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 13: SW/HW System Interface

March 2023, Revision 1.0

Notices and Disclaimers

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Code names are used by Intel to identify products, technologies, or services that are in development and not publicly available. These are not "commercial" names and not intended to function as trademarks

Customer is responsible for safety of the overall system, including compliance with applicable safety-related requirements or standards.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document, with the sole exceptions that a) you may publish an unmodified copy and b) code included in this document is licensed subject to Zero-Clause BSD open source license (0BSD). You may create software implementations based on this document and in compliance with the foregoing that are intended to execute on the Intel product(s) referenced in this document. No rights are granted to create modifications or derivatives of this document.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined". Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

Table of Contents

SW/HW System Interface

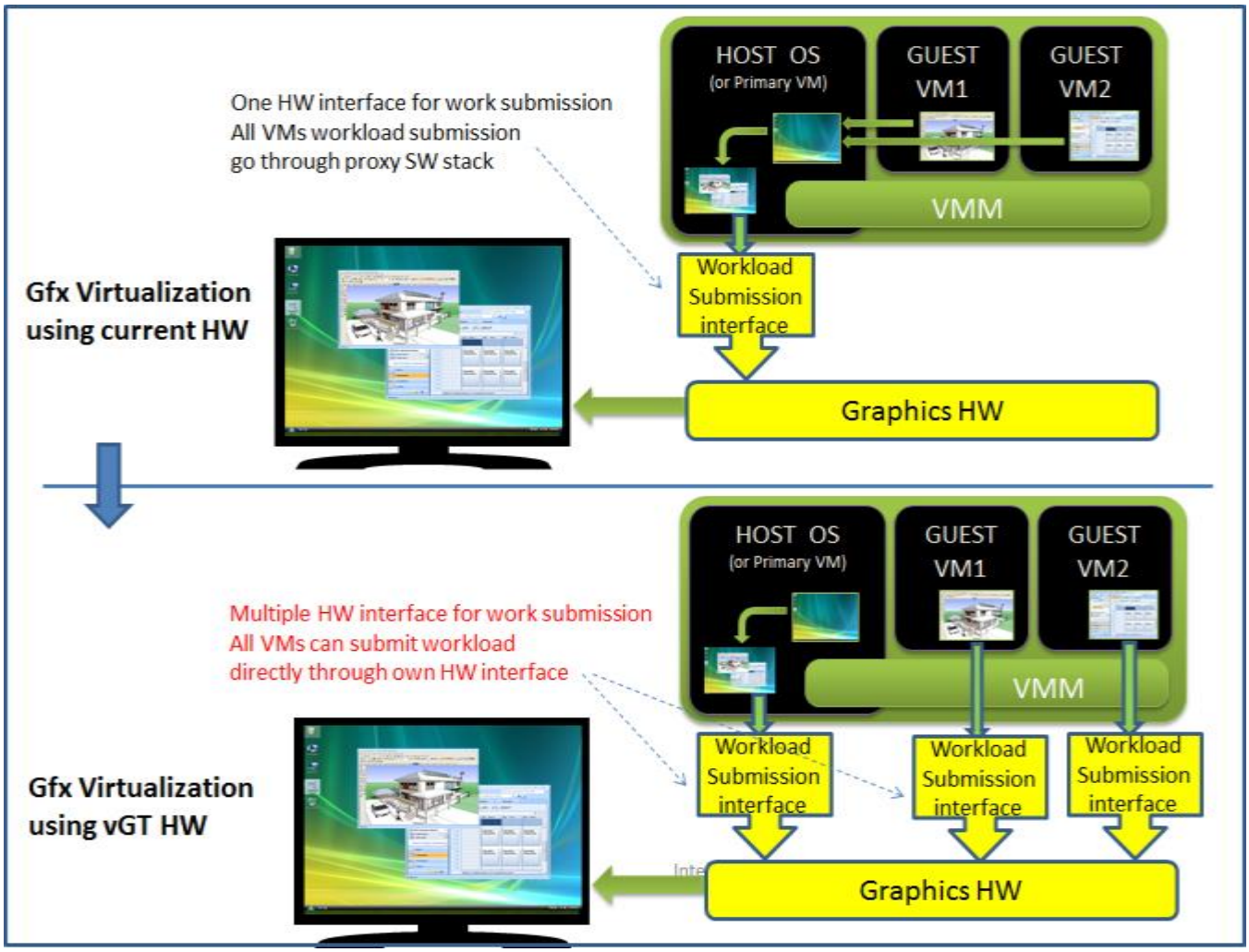
HW Graphics Virtualization

Graphics virtualization allows multiple Virtual Machines (VMs) to access high-quality, high-performance graphics, with minimal software overhead. The graphics virtualization feature adds hardware and firmware to improve performance and enable VMMs to support Intel(R) HD Graphics, Iris0x2122 graphics, and Iris0x2122 Pro graphics in a standard way, eliminating special requirements that could be barriers to adoption.

Introduction

Intel platforms have supported Virtualization Technology for Directed I/O (VT-d) since 2007. In the original VT-d model, graphics is exposed as a single device that can be assigned to only one VM, thus limiting how workloads are submitted to hardware. Such limitations can degrade performance for all VMs, except the one that owns the graphics hardware.

The figure below shows the new concept of graphics virtualization, where each VM can access fully accelerated graphics capabilities.

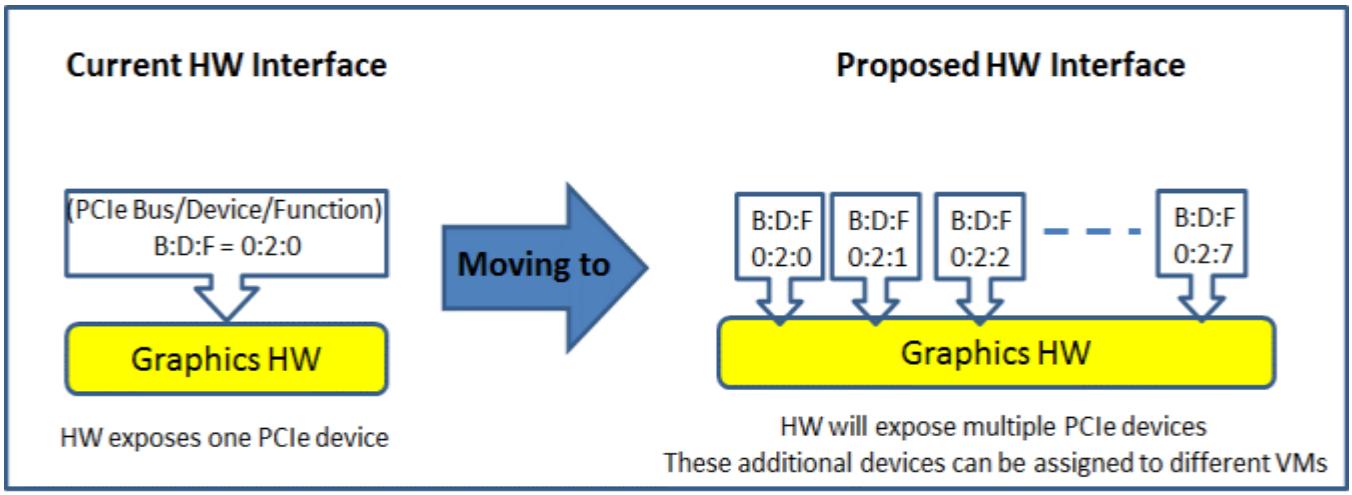


Fully Virtualized Graphics - Conceptual View

The capability depicted above requires the following infrastructure:

- Graphics hardware that exposes multiple hardware interfaces, for assignment to different VMs;
- Resources to submit the graphics workload:
    - Memory (aperture) allowing each VM to submit workload and associated data surfaces;
    - A graphics translation table (GTT) to manage aperture pages;
    - Some MMIO registers exposed to each VM;
- A signaling mechanism to invoke the hardware to execute the workload;
- A signaling mechanism for the hardware to convey relevant information by sending interrupts;
- A mechanism to display the output produced by the execution of the workload.

The figure below shows the proposed hardware interface used to allow multiple software stacks to each get their own "graphics device."



Hardware Interface

Display Output from a Virtualized Environment

There are several possible ways to display content from a virtualized graphics environment:

- Rely on the virtual machine monitor (VMM) or PF driver to composite rendered surfaces from different VMs;
- Assign a display engine exclusively to a virtual function (VF), preferably a trusted VF from the VMM perspective, and use the corresponding planes/pipes to output data;
- Assign a display pipe or plane to a VF.

For In-Vehicle Infotainment (IVI), some applications may require keeping keep the VMM very light weight, and letting a trusted VM manage the display (including composition). In such a scenario, the main dashboard might be owned by the trusted VM, while ancillary displays would be managed by a different VM.

Exposing the Virtualization Capable Hardware

The graphics virtualization infrastructure is exposed to system software by the Single Root I/O virtualization standard (SR-IOV), of the PCIe standard. The exposure is accomplished using a PCIe device Physical function that includes the SR-IOV Extended Capability structure, within the PCIe Extended Capabilities list.

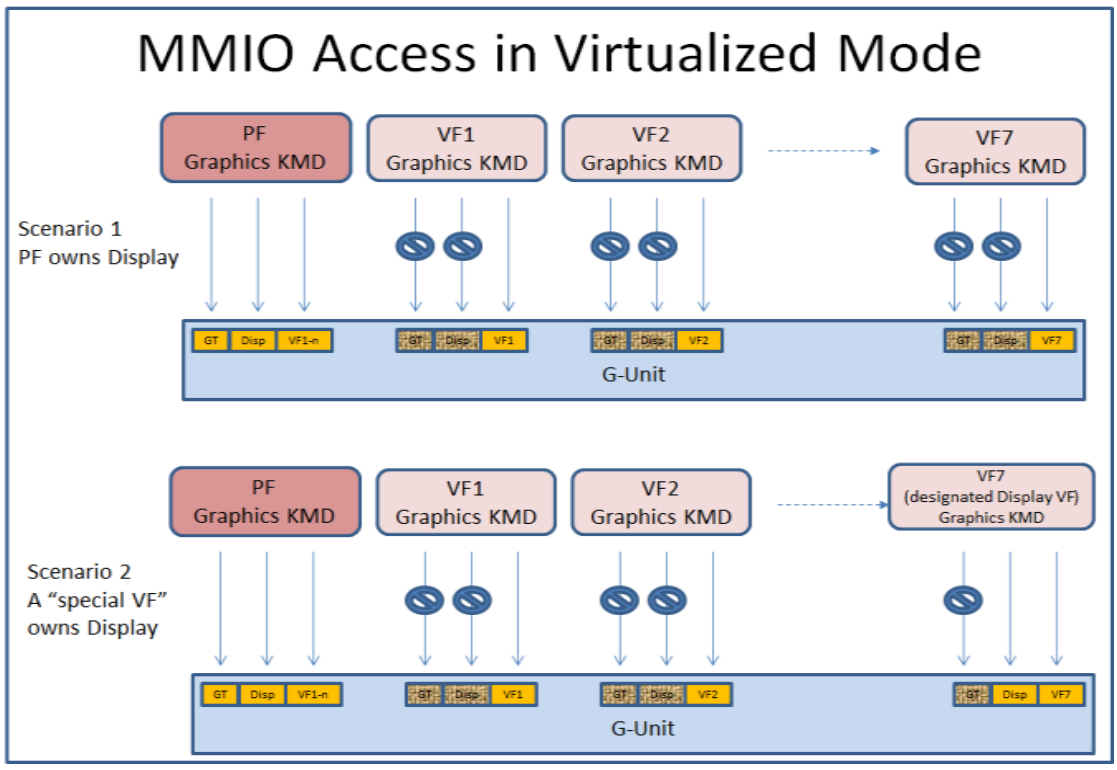
MMIO Space

The MMIO register space allocates 8MB for graphics hardware, but currently only a subset of that allocation is used for MMIO. All GT, Media, and Display MMIO registers are allocated within the lower 3MB of the MMIO range, so each VF needs a minimum of 4MB of MMIO space, after rounding up to the next power of two. To simplify design and validation, and provide additional MMIO space in the future, both VF and PF are allocated 8MB of MMIO register space each.

From a global perspective, only the PF graphics driver needs to access most MMIO registers. From a security viewpoint, the VF driver and software stack are less trusted than PF, so VF does not have access to the full graphics device MMIO range. However, the VM controlling a VF needs to access a small subset of MMIO registers, such as those used to communicate with GuC, and create a VMM-independent PF to VF communication channel. The registers are replicated within the MMIO space of each VF, and only those registers can be accessed by the VM via the VF MMIO.

Limiting VF access to hardware registers requires closing two access paths: direct CPU access and access via a graphics engine. This section covers restricting the CPU access path; the Privilege section covers restricting the graphics engines path.

The figure below shows how the various stacks view the MMIO space.



MMIO Access in Virtualized Mode

As with the PF, VF hardware allocates the MMIO register space and the Global GTT in the same PCI BAR (GTTMMADR), which is 16MB per VF.

Global Address space and Graphics Aperture

The graphics hardware currently supports graphics aperture, and global graphics address space, up to 4GB. When SRIOV is enabled, the global address space is 4GB and is shared among all Functions. System software running in a VM submits commands and data required for a workload through the graphics aperture. When the VMM assigns a VF to a VM, the VMM exposes the VF's graphics aperture range (VF GMADR) and GGTT range (VF GTTMMADR), so the VF graphics driver can access and manage the global address space assigned to it by the Host KMD:

- The Host graphics driver (PF KMD) manages the allocation of global address pages to Physical and Virtual Functions
- A field in the Global GTT contains the number of the Function to which that page has been assigned, and that page may only be accessed by that VF, or the PF
- The Guest graphics driver (VF KMD) can modify the address mapping (Global->GPA) for global address pages that have already been assigned to that VF by the Host KMD (Function field matches the VF)
- Graphics aperture accesses require two translations - using the VF-GGTT to translate from Graphics address to GPA, then using the VF's BDF to translate from GPA to HPA.
- When the Host accesses a global page via the PF, VTd translation will be based on the Function Number stored in the GGTT entry.

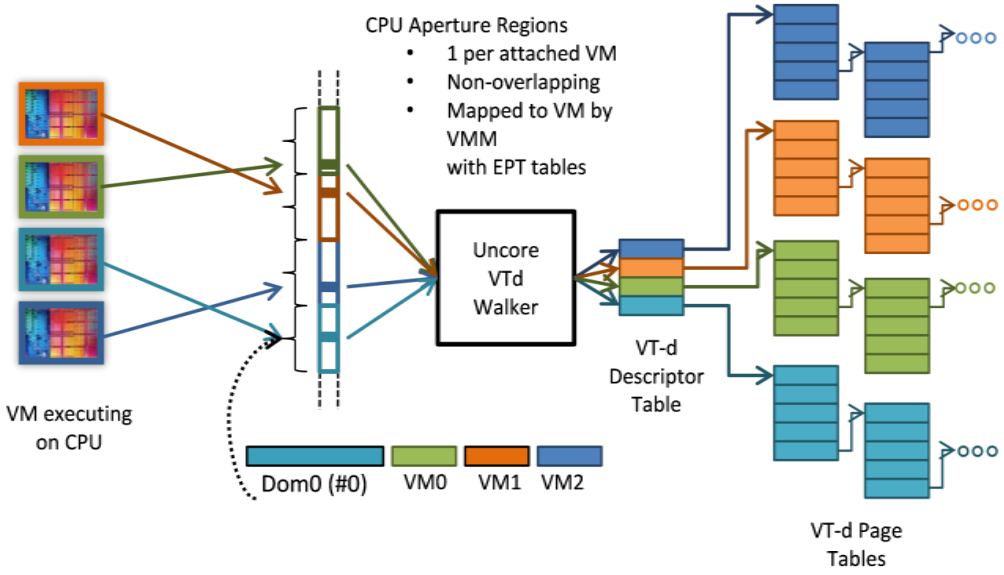
A Guest Kernel Mode Driver can update GGTT Entries only for pages that have been assigned to that Guest, as determined by the Function Number field in the GGTT Entry. If the Function Number in the entry matches the Function Number of the VF assigned to the Guest, then the access is allowed, otherwise it is not. The exact behavior for matching and non-matching function numbers is described below.

