Intel® Arc™ A-Series Graphics and Intel Data Center GPU Flex Series

Open-Source Programmer's Reference Manual

For the discrete GPUs code named "Alchemist" and "Arctic Sound-M"

Volume 8: Command Stream Programming

March 2023, Revision 1.0

Notices and Disclaimers

Table of Contents

Command Stream Programming 1

Introduction 1

Workload Submission and Execution Status 2

Scheduling and Execlists 2

Preemption 11

Execution Status 11

Commands and Programming Interface 14

Command Buffers 14

Command Streamer Command Formats 18

Execution Control Infrastructure 21

MI Commands for Graphics Processing Engines 32

Register Access and User Mode Privileges 33

Context Management 46

Global State 47

Context State 52

Producer-Consumer Data ordering for MI Commands 56

Memory Data Ordering 57

MMIO Data Ordering 58

Command Fetch 59

Advanced Command Prefetch 60

Self-Modifying Code 60

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. Graphics, Video Decode engine, Video Enhancement Engine and Blitter engine.

Some product SKUs 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.

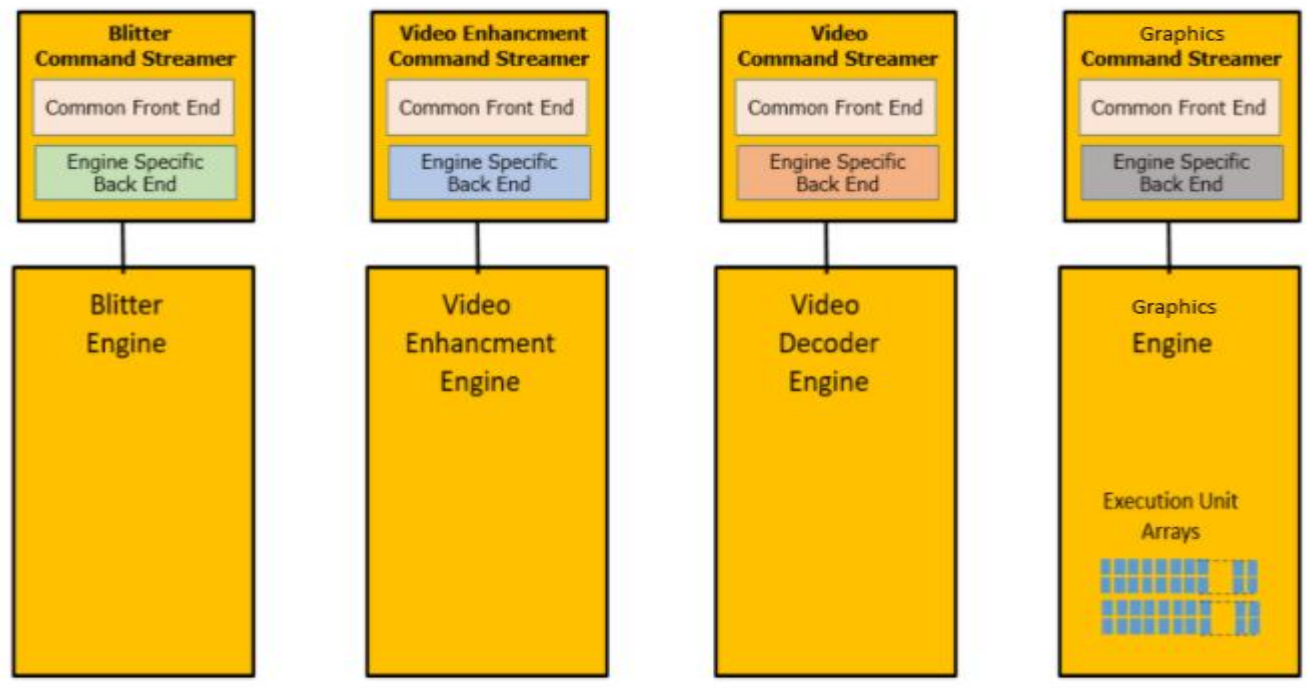


Figure 1: High level view of Command Streamer

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, synchronization, etc.).

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

Workload Submission and Execution Status

This section describes the interface to submit work and obtain status

Scheduling and 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 (Graphics, 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 into 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 state 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.

Execution List Submission Port (ELSP)

Execution List Submit Port is a MMIO register in every engine. Contexts are queued in for submission to the engine by writing the context descriptor to the engine's ELSP. ELSP provides flexibility to queue up to eight contexts at a time for submission called Submission Queue (SQ). SQ is a staging buffer in the engine which can hold up to eight contexts (elements) for submission. ELSP provides a mechanism to load the elements of the SQ in a cyclic order which wraps around to E0 on writing to E7 (E0, E1 ... E7, E0 ..). Each element of the SQ is also individually MMIO mapped which can be written to or read from, this provides an alternate flexible mechanism to independently modify any element of the SQ in any order avoiding ELSP port. The valid bits of the unused context descriptors should be set to '0', in a pathological case all the eight elements can have the valid bits of the context descriptors set to '0' (Empty Submission Queue).

Writing to SQ through ELSP or direct MMIO itself doesn't trigger the engine to start executing the elements from the SQ. Engine has to be explicitly notified to start executing the elements from the SQ by writing to the "Load" bit in the Execlist Control Register. Engine on detecting Load notification will sample the SQ to its internal execution staging buffer called Execution Queue (EQ). On loading Execution Queue becomes valid. Engine will start executing elements with valid context descriptors in serial order from the valid EQ, starting from E0 followed by E1 followed by E2 and so on to E7, invalid elements are simply skipped by the engine. EQ becomes invalid following execution of E7 making engine idle.

Moving from one element to the other element in an EQ is called synchronous context switch. Once a context is switched out, the relevant context state and context descriptor doesn't exist in EQ, only way the context can be brought back for execution into EQ is through a new "Load" from SQ. Refer context switch section for more details.

SW can modify SQ as many times as needed before issuing the "Load" command to the engine for execution. SQ contents are retained and not destructed on issuing "Load" and SQ contents are also retained across power flows.

Issuing a "Load" command while there is an ongoing element execution from EQ will result in immediate sampling of SQ to EQ and also results in preemption of the executing context on appropriate boundary. Once preemption of the ongoing executing context is complete, engine will start executing from Element-0 of the updated EQ. Preemption of a context is called asynchronous context switch and refer context switch section for more details. Multiple loads occurring while there is an ongoing preemption of an executing context will result in EQ getting updated multiple times, engine will only execute the latest EQ available upon completion of the preemption and will not get to see the intermediate updates.

Execlist Submit Port Register

Execlist Submission Queue Contents

Execlist Control Register

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.

If supported in register programming, the **Context Restore Inhibit** bit in the Context/Save image in memory can be set to prevent restoring garbage engine 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.

Context Descriptor.ContexID (DW1) 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
SW Counter						Reserved			SW Context ID																Reserved	Virtual Function Number					

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)

SW Context ID represents a software assigned unique context ID.

Virtual Function Number represents function number when virtualization is enabled. Set to zero when virtualization is not enabled.

Direct Submission (Ring 3 to GuC)

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 it's blocked on one.

Proxy Submission (In order submission from Kernel Mode Driver to GuC)

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 synchronization 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 Memory Object Overview/Logical Contexts chapter.

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

RINGBUF -- Ring Buffer 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 (Graphics, 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.

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.
- A context switch status QW (8 bytes) for the context that was just switched away (including Idle to Active) will be reported to the GUC. Firmware running on GUC or the SW running on host must read the reported context status QW through "CSB Read Port" and "CSB FIFO Status" MMIO registers implemented in GUC. Refer "Command Streamer Status Information" section under GUC for more details.

Format of Context Status QWord

Context Status

Context Status should be inferred as described in the table below.

IDLE_CTXID Encoding

IDLE_CTXID
0xFFFF

S.No	Switch to New Queue	Ctid Away	Ctxid To	Switch Detail	Description
1	1	IDLE_CTXID	0xAB	0	**Idle to Active** Ctxid Away: IDLE_CTXID indicates HW was idle when switched to the new queue. Ctxid To: 0xAB is the context picked form the newly submitted queue to execute.
2	1	0xAB	IDLE_CTXID	0-5	**Preempt to Idle** Ctxid Away: 0xAB is the context that got switched out due to Preempt To Idle. Ctxid To: IDLE_CTXID indicates HW will go Idle following this context switch. Switch To New Queue field status set distinguishes between Preempt To Idle Vs Active To Idle Switch.
3	1	IDLE_CTXID	IDLE_CTXID	X	**Preempt To Idle, Idle to Active** Preempt To Idle has occurred when HW was idle. Ctxid Away: IDLE_CTXID indicates HW was idle when switched to the new queue (Preempt To Idle). Ctxid To: IDLE_CTXID indicates HW will go Idle following this context switch.
4	1	0xAB	0x7BC	0	**Switched to New queue and also the earlier context is complete.** Ctxid Away: 0xAB is the context that got switched out due to submission of new queue and also the context is complete. Ctxid To: 0x7BC is the context picked form the newly submitted queue to execute.

5	1	0xAB	0x7BC	5	**Switched to new queue with the earlier context preempted.** Ctxid Away: 0xAB is the context that got preempted and switched out due to submission of new queue. Ctxid To: 0x7BC is the context picked form the newly submitted queue to execute.
6	1	0xABC	0x7BC	1-4	**Switched to new queue, at the time of switch, executing context was waiting on an un-successful wait.** Ctxid Away: 0xAB is the context that got switched out on an un-successful wait due to submission of new queue. Ctxid To: 0x7BC is the context picked form the newly submitted queue to execute.
7	1	0xAB	0xAB	-NA-	**Lite restore.** Switched to new queue.
8	0	0xAB	0x7AC	0	**Element (Synchronous context) switch on context complete.** Ctxid Away: 0xAB is the context that got switched out due to context complete. Ctxid To: 0x7AC is the next element (context) form the execution queue selected to execute.
9	0	0xAB	0x7AC	1-4	**Element (Synchronous context) switch on un-successful wait.** Ctxid Away: 0xAC is the context that got switched out due to un-successful wait. Ctxid To: 0x7AC is the next element (context) form the execution queue selected to execute.
10	0	0xAB	IDLE_CTXID	0-4, 5*	**Active to Idle.** Ctxid Away: 0xAB is the context that got switched out due to context complete or un-successful wait. Ctxid To: IDLE_CTXID indicates HW will go Idle following this context switch. Switch To New Queue field reset status distinguishes between Active To Idle Switch Vs Preempt To Idle. Switch Detail as 5 is possible on Preempt to Idle.

