling SW triggers preemption by submitting a new pending execlist to ELSP (ExecList Submit Port). HW triggers preemption on a preemptable command on detecting the availability of the new pending execlist, following preemption context switch happens to the newly submitted execlist. As part of the context switch preempted context state is saved to the preempted context LRCA, context state contains the details such that on resubmission of the preempted context HW can resume execution from the point where it was preempted.

Example:

```
Ring Buffer

MI_ARB_ON_OFF    // OFF
MI_BATCH_START   // Media Workload
MI_ARB_ON_OFF    // ON
MI_ARB_CHK       // Preemptable command outside media command buffer.
```

The following tables list the Preemptable Commands in ExecList mode of scheduling:

Engine (below)	Preemptable Commands											
	MI_ARB_CHECK	Element Boundary	Semaphore Wait	Wait for Event	3DPRIMITIVE	GPGPU_WALKER	PIPE_CONTROL ***	MEDIA STATE FLUSH	MEDIA_OBJECT_WALKER/ MEDIA_OBJECT	PIPELINE_SELECT	Any Non-Pipelined State ****	3DSTATE_PTBR_TILE_PASS_INFO
Render	AP	AP	Unsuccessful & AP	Unsuccessful & AP	Object Level (if enabled *)	Mid-Thread (if enabled **)	PIPESEL-GPGPU MODE / PIPESEL-MEDIA MODE	Mid-Thread (if enabled **)	Thread Group	PIPESEL-GPGPU MODE / PIPESEL-MEDIA MODE	PIPESEL-GPGPU MODE / PIPESEL-MEDIA MODE	N/A
Position	AP	N/A	Unsuccessful & AP	Unsuccessful & AP	Object Level (if enabled *)	N/A	N/A	N/A	N/A	N/A	N/A	AP
Blitter	AP	AP	Unsuccessful & AP	Unsuccessful & AP	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Media (VDbox)	AP	AP	Unsuccessful & AP	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Video Enhancement	AP	AP	Unsuccessful & AP	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Table Notes:

AP - Allow Preemption if arbitration is enabled.

* 0x20EC bit 0 determines whether the level of preemption is command or object level.

** 0x20E4 bits 2:1 determine the level of preemption for GPGPU workloads.

*** MI_ATOMIC and MI_SEMAPHORE_SIGNAL commands with Post Sync Op bit set are treated as PIPE_CONTROL command with Post Sync Operation as Atomics or Semaphore Signal.

**** Any Header with the value [31:29] = "011", [28:27] = "00" OR "11" and [26:24] = "001". Refer to Graphics Command Formats

Execution Status

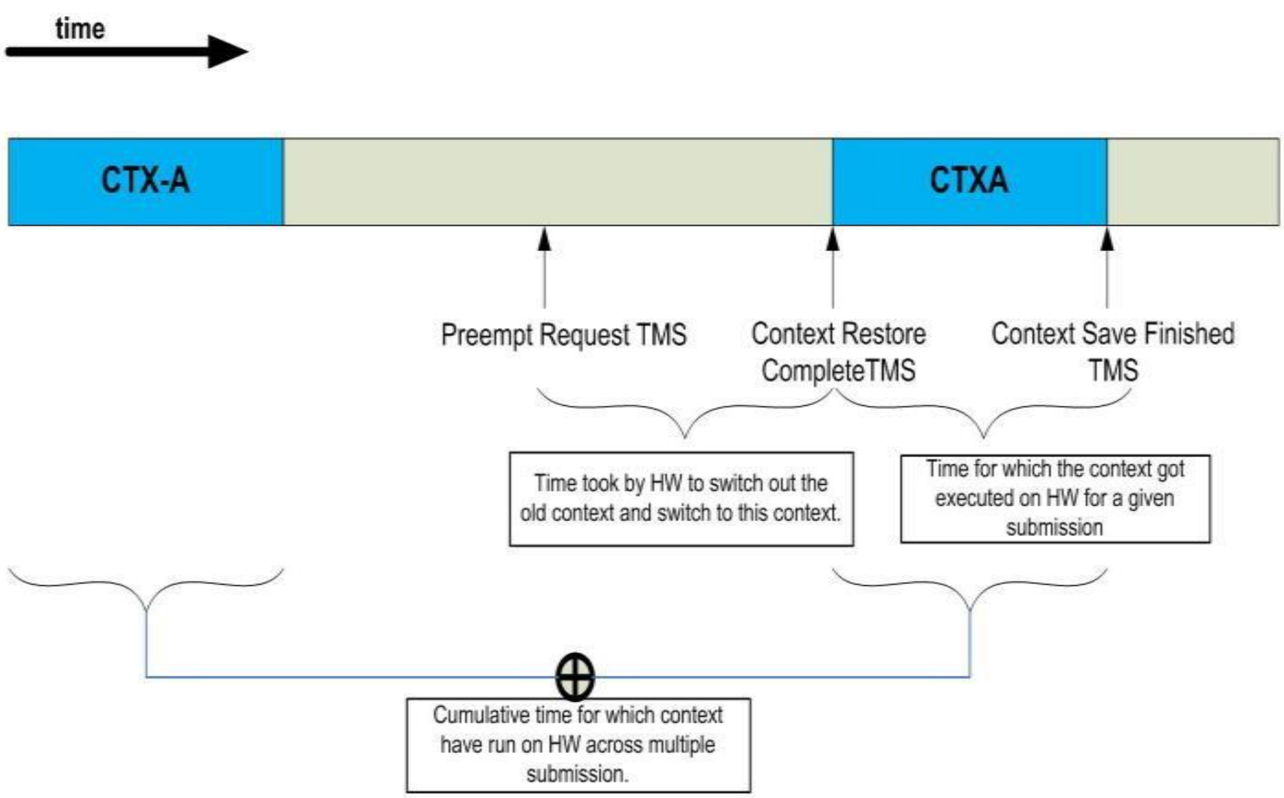
This section describes the infrastructure used to report status that the hardware provides