VF GTTMMADR Accesses

Read w/Matching VF	Read w/Mismatch VF	Write with Matching VF	Write w/Mismatch VF
Returns all fields except Function Number[7:2], which is returned as 0	Returns all 0's	Updates all fields except Function Number[7:2] and Valid[0]	No Updates

VF Aperture Operations

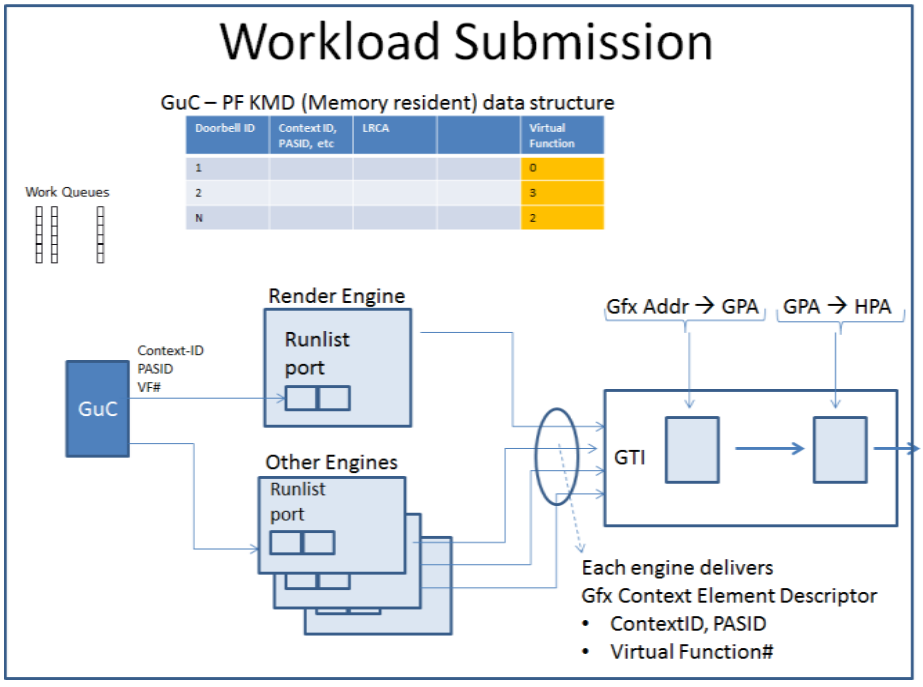
Graphics hardware supports the ability to determine the BDF used for aperture accesses, which may happen simultaneously (as shown in the following figure). The BDF to be associated with an aperture access is based on the per-VF BAR range.



VT-d Translation of Aperture Accesses

VM Workload Scheduling

Allow PFs to submit workloads: for example, to do compositing or other operations that require using data from different VMs. The figure below shows the basic conceptual model for workload submission.



Workload Submission

Memory-based doorbell queues are the primary mechanism for scheduling workloads on the graphics hardware. The PF KMD assigns some doorbells to each VM for requesting execution. Within a VM, the graphics driver allocates available doorbells to the entities that submit work. For example, the PF KMD may have 256 doorbells available and assign 32 to a VM, then the VM graphics driver may keep 16 doorbells for itself and grants 16 for direct submission by Ring3 (UMD) applications.

The graphics KMD and GuC have a shared data structure that stores workload-related characteristics (ContextID, PASID, Doorbell#, LRCA location, etc). The table for each workload queue also adds the VF number. When a doorbell ring results in a workload submittal, the GuC also submits a VF number from the queue table into the execlist port. The supplied VF number is added to the context attribute and relayed to GTI. In virtualized mode, GTI performs a nested translation: the first-level translation is based on PASID (advanced mode) or PPGTT (legacy mode); the second level translation is a standard VT-d translation based on the BDF, where "F" comes from the VF number.

VF Workload Execution in an Engine

Gang-scheduling all engines would be inefficient, so GuC supports the ability to schedule a workload from any VF to any engine. Hence each engine can be executing context on behalf of different VFs. Each engine delivers the PASID, the directory pointer for first-level translation, and BDF with VF for second-level translation. On a context switch, TLBs are flushed to avoid cross-domain access from stale TLBs.

Managing Privilege Levels

The hardware enforces privilege separation between Ring3 (UMD) and Ring0 (KMD) commands as follows:

- Only KMD is allowed to issue certain commands that are considered privileged.
- Only KMD commands are allowed to update the majority of graphics registers that are considered privileged.
- UMD-issued commands are allowed to access and update a small set of registers that are considered non-privileged.

This section describes how virtualization handles privileged commands and registers.

In general, KMD may create privileged command buffers and UMD creates non-privileged command buffers. From a hardware perspective, command buffers using the Global GTT are at a higher privilege level, while command buffers using Per Process GTT are non-privileged.

The MI_BATCH_BUFFER command contains an Address Space Indicator field that can determine whether the batch buffer uses the GGTT space or uses the PP GTT space. When a batch buffer command executes another batch buffer, the address space of the "child batch buffer" is determined by combining the Address Space Indicator fields of the parent batch buffer and child batch buffer. The privilege of a chained or second-level batch buffer is either equal to, or lower than, the privilege of the parent (e.g. a child cannot be GGTT if the parent is PP GTT).

The following table shows how privileged commands are handled in both virtualized and non-virtualized scenarios. In the non-virtualized case, a privileged command or access to GGTT in a non-privileged buffer

results in a no-op or modification to PPGTT access. In the virtualized scenario, the KMD in the VM can insert commands that use the VF GGTT, so commands that use GGTT can go through, with the restriction that they use the VF's GGTT.

Privileged Commands in Virtualized and Non-Virtualized Scenarios

Category	Command	Behavior in Non-privileged BB	Behavior in VF with privileged BB/Ring Buffer
GTT Update	MI_UPDATE_GTT	No-op	Allow only for PF
Display	MI_DISPLAY_FLIP	No-op	Allow only for PF
	MI_LOAD_SCAN_LINES_INCL/EXCL	Non-privileged instruction, Allowed	Allow only for PF
	MI_WAIT_FOR_EVENT	Non-privileged instruction, Allowed	Allow only for PF
Register Update	MI_LOAD_REGISTER_IMM	No-op if target is a Privileged register	Continue UMD static whitelist and VF KMD programmed whitelist. (Even though OACONTROL (0x2B00) register is part of the UMD static whitelist, this register is privileged and only allowed by PF KMD and will not be accessible by VF KMD or UMD.) VF KMD can access some registers using an extended programmable whitelist infrastructure. Allow accesses within engine MMIO (e.g. RenderCS can write to allowed registers within the render engine). Allow register accesses to GAM/GAMT to enable context save/restore
	MI_LOAD_REGISTER_REG	No-op if target is a Privileged register	
	MI_LOAD_REGISTER_MEM	No-op if GGTT is used or the target is a Privileged register	
Memory write	MI_STORE_DATA_INDEX	No-op	PF allowed to access Global or Per-Process Hardware Status Page. VF allowed only to access Per-Process Hardware Status Page.
	MI_STORE_REGISTER_MEM	No-op if GGTT is used	Allow - target memory is always within VF
	MI_STORE_DATA_IMM	No-op if GGTT is used	Allow to VF GGTT or PPGTT

Category	Command	Behavior in Non-privileged BB	Behavior in VF with privileged BB/Ring Buffer
	MI_RS_STORE_DATA_IMM	No-op if GGTT is used	Allow to VF GGTT or PPGTT
Sync	MI_SEMAPHOR_MBOX, MI_SEMPHORE_ SIGNAL,/ WAIT	No-op if GGTT is used	Not privileged. Semaphore address may be in GGTT or PP GTT
	PIPE_CONTROL	Send Flush down. Post Sync is No-op if GGTT or use store data idx. Post sync LRI to privileged register is discarded.	Allow - notify enable INT goes to GuC. Post sync op for register write is subjected to check (like MI_LOAD_REGISTER_IMM) PF allowed to access Global or Per-Process Hardware Status Page for "Post-Sync Operation" with "Store Data Index". VF allowed only to access Per-Process Hardware Status Page when "Post-Sync Operation" is with "Store Data Index".
	MI_CLFLUSH	Allow	Allow
	MI_FLUSH_DW	No-op if GGTT or use Store data idx is enabled	Allow to VF GGTT or PPGTT PF allowed to access Global or Per-Process Hardware Status Page for "Post-Sync Operation" with "Store Data Index". VF allowed only to access Per-Process Hardware Status Page when "Post-Sync Operation" is with "Store Data Index".
	MI_ARB_CHECK	Allow	Allow
	MI_ARB_ON_OFF	No-op	Disallow - privileged
	MI_ATOMIC	No-op if GGTT	Allow - VF GGTT or PP GTT
Commands	MI_BATCH_BUFFER_START	Priv < = Parent Privilege	Allow - not privileged
	MI_CONDITIONAL_BATCH_BUFFER_END	No-op if GGTT	Allow - not privileged
	MI_USER_INTERRUPT	Allow	Allow - GuC gets interrupt and notifies VF
	MI_NOOP	Allow	Allow
	MI_COPY_MEM_MEM	No-op if src or dest addr is	Allow

Category	Command	Behavior in Non-privileged BB	Behavior in VF with privileged BB/Ring Buffer
		GGTT	
	CRYPTO_INLINE_STATUS_READ	No-op if GGTT	Allow - status can be within VF GGTT or PPGTT
Performance	MI_REPORT_PERF_COUNT	No-op if GGTT	No-op if GGTT and access are from VF. VF are only allowed with PPGTT memory type. GGTT access are only allowed from PF ring buffer or PF privilged batch buffer.
Legacy	MI_SET_CONTEXT	No-op	Allow within VF ring buffer, but not in non-privileged BB
GHWSP Access - (Global Hadware Status page-setup through HWS_PGA MMIO register)	Updates to Context Status Buffer on a context switch	-N.A-	Context Swtich Status update will only happen on a PF context switch. Context Switch status updates will not happen on a VF context switch.
	Interrupt status dword write to Hardware Status Page on an interrupt when enabled through HWSTAM (Hardwre status mask register).	-N.A-	SW must not unmask HWSTAM for any interrupts in virtualized mode of operation.
	Index writes in to GHWSP through MI_STORE_DATA_INDEX, PIPE_CONTROL and MI_FLUHS_DW	No-op if GGTT	No-op if access from VF. PF allowed to access.

Access to device registers is by either of two methods: directly from the host CPU, or from a graphics engine using an MI_LOAD_REGISTER_* command. As described for the MMIO Space, the Host path allows VF software limited access to only a subset of device registers. This section describes details for limiting register access through an engine.

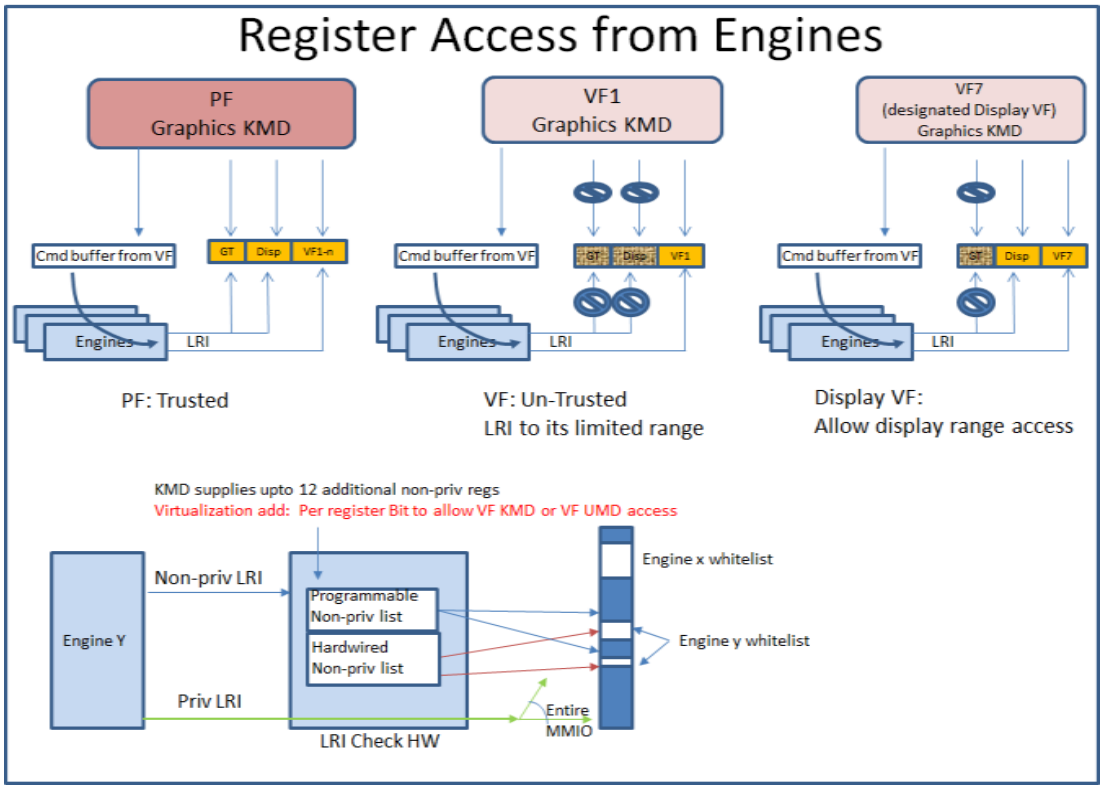
The hardware treats most registers as privileged. Functionally, UMD must be able to access certain registers, so a whitelist infrastructure provides limited register access, as described below:

- Specific registers in each engine are marked as non-privileged. Hardware allows non-privileged command buffers to complete accesses to these registers.
- However, such a hardwired whitelist is restrictive, and can be problematic if access to additional registers becomes necessary. So each engine provides a programmable means of converting twelve additional registers from privileged to non-privileged - KMD is expected to do the programming. The converted registers are saved and restored as part of power context, not process context.

In a virtualized environment, an engine is allowed four levels of register accesses as follows:

- UMDs for both PF and VF can access the standard UMD whitelist.
- A non-display VF KMD needs access to a larger set of registers than the UMD whitelist, but not the entire register range. Additional infrastructure to allow the extended register accesses is described later in this document.
- A display VF KMD can access the same registers as a non-display VF KMD, in addition to registers associated with the Display Engine (e.g., to schedule flips via LRI).
- PF KMD can access any register.