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 to the Global 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 table lists Preemptable Commands in ExecList mode of scheduling:

Command Streamer Preemptable Commands

Preemptable Command	Condition
MI_ARB_CHECK	AP
Element Boundary	AP (if allowed)
Semaphore Wait	Unsuccessful & AP
Wait for Event	Unsuccessful & AP (if allowed)

Table Notes:

AP - Allow Preemption if arbitration is enabled.

For additional preemptable commands specific to any engine type, refer to the engine specific command interface documentation.

Execution Status

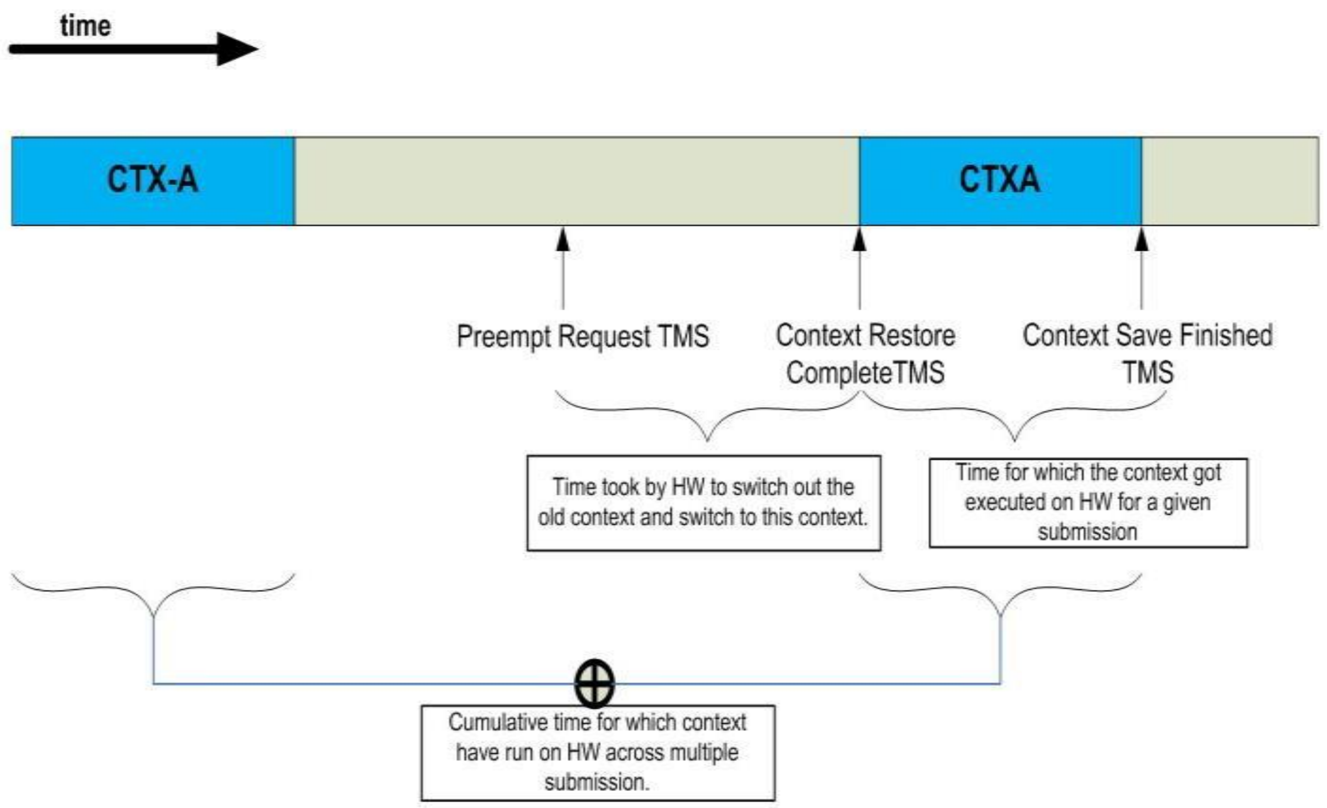
This section describes the infrastructure used to report status that the hardware provides

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.

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


time
CTX-A
CTXA
Preempt Request TMS
Context Restore CompleteTMS
Context Save Finished TMS
Time took by HW to switch out the old context and switch to this context.
Time for which the context got executed on HW for a given submission
Cumulative time for which context have run on HW across multiple submission.

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 memory. This page exists primarily to allow the device to report status via GGTT writes.

The address of this page is programmed via the HWS_PGA MI register.

Hardware Status Page Address Register

Interrupt Control Registers

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

Bit Definition for Interrupt Control Registers:

Engine Interrupt Vector Definition Table

Structures
Blitter Interrupt Vector
Render Engine Interrupt Vector
VideoDecoder Interrupt Vector
VideoEnhancement Interrupt Vector
Compute Engine 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	**Primary Error**	Set when error occurs, cleared when error cleared.
2	Reserved	
0	**User Interrupt**	0

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 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
ComputeCS 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

Commands and Programming Interface

This section describes the command supported by command streamer and the programming interface.

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.

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).	

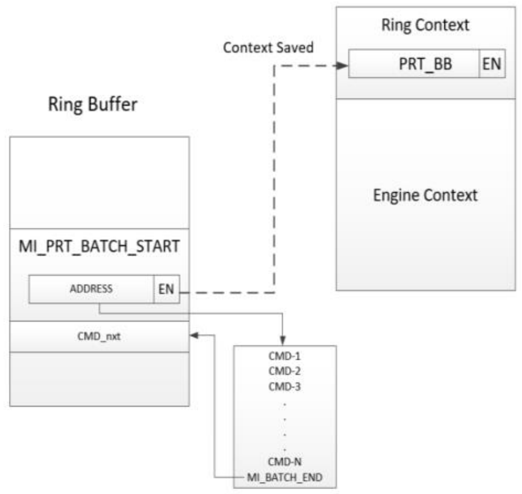
Persistent Batch Buffer

Persistent batch buffer provides a mechanism to jump and execute a batch of commands from a buffer in graphics memory (PPGTT or GGTT) from within a command sequence, like a batch buffer. Command sequence in a persistent batch buffer is ended through MI_BATCH_BUFFER_END command. Primary differentiating feature it supports is when enabled through “Persistence Enable” in the command, the persistent batch buffer details are saved as part of the context state when executed and the persistent batch buffer gets executed on subsequent context restore of the corresponding context before resuming the regular command buffer execution (ring buffer or batch buffer). Persistent batch buffer can be programmed form within a Ring Buffer or a batch buffer. Persistent batch buffer can be invoked from the command sequence several times with different start address, while the hardware will only context save the details of the most recently executed persistent batch buffer.

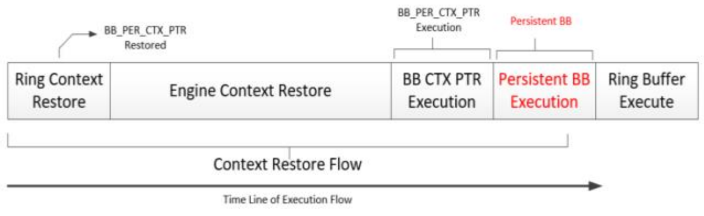
MI_PRT_BATCH_BUFFER_START

Persistent Batch Buffer State Register

The diagram below shows the overall working mechanism of Persistent Batch Buffer.



Execution of Persistent Batch Buffer from a Ring Buffer



Persistent Batch Buffer Execution on a Context Restore

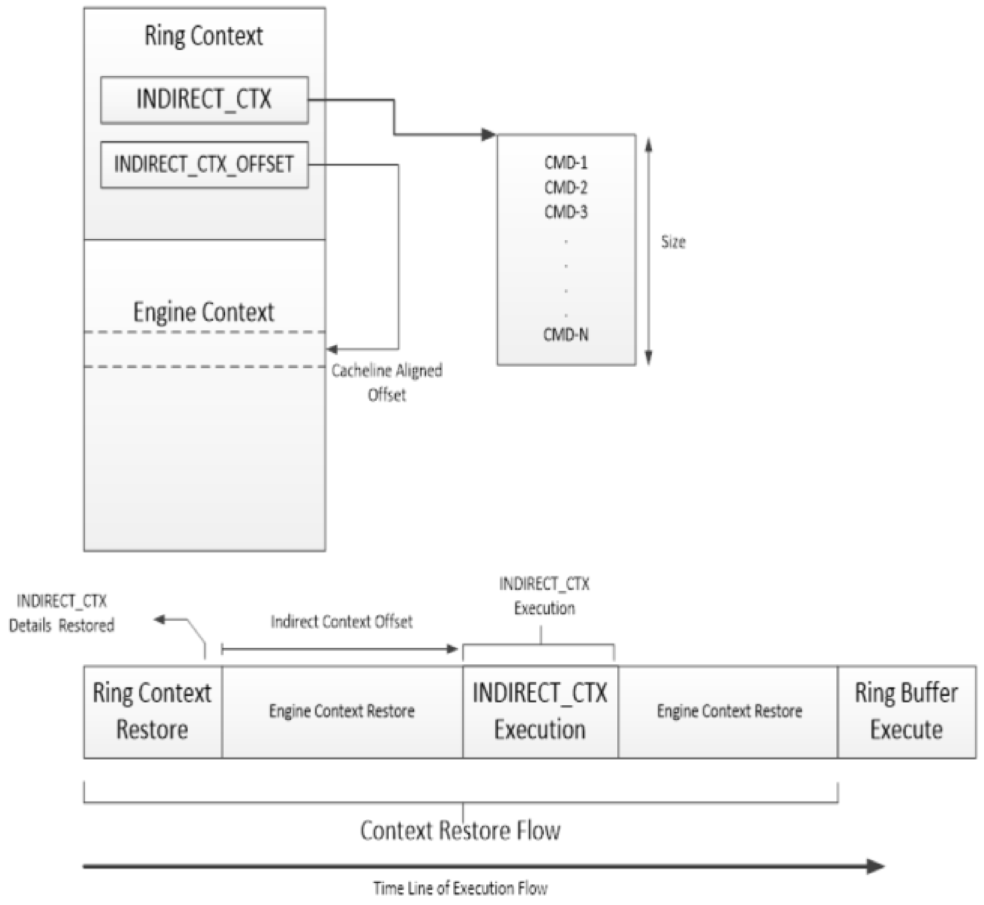
Workaround Batch Buffers

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)