The Per-Process Hardware Status Page

The layout of the Per-Process Hardware Status Page is defined at **PPHWSP_LAYOUT**.

The DWord offset values in the PPHWSP_LAYOUT are in decimal.

Figure below explains the different timestamp values reported to PPHWSP on a context switch.



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

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

Hardware Status Page

The hardware status page is a naturally-aligned 4KB page residing in snooped 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.

Interrupt Control Registers

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

Bit Defintion for Interrupt Control Registers:

Engine Interrupt Vector Definition Table

Blitter Interrupt Vector
Render Engine Interrupt Vector
VideoDecoder Interrupt Vector
VideoEnhancement Interrupt Vector

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)
9	Reserved	
8	**Context Switch Interrupt.** Set when a context switch has just occurred.	Not supported to be unmasked.
7	**Page Fault.** This bit is set whenever there is a pending PPGTT (page or directory) fault. This interrupt is for handling Legacy Page Fault interface for all Command Streamers (BCS, RCS, VCS, VECS). When Fault Repair Mode is enabled, Interrupt mask register value is not looked at to generate interrupt due to page fault. Please refer to vol1c "Page Fault Support" section for more details.	Set when event occurs, cleared when event cleared. Not supported to be unmasked.
6	**Media Decode Pipeline Counter Exceeded Notify Interrupt.** The counter threshold for the execution of the media pipeline is exceeded. Driver needs to attempt hang recovery.	Not supported to be unmasked. Only for Media Pipe.
5	**L3 Parity interrupt**	Only for Render Pipe
4	**Flush Notify Enable**	0
3	**Master Error**	Set when error occurs, cleared when error cleared.
2	Reserved	
0	**User Interrupt**	0

Command Streamer > Hardware Status Mask Register

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 (except for the unrecoverable bits described below).

The following structures describe the Hardware-Detected Error bits:

The following structures describe the Hardware-Detected Error bits:

Error Bits
RCS Hardware-Detected Error Bit Definitions Structure
BCS Hardware-Detected Error Bit Definitions Structure
VCS Hardware-Detected Error Bit Definitions Structure
VECS Hardware-Detected Error Bit Definitions Structure

The following are the EIR, EMR and ESR registers:

Registers
EIR - Error Identity Register
EMR - Error Mask Register
ESR - Error Status Register

Producer-Consumer Data ordering for MI Commands

This section details the explicit data ordering enforced by HW for produce-consume of data between MI commands and explicit programming notes for data ordering not explicitly enforced by HW.

This section describes the MI commands that result in modification of data in Graphics memory or MMIO registers. These commands can be treated as producers of data for which consumers can either be SW or subsequent commands (MI or non-MI) executed by HW.

Operations (memory update or MMIO update) resulting from a command execution can be classified in to posted or non-posted.

- An operation is classified as posted if the operation initiated by the command is not guaranteed to complete (data change to be reflected) before HW moves on to the following command to execute, the posted operation is guaranteed to complete eventually. Posted operations can be forced to complete through explicit or implicit means, detailed in following section.
    - For example, a memory write is called posted if the hardware moves on to the next command after generating a memory write without waiting for the memory modification to reach a global observable point.
- An operation is classified as non-posted if the operation initiated by the command is completed before HW moves on to execute the following command.
    - For example, a memory write is called non-posted if the hardware waits for the memory write to reach a global observable point before it moves on to the next command to execute.

There are certain commands which supported both posted and non-posted operations and can be programmed by SW to select the appropriate behavior based on the usage model.

Memory Data Ordering

This section details the produce-consume data for MI commands accessing memory.