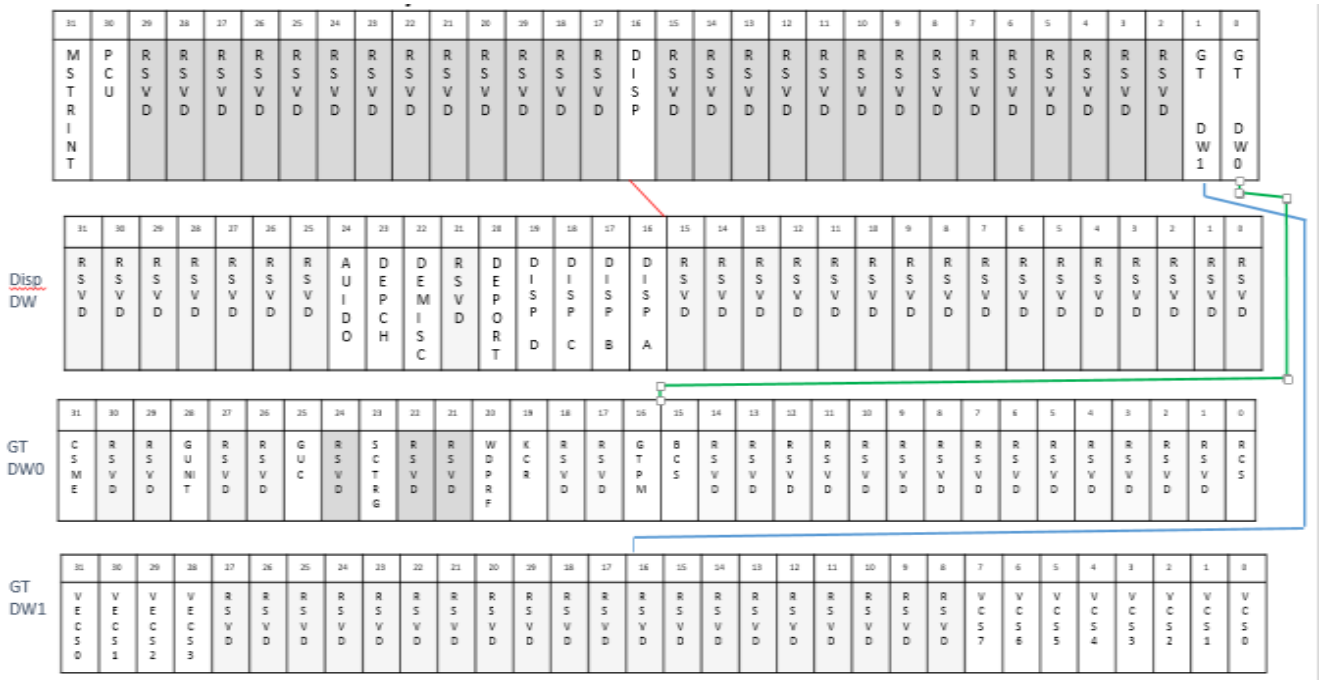
The programmable whitelist registers are modified by adding a single bit in each register, to specify whether the non-privileging action is targeted for UMD or KMD access. The added bit allows a trusted agent to set up some of the twelve programmable registers to allow VF KMD access to specific registers. The PF driver or GuC is responsible for setting up the programmable register list.



Register Access from Engines

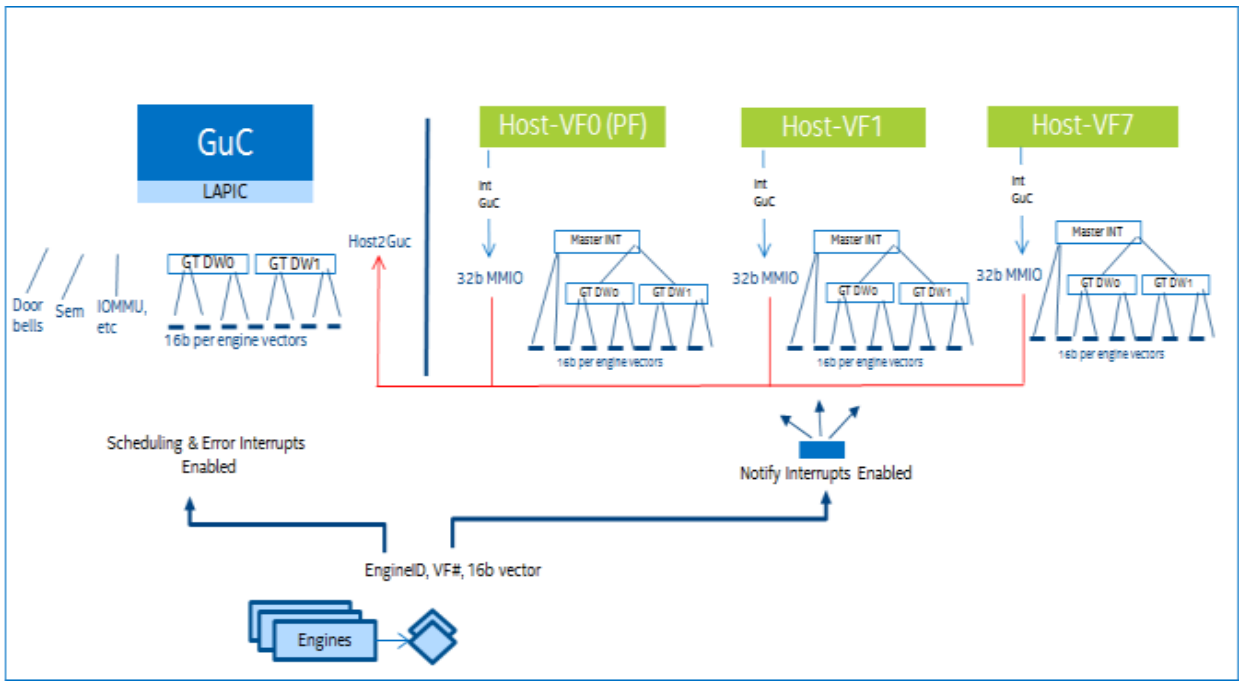
Interrupt Interface

Hardware exposes interrupts to the Host as shown in the figure below.



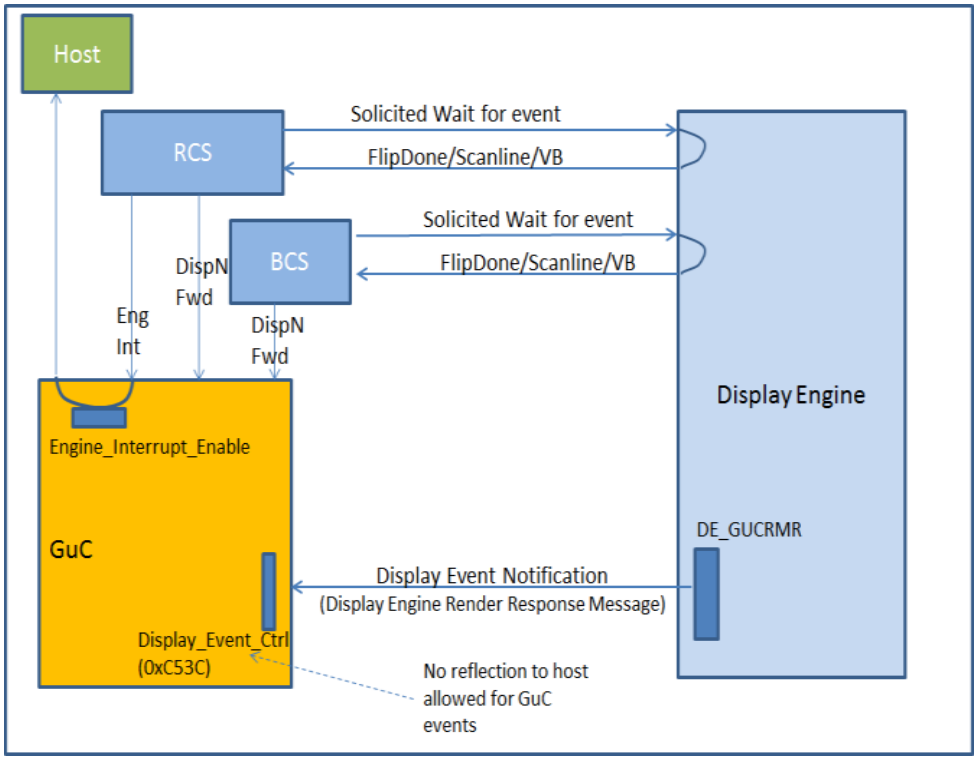
Pre-graphics Virtualization Scenario for Interrupt Delivery

Each VF's address space replicates the virtualization interrupt infrastructure. This replication keeps hardware and interrupt delivery routing simple, trading cost for simplicity. The following figure shows the infrastructure replicated in each VF's domain.



Interrupt Infrastructure per Virtual Function

Display hardware propagates display events only to PF. GuC can virtualize some display events to a VF if desired.



Display <=> Guc Interrupt Messages

Resets

Hardware supports the following reset variations:

- Conventional Reset
- Function Level Reset targeting a PF
- Function Level Reset targeting a VF
- Clearing the VF Enable bit with the SR-IOV Control register
- Engine-specific Reset

Conventional Reset (Bus Reset)

A conventional reset returns all functions (including PFs and VFs) to their original power-on state, and clears VF Enable in the PF - so VFs do not exist after a conventional reset.

FLR Targeting a PF

A Functional Level Reset (FLR) targeting a PF resets the PF to the PCIe Specification. The PF FLR resets the SR-IOV Extended Capability, including VF Enable, so VFs do not exist after this type of reset. For a single-

function integrated device (not counting VFs), PF FLR is almost the same as Conventional Reset, except that "Sticky" bits in Cfg space are preserved.

FLR Targeting a VF

The SR-IOV requires that all VFs support FLR. A VF FLR must clear the VF's internal state, but FLR does not affect the VF's existence in PCI Configuration Space or PCI Bus address space. The VF FLR does not affect the VF's BARn values or VF MSE in the PF's SR-IOV Extended Capability. Also, a VF FLR does not affect other VFs.

VMMs can use FLR to reset a VF before assigning that VF to a new VM. For this use of VF FLR, hardware functions normally without a reset, so the reset just clears leftover states from the previous VM.

Initiating a VF FLR triggers the following events:

- The MMIO space BARs retain their contents.
- Host invocation of VF FLR generates an interrupt to GuC.
- GuC interrupts the PF KMD to indicate that a VF is going through FLR sequence. The VF number is passed in the interrupt data register.
- PF KMD deactivates all doorbells associated with the VF FLR, by clearing the "cookie" value in DW0 of each affected doorbell address. Clearing the cookie value causes the doorbell controller to clear the doorbell's Valid bit, which software cannot directly clear.
    - Note: GuC may be able to disable the doorbell by clearing the cookie value in memory, if future driver architecture allows GuC firmware (instead of KMD) to allocate the doorbells. This software detail is addressed in the SAS.
- GuC discards all unscheduled workloads associated with the VF FLR.
- GuC pre-empts any workloads associated with the VF FLR that are currently running on engines. Note that software may choose use Engine-specific reset to terminate the workload (must not affect any workloads running on behalf of a different VF).
- Each CS running a context associated with the VF undergoing FLR initiates a TLB Invalidate that clears the PPGTT TLB associated with the CS engine.
    - Note: an engine-specific reset would clear out the GAM resources for that Engine, including TLB.
    - Clearing the PPGTT TLB for the CS engine does not affect second-level (VTd) intermediate walker caches.
- The VMM invalidates second-level TLB caches, if necessary.
- GuC discards any pending semaphores associated with VF workloads.
- GuC clears the VF interrupt interface: VF primary control, VF Interrupt additional info, and the VF GuC interrupt port.
- GuC interrupts PF KMD to indicate that cleanup is complete.
- Gunit resets VF-dedicated MMIO registers.

The above VF FLR handling occurs when hardware and GuC firmware are functioning properly. To catch possible errors, use PF KMD to monitor GuC firmware at regular intervals. If GuC does not respond in a timely manner, initiate an all-engine reset including reset of the GuC hardware, or use PF KMD to trigger a conventional reset (bus reset) of the entire device. When using PF to trigger a conventional reset, PF KMD must be able to notify the VMM, since the VMM must re-enable and rebuild the Virtual Functions.

Clearing SR-IOV VF Enable

When a VF Enable is cleared after being set, all of the VFs associated with PF cease to exist and must no longer issue PCIe transactions or respond to Configuration Space or Memory Space accesses. VFs must not retain any state (including sticky bits) after VF Enable has been cleared. These conditions are ensured by the following actions:

- VF Enable (Gunit) must qualify all configuration and memory accesses targeting VFs.
- Clearing VF Enable triggers an interrupt from Gunit to GuC.
- GuC firmware clears the internal state for all VFs, using the same flow described earlier for FLR targeting a VF. GuC firmware must follow the VF FLR flow for all previously enabled VFs.

Engine-Specific Reset

The PF KMD or GuC can reset specific hardware engines (e.g. RCS, BCS, VCS) when an engine is not responding. VF KMD is not allowed to directly reset any engine, since that engine may be executing work on behalf of a different VM. Instead, VF KMD interrupts GuC using the existing Host->GuC interrupt mechanism. GuC makes sure an engine is reset only if that engine is running on behalf of the interrupting VM/VF, and not running for any others.

Stolen Memory Management

Virtualization does not change stolen memory allocation. The BIOS allocates stolen memory and updates the BDSM and BGSM registers accordingly. The existence of stolen memory is exposed only to the PF driver.

Stolen memory segments are managed as follows:

GSM

- Global GTT is located in the stolen memory.
- Each VF can update its section of the GTT through the VF GTTADDR range.

DSM

- Only the PF driver can access DSM memory.
- VF must not map its Global GTT entries to DSM physical address.

Local Memory Virtualization

The GPU Device can be associated with a dedicated memory resource that is not shared with other system devices and not managed directly by the Operating System or Hypervisor / Virtual Machine Manager (VMM). Such a resource is referred to as "Local Memory". The most straightforward way to provide Local Memory is to directly attach dedicated memory, such as DDR, GDDR, or HBM, to the GPU device. But it is also possible to create a pool of Local Memory by carving out of stealing it from the normal System Memory pool (i.e., system DRAM).

Local Memory allocation is managed by the Device Driver instead of the OS or VMM. In a non-virtualized environment, the driver can differentiate Local vs System memory allocations using a bit in the 1st-level page tables (PPGTT or GGTT) and that is all that is required. Simple device virtualization using Direct Device Assignment or Pass-Thru models use the same mechanism, since the entire Local Memory is owned by the device and that device is attached to only one Guest VM.

However, with more complicated virtualization environments such as Single Root I/O Virtualization (SRIOV), the pool of Local Memory must be allocated among all Guests using the device, while preserving isolation requirements between the Guests. Specifically, a given Guest VM must be able to access Local Memory that was explicitly allocated to that Guest, but not be able to access any Local Memory that was allocated to a different Guest VM, nor to the Host/Hypervisor.

Since Local Memory is dedicated to the device, the Hypervisor does not manage this allocation. The GPU Driver and supporting HW must provide the mechanisms to provide and enforce the allocation of Local Memory.

Local Memory Translation Table

In order to flexibly allocate a limited amount of memory that is local to the GPU device, and isolate that allocation from other Virtual Machines, a new 2nd-Level "Local Memory Translation Table" (LMTT) is implemented in GT HW and SW. This table is used for Local Memory in place of the VTd 2nd-level table used with memory managed by system software (OS and VMM).

The key characteristics of the Local Memory Translation Tables (LMTT) are:

- The LMTT tables are managed by the Host KMD, in coordination with the VMM or Host OS
- A separate LMTT structure is allocated for each Guest VM or Assignable Interface that receives Local Memory resources
- The parameter Local Memory Guest Address Width (LMGAW) defines the number of address bits available for Guest view of Local Memory
- The parameter Local Memory Host Address Width (LMHAW) defines the number of address bits available for the final Host view of Local Memory
    - In the initial implementation, LMGAW and LMHAW are expected to be equal, so any Guest can access all of Local Memory, but this is not required architecturally.
- LMTT is a multi-level structure, residing entirely in Local Memory

- The top level is a Directory, with an entry for each Guest VM/VF
    - For SRIOV, the Directory Entries are indexed by the Function Number (63 VF + 1 PF = 64 Entries)
    - Each LMTT Directory Entry contains:
        - Location of the next-level LMTT structure for the specified VM, in physical Local Memory, in multiples of 64KB (LMHAW-16 bits)
        - Valid bit
    - With future SIOV support, Directory Entries are indexed by the PASID (20-bit PASID = 1M Entries), and must be naturally aligned.
    - LMTT Directory must be aligned on a 64KB boundary in Local Memory
    - The location of the LMTT Directory is stored in two MMIO registers accessible only by the Host KMD: one in Gunit, and one in GT (GAM)
- LMTT Leaf level translates a 2MB page of Local Memory in Guest address space to physical Local Memory
    - Each LMTT Entry is 32-bits wide and contains
        - Final translation of 2MB page from Guest Local Memory into the final Host Local Memory Page (address bits LMHAW-1:21).
        - Valid bit

LMTT Attributes

LMGAW	LMHAW	Directory Entry Size	Directory Size	L2 Entry Size	L2 Size	Leaf Entry Size	Leaf Table Size
37 bits (128GB)	37 bits (128GB)	32 bits	64 Entries 256 Bytes	N/A	N/A	32 bits	64k Entries 256KB

When a Guest VM is created, the VMM will virtualize an LMEM_BAR for the Guest to configure. This is referred to as LMEM_BAR(g) to differentiate from the "real" physical LMEM_BAR associated with the device (LMEM_BAR(h)). The Guest OS will assign a GPA to the LMEM_BAR(g) as it would any other PCI MMIO resource.