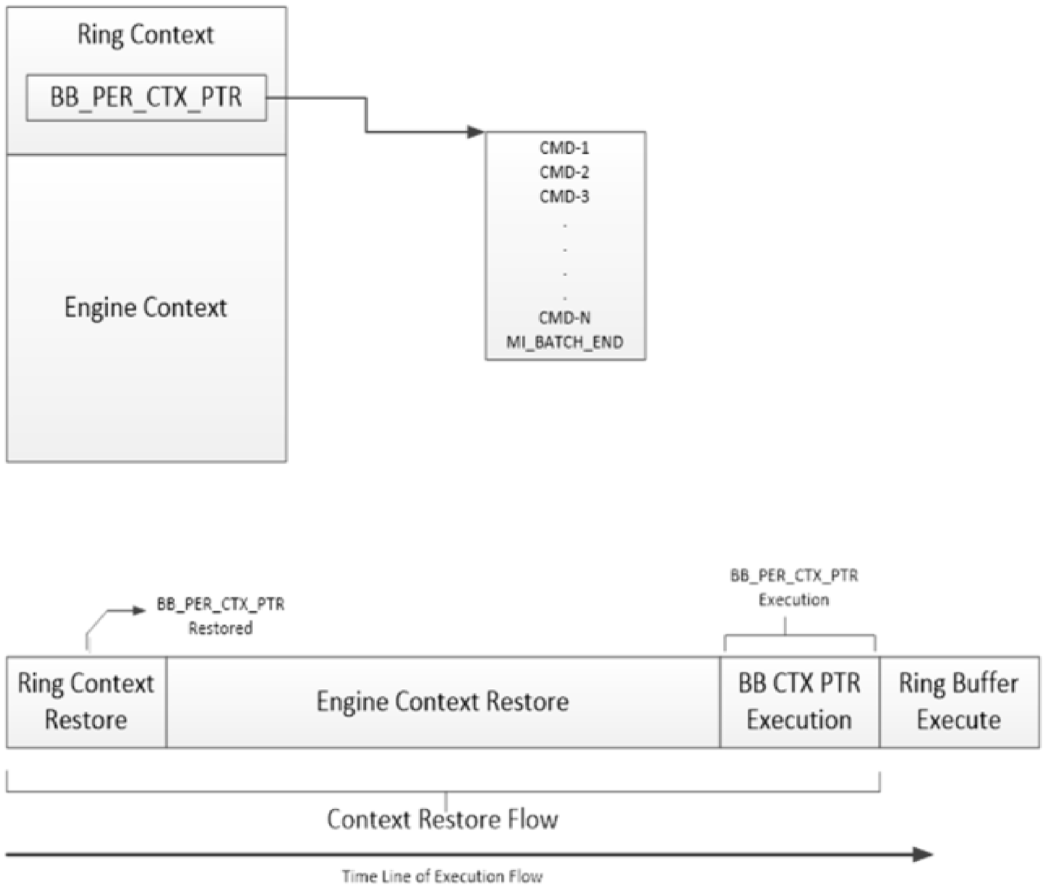
As shown in the figure below, this workaround buffer can be invoked at any cacheline aligned offset in the engine context.



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.

Command Streamer Command Formats

This section describes the general format of the command streamer commands.

Command streamer 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.

Command streamer 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 command streamer 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

Engine Command Header Format

Type	Bits	
	31:29	28:0
Memory Interface (MI)	000	
Engine Command	010, 011	
Reserved	001, 100, 101, 110, 111	

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	All
02h	MI_USER_INTERRUPT	All
03h	MI_WAIT_FOR_EVENT	Render, Blitter
04h	MI_WAIT_FOR_EVENT_2	Render, Blitter
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

Opcode (28:23)	Command	Pipes
1Dh	MI_FORCE_WAKEUP	All except Render
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
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
39h	MI_PRT_BATCH_BUFFER_START	All
37h-38h	Reserved	
39h	Reserved	All
39h-3Fh	Reserved	

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.

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

Command Streamers supports predication logic to conditionally execute the commands from the ring buffers and from batch buffers. This provides programmability to SW to dynamically control the execution flows from the submitted command buffers while getting executed on the graphics hardware.

Predication Control in commands is provided in two ways:

. Explicit bit in the command to enable predication based on a predication result.

. Commands that don't have explicit predication enable gets always predicated based on a predication result.

Table below lists the commands that explicitly support predication control and the associated predication result based of which predication logic is computed. When the predication logic gets evaluated to true, the command doesn't get executed and gets NOOP'ed resulting in no action taken.

Command's Listing Predication Control

Command	Predication Condition
3DPRIMITIVE	MI_PREDICATE_RESULT[0] == '0
3DSTATE_WM_HZ_OP	MI_PREDICATE_RESULT[0] == '0
MI_STORE_REGISTER_MEM	MI_PREDICATE_RESULT[0] == '0
COMPUTE_WALKER	MI_PREDICATE_RESULT[0] == '0
PIPE_CONTROL	MI_PREDICATE_RESULT[0] == '0
MI_BATCH_BUFFER_START	MI_SET_PREDICATE_RESULT[0] == '1
MI_BATCH_BUFFER_END	MI_SET_PREDICATE_RESULT[0] == '1
MI_CONDITIONAL_BATCH_BUFFER_END	MI_SET_PREDICATE_RESULT[0] == '1
All Other Commands	MI_SET_PREDICATE_RESULT[0] == '1

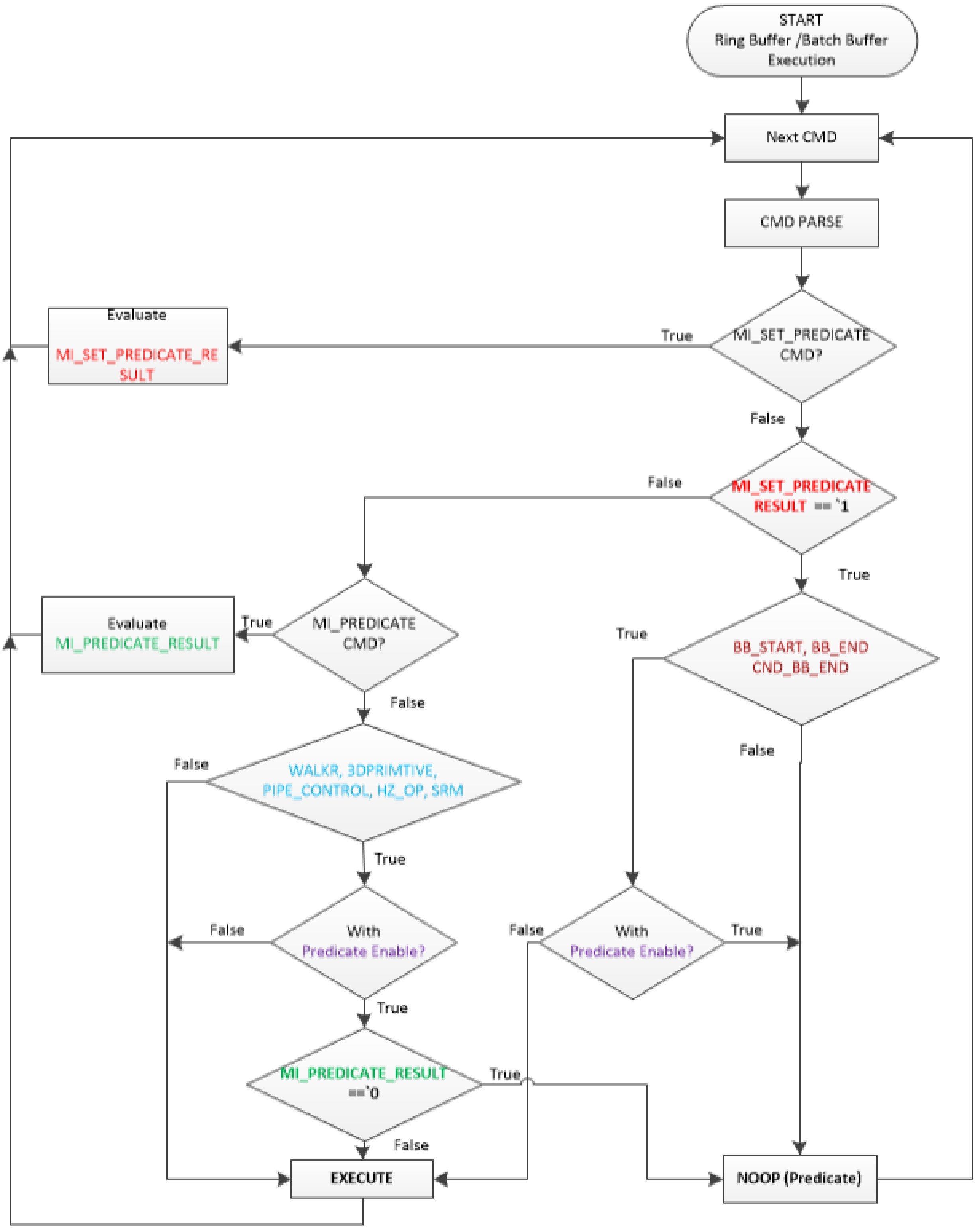
MI_PREDICATE_RESULT and MI_SET_PREDICATE_RESULT registers gets updated as a result of execution of **MI_PREDICATE** and **MI_SET_PREDICATE** commands respectively. These registers are also SW R/W. MI_PREDICATE_RESULT and MI_PREDICATE_RESULT_2 are SW R/W registers inputted by MI_SET_PREDICATE command to compute MI_SET_PREDICATE_RESULT value.

The table below lists the predication result registers and their details.

Predication Result Table

Predication Result Register	Engines Supported	Context Specific	Modification
MI_PREDICATE_RESULT	RenderCS, ComputeCS	Engine Context	MI_PREDICATE (cmd), MMIO R/W
MI_SET_PREDICATE_RESULT	All	Engine Context	MI_SET_PREDICATE(cmd), MMIO R/W
MI_PREDICATE_RESULT_1	All	Engine Context	MMIO R/W
MI_PREDICATE_RESULT_2	All	Power Context	MMIO R/W

The Flow Chart below shows how the various predication logic gets applied hierarchically in the command execution.



Predication Logic Flow Chart

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 commands.