Memory Data Producer

This section describes the MI commands that modify data in graphics memory. Few commands always generate posted memory writes whereas few commands provide programmable option to generate posted Vs non-posted memory writes.

- A memory write is called posted if the hardware moves on to the next command after generating a memory write and doesn't wait for the memory modification to reach a global observable point. Since HW doesn't wait for the memory write completion it can execute the next command immediately without incurring any additional latency. Read after Write hazard is applicable in this scenario.
- A memory write is called non-posted if the hardware waits for the memory write to reach a global observable point before it moves on to the next command to execute. Since HW waits for the memory write completion before it goes on to the next command, it will incur additional latency causing a stall at top of the pipe. Read after write hazard will not happen in this scenario.

A write completion of a non-posted memory write will guarantee all the prior posted memory writes are to global observable (GO) point.

For optimal performance SW must use commands generating non-posted memory writes at the minimal. For example, a single non-posted memory write can be used just before the consume point to flush out all the prior posted memory writes to global observable point. Based on the usage model SW can use a combination of commands that generate posted memory writes and non-posted memory writes for optimal performance.

Table below lists the MI Commands that can update/modify the data in graphics memory and the associated type of memory write.

Command	Memory Write Type
MI_STORE_REGISTER_MEM	Posted
MI_COPY_MEM_MEM	Posted
MI_STORE_DATA_INDEX	Posted
MI_STORE_DATA_IMM	Posted
MI_REPORT_HEAD	Posted
MI_UPDATE_GTT	Posted
MI_REPORT_PERF_COUNT	Posted
MI_ATOMIC	Posted, Non-Posted
MI_FLUSH_DW (With Post-Sync Operation)	Non-Posted
PIPE_CONTROL (non-stalling, with Post-Sync Operation)	Posted
PIPE_CONTROL (Stalling, Post-Sync Operation)	Non-Posted

Apart from the MI commands that generate Non-Posted memory writes listed in the above table, execution of following commands will also implicitly ensure all prior posted writes are to Global Observable point.

Command
PIPE_CONTROL (Stalling)
MI_FLUSH_DWORD

Memory Data - Consumer

Table below lists the MI command that read the data from graphics memory as part of the command execution. Data in memory should be coherent prior to execution of these command to achieve expected functional behavior upon execution of these commands, Graphics memory writes by the earlier executed MI commands must be GO prior to execution of these commands. However starting from BDW hardware has started explicitly enforcing data ordering for few of the commands (based on the prevalent usage models) and mentioned in the table below.

Command	Coherency Requirement
MI_LOAD_REGISTER_MEM	HW implicitly ensures memory writes by the prior MI commands by the corresponding engine are coherent for this command execution.
MI_BATCH_BUFFER_START	SW must ensure the data cohrency.
MI_CONDITIONAL_BATCH_BUFFER_END	SW must ensure the data cohrency.
MI_ATOMIC	HW implicitly ensures memory writes by the prior MI commands by the corresponding engine are coherent for this command execution.
MI_SEMAPHORE_WAIT	HW implicitly ensures memory writes by the prior MI commands by the corresponding engine are coherent for this command execution.

SW can use any of the MI commands that generate non-posted memory writes or the commands that implicitly force prior memory writes to GO to ensure data is coherent in memory prior to execution of these commands.

MMIO Data Ordering

This section details the produce-consume data for MI commands accessing MMIO registers.

MMIO Data Producer

Table below lists the MI commands that modify data in MMIO registers and also states if the MMIO writes generated are posted Vs non-posted.

- A MMIO write is called non-posted if the hardware waits for the MMIO update to occur before it moves on to the next command to execute.
- A MMIO write is called posted if the hardware moves on to the next command after generating a MMIO write without waiting for the MMIO update to occur.

All the MI commands listed below generate non-posted MMIO writes and hence HW guarantees the MMIO modification has taken place before HW moves on the following command.

MI_LOAD_REGISTER_MEM supports both posted and non-posted behavior and can be configured through "Async Mode Enable" bit in the command header.

Command	MMIO Write Type
MI_LOAD_REGISTER_IMM	Non-Posted
PIPE_CONTROL	Non-Posted
MI_LOAD_REGISTER_MEM	Posted, Non-Posted
MI_MATH	Non-Posted
MI_LOAD_REGISTER_REG	Non-Posted

MMIO Data Consumer

All the commands that modify the MMIO are non-posted and hence any MI command consumer of MMIO data will always get the latest updated value.

Software must take care of appropriately programming the "Async Mode Enable" bit in MI_LOAD_REGISTER_MEM command based on the requirements to enforce data ordering between producer and consumer. Table below lists the MI commands that consume the MMIO data.

Command
MI_STORE_REGISTER_MEM
MI_PREDICATE
MI_LOAD_REGISTER_REG
MI_MATH
MI_SET_PREDICATE
MI_SEMAPHORE_WAIT (register poll)