When a Command Streamer loads a new context, the VF# is extracted from the Context Descriptor and passed to the GAM, which tracks this information separately for each Engine.

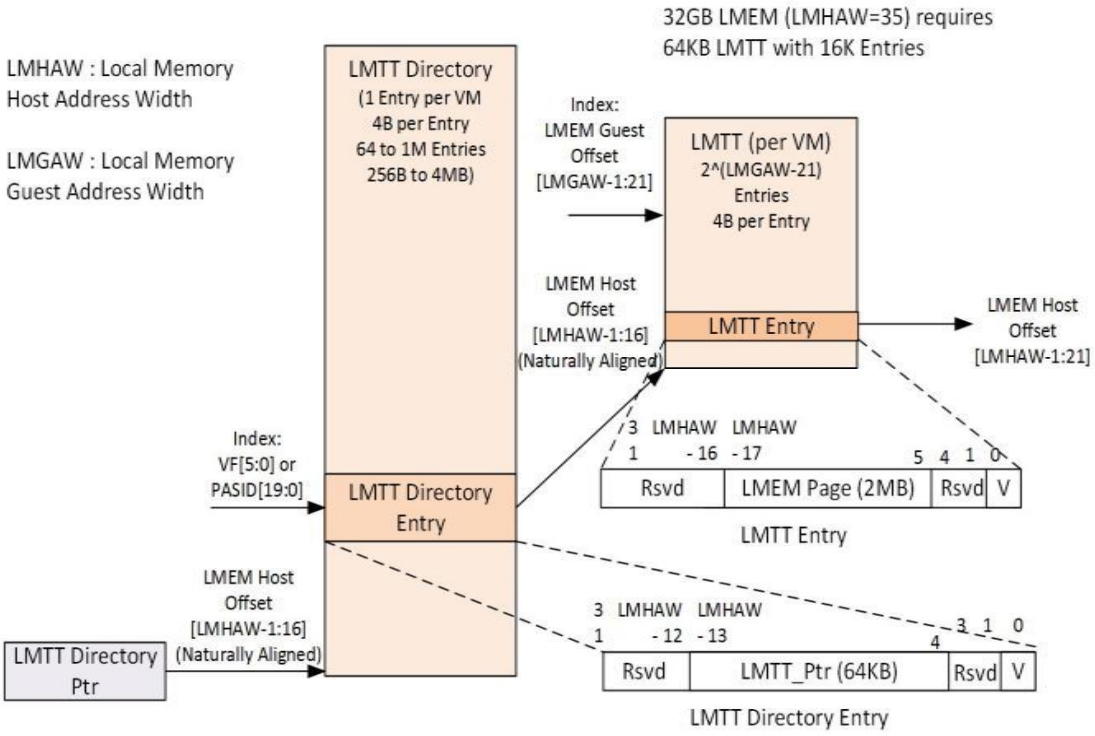
When GAM HW receives a request from an "Engine" (e.g., Render or VDBox) it performs 1st-level translation using the appropriate PPGTT. Each PPGTT Entry includes a new "LMEM" bit that indicates whether the page is allocated in Local Memory, or System Memory. If the page resides in Local Memory, then LMTT is used for 2nd level translation, otherwise the standard VTd tables are used via Host-based IOMMU (there is no IOMMU functionality within the discrete GPU device).

TLBs in GAM work essentially the same as today, except that the TLB tag data must include the new LMEM information from the page table entry. There are no changes required to TLB invalidations -

appropriate entries are automatically invalidated whenever a new context starts (whether part of a new VM or not), and same SW controls (KMD and GuC FW) can also be used to force invalidations when necessary.

The following figure illustrates the structure and function of the LMTT:

Two-Level LMTT Structure

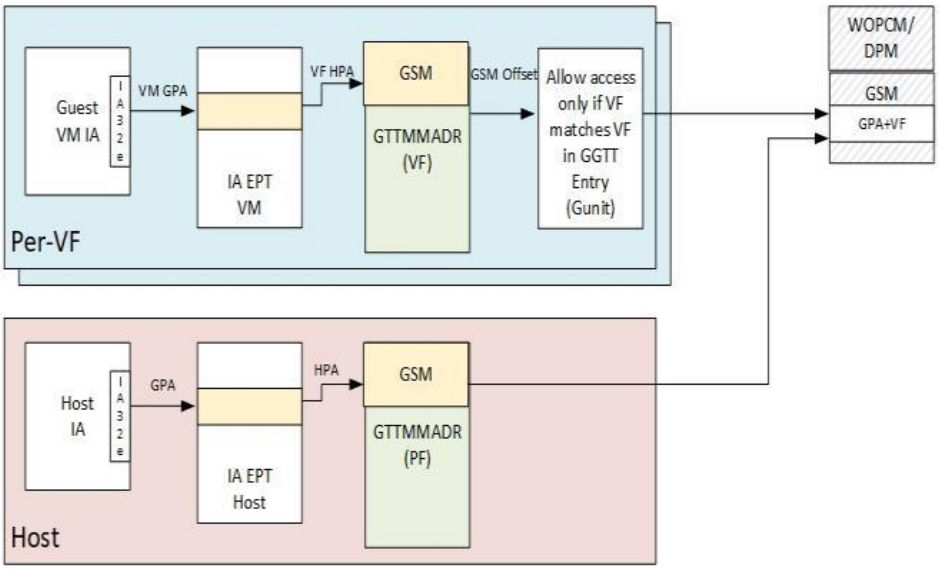


Multi-Level LMTT Structure

Global Memory Space

Global Address space, and the associated Global GTT, are shared for all Global allocations for all Guest VMs or AIs, as is already defined for SRIOV in monolithic (non-G-die) configurations with UMA memory. In the current SRIOV definition, the Global GTT is managed by the Host KMD, which determines which global address can be accessed by which Guests. The Global GTT Entry includes the Function Number of the guest (or Host) to which that global page has been allocated. The Host KMD allocates the GGTT entries for each Virtual Function. A Guest Kernel Mode Driver is allowed to directly update GGTT Entries that have already been allocated to that Guest VM/VF. A VM will be allowed to access the GGTT range (GSM) within the GTTMMADR BAR of the VF to which it has been mapped. A read or write to a specific GGTT Entry within a VF GSM range results in Gunit HW reading that entry and ensuring the Function Number field within the Entry matches the VF# of the access. If it does, the access is allowed to proceed, else it is blocked (writes dropped, reads return all zeros).

The diagram below illustrates the flow for access from IA Guest or Host driver to the GGTT in GSM range.



When GAM receives a request from a global agent, such as GuC or OA, or from an Engine that is operating on behalf of the Host, that targets Global Memory, GAM will use the Function Number from the GGTT for 2nd-level translation, instead of the Host's Function Number of 0. This allows the Host to share global space with all Guests.

With Type 2 Local Memory, we build on the above mechanism, and add another bit ("LMEM" bit 1) to the GGTT Entry to distinguish Global addresses that are allocated in Local Memory vs those in regular system memory. If the Global Address is in Local Memory (Local = 1), then 2nd-level translation is performed using the LMTT, as described above. If the Global Address is not Local (LMEM = 0), then the address is a GPA and is further translated through VTd 2nd-level tables using via the Host IOMMU - there is no need for IOMMU within the ATS device.

Note: The astute reader may wonder why a new bit is required to distinguish Local vs System Memory? Why can't HW just compare the physical address in the GGTT Entry against the PF LMEM_BAR, similar to PPGTT handling? That wouldn't work because the GGTT Entry may contain either a GPA pointing to DDR, or an HPA pointing to Local Memory, and nothing prevents the DDR GPA from overlapping the LMEM_BAR range.

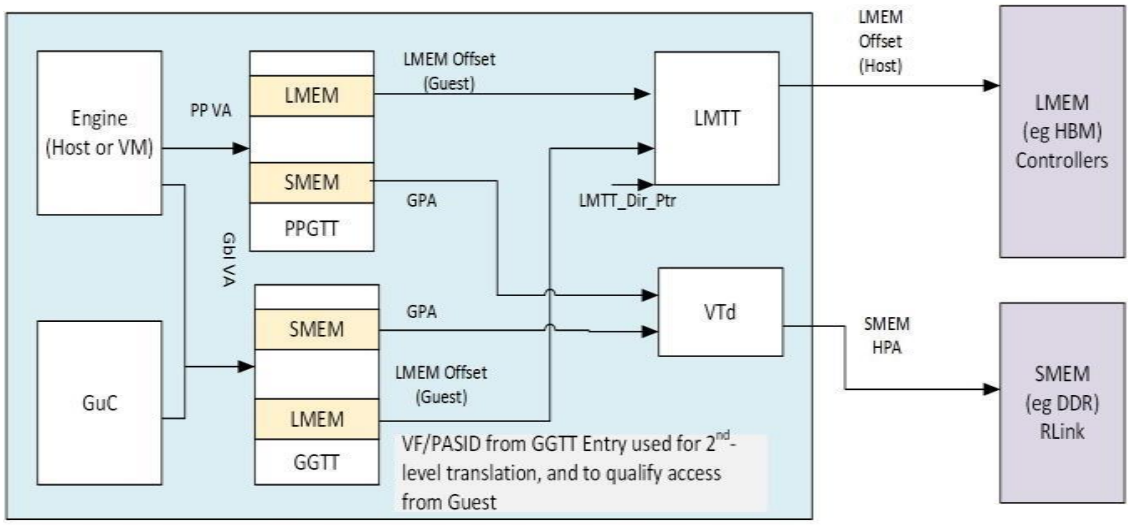
Because the global address space is shared by all contexts and all Guest VMs, and is limited to 4GB total, the GGTT page size granularity remains 4KB. When allocating global surfaces larger than 4KB, the Host KMD should attempt to choose contiguous 4KB pages up to at least 64KB, which is the minimum page size for which HBM accesses are optimized.

The updated GGTT Entry format:

GGTT Entry

63:54	53:46	45:12	11:2	1	0
PASID[19:10]	Rsvd	GPA/LMEM[45:12]	VF/PASID[9:0]	L	V

The following diagram illustrates the key structures and flows described above:

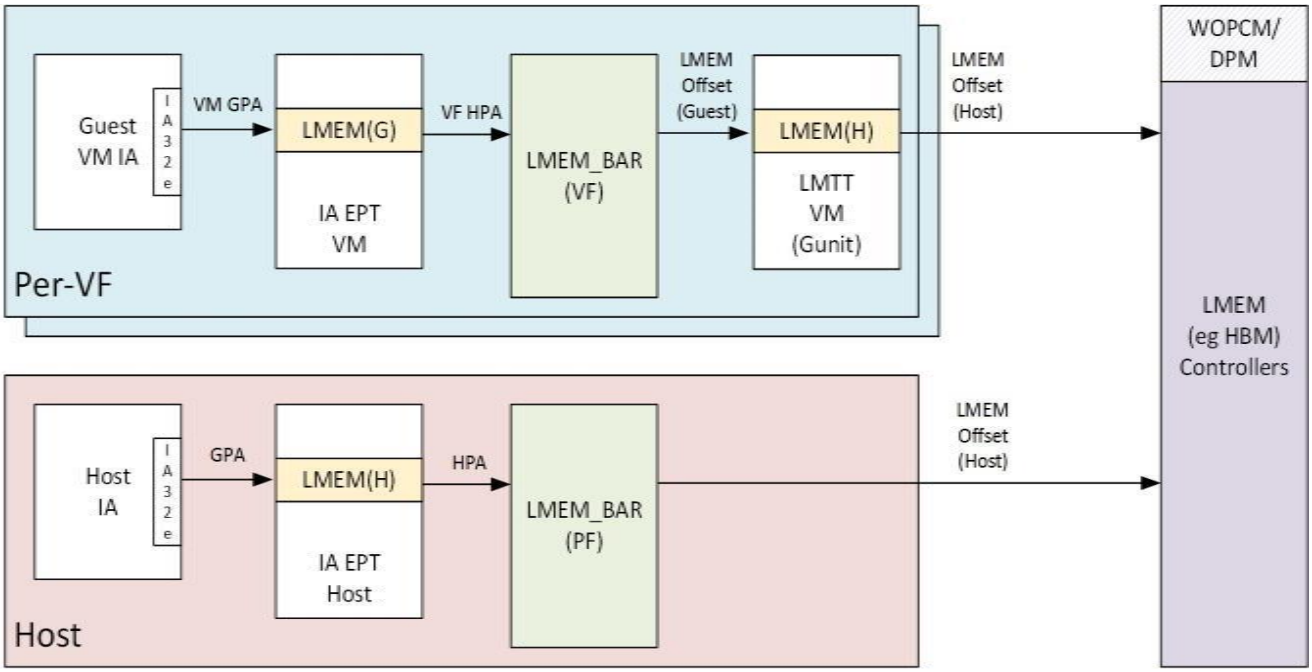


IA CPU Accesses to Local Memory

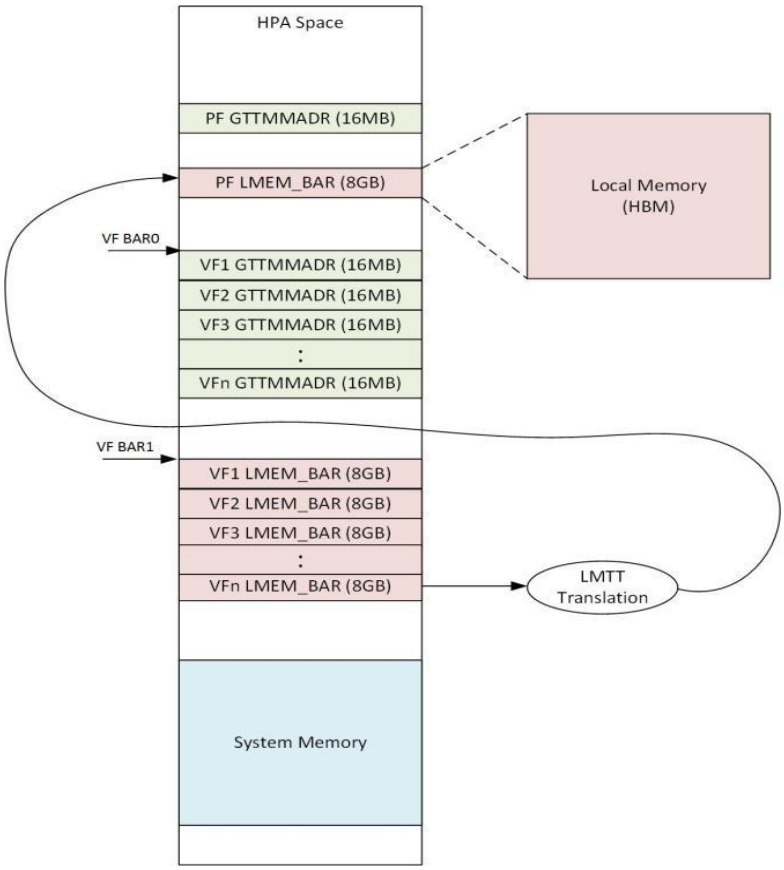
With SRIOV and Local Memory, the VF GMADR BARs are replaced by VF LMEM_BARs. The VMM virtualizes these VF BARs for each Guest VM. When a VM accesses its virtualized LMEM_BAR(G), the IA EPT for that VM will translate the request to the VF LMEM_BAR to which the VM has been assigned. The address of this request is in HPA space but does not yet map to the physical Local Memory, which resides at the PF LMEM_BAR range. So SGunit HW must also use the LMTT to translate from the Guest view of HBM to the actual location. The index into the LMTT_Directory structure is the VF# that contains the VF LMEM_BAR that was targeted by the access. The location of the LMTT_Directory structure is stored in a PF MMIO register (not accessible by a Guest VM).