MI_PREDICATE

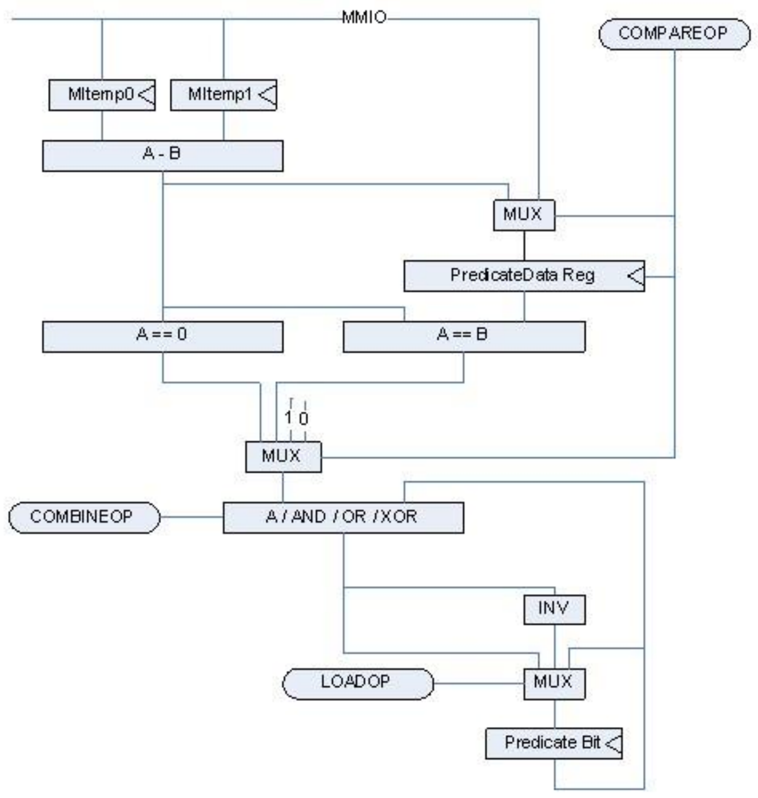
Predicated Rendering Support in HW

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 any command that may be predicated. 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 PIPE_CONTROL store-DWord operations, software can use the new **Pipe Control Flush Enable** bit in the PIPE_CONTROL command.

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:

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.

Access to this ALU is thru the **MI_MATH** command. The MI_MATH command supports MOCS programming and predication.

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.

In addition to the operations above, the ALU supports both a logical and arithmetic shift operations.

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.

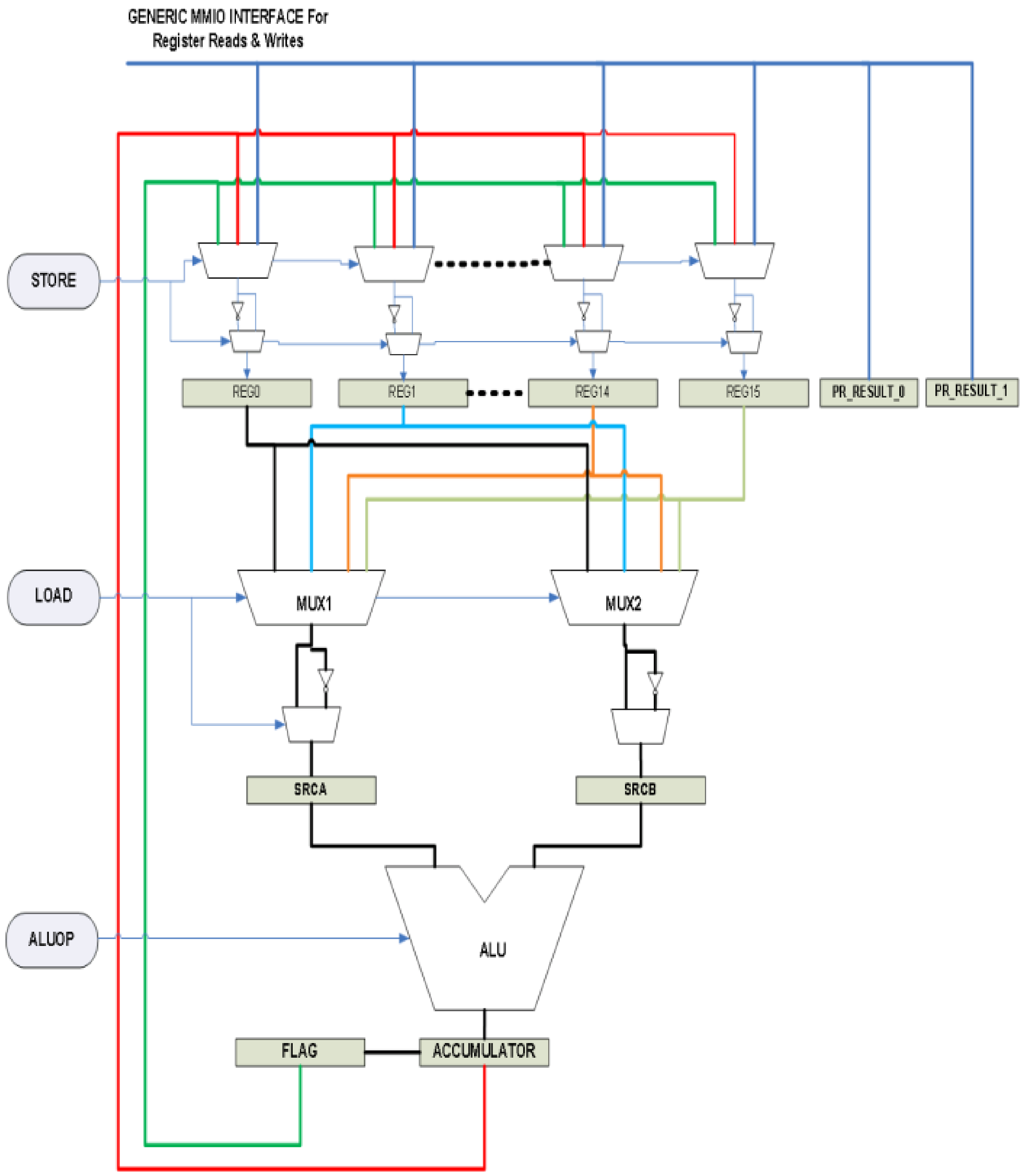
See the ALU Programming section in the Render Engine Command Streamer, for a description of the ALU programming model. Programming model is the same for all command streamers that support ALU, but each command streamer uses its own MMIO address range to address the registers. The following subsections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

NOOP and FENCE Operations

NOOP operation has does no operation but will delay and add operation idle time between commands.

FENCE_RD command will ensure any pending reads due to LOADIND command is complete. FENCE_WR command will ensure any pending writes due to STOREIND command is complete. Otherwise, these commands have the same behavior and performance of a NOOP.

Opcode	Operand1	Operand2
31:20	19:10	9:0
NOOP	N/A	N/A
FENCE_RD	N/A	N/A
FRNCE_WR	N/A	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.

Opcode	Operand1	Operand2
31:20	19:10	9:0
ADD	N/A	N/A
SUB*	N/A	N/A
OR	N/A	N/A
XOR	N/A	N/A
SHL	N/A	N/A

Opcode	Operand1	Operand2
SHR	N/A	N/A
SAR	N/A	N/A

*Note: Accumulator is loaded with value of SRCA - SRCB on a subtraction.

For all shift commands(SHL, SHR and SAR), SRCB is the shift value and SRCA is the value to be shifted. The shift commands only support values of 1, 2, 4, 8,16 and 32 in SRCB. HW will pick the next closest lower value if any other value is programmed. i.e. a value of 15 will shift by 8. Value of zero means no shift.

SHL is a logical shift left where zeros are inserted from least significant bit to the most significant bit

SHR is a logical shift right where zeros are inserted from most significant bit to the lowest significant bit.

SAR is an arithmetic right shift where the most significant bit is replicated from the most significant bit to the least significant bit.

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>


  // Loads any GPR (generic register) value from the indirect address specified by the
Accumulator. Memory address in Accumulator is always treated as 64b aligned on the memory
read.

     LOADIND   <R0.. R15>, *ACCU
```

HW will ensure any pending data to be loaded into a GPR register will be coherent with any command either loading or storing to that GPR. This includes STORE, STOREINV, LOAD, LOADINV,

LOADIND and STOREIND operations. This ensures we will avoid any RW or WW hazards. For example, LOAD SRCA, REG0 following a LOADIND REG0, ACCU operation will stall till REG0 data has returned from memory.

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

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>


// Stores any GPR (generic register) value to the indirect address specified by the
Accumulator. Memory address in Accumulator is always treated as 64b aligned on memory write.
    STOREIND  *ACCU, <R0.. R15>
```

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

Table for ALU OPCODE Encodings

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	OPCODE ENCODING	PREFIX(11:10)	OPCODE(9:7)	SUB-OPCODE(6:0)
NOOP	0x000	0	0	0
FENCE_RD	0x001	0	0	1
FENCE_WR	0x002	0	0	2
LOAD	0x080	0	1	0
LOADINV	0x480	1	1	0
LOAD0	0x081	0	1	1
LOAD1	0x481	1	1	1
LOADIND	0x082	0	1	2
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
SHL	0x105	0	2	5
SHR	0x106	0	2	6
SAR	0x107	0	2	7
STORE	0x180	0	3	0
STOREINV	0x580	1	3	0
STOREIND	0x181	0	3	1

Table for Register Encodings

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

Command
MI_NOOP
MI_ARB_CHECK
MI_ARB_ON_OFF
MI_BATCH_BUFFER_START
MI_CONDITIONAL_BATCH_BUFFER_END
MI_DISPLAY_FLIP(Graphics/Copy Only)
MI_LOAD_SCAN_LINES_EXCL(Graphics/Copy Only)
MI_LOAD_SCAN_LINES_INCL(Graphics/Copy Only)
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
MI_SET_PREDICATE
MI_UPDATE_GTT
MI_PRT_BATCH_BUFFER_START

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
BCS	0x22000	Blitter Command Streamer
CCS0	0x1A000	Compute Command Streamer 0
CCS1	0x1C000	Compute Command Streamer 1
CCS2	0x1E000	Compute Command Streamer 2
CCS3	0x26000	Compute Command Streamer 3
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
AV1/VDBOX0	0x1C2B00	AV1/Video Decode Block
AV1/VDBOX1	0x1C6B00	
AV1/VDBOX2	0x1D2B00	
AV1/VDBOX3	0x1D6B00	
AV1/VDBOX4	0x1E2B00	
AV1/VDBOX5	0x1E6B00	
AV1/VDBOX6	0x1F2B00	
AV1/VDBOX7	0x1F6B00	
HEVC	0x1C2800	
HEVC1	0x1C6800	
HEVC2	0x1D2800	
HEVC3	0x1D6800	

Unit	MMIO Base Offset	Description
HEVC4	0x1E2800	
HEVC5	0x1E6800	
HEVC6	0x1F2800	
HEVC7	0x1F6800	

Read Only User Mode Privilege MMIO Access

The tables below specify the offsets that are allowed for MMIO reads within a non-privileged batch buffer (PPGTT). This is in addition to what is already whitelisted for writes in the User Mode Privileged Commands section. Refer to Register Access and User Mode Privileges section for Base address for the below offsets.

CS means all command streamers.

Read Only Whitelist

Name	Base Address (default=none)	MMIO Offset (hex)	Size in DW
All Command Streamers			
OAG_PERF_<x>		2700	64
OAG_PERF_<x>		2B00	320
OAG_PERF_<x>		D900	192
OASTATUS		DAFC	1
OAHEADPTR		DB00	1
OATAILPTR		DB04	1
GFXREG_GT		145040	7
GFXREG_IA		145828	13
GFXREG_IO		145928	24
RP_STATUS0		A01C	1
GFXREG_GT_GFX_RC6		138108	1
GFXREG_GT_GFX_RC6P		13810C	1
Perf Profiler Timer Reg		D00	1
Graphics CS			
GPU_TIMESTAMP		2358	2
GPU_TIMESTAMP		18358	2
CS_ENGINE_ID		208C	1
CS_ENGINE_ID		1808C	1
OAR_PERF_<x>		2800	192
GFXREG_UNSLICE_FF_CTRL_FLC_THRSHLD1		A288	1
GFXREG_UNSLICE_FF_CTRL_FLC_THRSHLD2		A28C	1
GFXREG_UNSLICE_FF_COUNT1		A538	1

Name	Base Address (default=none)	MMIO Offset (hex)	Size in DW
GFXREG_UNSLICE_FF_COUNT2		A53C	1
GFXREG_RPPREVUP		A058	1
GFXREG_RPPREVDN		A064	1
GFXREG_RPUPEI		A068	1
GFXREG_RPDNEI		A06C	1
GFXREG_GT_GFX_RC6		138108	1
GFXREG_GT_GFX_RC6P		13810C	1
CS_CTX_TIMESTAMP		23A8	1
ComputeCS			
GPU_TIMESTAMP	CCS	358	2
CS_ENGINE_ID	CCS	8C	1
GFXREG_UNSLICE_FF_CTRL_FLC_THRSHLD1		A288	1
GFXREG_UNSLICE_FF_CTRL_FLC_THRSHLD2		A28C	1
GFXREG_UNSLICE_FF_COUNT1		A538	1
GFXREG_UNSLICE_FF_COUNT2		A53C	1
GFXREG_RPPREVUP		A058	1
GFXREG_RPPREVDN		A064	1
GFXREG_RPUPEI		A068	1
GFXREG_RPDNEI		A06C	1
GFXREG_GT_GFX_RC6		138108	1
GFXREG_GT_GFX_RC6P		13810C	1
CS_CTX_TIMESTAMP	CCS	3A8	1
OAC_PERF_<x>		15000	160
BlitterCS			
GPU_TIMESTAMP		22358	2
CS_ENGINE_ID		2208C	1
CS_CTX_TIMESTAMP	BCS	3A8	1
RP_STATUS0		A01C	1
PERFCNT1_LSB		91B8	1
PERFCNT1_MSB		91BC	1
PERFCNT2_LSB		91C0	1
PERFCNT2_MSB		91C4	1
GFXREG_GT		145040	7
GFXREG_IA		145828	13
GFXREG_IO		145928	24
GFXREG_UNSLICE_FF_CTRL_FLC_THRSHLD1		A288	1
GFXREG_UNSLICE_FF_CTRL_FLC_THRSHLD2		A28C	1

Name	Base Address (default=none)	MMIO Offset (hex)	Size in DW
GFXREG_UNSLICE_FF_COUNT1		A538	1
GFXREG_UNSLICE_FF_COUNT2		A53C	1
GFXREG_RPPREVUP		A058	1
GFXREG_RPPREVDN		A064	1
GFXREG_RPUPEI		A068	1
GFXREG_RPDNEI		A06C	1
VideoCS			
GPU_TIMESTAMP	VCS	358	2
CS_CTX_TIMESTAMP	VCS	3A8	1
PERFCNT1_LSB		91B8	1
PERFCNT1_MSB		91BC	1
PERFCNT2_LSB		91C0	1
PERFCNT2_MSB		91C4	1
GFXREG_UNSLICE_FF_CTRL_FLC_THRSHLD1		A288	1
GFXREG_UNSLICE_FF_CTRL_FLC_THRSHLD2		A28C	1
GFXREG_UNSLICE_FF_COUNT1		A538	1
GFXREG_UNSLICE_FF_COUNT2		A53C	1
GFXREG_RPPREVUP		A058	1
GFXREG_RPPREVDN		A064	1
GFXREG_RPUPEI		A068	1
GFXREG_RPDNEI		A06C	1
CS_ENGINE_ID	VCS	8C	1
VideoEnhancementCS			
GPU_TIMESTAMP	VECS	358	2
CS_CTX_TIMESTAMP	VECS	3A8	1
PERFCNT1_LSB		91B8	1
PERFCNT1_MSB		91BC	1
PERFCNT2_LSB		91C0	1
PERFCNT2_MSB		91C4	1
GFXREG_UNSLICE_FF_CTRL_FLC_THRSHLD1		A288	1
GFXREG_UNSLICE_FF_CTRL_FLC_THRSHLD2		A28C	1
GFXREG_UNSLICE_FF_COUNT1		A538	1
GFXREG_UNSLICE_FF_COUNT2		A53C	1
GFXREG_RPPREVUP		A058	1
GFXREG_RPPREVDN		A064	1
GFXREG_RPUPEI		A068	1
GFXREG_RPDNEI		A06C	1

Name	Base Address (default=none)	MMIO Offset (hex)	Size in DW
GFXREG_GT_GFX_RC6		138108	1
GFXREG_GT_GFX_RC6P		13810C	1
CS_ENGINE_ID	VECS	8C	1

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 Violation 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, Compute CS
PIPE_CONTROL	Still send flush down, Post-Sync Operation is NOOP if	Render CS, ComputeCS

User Mode Privileged Command	Function in Non-Privileged Batch Buffers	Source
	Use Global GTT or Use "Store Data Index" is enabled. Post-Sync Operation LRI to **Privileged Register** is discarded.	
MI_ATOMIC	Command is converted to NOOP if **Use Global GTT** is enabled.	*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.

The tables below also are part of the allowed registers allowed to be read by a non-Privileged (PPGTT) batch buffer. Refer to Read Only User Mode Privilege MMIO Access section for the rest of the allowable registers for read access.

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

MMIO Name	MMIO Offset	Size in DWords
Cache_Mode_0	0x7000	1
Cache_Mode_1	0x7004	1
GT_MODE	0x7008	1
NOPID	0x2094	1
INSTPM	0x20C0	1
IA_VERTICES_COUNT	0x2310	2
IA_PRIMIVTIVES_COUNT	0x2318	2
VS_INVOCATION_COUNT	0x2320	2
HS_INVOCATION_COUNT	0x2300	2
DS_INVOCATION_COUNT	0x2308	2

MMIO Name	MMIO Offset	Size in DWords
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_PRIMITIVES_COUNT	0x2340	2
PS_INVOCATION_COUNT	0x2348	2
PS_DEPTH_COUNT	0x2350	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
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

MMIO Name	MMIO Offset	Size in DWords
GPUGPU_DISPATCHDIMZ	0x2508	1
MI_PREDICATE_SRC0	0x2400	1
MI_PREDICATE_SRC0	0x2404	1
MI_PREDICATE_SRC1	0x2408	1
MI_PREDICATE_SRC1	0x240C	1
MI_PREDICATE_DATA	0x2410	1
MI_PREDICATE_DATA	0x2414	1
MI_PREDICATE_RESULT	0x2418	1
MI_PREDICATE_RESULT_1	0x241C	1
MI_PREDICATE_RESULT_2	0x23BC	1
3DPRIM_END_OFFSET	0x2420	1
3DPRIM_START_VERTEX	0x2430	1
3DPRIM_VERTEX_COUNT	0x2434	1
3DPRIM_INSTANCE_COUNT	0x2438	1
3DPRIM_START_INSTANCE	0x243C	1
3DPRIM_BASE_VERTEX	0x2440	1
3DPRIM_XP0	0x2690	1
3DPRIM_XP1	0x2694	1
3DPRIM_XP2	0x2698	1
GPGPU_THREADS_DISPATCHED	0x2290	2
BB_OFFSET	0x2158	1
CS_GPR (1-16)	0x2600	32
OA_CTX_CONTROL	0x2360	1
OA_CTX_CONTROL_MSG	0x2AA0	1
OACTXID	0x2364	1
OAR_OACONTROL	0x2960	1
OAR_OASTATUS	0x2968	1
PR_CTR_CTL_RCSUNIT	0x2178	1
PR_CTR_THRSH_RCSUNIT	0x217C	1
Deprecated Register	0xE518	1
PTBR_PAGE_POOL_SIZE_REGISTER	0x17520	1
PSS_MODE	0x7038	1
CMD_BUFF_CTL	0x2084	1
Z_DISCARD_EN	0x7040	1
TRTT_CR	0x4400	1
TRTT_VA_RANGE	0x4404	1
TRTT_L3_BASE_LOW	0x4408	1

MMIO Name	MMIO Offset	Size in DWords
TRTT_L3_BASE_HIGH	0x440C	1
TR_NULL_GFX	0x4410	1
TRTT_INVAL	0x4414	1
LSQCREG1	0xB100	1
LSQCREG4	0xB118	1
LSQCREG5	0xB158	1
LSQCREG6	0xB15C	1
L3ALLOCREG	0xB134	1
L3TCCNTLREG	0xB138	1
CS_MI_ADDRESS_OFFSET	0x23B4	1
MI_SET_PREDICATE_RESULT	0x23B8	1
WPARID	0x221C	1
PREDICATION_MASK	0x21FC	1
TASK_INVOCATION_COUNT	0x26E8	2
MESH_INVOCATION_COUNT	0x26E0	2
3DMESH_TG_COUNT	0x26F0	1
3DMESH_STARTING_TGID	0x26F4	1
MESH_PRIMITIVE_COUNT	0x26D8	2

MMIO Offset mentioned against the register are offset from the corresponding Command Streamers MMIO Base Address. Refer to Register Access and User Mode Privileges section for Base address for the below offsets.