The PF GMADR BAR is similarly replaced with a PF LMEM_BAR, which provides direct access to Local Memory. No LMTT translation is required for Host/PF access to Local Memory. SGunit HW simply strips off the LMEM_BAR base address to generate the final offset into Local Memory. HW does however need to protect certain non-IA accessible ranges such as WOPCM, DPM, and the GGTT (GGTT is only accessible to host via GSM range of GTTMMADR).

The following diagram illustrates the for SRIOV IA accesses to Local Memory:



The following Diagram shows how the various SRIOV BARs fit into HPA Space, using default BAR sizes (LMEM_BARs are configurable):



MMIO

This is the MMIO chapter for System Interfaces Volume.

Force Wake and Steering Table

MMIO Range Start	MMIO Range End	# Bytes	Wake Target	Replicated / Multicast ?	Replication Group Type	Inst. Count	Steering
00000000	00000AFF	2816					
00000B00	00000BFF	256	AON	Yes	SQIDI	8	subsliceid[0..7]
00000C00	00000DFF	512	AON	No	-	1	-
00000E00	00000FFF	512	AON	No	-	1	-
00001000	00001FFF	4096	AON	Yes	SQIDI	8	subsliceid[0..7]
00002000	000026FF	1792	RENDER*	No	-	1	-
00002700	000027FF	256	GT	No	-	1	-
00002800	00002AFF	768	GT	No	-	1	-
00002B00	00002FFF	1280	GT	No	-	1	-
00003000	00003FFF	4096	GT	No	-	1	-
00004000	000041FF	512	GT	Yes	MSLICE	4	sliceid[0..3]
00004200	000043FF	512	GT	Yes	MSLICE	4	sliceid[0..3]
00004400	000048FF	1280	GT	Yes	MSLICE	4	sliceid[0..3]
00004900	00004AFF	512	GT	Yes	MSLICE	4	sliceid[0..3]
00004B00	00004FFF	1280					
00005000	000050FF	256	AON	No	-	1	-
00005100	000051FF	256	AON	No	-	1	-
00005200	000052FF	256	RENDER	Yes	GSLICE	8	sliceid[0..7]
00005300	000053FF	256	RENDER*	No	-	1	-
00005400	000054FF	256	RENDER	Yes	GSLICE	8	sliceid[0..7]
00005500	00005FFF	2816	RENDER	Yes	GSLICE	8	sliceid[0..7]
00006000	00006FFF	4096	RENDER	Yes	GSLICE	8	sliceid[0..7]
00007000	00007FFF	4096	RENDER	Yes	GSLICE	8	sliceid[0..7]
00008000	000080FF	256	GT	No	-	1	-
00008100	0000813F	64	GT	No	-	1	-
00008140	0000814F	16	RENDER	Yes	GSLICE	8	sliceid[0..7]
00008150	0000815F	16	RENDER	Yes	DSS	32	sliceid[0..7], subsliceid[0..3]
00008160	0000817F	32					
00008180	000081FF	128	AON	No	-	1	-
00008200	000082FF	256	GT	No	-	1	-
00008300	000084FF	512	RENDER*	No	-	1	-
00008500	000085FF	256	GT	No	-	1	-
00008600	000086FF	256	GT	No	-	1	-

MMIO Range Start	MMIO Range End	# Bytes	Wake Target	Replicated / Multicast ?	Replication Group Type	Inst. Count	Steering
00008700	000087FF	256	GT	Yes	SQIDI	8	subsliceid[0..7]
00008800	0000883F	64					
00008840	000089FF	448					
00008A00	00008BFF	512					
00008C00	00008C7F	128					
00008C80	00008CFF	128	GT	Yes	L3BANK	32	sliceid[0..3], subsliceid[0..7]
00008D00	00008D7F	128	RENDER	Yes	DSS	32	sliceid[0..7], subsliceid[0..3]
00008D80	00008DFF	128	RENDER	Yes	DSS	32	sliceid[0..7], subsliceid[0..3]
00008E00	00008FFF	512					
00009000	000093FF	1024	GT	No	-	1	-
00009400	0000947F	128	GT	No	-	1	-
00009480	000094CF	80					
000094D0	0000951F	80	RENDER	Yes	GSLICE	8	sliceid[0..7]
00009520	0000955F	64	RENDER	Yes	DSS	32	sliceid[0..7], subsliceid[0..3]
00009560	000095FF	160	AON	No	-	1	-
00009600	0000967F	128					
00009680	000096FF	128	RENDER	Yes	DSS	32	sliceid[0..7], subsliceid[0..3]
00009700	000097FF	256					
00009800	00009FFF	2048	GT	No	-	1	-
0000A000	0000AFFF	4096	GT	No	-	1	-
0000B000	0000B0FF	256	GT	Yes	LNCF	8	sliceid[0..3], subsliceid[0..1]
0000B100	0000B3FF	768	GT	Yes	L3BANK	32	sliceid[0..3], subsliceid[0..7]
0000B400	0000B4FF	256	GT	No	-	1	-
0000B500	0000BFFF	2816					
0000C000	0000C7FF	2048	GT	No	-	1	-
0000C800	0000CFFF	2048	GT	Yes	MSLICE	4	sliceid[0..3]
0000D000	0000D3FF	1024	AON	No	-	1	-
0000D400	0000D7FF	1024	AON	No	-	1	-
0000D800	0000D87F	128	RENDER	Yes	GSLICE	8	sliceid[0..7]
0000D880	0000D8FF	128	GT	Yes	LNCF	8	sliceid[0..3], subsliceid[0..1]
0000D900	0000DBFF	768	GT	No	-	1	-
0000DC00	0000DCFF	256	RENDER	Yes	GSLICE	8	sliceid[0..7]

MMIO Range Start	MMIO Range End	# Bytes	Wake Target	Replicated / Multicast ?	Replication Group Type	Inst. Count	Steering
0000DD00	0000DDFF	256	GT	Yes	MSLICE	4	sliceid[0..3]
0000DE00	0000DE7F	128					
0000DE80	0000DEFF	128	RENDER	Yes	DSS	32	sliceid[0..7], subsliceid[0..3]
0000DF00	0000DFFF	256	RENDER	Yes	DSS	32	sliceid[0..7], subsliceid[0..3]
0000E000	0000E0FF	256					
0000E100	0000E1FF	256	RENDER	Yes	DSS	32	sliceid[0..7], subsliceid[0..3]
0000E200	0000E3FF	512	RENDER	Yes	DSS	32	sliceid[0..7], subsliceid[0..3]
0000E400	0000E7FF	1024	RENDER	Yes	DSS	32	sliceid[0..7], subsliceid[0..3]
0000E800	0000E8FF	256	RENDER	Yes	DSS	32	sliceid[0..7], subsliceid[0..3]
0000E900	0000E9FF	256	GT	Yes	MSLICE	4	sliceid[0..3]
0000EA00	0000EFFF	1536					
0000F000	0000F0FF	256	GT	Yes	MSLICE	4	sliceid[0..3]
0000F100	0000FFFF	3840	GT	Yes	MSLICE	4	sliceid[0..3]
00010000	00011FFF	8192					
00012000	000127FF	2048	AON	No	-	1	-
00012800	00012FFF	2048					
00013000	000131FF	512	VD0	No	-	1	-
00013200	000133FF	512	VD2	No	-	1	-
00013400	00013FFF	3072					
00014000	000141FF	512	VD0	No	-	1	-
00014200	000143FF	512	VD2	No	-	1	-
00014400	000145FF	512	VD4	No	-	1	-
00014600	000147FF	512	VD6	No	-	1	-
00014800	00014FFF	2048	RENDER*	No	-	1	-
00015000	000153FF	1024	GT	No	-	1	-
00015400	000157FF	1024	GT	No	-	1	-
00015800	00015BFF	1024	GT	No	-	1	-
00015C00	00015FFF	1024	GT	No	-	1	-
00016000	00016DFF	3584					
00016E00	00016FFF	512	RENDER*	No	-	1	-
00017000	00017FFF	4096	RENDER	Yes	GSLICE	8	sliceid[0..7]
00018000	00019FFF	8192	RENDER*	No	-	1	-
0001A000	0001BFFF	8192	RENDER*	No	-	1	-
0001C000	0001DFFF	8192	RENDER*	No	-	1	-

MMIO Range Start	MMIO Range End	# Bytes	Wake Target	Replicated / Multicast ?	Replication Group Type	Inst. Count	Steering
0001E000	0001FFFF	8192	RENDER*	No	-	1	-
00020000	00020FFF	4096	VD0	No	-	1	-
00021000	00021FFF	4096					
00022000	00022FFF	4096	GT	No	-	1	-
00023000	00023FFF	4096	GT	No	-	1	-
00024000	0002407F	128	AON	No	-	1	-
00024080	0002417F	256					
00024180	000241FF	128	GT	No	-	1	-
00024200	000249FF	2048					
00024A00	00024A7F	128	RENDER	Yes	DSS	32	sliceid[0..7], subsliceid[0..3]
00024A80	000251FF	1920					
00025200	0002527F	128	GT	No	-	1	-
00025280	000252FF	128	GT	No	-	1	-
00025300	000255FF	768					
00025600	0002567F	128					
00025680	000256FF	128					
00025700	000259FF	768					
00025A00	00025A7F	128					
00025A80	00025AFF	128					
00026000	00027FFF	8192	RENDER*	No	-	1	-
00028000	0002FFFF	32768					
00030000	0003FFFF	65536	GT	No	-	1	-
001C0000	001C07FF	2048	VD0	No	-	1	-
001C0800	001C0FFF	2048	VD0	No	-	1	-
001C1000	001C1FFF	4096	VD0	No	-	1	-
001C2000	001C27FF	2048	VD0	No	-	1	-
001C2800	001C2AFF	768	VD0	No	-	1	-
001C2B00	001C2BFF	256	VD0	No	-	1	-
001C2C00	001C2CFF	256					
001C2D00	001C2DFF	256	VD0	No	-	1	-
001C2E00	001C3DFF	4096	VD0	No	-	1	-
001C3E00	001C3EFF	256					
001C3F00	001C3FFF	256	VD0	No	-	1	-
001C4000	001C47FF	2048	VD1	No	-	1	-
001C4800	001C4FFF	2048	VD1	No	-	1	-

MMIO Range Start	MMIO Range End	# Bytes	Wake Target	Replicated / Multicast ?	Replication Group Type	Inst. Count	Steering
001C5000	001C5FFF	4096	VD1	No	-	1	-
001C6000	001C67FF	2048	VD1	No	-	1	-
001C6800	001C6AFF	768	VD1	No	-	1	-
001C6B00	001C6BFF	256	VD1	No	-	1	-
001C6C00	001C6CFF	256					
001C6D00	001C6DFF	256	VD1	No	-	1	-
001C6E00	001C7EFF	4352					
001C7F00	001C7FFF	256					
001C8000	001C9FFF	8192	VE0	No	-	1	-
001CA000	001CA0FF	256	VE0	No	-	1	-
001CA100	001CBEFF	7680					
001CBF00	001CBFFF	256					
001CC000	001CCFFF	4096	VD0	No	-	1	-
001CD000	001CDFFF	4096	VD2	No	-	1	-
001CE000	001CEFFF	4096	VD4	No	-	1	-
001CF000	001CFFFF	4096	VD6	No	-	1	-
001D0000	001D07FF	2048	VD2	No	-	1	-
001D0800	001D0FFF	2048	VD2	No	-	1	-
001D1000	001D1FFF	4096	VD2	No	-	1	-
001D2000	001D27FF	2048	VD2	No	-	1	-
001D2800	001D2AFF	768	VD2	No	-	1	-
001D2B00	001D2BFF	256	VD2	No	-	1	-
001D2C00	001D2CFF	256					
001D2D00	001D2DFF	256	VD2	No	-	1	-
001D2E00	001D3DFF	4096					
001D3E00	001D3EFF	256					
001D3F00	001D3FFF	256	VD2	No	-	1	-
001D4000	001D47FF	2048	VD3	No	-	1	-
001D4800	001D4FFF	2048	VD3	No	-	1	-
001D5000	001D5FFF	4096	VD3	No	-	1	-
001D6800	001D6AFF	768	VD3	No	-	1	-
001D6B00	001D6BFF	256	VD3	No	-	1	-
001D6C00	001D6CFF	256					
001D6D00	001D6DFF	256	VD3	No	-	1	-
001D6E00	001D7EFF	4352					
001D7F00	001D7FFF	256					
001D8000	001D9FFF	8192	VE1	No	-	1	-
001DA000	001DA0FF	256	VE1	No	-	1	-

MMIO Range Start	MMIO Range End	# Bytes	Wake Target	Replicated / Multicast ?	Replication Group Type	Inst. Count	Steering
001DA100	001DBEFF	7680					
001DBF00	001DBFFF	256					
001DC000	001DFFFF	16384					
001E0000	001E07FF	2048	VD4	No	-	1	-
001E0800	001E0FFF	2048	VD4	No	-	1	-
001E1000	001E1FFF	4096	VD4	No	-	1	-
001E2800	001E2AFF	768	VD4	No	-	1	-
001E2B00	001E2BFF	256	VD4	No	-	1	-
001E2C00	001E2CFF	256					
001E2D00	001E2DFF	256	VD4	No	-	1	-
001E2E00	001E3EFF	4352					
001E3F00	001E3FFF	256	VD4	No	-	1	-
001E4000	001E47FF	2048	VD5	No	-	1	-
001E4800	001E4FFF	2048	VD5	No	-	1	-
001E5000	001E5FFF	4096	VD5	No	-	1	-
001E6800	001E6AFF	768	VD5	No	-	1	-
001E6B00	001E6BFF	256	VD5	No	-	1	-
001E6C00	001E6CFF	256					
001E6D00	001E6DFF	256	VD5	No	-	1	-
001E6E00	001E7EFF	4352					
001E7F00	001E7FFF	256					
001E8000	001E9FFF	8192	VE2	No	-	1	-
001EA000	001EA0FF	256	VE2	No	-	1	-
001EA100	001EBEFF	7680					
001EBF00	001EBFFF	256					
001EC000	001EFFFF	16384					
001F0000	001F07FF	2048	VD6	No	-	1	-
001F0800	001F0FFF	2048	VD6	No	-	1	-
001F1000	001F1FFF	4096	VD6	No	-	1	-
001F2000	001F27FF	2048	VD6	No	-	1	-
001F2800	001F2AFF	768	VD6	No	-	1	-
001F2B00	001F2BFF	256	VD6	No	-	1	-
001F2C00	001F2CFF	256					
001F2D00	001F2DFF	256	VD6	No	-	1	-
001F2E00	001F3EFF	4352					
001F3F00	001F3FFF	256	VD6	No	-	1	-
001F4000	001F47FF	2048	VD7	No	-	1	-
001F4800	001F4FFF	2048	VD7	No	-	1	-