User Mode Non-Privileged Registers for Compute Command Streamer (CCS)

MMIO Name	MMIO Offset	Size in DWords
NOPID	0x00094	1
INSTPM	0x000C0	1
GPUGPU_DISPATCHDIMX	0x00500	1
GPUGPU_DISPATCHDIMY	0x00504	1
GPUGPU_DISPATCHDIMZ	0x00508	1
MI_PREDICATE_SRC0	0x00400	1
MI_PREDICATE_SRC0	0x00404	1
MI_PREDICATE_SRC1	0x00408	1
MI_PREDICATE_SRC1	0x0040C	1
MI_PREDICATE_DATA	0x00410	1
MI_PREDICATE_DATA	0x00414	1
MI_PREDICATE_RESULT	0x00418	1

MMIO Name	MMIO Offset	Size in DWords
MI_PREDICATE_RESULT_1	0x0041C	1
MI_PREDICATE_RESULT_2	0x003BC	1
GPGPU_THREADS_DISPATCHED	0x00290	2
BB_OFFSET	0x00158	1
CS_GPR (1-16)	0x00600	32
PR_CTR_CTL_RCSUNIT	0x00178	1
PR_CTR_THRSH_RCSUNIT	0x0017C	1
CMD_BUFF_CTL	0x00084	1
CS_MI_ADDRESS_OFFSET	0x003B4	1
MI_SET_PREDICATE_RESULT	0x003B8	1
WPARID	0x0021C	1
PREDICATION_MASK	0x001FC	1
OA_CTX_CONTROL(CCS)	0x00360	1
OA_CTXID	0x00364	1

** These registers MMIO Offset accessed are same across the ComputeCS's.

MMIO Name	MMIO Offset	Size in DWords
OA_CTX_CONTROL_MSG**	0x151E0	1
OACONTROL_CCS0_OA**	0x15114	1
OASTATUS_CCS0_OA**	0x1511C	1

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

ComputeCS0

MMIO Name	MMIO Offset	Size in DWords
TRTT_CR*	0x4580	1
TRTT_VA_RANGE*	0x4584	1
TRTT_L3_BASE_LOW*	0x4588	1
TRTT_L3_BASE_HIGH*	0x458C	1
TRTT_NULL*	0x4590	1
TRTT_INVAL*	0x4594	1

ComputeCS1

MMIO Name	MMIO Offset	Size in DWords
TRTT_CR*	0x45A0	1
TRTT_VA_RANGE*	0x45A4	1

MMIO Name	MMIO Offset	Size in DWords
TRTT_L3_BASE_LOW*	0x45A8	1
TRTT_L3_BASE_HIGH*	0x45AC	1
TRTT_NULL*	0x45B0	1
TRTT_INVAL*	0x45B4	1

ComputeCS2

MMIO Name	MMIO Offset	Size in DWords
TRTT_CR*	0x45C0	1
TRTT_VA_RANGE*	0x45C4	1
TRTT_L3_BASE_LOW*	0x45C8	1
TRTT_L3_BASE_HIGH*	0x45CC	1
TRTT_NULL*	0x45D0	1
TRTT_INVAL*	0x45D4	1

ComputeCS3

MMIO Name	MMIO Offset	Size in DWords
TRTT_CR*	0x45E0	1
TRTT_VA_RANGE*	0x45E4	1
TRTT_L3_BASE_LOW*	0x45E8	1
TRTT_L3_BASE_HIGH*	0x45EC	1
TRTT_NULL*	0x45F0	1
TRTT_INVAL*	0x45F4	1

User Mode Non-Privileged Registers for Blitter Command Streamer(BCS)

MMIO Name	MMIO Offset	Size in DWords
BCS_GPR	0x22600	32
BCS_SWCTRL	0x22200	1
BLIT_CCTL	0x22204	1
PR_CTR_CTL_BCSUNIT	0x22178	1
PR_CTR_THRSH_BCSUNIT	0x2217C	1
BLT_TRTT_CR	0x4480	1
BLT_TRTT_VA_RANGE	0x4484	1
BLT_TRTT_L3_BASE_LOW	0x4488	1
BLT_TRTT_L3_BASE_HIGH	0x448C	1
BLT_TRTT_NULL	0x4490	1
BLT_TRTT_INV	0x4494	1
NOPID	0x22094	1

MMIO Name	MMIO Offset	Size in DWords
MI_PREDICATE_RESULT_1	0x2241C	1
MI_PREDICATE_RESULT_2	0x223BC	1
INSTPM	0x220C0	1
CS_MI_ADDRESS_OFFSET	0x223B4	1
MI_SET_PREDICATE_RESULT	0x223B8	1
WPARID	0x2221C	1
PREDICATION_MASK	0x221FC	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
NOPID	VECS	0x094	1
MI_PREDICATE_RESULT_1	VECS	0x41C	1
MI_PREDICATE_RESULT_2	VECS	0x3BC	1
INSTPM	VECS	0x0C0	1
CS_MI_ADDRESS_OFFSET	VECS	0x3B4	1
MI_SET_PREDICATE_RESULT	VECS	0x3B8	1
WPARID	VECS	0x21C	1
PREDICATION_MASK	VECS	0x1FC	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
NOPID	VCS	0x094	1
MI_PREDICATE_RESULT_1	VCS	0x41C	1
MI_PREDICATE_RESULT_2	VCS	0x3BC	1
INSTPM	VCS	0x0C0	1

MMIO Name	Unit Base	MMIO Range	Size in DWords
CS_MI_ADDRESS_OFFSET	VCS	0x3B4	1
MI_SET_PREDICATE_RESULT	VCS	0x3B8	1
WPARID	VCS	0x21C	1
PREDICATION_MASK	VCS	0x1FC	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.

VEBOX-0

MMIO Name	MMIO Offset	Size in DWords
TRTT_CR*	0x4460	1
TRTT_VA_RANGE*	0x4464	1
TRTT_L3_BASE_LOW*	0x4468	1
TRTT_L3_BASE_HIGH*	0x446C	1
TRTT_NULL*	0x4470	1
TRTT_INVAL*	0x4474	1

VDBOX-0

MMIO Name	MMIO Offset	Size in DWords
TRTT_CR*	0x4420	1
TRTT_VA_RANGE*	0x4424	1
TRTT_L3_BASE_LOW*	0x4428	1
TRTT_L3_BASE_HIGH*	0x442C	1
TRTT_NULL*	0x4430	1
TRTT_INVAL*	0x4434	1

VDBOX-1

MMIO Name	MMIO Offset	Size in DWords
TRTT_CR*	0x4440	1
TRTT_VA_RANGE*	0x4444	1
TRTT_L3_BASE_LOW*	0x4448	1
TRTT_L3_BASE_HIGH*	0x444C	1
TRTT_NULL*	0x4450	1
TRTT_INVAL*	0x4454	1

VEBOX-1

MMIO Name	MMIO Base	MMIO Offset	Size in DWords
TRTT_CR*	n/a	0x4560	1
TRTT_VA_RANGE*	n/a	0x4564	1
TRTT_L3_BASE_LOW*	n/a	0x4568	1
TRTT_L3_BASE_HIGH*	n/a	0x456C	1
TRTT_NULL*	n/a	0x4570	1
TRTT_INVAL*	n/a	0x4574	1

VDBOX-2

MMIO Name	MMIO Base	MMIO Offset	Size in DWords
TRTT_CR*	n/a	0x4520	1
TRTT_VA_RANGE*	n/a	0x4524	1
TRTT_L3_BASE_LOW*	n/a	0x4528	1
TRTT_L3_BASE_HIGH*	n/a	0x452C	1
TRTT_NULL*	n/a	0x4530	1
TRTT_INVAL*	n/a	0x4534	1

VDBOX-3

MMIO Name	MMIO Base	MMIO Offset	Size in DWords
TRTT_CR*	n/a	0x4540	1
TRTT_VA_RANGE*	n/a	0x4544	1
TRTT_L3_BASE_LOW*	n/a	0x4548	1
TRTT_L3_BASE_HIGH*	n/a	0x454C	1
TRTT_NULL*	n/a	0x4550	1
TRTT_INVAL*	n/a	0x4554	1

Context Management

When the scheduler submits a list of workloads through the execution list, the Command streamer hardware executes one context at a time.

An engine starts executing a context by loading the state (LRCA) in memory that is pointed to by the context descriptor.

The structure of the LRCA is described in subsequent sections.

Global State

There is only one copy of state variables across contexts running on an engine and changing the settings of these variables requires explicit programming of these state variables. Typically, global state variables are programmed only once either at the time of power-on or at the time of GFX driver initialization. Examples of global sate include:

- MI registers (HWSTAM, SEMA_WAIT_POLL etc.)
- Configuration Registers (GFX_MODE etc.)

The global state of an engine is context save/restored during power-off/on regimes.

Following subsections describe the power context images of engines across generations.

Power Context Image

This section lists the power context image of Video Engine, Copy Engine and Video Enhancement Engine across generations.

CSFE Power context without Display

CSFE Power Context Image with Display

Description	Offset	Unit	# of DW	Address Offset (PWR)	CSFE/CSBE
NOOP		CS	1	0000	CSFE
Load_Register_Immediate header	0x1100_00B3	CS	1	0001	CSFE
GFX_MODE	0x029C	CS	2	0002	CSFE
GHWSP	0x0080	CS	2	0004	CSFE
RC_PWRCTX_MAXCNT	0x0054	CS	2	0008	CSFE
CTX_WA_PTR	0x0058	CS	2	000A	CSFE
NOPID	0x0094	CS	2	000C	CSFE
HWSTAM	0x0098	CS	2	000E	CSFE
EIR	0x00B0	CS	2	0012	CSFE
EMR	0x00B4	CS	2	0014	CSFE
CMD_CCTL_0	0x00C4	CS	2	0016	CSFE
PREEMPT_DLY	0x0214	CS	2	0018	CSFE
CTXT_PREMP_DBG	0x0248	CS	2	001A	CSFE
WAIT_FOR_RC6_EXIT	0x00CC	CS	2	001C	CSFE
RCS_CTXID_PREEMPTION_HINT	0x04CC	CS	2	001E	CSFE
CS_PREEMPTION_HINT_UDW	0x04C8	CS	2	0020	CSFE
CS_PREEMPTION_HINT	0x04BC	CS	2	0022	CSFE
CCID Register	0x0180	CS	2	0024	CSFE
MI_PREDICATE_RESULT_2	0x03BC	CS	2	0026	CSFE

Description	Offset	Unit	# of DW	Address Offset (PWR)	CSFE/CSBE
CTXT_ST_PTR	0x03A0	CS	2	0028	CSFE
CTXT_ST_BUF	0x0370	CS	24	002A	CSFE
CTXT_ST_BUF	0x03C0	CS	24	0042	CSFE
SEMA_WAIT_POLL	0x024C	CS	2	005A	CSFE
IDLEDELAY	0x023C	CS	2	005C	CSFE
RCS_FORCE_TO_NONPRIV_0_11	0x04D0	CS	24	005E	CSFE
RCS_FORCE_TO_NONPRIV_12_15	0x010	CS	8	0076	CSFE
RCS_FORCE_TO_NONPRIV_16_19	0x1D0	CS	8	007E	CSFE
EXECLIST_STATUS_REGISTER	0x0234	CS	2	0086	CSFE
CXT_OFFSET	0x01AC	CS	2	008A	CSBE
STOP_PARSER_CONTROL	0x0424	CS	2	008C	CSBE
STOP_PARSER_HINT_ADDR	0x0428	Cs	4	008E	CSBE
EXECLIST_SQ_CONTENTS	0x0510-0x054F	CS	32	0092	CSFE
CSB_INTERRUPT_MASK	0x0218	CS	2	00B2	CSFE
EQ_ELEMENT_MASK	0x056C	CS	2	00B4	CSFE
NOOP		CS	8	00B8	CSFE
NOOP		CS	2	00B8	CSFE

CSFE Power Context with Display

CSFE Power Context Image with Display

Description	Offset	Unit	# of DW	Address Offset (PWR)	CSFE/CSBE
NOOP		CS	1	0000	CSFE
Load_Register_Immediate header	0x1100_00C5	CS	1	0001	CSFE
GFX_MODE	0x029C	CS	2	0002	CSFE
GHWSP	0x0080	CS	2	0004	CSFE
RC_PWRCTX_MAXCNT	0x0054	CS	2	0008	CSFE
CTX_WA_PTR	0x0058	CS	2	000A	CSFE
NOPID	0x0094	CS	2	000C	CSFE
HWSTAM	0x0098	CS	2	000E	CSFE
EIR	0x00B0	CS	2	0012	CSFE
EMR	0x00B4	CS	2	0014	CSFE
CMD_CCTL_0	0x00C4	CS	2	0016	CSFE
PREEMPT_DLY	0x0214	CS	2	0018	CSFE
CTXT_PREMP_DBG	0x0248	CS	2	001A	CSFE
SYNC_FLIP_STATUS	0x02D0	CS	2	001C	CSFE
SYNC_FLIP_STATUS_1	0x02D4	CS	2	001E	CSFE

Description	Offset	Unit	# of DW	Address Offset (PWR)	CSFE/CSBE
SYNC_FLIP_STATUS_2	**0x02EC**	**CS**	**2**	**0020**	**CSFE**
WAIT_FOR_RC6_EXIT	**0x00CC**	**CS**	**2**	**0022**	**CSFE**
RCS_CTXID_PREEMPTION_HINT	**0x04CC**	**CS**	**2**	**0024**	**CSFE**
CS_PREEMPTION_HINT_UDW	**0x04C8**	**CS**	**2**	**0026**	**CSFE**
CS_PREEMPTION_HINT	**0x04BC**	**CS**	**2**	**0028**	**CSFE**
CCID Register	**0x0180**	**CS**	**2**	**002A**	**CSFE**
MI_PREDICATE_RESULT_2	**0x03BC**	**CS**	**2**	**002C**	**CSFE**
CTXT_ST_PTR	**0x03A0**	**CS**	**2**	**002E**	**CSFE**
CTXT_ST_BUF	**0x0370**	**CS**	**24**	**0030**	**CSFE**
CTXT_ST_BUF	**0x03C0**	**CS**	**24**	**0048**	**CSFE**
SEMA_WAIT_POLL	**0x024C**	**CS**	**2**	**0060**	**CSFE**
IDLEDELAY	**0x023C**	**CS**	**2**	**0062**	**CSFE**
DISPLAY MESSAGE FORWARD STATUS	**0x02E8**	**CS**	**2**	**0064**	**CSFE**
RCS_FORCE_TO_NONPRIV_0_11	**0x04D0**	**CS**	**24**	**0066**	**CSFE**
RCS_FORCE_TO_NONPRIV_12_15	**0x010**	**CS**	**8**	**007E**	**CSFE**
RCS_FORCE_TO_NONPRIV_16_19	**0x1D0**	**CS**	**8**	**0086**	**CSFE**
EXECLIST_STATUS_REGISTER	**0x0234**	**CS**	**2**	**008E**	**CSFE**
CXT_OFFSET	**0x01AC**	**CS**	**2**	**0092**	**CSBE**
STOP_PARSER_CONTROL	**0x0424**	**CS**	**2**	**0094**	**CSBE**
STOP_PARSER_HINT_ADDR	**0x0428**	**Cs**	**4**	**0098**	**CSBE**
SYNC_FLIP_STATUS_3	**0x02B8**	**CS**	**2**	**009A**	**CSFE**
SYNC_FLIP_STATUS_4	**0x02C0**	**CS**	**2**	**009C**	**CSFE**
SYNC_FLIP_STATUS_5	**0x02C4**	**CS**	**2**	**009E**	**CSFE**
SYNC_FLIP_STATUS_6	**0x01F8**	**CS**	**2**	**00A0**	**CSFE**
DISPLAY MESSAGE FORWARD STATUS_2	**0x0188**	**CS**	**2**	**00A2**	**CSFE**
DISPLAY MESSAGE FORWARD STATUS_3	**0x018C**	**CS**	**2**	**00A4**	**CSFE**
EXECLIST_SQ_CONTENTS	**0x0510-0x054F**	**CS**	**32**	**00A6**	**CSFE**
CSB_INTERRUPT_MASK	**0x0218**	**CS**	**2**	**00C6**	**CSFE**
EQ_ELEMENT_MASK	**0x056C**	**CS**	**2**	**00C8**	**CSFE**
NOOP		**CS**	**4**	**00D0**	**CSFE**
NOOP		CS	2	00D2	CSFE

CSFE Power Context

CSFE Power Context Image

Description	Offset	Unit	# of DW	Address Offset (PWR)	CSFE/CSBE
NOOP		CS	1	0	CSFE
Load_Register_Immediate header	0x1100_00DB	CS	1	001	CSFE
GFX_MODE	0x029C	CS	2	0002	CSFE
GHWSP	0x0080	CS	2	0004	CSFE
RING_BUFFER_CONTROL (Ring Always Disabled)	0x003C	CS	2	0006	CSFE
Ring Head Pointer Register	0x0034	CS	2	0008	CSFE
Ring Tail Pointer Register	0x0030	CS	2	000A	CSFE
RING_BUFFER_START	0x0038	CS	2	000C	CSFE
RING_BUFFER_CONTROL (Original status)	0x003C	CS	2	000E	CSFE
Batch Buffer Current Head Register (UDW)	0x0168	CS	2	0010	CSFE
Batch Buffer Current Head Register	0x0140	CS	2	0012	CSFE
Batch Buffer State Register	0x0110	CS	2	0014	CSFE
SECOND_BB_ADDR_UDW	0x011C	CS	2	0016	CSFE
SECOND_BB_ADDR	0x0114	CS	2	0018	CSFE
SECOND_BB_STATE	0x0118	CS	2	001A	CSFE
RC_PWRCTX_MAXCNT	0x0054	CS	2	001E	CSFE
CTX_WA_PTR	0x0058	CS	2	0020	CSFE
NOPID	0x0094	CS	2	0022	CSFE
HWSTAM	0x0098	CS	2	0024	CSFE
IMR	0x00A8	CS	2	0026	CSFE
EIR	0x00B0	CS	2	0028	CSFE
EMR	0x00B4	CS	2	002A	CSFE
CMD_CCTL_0	0x00C4	CS	2	002C	CSFE
UHPTR	0x0134	CS	2	002E	CSFE
BB_PREEMPT_ADDR_UDW	0x016C	CS	2	0030	CSFE
BB_PREEMPT_ADDR	0x0148	CS	2	0032	CSFE
RING_BUFFER_HEAD_PREEMPT_REG	0x014C	CS	2	0034	CSFE
PREEMPT_DLY	0x0214	CS	2	0036	CSFE
CTXT_PREMP_DBG	0x0248	CS	2	0038	CSFE
SYNC_FLIP_STATUS	0x02D0	CS	2	003A	CSFE
SYNC_FLIP_STATUS_1	0x02D4	CS	2	003C	CSFE

Description	Offset	Unit	# of DW	Address Offset (PWR)	CSFE/CSBE
SYNC_FLIP_STATUS_2	0x02EC	CS	2	003E	CSFE
WAIT_FOR_RC6_EXIT	0x00CC	CS	2	0040	CSFE
RCS_CTXID_PREEMPTION_HINT	0x04CC	CS	2	0042	CSFE
CS_PREEMPTION_HINT_UDW	0x04C8	CS	2	0044	CSFE
CS_PREEMPTION_HINT	0x04BC	CS	2	0046	CSFE
CCID Register	0x0180	CS	2	0048	CSFE
SBB_PREEMPT_ADDRESS_UDW	0x0138	CS	2	004A	CSFE
SBB_PREEMPT_ADDRESS	0x013C	CS	2	004C	CSFE
MI_PREDICATE_RESULT_2	0x03BC	CS	2	004E	CSFE
CTXT_ST_PTR	0x03A0	CS	2	0050	CSFE
CTXT_ST_BUF	0x0370	CS	24	0052	CSFE
CTXT_ST_BUF	0x03C0	CS	24	006A	CSFE
SEMA_WAIT_POLL	0x024C	CS	2	0082	CSFE
IDLEDELAY	0x023C	CS	2	0084	CSFE
DISPLAY MESSAGE FORWARD STATUS	0x02E8	CS	2	0086	CSFE
RCS_FORCE_TO_NONPRIV	0x04D0	CS	24	0088	CSFE
EXECLIST_STATUS_REGISTER	0x0234	CS	2	00A0	CSFE
CXT_OFFSET	0x01AC	CS	2	00A4	CSBE
STOP_PARSER_CONTROL	0x0424	CS	2	00A6	CSBE
STOP_PARSER_HINT_ADDR	0x0428	Cs	4	00A8	CSBE
SYNC_FLIP_STATUS_3	0x02B8	CS	2	00AC	CSFE
SYNC_FLIP_STATUS_4	0x02C0	CS	2	00AE	CSFE
SYNC_FLIP_STATUS_5	0x02C4	CS	2	00B0	CSFE
SYNC_FLIP_STATUS_6	0x01F8	CS	2	00B2	CSFE
DISPLAY MESSAGE FORWARD STATUS_2	0x0188	CS	2	00B4	CSFE
DISPLAY MESSAGE FORWARD STATUS_3	0x018C	CS	2	00B6	CSFE
EXECLIST_SQ_CONTENTS	0x0510-0x054F	CS	32	00B8	CSFE
CSB_INTERRUPT_MASK	0x0218	CS	2	00D8	CSFE
EQ_ELEMENT_MASK	0x056C	CS	2	00DA	CSFE

Context State

Context state is associated with a specific context. Context state can be programmed through Command Stream only when the associated context is being actively executed in the engine.

Context state is save/restored through Logical Context.

Logical Contexts

A logical context is an area in memory used to store hardware context state information and the context is referenced via a context descriptor. Context descriptor carries graphics memory address. Logical Context is always in global virtual memory. GFX device provides means to save and restore the hardware context state to logical context. Context state save/restore mechanism is used by SW to avoid re-programming the HW state across context switches for a given context.

CSFE Execlist Context

This section details the CSFE Execlist Context which is the common layout referred to as part of the VDBOX, Copy Engine and Video Enhancement context images.