MMIO Range Start	MMIO Range End	# Bytes	Wake Target	Replicated / Multicast ?	Replication Group Type	Inst. Count	Steering
001F5000	001F5FFF	4096	VD7	No	-	1	-
001F6800	001F6AFF	768	VD7	No	-	1	-
001F6B00	001F6BFF	256	VD7	No	-	1	-
001F6C00	001F6CFF	256					
001F6D00	001F6DFF	256	VD7	No	-	1	-
001F6E00	001F7EFF	4352					
001F7F00	001F7FFF	256					
001F8000	001F9FFF	8192	VE3	No	-	1	-
001FA000	001FA0FF	256	VE3	No	-	1	-
001FA100	001FBEFF	7680					
001FBF00	001FBFFF	256					
001FC000	001FFFFF	16384					
00200000	0023FFFF	262144					

Render*- physically located in GTI but logically managed as if part of render for forcewake/shadowing by MGSR (host SW should use Render wake before access)

- The Steering Control Registers reside at the following locations:
  - MGSR access point (access initiated by agent outside of GT):

#	Steering Reg Addr	Description
1	0xFD0	Access steering towards MCFG endpoints only.
2	0xFD4	Access steering towards SF endpoints only, HW (ITP, DFX, Pcode/CSE FW accesses)
3	0xFD8	Access steering towards SF endpoints only, IA
4	0xFE0	Access steering towards GAM (GAMCTRL, GAMREQSTRM, GAMCMDI, GAMWKRS, and GAMXB).
5	0xFDC	Access steering towards all other endpoints

    - **Note:** 0xFD4 and 0xFD8 steering registers are provided to allow concurrent steering access from driver and any other agent.
    - GuC access **point**:

#	Steering Reg Addr	Description
1	0xC060	Access steering towards all GT endpoints

- CS access point:

#	Steering Reg Addr	Description
1	0x20CC	Access steering towards all GT endpoints

- **Note:** All Steering Control Registers contain the following fields:

Field	Description
multicast	1: Access will be multicast to all replicated endpoints: • *WRITE* op cycles go to all endpoint instances; sliceid[]/subsliceid[] fields ignored. • *READ* op cycles go to all endpoint instances, and responses are returned from all instances; The MsgCh selects single instance's response as the final read return, based on sliceid[]/subsliceid[] fields. 0: Access will be steered using sliceid[] and subsliceid[] fields below: • Both *WRITE* and *READ* cycles go to a single instance of an endpoint, based on sliceid[]/subsliceid[] steering. Default: 1 Note: The multicast field has no impact for a non-replicated target.
sliceid[]	Default: 0
subsliceid[]	Default: 0

- The following Replication Group Types exist for multicast MMIO endpoints:

- Note: GT is organized into "quadrants"; where each quadrant contains (8DSS w/128EUs, + 2 geometry slices) + 1 Mslice + 8 L3$ banks. Thus, the 0th quadrant contains DSS7:0, Gslice1:0, MSlice0, and L3Banks 7:0. The 3rd quadrant contains DSS 31:24, Gslice 7:6; MSlice 3; L3 Banks31:24. For the discussion below, we will use the following "helper" values:
  - combined_dss_enable = dss_g_enable | dss_c_enable => bit field indicating DSS that are "enabled" for either geometry or compute or both
  - quadrant_dss_enabled[n] = (((combined_dss_enable » 8*n) && 0xFF) != 0) where n is quadrant index in range of 0..3

Replication Group Type	Description / Notes

SQIDI	• 8 instances of each endpoint
	  ○ subsliceid: 0..7 to access instances 0..7 in Alchemist-512
	  ○ subsliceid: 2..5 to access instances 2..5 in Alchemist-256 (instances 0, 1, 6, 7 are terminated by the MsgCh)
	  ○ subsliceid: 2,3 to access instances 2,3 in Alchemist-128 (instances 0, 1, 4-7 are terminated by the MsgCh)

GSLICE	• 8 instances of endpoint ○ sliceid: 0..7 to access SLICE instances 0..7 • MsgCh terminated when all 4 fuse_dss_g_enable and fuse_dss_c_enables are '0' for that instance

DSS	• 4 instances per GSLICE
	  ○ sliceid: 0..7 to access each GSLICE 0..7
	  ○ subsliceid: 0..3 to access each DSS 0..3 per GSLICE
	• MsgCh terminated when fuse_dss_g_enable and fuse_dss_c_enables is '0' for that instance

L3BANK	• Access to the nth L3Bank maps onto a 5 bit l3BankIndex which is split into between the steering register fields as follows: l3BankIndex[4:2]=> sliceid[2:0]; l3BankIndex [1:0]=> subsliceid[1:0]. bit 3 of sliceid and bit 2 of subsliceid in steering register is unused in this mapping. • Termination: All 8 L3 banks in a quadrant are disabled/terminated ONLY if BOTH meml3 for that quadrant is disabled AND all DSS are disabled on that quadrant.
MSLICE	• Steering: sliceid (0..3) • Termination: An MSLICE is disabled/terminated ONLY if BOTH meml3 for that quadrant is disabled AND all DSS are disabled on that quadrant. GAM* note: There is one "primary" GAM, and software must set the steering back to that primary GAM before reading any GAM register: Primary GAM instance is 0 unless otherwise noted. Primary GAM instance is 1

LNCF	• Steering: sliceid (0,1) to access 2 LNCFs individually in MSLICE0; sliceid (2,3) to access 2 LNCFs individually in MSLICE1; sliceid (4,5) to access 2 LNCFs individually in MSLICE2; sliceid (6,7) to access 2 LNCFs individually in MSLICE3; • Termination: An LNCF is disabled/terminated ONLY if BOTH meml3 for that quadrant is disabled AND all DSS are disabled on that quadrant.

- Fuse reflections (how to tell when an endpoint is disabled):

Fuse	Register reflection
fuse_dss_c_enable[31:0]	0x9144[31:0]
fuse_dss_g_enable[31:0]	0x913C[31:0]
fuse_meml3_en[3:0]	0x9118[3:0]

Note: MsgCh termination also occurs when the domains are powered down. (i.e., not necessarily because the domain is disabled/fused off.) If reading/writing the registers is needed, then force-wake of the domain is required. Force-wake is not required for shadow register accesses coming through MGSR.

- The following table captures the force-wake and corresponding acknowledgment register locations for the various domains:

Domain	Driver ForceWake Req	Driver ForceWake Ack	GuC ForceWake Req	GuC ForceWake Status	Comment
AON	NA	NA	NA	NA	Registers sit outside of the C6 boundary. No ForceWake required.
GT	0xA188	0x00130044	NA	NA	
Render	0xA278	0x0D84	0xA27C	0xA2A0[1]	
VDBOX0	0xA540	0x0D50	0xA274[0]	0xA2A0[0]	
VDBOX1	0xA544	0x0D54	0xA274[1]	0xA2A0[0]	
VDBOX2	0xA548	0x0D58	0xA274[2]	0xA2A0[2]	
VDBOX3	0xA54C	0x0D5C	0xA274[3]	0xA2A0[2]	As available in the product
VDBOX4	0xA550	0x0D60	0xA274[4]	0xA2A0[3]	As available in the product
VDBOX5	0xA554	0x0D64	0xA274[5]	0xA2A0[3]	As available in the product

Domain	Driver ForceWake Req	Driver ForceWake Ack	GuC ForceWake Req	GuC ForceWake Status	Comment
VDBOX6	0xA558	0x0D68	0xA274[6]	0xA2A0[4]	As available in the product
VDBOX7	0xA55C	0x0D6C	0xA274[7]	0xA2A0[4]	As available in the product
VEBOX0	0xA560	0x0D70	0xA274[8]	0xA2A0[0]	As available in the product
VEBOX1	0xA564	0x0D74	0xA274[9]	0xA2A0[2]	As available in the product
VEBOX2	0xA568	0x0D78	0xA274[10]	0xA2A0[3]	As available in the product
VEBOX3	0xA56C	0x0D7C	0xA274[11]	0xA2A0[4]	As available in the product

- Miscellaneous Notes:
  - The MsgCh network has termination points, where cycles to endpoints that are disabled (fused-off, powered off, etc…) are gracefully completed. The termination node on the network will sink P cycles, and return dummy completions for NP cycles, on behalf of the disabled endpoints.
  - Access requirements to registers that are part of GTMMADDR but not listed in the GT MMIO map table is defined elsewhere. This descriptions in this document only cover GT range (GT MMIO map xls.)

Multicast Steering and Die Recovery

Some units in GT are replicated multiple times in the design, each with their own register storage local to that instance.

- In some cases, each replica/instance gets its own MMIO address range of offsets – for example, the multiple CCS command streamers, multiple VDBox/VEBox instances. For those, direct register access targets the only instance of that registers. The programming model described on this page is moot for those cases where each register has unique address.
- In other cases, the multiple instances of the unit use the same MMIO address on message channel. For these cases, the message channel provides additional capabilities to address the instances for read/write operations in either multicast (targeting all instances) or unicast modes (target specific instance) via a set of "steering registers" which can be configured to direct the access as desired. The steering registers have 3 fields: Multicast/Unicast, Sliceid, Subsliceid.
  - Multicast write access - write goes to all instances; sliceid/subsliceid fields are ignored
  - Multicast read access – read goes to all instances and all instances generate read response; message channel selects single instance's response as the final read return (based on the steering register slice/subslice fields)
  - Unicast write access – write goes to only the instance specified in the steering register
  - Unicast read access – read goes to only the instance specified in the steering register

- In some replicated units, all of the replicated instances always "enabled" from a message channel perspective (never fused off/separately power gated) and thus all instances are always accessible if the containing power well is on (e.g., if GT is out of RC6)
- In some replicated units, there are die recovery/fuse down modes where some instances are fused off/disabled. For the latter, GT also contains MMIO registers which allow SW to detect which instances are fused as enabled/disabled (generally 1-hot). When this fuse down case applies, message channel is aware of the fusing and provides automatic termination of cycles toward disabled instances (writes get dropped with dummy NP completion if NP write; reads get dummy completion with 0 read return value from that instance). The fuse mirror register provides a mechanism for SW to know which instances are valid and to program the steering register toward enabled instances when needed – see comments below.

General rules:

- Some of these replicated registers are control registers which are generally expected to be all programmed with the same value – for these, writes should generally be multicast and reads can target any enabled instance (since all instances should contain the same value from prior multicast write).
- Some replicated registers are status registers and are expected to have different values as part of normal usage (for example, INSTDONE registers related to Sampler, Slice common; TDL thread status, etc.). For these typical usage model would be to either iterate over all enabled instances or select specific single instance to target.
- If an instance is disabled (access terminated on message channel via the fuse info above or if containing power well is power gated), reads from that instance will return 0s and writes are silently dropped. Since the default for the steering registers is multicast read with sliceid=subsliceid=0, the default hardware behavior is to return data from instance that corresponds with sliceid/subsliceid = 0. If that instance is disabled, message channel will return a dummy response (0). In order to get correct/valid value the steering registers must be used to access a valid instance.
  - Note that a common usage model is for SW/FW to initializing specific bits in control register by reading the current/default value, then modifying the value in memory (set/clear few bits), and then write the result back.
  - For these cases, SW must ensure that it uses the steering registers to steer to an enabled instance when performing the initial read.
- When performing engine and power context save restore, GT hardware is aware of the fuses and internally targets reads for context save toward the first enabled instance.
- In cases where steering registers are being programmed, caution must be exercised to ensure that there is no race condition/concurrent access between two different initiators using a given steering register. SW must protect against concurrent access by multiple threads to any given steering register. System level flows must also guard against concurrent access by Firmware (CSC/FSP FW, Punit pCode) and driver tools to any given steering register.

- Multicast is the hardware default. If an agent sets a steering register to unicast mode, they should generally set it back to multicast after completion.
- In some projects there are separate steering registers listed are intended to allow for some degree of concurrency between different usages targeting different destinations in GT by replication group.
    - MGSR uses the MMIO offset requested in the inbound cycle to select which steering register to use for routing.
    - MGSR uses SAI policy registers to identify sources as “IA” (low privilege cfg_src on message channel) vs “HW” (high privilege – includes trusted firmware such as CSC/FSP, Pcode)
    - See project specific documentation for the list of steering registers and their intended use.

SW Virtualization Reserved MMIO range

The MMIO address range from 0x178000 thru 0x178FFF is reserved for communication between a VMM and the GPU Driver executing on a Virtual Machine.

HW does not actually implement anything within this range. Instead, in a SW Virtualized environment, if a VM driver issues a read to this MMIO address range, the VMM will trap that access, and provide whatever data it wishes to pass to the VM driver. In a non-SW-Virtualizated environment (including an SR-IOV Virtualized environment), reads will return zeros, like any other unimplemented MMIO address. Writes to this range are always ignored.

It is important that no "real" HW MMIO register be defined within this range, as it would be inaccessible in a SW-virtualized environment.

Register Address Maps

Graphics Register Address Map

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

VGA and Extended VGA Register Map

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

VGA and Extended VGA I/O and Memory Register Map

Address	Register Name (Read)	Register Name (Write)
2D Registers		
3B0h-3B3h	Reserved	Reserved
3B4h	VGA CRTC Index (CRX) (monochrome)	VGA CRTC Index (CRX) (monochrome)
3B5h	VGA CRTC Data (monochrome)	VGA CRTC Data (monochrome)
3B6h-3B9h	Reserved	Reserved
3Bah	VGA Status Register (ST01)	VGA Feature Control Register (FCR)
3BBh-3BFh	Reserved	Reserved
3C0h	VGA Attribute Controller Index (ARX)	VGA Attribute Controller Index (ARX)/ VGA Attribute Controller Data (alternating writes select ARX or write ARxx Data)
3C1h	VGA Attribute Controller Data (read ARxx data)	Reserved
3C2h	VGA Feature Read Register (ST00)	VGA Miscellaneous Output Register (MSR)
3C3h	Reserved	Reserved
3C4h	VGA Sequencer Index (SRX)	VGA Sequencer Index (SRX)
3C5h	VGA Sequencer Data (SRxx)	VGA Sequencer Data (SRxx)
3C6h	VGA Color Palette Mask (DACMASK)	VGA Color Palette Mask (DACMASK)
3C7h	VGA Color Palette State (DACSTATE)	VGA Color Palette Read Mode Index (DACRX)
3C8h	VGA Color Palette Write Mode Index (DACWX)	VGA Color Palette Write Mode Index (DACWX)
3C9h	VGA Color Palette Data (DACDATA)	VGA Color Palette Data (DACDATA)
3CAh	VGA Feature Control Register (FCR)	Reserved
3CBh	Reserved	Reserved
3CCh	VGA Miscellaneous Output Register (MSR)	Reserved
3CDh	Reserved	Reserved
3CEh	VGA Graphics Controller Index (GRX)	VGA Graphics Controller Index (GRX)
3CFh	VGA Graphics Controller Data (GRxx)	VGA Graphics Controller Data (GRxx)
3D0h-3D1h	Reserved	Reserved
2D Registers		
3D4h	VGA CRTC Index (CRX)	VGA CRTC Index (CRX)
3D5h	VGA CRTC Data (CRxx)	VGA CRTC Data (CRxx)
System Configuration Registers		

Address	Register Name (Read)	Register Name (Write)
3D6h	GFX/2D Configurations Extensions Index (XRX)	GFX/2D Configurations Extensions Index (XRX)
3D7h	GFX/2D Configurations Extensions Data (XRxx)	GFX/2D Configurations Extensions Data (XRxx)
2D Registers		
3D8h-3D9h	Reserved	Reserved
3DAh	VGA Status Register (ST01)	VGA Feature Control Register (FCR)
3DBh-3DFh	Reserved	Reserved

Indirect VGA and Extended VGA Register Indices

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

2D Sequence Registers (3C4h / 3C5h)

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

2D Graphics Controller Registers (3CEh / 3CFh)

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

Index	Sym	Register Name
11h	GR11	Page Selector
18h	GR18	Software Flags

2D Attribute Controller Registers (3C0h / 3C1h)

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

2D CRT Controller Registers (3B4h / 3D4h / 3B5h / 3D5h)

Index	Sym	Register Name
00h	CR00	Horizontal Total
01h	CR01	Horizontal Display Enable End
02h	CR02	Horizontal Blanking Start
03h	CR03	Horizontal Blanking End
04h	CR04	Horizontal Sync Start
05h	CR05	Horizontal Sync End
06h	CR06	Vertical Total
07h	CR07	Overflow

Index	Sym	Register Name
08h	CR08	Preset Row Scan
09h	CR09	Maximum Scan Line
0Ah	CR0A	Text Cursor Start
0Bh	CR0B	Text Cursor End
0Ch	CR0C	Start Address High
0Dh	CR0D	Start Address Low
0Eh	CR0E	Text Cursor Location High
0Fh	CR0F	Text Cursor Location Low
10h	CR10	Vertical Sync Start
11h	CR11	Vertical Sync End
12h	CR12	Vertical Display Enable End
13h	CR13	Offset
14h	CR14	Underline Location
15h	CR15	Vertical Blanking Start
16h	CR16	Vertical Blanking End
17h	CR17	CRT Mode
18h	CR18	Line Compare
22h	CR22	Memory Read Latch Data

GUC

GuC Introduction

GuC is an embedded micro-controller in the graphics sub-system that is designed to perform graphics workload scheduling on the various graphics parallel engines. In this scheduling model, host software submits work through one of the 256 graphics doorbells and this invokes the micro-kernel running on the GuC core to perform the scheduling operation on the appropriate graphics engine.

Scheduling operations include determining which workload to run next, submitting a workload to a command streamer, pre-empting existing workloads running on an engine, monitoring progress and notifying host SW when work is done. To perform these actions, the GuC requires access to a wide range of assets within the graphics subsystem. The GuC has access to the entire graphics device MMIO register space to allow it to schedule work on any graphics engine.

The code that runs on the GuC is provided by the graphics driver (KMD) during the boot-up and graphics initialization phase. Code provided by the driver is copied from graphics memory and authenticated before execution.

From a functional perspective, the GuC sub-system has the following blocks:

- A Shim block that provides an interface between the micro-controller and rest of the graphics assets.

- An interrupt block that aggregates all the notifications coming from various graphics engines and communicates them to the GuC micro-controller for action. The interrupt block supports (programmable) prioritized delivery of events.
- A DMA engine to allow efficient copy of large blocks of data between memory and internal SRAM. During GuC initialization phase, this DMA engine is available to the host SW to load the GuC micro-kernel. Once the micro-kernel is successfully loaded into GuC, the access to the DMA engine is restricted to the code running on the GuC.
- It also has additional infrastructure to receive notification that are required for scheduling (semaphores from engines, page faults/faults-cleared from Memory interface, etc)
- A GuC power management unit that determines when all the GuC components are idle and supports the power management protocol with the Power Management unit.

Once code is loaded successfully, the primary method of communication with GuC is through the workload doorbells and a GuC/host interrupt mechanism. GuC automatically saves and restores its code image across RC6 power states, so no host intervention is required during these power transitions.

Terminology

Description	Software Use	Must Be Implemented As
Read/Write, R/W	This bit can be read or written.	
Reserved	Do not assume a value for these bits. Writes have no effect.	Writes are ignored. Reads return zero.
Reserved: must be zero, MBZ	Software must always write a zero to these bits. This allows new features to be added using these bits that will be disabled when using old software and as the default case.	Writes are ignored. Reads return zero. Maybe be connected as Read/Write in future projects.
Reserved: PBC, software must preserve contents	Software must write the original value back to this bit. This allows new features to be added using these bits.	Read only or test mode Read/Write.
Read Only	This bit is read only. The read value is determined by hardware. Writes to this bit have no effect.	According to each specific bit. The bit value is determined by hardware and not affected by register writes to the actual bit.
Read/Clear, Read/Write Clear	This bit can be read. Writes to it with a one cause the bit to clear.	Hardware events cause the bit to be set and the bit is cleared on a write operation where the corresponding bit has a one for a value.
Double Buffered	Write when desired. Read gives the unbuffered value (written value) unless specified otherwise. Written values will update to take effect after a certain point. Some have a specific arming sequence where a write to another register is required before the update	Two stages of registers used. First stage is written into and used for readback (unless specified otherwise). First stage value is transferred into second stage at the update point. Second stage value is used to control hardware. Arm/disarm flag for specific arming sequences.

Description	Software Use	Must Be Implemented As
	can take place. This is used to ensure atomic updates of several registers.	

Arming Doorbells

As indicated in the Workload submission section, doorbell rings signal request for work to be submitted to hardware.

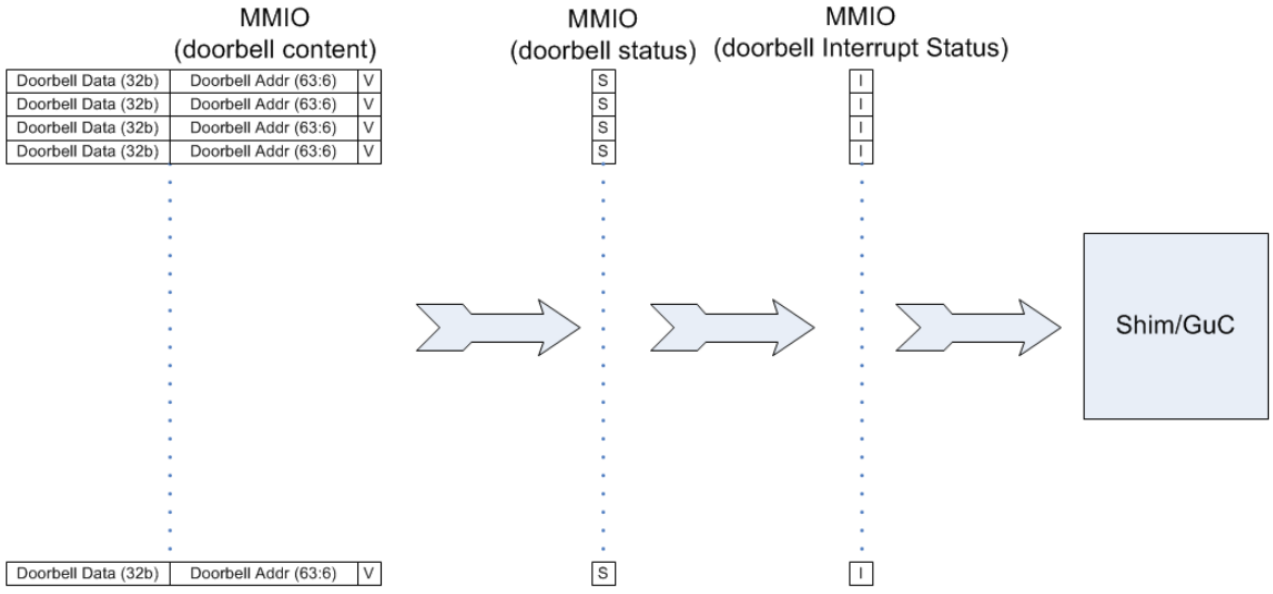
Doorbells need to be configured (armed) before they can be rung. Following sections describe the sequence.

DoorbellStructures

Doorbell structures for reporting have been separated into three logical blocks in the hardware. First stage is where doorbell address is stored along with Valid bit. Once Valid is set (written by KMD/GuC), doorbell block will start monitoring and store the cookie value (DW[0] of the doorbell line).

Once a doorbell is triggered corresponding Status bit is set. Eventually status is propagated to interrupt status bit, meanwhile multiple triggers on doorbell will be collapsed within the status bits.

A read from interrupt status stage to doorbell status stage moves the bits to interrupt status register and clears the doorbell status register.



All stages are exposed via MMIO:

Doorbell Address. Read/Write support for the driver, but thru GuC/Shim (no direct updates).

Valid. Read/Write support however there are restrictions which hardware needs to ensure:

- 0=>1 transition only from GuC via message channel (not allowed for any other message channel client and memory reads of related doorbell does not change the doorbell value from 0=>1.
- 1=>0 transition is only from memory read that doorbell block does when it needs to acquire the doorbell cookie.

Hardware has to guarantee proper clients make the transitions.

Doorbell Status. Read Only (updates are HW managed).

Doorbell Interrupt Status. Read/Write - it is set by hardware and cleared by driver/GuC.

Arming MMIO Based Doorbells

In terms of functionality, MMIO Doorbells are no different from Memory based Doorbell (i.e., Both are used to submit work to the GuC/Hardware). Main differences are in terms of DB arming/Ring protocol. Main motivation for moving to MMIO based Doorbell to not rely on SNOOP/Cache protocol which is not possible for DGPU connected via standard PCIe protocol.

Instead of Memory, 4 MB of MMIO space/region is used for mapping Doorbell. This range will be available starting offset 0x400000h from base of MMIO BAR. Each 4k page of MMIO will be represent as a single DB. Per MMIO BAR, Maximum of 1024 DB allocation possible.

Note:

- For SRIOV 63, we have MMIO BAR per VF.. With each VF BAR has 4 MB range for DB. so per VF Maximum DB allocated is still 1024 but total DB allocated across all VF/PF = 1024 * 64 = 64k DB.

Even though we have 1024 DB allocation, but arming supported for only 256 physical DB. This is done via new 256 DB registers located in SGUNIT(3F_F000h to 3F_F7F8h). Picture below show relationship between 1024 MMIO DB page and 256 Physical DB.

Allocating DB

Steps to allocate DB.

1. On Acquire doorbell, KMD select a "free/available" Doorbell Page (**Page#**) from the 4MB DB space allocated in MMIO BAR.

2. Depending on need of KMD will map that MMIO page in User Mode Process space for application or Ring3 UMD module. OR KMD keep MMIO page mapped in Ring0 space only.

Note: if all DB are allocated and new request came for allocation than KMD will fail that request for simplification. Below is diagram for Allocating DB page.

Arming DB

Steps to ARM DB.

[1] Once DB page was allocated by KMD via above step (DB page Allocation)

[2] KMD calculate '*offset*' from MMIO_BAR BASE (DB page# * 4KB/PAGE_SIZE)

[3] KMD program that '*offset*' *into* "DoorbellTriggerAddressGPA" of UK_GEN11_KM_CONTEXT_DESCRIPTOR

Note: DoorbellTriggerAddressGPA will hold MMIO_OFFSET. So its not GPA but kept same to support backward compatibility of passing GPA in case of Memory based DB.

[4] KMD does similar H2G protocol to request GuC for arm specific physical DB with UK_GEN11_KM_CONTEXT_DESCRIPTOR

[5] GuC FW validate/compare KMD selected physical DB ID for a given VF (i.e., VF is allowed to arm that Physical DB or not).

Note: Needed only for Virtualization based on provision scheme by PF KMD.

[6] GuC FW programs Physical DB register with the VF# (derived from H2G VF#) + MMIO Offset(DoorbellTriggerAddressGPA) + VALID bit.

GuC Shim (GUCSHIM) Register Functions

The GuC Shim provides the interface between GuC and the rest of GT. It is comprised of the various status registers that communicate the current state of the GuC, the infrastructure to setup the address space for GuC operation and interface with message channel.

The following table provides a view of the GuC address space

Address Top	Address Bottom	Space	Description
0xFFFF_FFFF	0xFF80_0000	Graphics MMIO	8 MB off the top of the 4GB space
0xFF80_FFFF	WOPCM_TOP	DRAM - Graphics Memory	Section should be decoded as: **Upper bound:** 4GB - 8MB(Gfx mmio) **Lower bound:**544KB Lower bound accounts for 32KB + 512KB
WOPCM_TOP-1	0x0008_8000	DRAM - WOPCM	Write Only Protected Content Memory (WO-PCM) This allows for code to straddle SRAM and memory (as described later) WOPCM_TOP = GUC_WOPCM_SIZE
0x0008_7FFF	0x0000_8000	SRAM space	512 KB SRAM This gets loaded with the GuC micro-kernel. The GuC may also use portion of the SRAM for its data, stack, and other required components.
0x0000_7FFF	0x0000_0000	Boot ROM	32 KB of BootROM for Initialization and authentication code that the GuC first jumps to is located here.

When using the GuC DMA engine to load the HuC uKernel, the status can be obtained by reading:

- GuC's BOOT_HASH_CHK register (0xc010 bit 8) to see if a HuC uKernel loading had been attempted, and
- HuC's HUC_STATUS2 register (0xd3b0 bit 7) to check whether or not the HuC uKernel was successfully loaded.

GUCSHIM Registers

Register
GUC_STATUS - Global MicroController Status
JMP_DEST - Jump Location
MIA_FORCE_FENCE - Minute IA Force Fence
MIA_INV_TLB - Minute IA Force TLB Invalidate
SOFT_SCRATCH - Soft Scratch
UOS_RSA_SCRATCH - RSA for uOS/Soft Scratch

Guc DMA (GUCDMA)

The DMA engine allows the MinuteIA core to move data back and forth efficiently from the various memory segments listed below. Note that the DMA engine supports more than current required usage models. Memory segments supported:

- Global GTT mapped memory
- Per Process GTT mapped memory
- WOPCM
- SRAM

The MinIA is a 32 bit engine so it cannot generate an address greater than 4 GB. Thus, any data that the MinIA core has to access must be located <4 GB in the graphics address space. The graphics address generated by the MinIA core goes through the regular graphics page table walk to derive the physical address that can be above 4 GB.

The DMA engine supports the full 48 bit addressing so it can be used by the MinIA core to get to the address regions above 4 GB. The MinIA core programs this DMA engine through registers that are mapped into the Gfx MMIO.

The DMA Registers

GUCDMA uses two 64-bit registers (4 DWord registers) to indicate the 48-bit Source and Destination addresses along with fetch type indication etc. Bit 1 of the DMA Control Register (described later) pins DMA Address Register 0 to Source addressing or Destination, and vice versa for DMA Address Register 1. By default, DMA Address Register 0 is assigned to Source addressing and DMA Address Register 1 to Destination addressing.

Register
DMA_ADDR_0_LOW - DMA Address Register 0 Low
DMA_ADDR_0_HIGH - DMA Address Register 0 High
DMA_ADDR_1_LOW - DMA Address Register 1 Low
DMA_ADDR_1_HIGH - DMA Address Register 1 High
DMA_COPY_SIZE - DMA Copy Size
DMA_CFG - DMA Configuration
DMA_CTRL - DMA Control

Programming Note	
Context:	The DMA Registers

Notes:

- The DMA engine can be deactivated by setting the Disable-GuC fuse. If this fuse is set, the DMA engine is rendered inoperable, so it cannot be used to load a GuC uOS or move any data. On a product that intends to use GuC, this fuse shall be zero.
- The lower 6 bits of *addressing* of both Source and Destination addresses must be same. There is no provision for barrel rotation across byte locations, during a DMA Transfer.
- The following restrictions shall be followed for placement of uOS and uApps:
  - uOS and uApps are always located on a 64-byte aligned address.
- uApps are not automatically loaded into SRAM by HW. uKernel must explicitly copy a uApp into SRAM from WOPCM when using it.
- Once the ukernel and the uApps have been loaded successfully into the WOPCM area, the HW shall not allow DMA operation to overwrite them.
- Before programming the DMA engine to access memory in the Per Process GTT address space, GuC SW must setup the PPGTT by programming registers: These registers are located in the GUC_PM unit (offsets: 0xC3B8 - 0xC3F0):
  - CTXT_INFO
  - PDP0, PDP1, PDP2, PDP3
  - PPGTT_ENABLE
- The GuC DMA engine also provides support for loading the micro-kernel. To load a third party ukernel, the third party GuC ukernel must be loaded first. An authenticated GuC ukernel can then be invoked to load a ukernel. (If a third party ukernel is loaded first, there is no way to clear the ME_DATA registers - thus locking out the ability to load a third party GuC).
- GuC DMA HW checks for the following illegal cases and rejects the DMA invocation (DMA will not happen):

Illegal Case
GuC WOPCM Base & GuC WOPCM Size is not programmed (for copy to/from WOPCM) 0xC050 and 0xC340
DMA copy into GuC WOPCM that does not fit into the GuC WOPCM
DMA copy into SRAM that falls off the SRAM edge (except for uKernel copy)
DMA size is set to 0

GuC Interrupt (GUCINT) Register Functions

This section discusses the register functions for GuC Interrupt. Registers in this section are:

Functionality
Interrupt Pin Assignment Registers
Doorbell Group Registers
Engine Interrupt Registers
GuC Timer Registers
GuC DMA Interrupt Registers
GuC Host Registers

Interrupt Overview

Core inside the GUC supports up to 32 interrupts in total which consists of SW Generated Interrupts, Internal Timers, and External Interrupts. External Interrupts to the core are line-based interrupts. Core supports up to five levels of interrupt priority and each interrupt can be assigned an interrupt priority level. Interrupt table below shows the distribution of interrupt allocation and the assigned interrupt priority level, this allocation is static and hardwired into the hardware logic during processor build time.