CSFE Execlist Context

Programming Note	
Context:	MMIO Offset information
MMIO offset mentioned for the registers in the below table are offset form the units "MMIO Base Offset" mentioned in the table " Base Offset for Video Command Streamers and Media Engine" in the section User Mode Privileged Commands. For Example: VECS has MMIO Base Offset as "0x1C_8000". In the below table "Context Control" register has 0x00244 as offset against it, actual MMIO Offset of "Context Control" register for VECS is 0xx1C_8244. Blitter Engine MMIO base offset must be considered as 0x2_2000.	

EXECLIST CONTEXT
EXECLIST CONTEXT(PPGTT Base)
ENGINE CONTEXT

Description	MMIO Offset/Command	Unit	# of DW	Offset
NOOP		CSEL	1	**0**
Load_Register_Immediate header	0x1108_1019	CSEL	1	**0001**
Load_Register_Immediate header	0x1108_101D	CSEL	1	**0001**
Context Control	0x00244	CSEL	2	**0002**
Ring Head Pointer Register	0x00034	CSEL	2	**0004**
Ring Tail Pointer Register	0x00030	CSEL	2	**0006**
RING_BUFFER_START	0x00038	CSEL	2	**0008**
RING_BUFFER_CONTROL	0x0003C	CSEL	2	**000A**
Batch Buffer Current Head Register (UDW)	0x00168	CSEL	2	**000C**
Batch Buffer Current Head Register	0x00140	CSEL	2	**000E**
Batch Buffer State Register	0x00110	CSEL	2	**0010**
BB_PER_CTX_PTR	0x001C0	CSEL	2	**0012**
CS_INDIRECT_CTX	0x001C4	CSEL	2	**0014**
CS_INDIRECT_CTX_OFFSET	0x001C8	CSEL	2	**0016**
CCID	0x00180	CSEL	2	**0018**
SEMAPHORE_TOKEN	0x002B4	CSEL	2	**001A**
PRT_BB_STATE	0x00120	CSEL	4	
NOOP		CSEL	4	**001C**
NOOP		CSEL	1	**0020**
Load_Register_Immediate header	0x1108_1011	CSEL	1	**0021**
CTX_TIMESTAMP	0x003A8	CSEL	2	**0022**
PDP3_UDW	0x0028C	CSEL	2	**0024**
PDP3_LDW	0x00288	CSEL	2	**0026**
PDP2_UDW	0x00284	CSEL	2	**0028**
PDP2_LDW	0x00280	CSEL	2	**002A**
PDP1_UDW	0x0027C	CSEL	2	**002C**
PDP1_LDW	0x00278	CSEL	2	**002E**
PDP0_UDW	0x00274	CSEL	2	**0030**
PDP0_LDW	0x00270	CSEL	2	**0032**
NOOP		CSEL	4	**0034**
NOOP		CSEL	8	**0038**
NOOP	{EXISTS IF (VCS, VECS)}	CSEL_BE	16	**0040**

Description	MMIO Offset/Command	Unit	# of DW	Offset
NOOP	{EXISTS IF (BCS)}	CSEL_BE	1	**0040**
Load_Register_Immediate header	0x1100_1003 {EXISTS IF (BCS)}	CSEL_BE	1	**0041**
BCS_SWCTRL	Base + 0x200 {EXISTS IF (BCS)}	CSEL_BE	2	**0042**
BLIT_CCTL	Base + 0x204 {EXISTS IF (BCS)}	CSEL_BE	2	**0044**
NOOP	{EXISTS IF (BCS)}	CSEL_BE	10	**0046**
MI_LOAD_REGISTER_MEM (INT_MASK_ENABLE from DWORD GFX_ADDR)	0x000A8	CSEL	4	**0050**
NOOP		CSEL	1	**0054**
MI_LOAD_REGISTER_IMM	0x1108_0001	CSEL	1	**0055**
INT_STATUS_REPORT_PTR	0x000AC	CSEL	2	**0056**
INT_SRC_REPORT_PTR	0x000A4	CSEL	2	**0058**
NOOP		CSEL	6	**005B**
NOOP		CSFE	1	**0060**
Load_Register_Immediate header	0x1108_106D	CSFE	1	
BB_STACK_WRITE_PORT	0x00588	CSFE	12	**0062**
EXCC	0x00028	CSFE	2	**006E**
MI_MODE	0x0009C	CSFE	2	**0070**
INSTPM	0x000C0	CSFE	2	**0072**
PR_CTR_CTL	0x00178	CSFE	2	**0074**
PR_CTR_THRSH	0x0017C	CSFE	2	**0076**
TIMESTAMP Register (LSB)	0x00358	CSFE	2	**0078**
BB_START_ADDR_UDW	0x00170	CSFE	2	**007A**
BB_START_ADDR	0x00150	CSFE	2	**007C**
BB_ADD_DIFF	0x00154	CSFE	2	**007E**
BB_OFFSET	0x00158	CSFE	2	**0080**
MI_PREDICATE_RESULT_1	0x0041C	CSFE	2	**0082**
CS_GPR (1-16)	0x00600	CSFE	64	**0084**
IPEHR	0x00068	CSFE	2	**00C4**
CMD_BUF_CTL (Dummy - To match RCS)	0x00084	CSFE	2	
CS_MI_ADDRESS_OFFSET	0x003B4	CSFE	2	
MI_SET_PREDICATE_RESULT	0x003B8	CSFE	2	
WPARID	0x0021C	CSFE	2	
PREDICATION_MASK	0x001FC	CSFE	2	
MI_FORCE_WAKEUP	{EXISTS IF (VCS, VECS)}	CSFE	2	

Description	MMIO Offset/Command	Unit	# of DW	Offset
NOOP	{EXISTS IF (BCS)}	CSFE	2	
NOOP		CSFE	6	
AUX_TTRTT_BASE_ADDRESS_SECTION (Separate section for each engine mentioned below)		CSFE	18	
NOOP*		CSFE	2	**00DA**
NOOP		CSFE	4	**00DC**

Video Enhancement Engine: AUX_TRTT_BASE_ADDRESS_SECTION

Description	MMIO Offset/Command	Unit	# of DW	Offset
NOOP		CSFE	1	**00B6**
Load_Register_Immediate header	0x1102_100B	CSFE	1	**00B7**
TRTT_CR	0x4460	CSFE	2	**00BC**
TRTT_VA_RANGE	0x4464	CSFE	2	**00BE**
TRTT_L3_BASE_LOW	0x4468	CSFE	2	**00C0**
TRTT_L3_BASE_HIGH	0x446C	CSFE	2	**00C2**
TRTT_NULL	0x4470	CSFE	2	**00C4**
TRTT_INVAL	0x4474	CSFE	2	**00C6**
NOOP		CSFE	4	**00B8**

Video Decode Engine:AUX_TRTT_BASE_ADDRESS_SECTION

Description	MMIO Offset/Command	Unit	# of DW	Offset
NOOP		CSFE	1	**00B6**
Load_Register_Immediate header	0x1102_100B	CSFE	1	**00B7**
TRTT_CR	0x4420	CSFE	2	**00BC**
TRTT_VA_RANGE	0x4424	CSFE	2	**00BE**
TRTT_L3_BASE_LOW	0x4428	CSFE	2	**00C0**
TRTT_L3_BASE_HIGH	0x442C	CSFE	2	**00C2**
TRTT_NULL	0x4430	CSFE	2	**00C4**
TRTT_INVAL	0x4434	CSFE	2	**00C6**
NOOP		CSFE	4	**00B8**

Copy Engine: AUX_TRTT_BASE_ADDRESS_SECTION

Description	MMIO Offset/Command	Unit	# of DW	Offset
NOOP		CSFE	1	**00B6**
Load_Register_Immediate header	0x1102_100B	CSFE	1	**00B7**
TRTT_CR	0x4480	CSFE	2	**00B8**
TRTT_VA_RANGE	0x4484	CSFE	2	**00BA**
TRTT_L3_BASE_LOW	0x4488	CSFE	2	**00BC**
TRTT_L3_BASE_HIGH	0x448C	CSFE	2	**00BE**
TRTT_NULL	0x4490	CSFE	2	**00C0**
TRTT_INVAL	0x4494	CSFE	2	**00C2**
NOOP		CSFE	2	**00C4**
NOOP		CSFE	2	**00C6**

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
MI_MATH (STOREIND)	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
MI_MATH (FENCE_WR)

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. 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 coherency.
MI_CONDITIONAL_BATCH_BUFFER_END	SW must ensure the data coherency.
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.
MI_COPY_MEM_MEM	
MI_MATH (LOADIND)	SW must ensure the data coherency.

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

The 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)

Command Fetch

Command parser implements a DMA engine to fetch the command data from memory. DMA engine pre-fetches eight cacheline worth of command data into its storage and keeps it ready to be executed, it keeps fetching command data as and when space is available in the storage.

Advanced Command Prefetch

Advanced command pre-fetch is an enhancement to the existing DMA engine to addresses its limitation of not being to stream pre-fetches on instructions causing jumps (Ex: Batch Buffer Start and Batch Buffer End). Advanced command pre-fetch is enabled by default; however, software is given flexibility to enable or disable advanced command pre-fetch at its convenience by following means:

- Global state through MMIO (GFX_MODE) which is part of power context. This MMIO bit must be enabled or disabled only through CPU path and must be done when there is no context active in hardware.
- Inline to the command sequence through MI_ARB_CHK, this state is maintained per context. This mechanism must be used by SW to disable pre-fetch around self-modification-code or around selective command sequences of interest.

In order for pre-fetch functionality to be enabled both global and per context state should be set to pre-fetch enable.

Self-Modifying Code

Self-modifying code (SMC) in context to command parser refers to a scenario where in a command sequence executed by the command parser modifies the upcoming commands to be executed by command parser. DMA pre-fetch of command data introduces certain programming restriction on placement of the SMC in the command sequence.

- The modifying commands and the modified commands should be far apart by the number of cachelines fetched by the CS for latency hiding (See Max Command FIFO Depth below) Or
- The modifying commands and the modified command must be executed after a batch buffer start (chained or nested)

Advanced command preparser adds additional limitation to the programming of self-modifying code. Software must explicitly disable the preparser logic before programming a batch buffer whose contents has been modified by the earlier programmed command sequence (self-modifying code). Preparser logic must be disabled using MI_ARB_CHECK command prior to programing the MI_BATCH_BUFFER_START command and pre-fetch logic must be enabled using MI_ARB_CHECK as the first command inside the batch buffer.

The depth depends on the parametrization within each command streamer as each have different requirements for latency hiding. Below is a table of the maximum size supported. Refer to the MI_BATCH_BUFFER_START for actual depths per command streamer.

Max Command FIFO Depth(64B)
16