Interrupt Table and Priority Assignments By Row

Interrupt Bit	Core External Interrupt Pin	Interrupt Type	Quantity	Priority Level
0	NA	SW Interrupt-0	1	L1
1	NA	SW Interrupt-1	1	L1
2	NA	Timer0 (Generated Internal to Core)	1	L1
3	NA	Profiling Interrupt (Generated Internal to Core)	1	L1
4..16	0..12	External Interrupts (Input Pins to Core)	13	L1
17..21	13..17	External Interrupts (Input Pins to Core)	5	L2
22..26	18..22	External Interrupts (Input Pins to Core)	5	L3
27..29	23..25	External Interrupts	3	L4

Interrupt Bit	Core External Interrupt Pin	Interrupt Type	Quantity	Priority Level
		(Input Pins to Core)		
30..31	26..27	External Interrupts (Input Pins to Core)	2	L5

Core coming out of reset will not be in a state to handle interrupts until it gets initialized to appropriate state. Firmware will explicitly set "Core Interrupt Ready" bit in register when the core is ready to accept interrupts until then GUC will withhold generating any interrupts to the core and keep accumulating the interrupts. GUC will fire all the pending interrupts to the core once the "Core Interrupt Ready" is asserted.

Interrupt agents to the GUC generate different interrupts based on their associated functionality. Since there are limited number of interrupt pins available on the core interface, the number of interrupts supported across the interrupt agents are logically grouped inside the GUC called Interrupt Groups 0..16 and represented as 17 external interrupts to the core. GUCINT provides "Interrupt Pin Assignment 0..4" registers to map each of these interrupt to any given "External Interrupt" pin on the core interface, this gives flexibility for the SW to move around a given interrupt on the "External Interrupt" interface of the core to get the desired priority level. Table below shows the 17 interrupt groups and their assigned index. "Interrupt Pin Assignment" registers have a field for each of the interrupt group against which SW must program the "External Interrupt" pin value to get the desired interrupt priority for a given interrupt.

Mapping of more than one interrupt to the same "External Interrupt" pin will result in logical "OR" of the corresponding interrupts connected to the "External Interrupt" pin. Default programing in the "Interrupt Pin Assignment 0..4" is set to "1F" disabling any interrupt generation on the external interrupt pins of the core.

[Register] Interrupt Pin Assignment 0

[Register] Interrupt Pin Assignment 1

[Register] Interrupt Pin Assignment 2

[Register] Interrupt Pin Assignment 3

[Register] Interrupt Pin Assignment 4

Interrupt Groups and Index Assignment

Interrupt Group	Interrupt Group Details
0	Door Bell Group1
1	Door Bell Group2
2	Door Bell Group3
3	Door Bell Group4
4	Door Bell Group5
5	Door Bell Group6
6	Door Bell Group7
7	Door Bell Group8
8	Semaphores
9	Display
10	Engine DW0
11	Engine DW1
12	DMA,Timer, InterGuC Messaging, Guc FLR Request, GuC PM FLush Request, Miscellaneous Int
13	Host Interrupt (PF)
14	Host Interrupt VF 1..31
15	Host Interrupt VF 32..63
16	IOMMU

Doorbell Group Registers

There is a Doorbell Control Register in GUCint, which has a single doorbell_rung bit for each one of the eight Doorbell Registers located in GTI (1900 - 191C). GTI sets this GuC Register bit when the corresponding Doorbell register GTI houses, has that GTI register value going from all 0s to having any one bit set (any one of the 32 doorbells in that doorbell group gets rung).

GTI could set multiple first_doorbell_rung bits in a single message to the GuC based Doorbell Control Register (corresponding to several doorbells rung in different doorbell group registers in GTI). Once a doorbell_rung bit is set for a group in the Doorbell Control Register, it is not updated until GUCINT reads the corresponding GTI doorbell register, at which time the corresponding rung_bit is reset.

The doorbell control register also holds 1 bit (send to Mini-Core) that routes interrupts to host or Mini-Core. By default, interrupts are sent to host. This bit must be set by software (running on Mini-Core or host) to route interrupts to Mini-Core.

GUC_DB_ISR_7 - GuC Doorbell Group 7 Interrupt Status

GUC_DB_ISR_6 - GuC Doorbell Group 6 Interrupt Status

GUC_DB_ISR_5 - GuC Doorbell Group 5 Interrupt Status

GUC_DB_ISR_4 - GuC Doorbell Group 4 Interrupt Status

GUC_DB_ISR_3 - GuC Doorbell Group 3 Interrupt Status

GUC_DB_ISR_2 - GuC Doorbell Group 2 Interrupt Status

GUC_DB_ISR_1 - GuC Doorbell Group 1 Interrupt Status

GUC_DB_ISR_0 - GuC Doorbell Group 0 Interrupt Status

DOORBELL_CTRL - Doorbell Control

Engine Interrupt Registers

GUC gets Engine Event interrupts from various engines (Render, Copy, Compute, Video Decode, Video Enhancement.).

GUCINT also gets Engine Event interrupts from OA.

The engines support a variety of interrupts that may not be interesting to GuC from a scheduling point of view. GUCINT provides an infrastructure to redirect engine interrupts to the host driver without invoking the Mini-Core firmware. This infrastructure allows software to specify on a per engine and per interrupt granularity the interrupts that must be delivered to Mini-Core or simply forwarded to the host (bypassing the MiniCore).

Command Streamer Status Information

During execution, the Command Streamer Status is sent to the GuC.

Programming Note	
Context:	Context Status Buffer Initialization
GuC CSB FIFO's are implemented on device reset domain, it's possible following GFX Reset (All engines and GuC are reset) there are unprocessed entries present in the engine CSB FIFO's. GuC FW as part of the GuC initialization flow must ensure the engine CSB FIFOs are drained and empty before scheduling contexts to the engines.	

CSB Read Port

The following RO registers are for use by GuC FW or host. SW must read twice to obtain a single CSB entry: the first read returns bits[31:0]; the second read returns [63:32].

CS CSB

BCS CSB

VCS CSB

VECS CSB

CSB FIFO Status Registers

The following RO registers hold the status of each Command Streamer's CSB FIFO:

CS CSB Fifo Status Register

BCS CSB Fifo Status Register

VCS CSB Fifo Status Register

VECS CSB Fifo Status Register

Guc DMA Interrupt Registers

GUC_DMA_IIR - GuC DMA Interrupt Input

The DMA generates this message interrupt at the completion of a programmed DMA transfer.

Guc Host Registers

GuC and Host(IA) communicate with each other through interrupts.

- A Host-to-GUC interrupt is generated by Host SW writing to 0xC4C8. The written data will get stored in 0xC590 and an interrupt will be generated to GuC.
- A GUC-to-Host interrupt is generated by GuC FW writing to 0xC4B8 - this generates a 16bit vector to the host (this 16b vector is shared by GuC HW and GuC FW. FW write can only set FW owned bits)

GUC_HOST_INTR_IIR - GuC Host Interrupt Input

Observability

Observability Overview

As GFX-enabled systems and usage models have grown in complexity over time, a number of hardware features have been added to provide more insight into hardware behavior while running a commercially available operating system. This chapter documents these features with pointers to relevant sections in other chapters. Supported observability features include:

Feature
Performance counters
Internal node tracing

Note: This chapter describes the registers and instructions used to monitor GPU performance. Please review other volumes in this specification to understand the terms, functionality, and details for specific Intel graphics devices.

DFD Configuration Restore

Since DFD logic does not usually add value to end user usage models and its configuration space is large (which would add latency to power management restore flows), it is typically not enabled during normal operation for optimal power & performance. Hence, additional steps are required when DFD functionality is needed in combination with system configurations where GT logic loses power/is reset. The basic strategy per scenario is detailed below.

GT Power-up/RC6 Exit

Strategy
Replicate failure without power management
Configure the DFD restore feature

Render Engine Power-up

Configure the RCS RC6 W/A batch buffer to restore render engine DFD configuration ONLY.

Media Engine Power-up

Configure the applicable media command stream W/A batch buffer to restore media engine DFD configuration ONLY.

Resume From Partial GT Power Down

For cases where SW is aware of power well state, re-apply DFD configuration.

For cases where SW is not aware of power well state, configure the per-context W/A batch buffer to apply the DFD configuration on every context load.

Trace

This section contains the following contents:

Feature
• Performance Visibility

Performance Visibility

Motivation For Hardware-Assisted Performance Visibility

As the focus on GFX performance and programmability has increased over time, the need for hardware (HW) support to rapidly identify bottlenecks in HW and efficiently tune the work sent to same has become correspondingly important. This describes the HW support for Performance Visibility.

Performance Event Counting

An earlier generation introduced dedicated GFX performance counters to address key issues associated with existing chipset CHAPs counters (lack of synchronization with GFX rendering work and low sampling frequency achievable when sampling via CPU MMIO read). Furthermore, reliance on SoC assets created a cross-IP dependency that was difficult to manage well. Hence, the approach since that earlier generation has been to use dedicated counters managed by the graphics device driver for graphics performance

measurement. The dedicated counter values are written to memory whenever an MI_REPORT_PERF_COUNT command is placed in the ring buffer.

While this approach eliminated much of the error associated with the previous approaches, it is still limited to sampling the counters only at the boundaries between ring commands. This inherently limited the ability of performance analysis tools to drill down into a primitive, which can contain thousands of triangles and require several hundreds of milliseconds to render. It is further worth noting that precise sampling via MI_REPORT_PERF command requires flushing the GFX pipeline before and after the work of interest. The overhead of flushing the GFX pipeline can become large if the work of interest is small, hence reducing the accuracy of the performance counter measurement. In such situations, the flush can be removed, or internally triggered reporting can be used with some resulting loss of precision in which draws/dispatches are being profiled.

Additionally, Intel design and architecture teams found that the existing silicon-based performance analysis tools provided only a general idea of where a problem may exist but were not able to pinpoint a problem. This was generally because the counter values are integrated across a very large time period, washing out the dynamic behavior of the workload.

All OA config registers are tied to GT global reset and hence are not affected by per-engine resets (e.g. render only reset).

OA Programming Guidelines

SW utilizing OA HW is expected to monitor the overflow/lost report status for the OABUFFER and respond as appropriate for the active usage model.

In order for OA counters to increment the 'Counter Stop-Resume Mechanism' bit of the OACTXCONTROL register must be set. This requires a RCS context with this bit set be loaded, and either RCS force wake be enabled or the RCS context be left active for the duration of the window this counter is needed for.

In general, OA is effectively unable to count between the power context save that happens prior to GFX entering RC6 and the power context restore that occurs on the next RC6 exit. This limitation results from the fact that the counters themselves are power context save/restored and hence the counts that (may) have accumulated in this time window are overwritten by the saved values that are read back from the power context save area. An example of the kind of information that can be missed is the GTI traffic resulting from the power context save of OA itself. The size of this performance counting blind spot is microarchitecturally minimized as much as reasonably possible but still varies from device to device.

Legacy OACS functionality is now logically split into two functions called OAG (OA Global) and OAR (OA Render). Summary of the blocks is as follows:

OAG:

- Handles OA buffer and timer/internally triggered sampling.
- Is unaffected by engine reset / power well status.
- Is inaccessible by non-privileged batch buffers but accessible by all command streamers / GuC / CPU.
- Implements free-running utilization counters.

- Is GT power context save/restored.
- Is only allowed to access global GTT memory.

OAR:

- Is expected to behave as a part of the render engine from a clocking/power well/reset perspective.
- Implements MI_REPORT_PERF command.
- Is render context save/restored, making all values reported by MI_REPORT_PERF per-context.
- Must be initialized to power-on default values as part of RCS golden context creation (please refer to RCS section describing golden context creation for full details) or implementation-specific undesirable behavior may occur.
- Doesn't support timer/internally triggered sampling.
- Can be enabled/disabled independent from OAG.
- Is only intended to be accessed by RCS, access from other command streamers / CPU may have implementation-specific negative side-effects.

OA Virtualization

Memory Address Space Considerations:

- Customer feedback has identified global (context insensitive) performance sampling as needed for load-balancing by a VMM/node manager. Given this request and the consideration that true system-level visibility is technically not allowed from a VM's perspective, global GTT OA requests shall be treated as coming from VF0. Global GTT accesses from OA that are "out of bounds" for VF0 shall be dropped in an implementation specific non-fatal way
- Current MDAPI driver support is critically dependent on the ability to issue performance counter queries from application / UMD code and return the performance counter data read into a user space memory location.
- The configuration of inputs to the B/C counter event blocks is global such that all VMs must derive B/C counter events from the same set of inputs. However, the Boolean equations configured in B-counter controls are per-VM.
- A-counter configuration (e.g. flex EU event configuration) is per context and hence specific to a VM.

Programming Note	
Context:	PPGTT Handling
OAG is expected to be used by host OS SW to perform global load-balancing / GFX utilization monitoring. OAR is part of render context and hence will be accessible / functional from the perspective of MDAPI driver running within a VM. Since the counts on B/C counters are context-save/restored, the activity on other VMs / contexts is no longer visible to any individual VM.	

HW Support

This section contains various reporting counters and registers for hardware support for Performance Visibility.

Performance Counter Report Formats

Counters layout for various values of select from the register:

OAR Report Format (Counter Select = 0b101):

A-Cntr 3 (low dword)	A-Cntr 2 (low dword)	A-Cntr 1 (low dword)	A-Cntr 0 (low dword)	GPU_TICKS	CTX ID	TIME_STAMP	RPT_ID
A-Cntr 11 (low dword)	A-Cntr 10 (low dword)	A-Cntr 9 (low dword)	A-Cntr 8 (low dword)	A-Cntr 7 (low dword)	A-Cntr 6 (low dword)	A-Cntr 5 (low dword)	A-Cntr 4 (low dword)
A-Cntr 19 (low dword)	A-Cntr 18 (low dword)	A-Cntr 17 (low dword)	A-Cntr 16 (low dword)	A-Cntr 15 (low dword)	A-Cntr 14 (low dword)	A-Cntr 13 (low dword)	A-Cntr 12 (low dword)
A-Cntr 27 (low dword)	A-Cntr 26 (low dword)	A-Cntr 25 (low dword)	A-Cntr 24 (low dword)	A-Cntr 23 (low dword)	A-Cntr 22 (low dword)	A-Cntr 21 (low dword)	A-Cntr 20 (low dword)
A-Cntr 35 (low dword)	A-Cntr 34 (low dword)	A-Cntr 33 (low dword)	A-Cntr 32 (low dword)	A-Cntr 31 (low dword)	A-Cntr 30 (low dword)	A-Cntr 29 (low dword)	A-Cntr 28 (low dword)
High bytes of A31-A28	High bytes of A27-A24	High bytes of A23-A20	High bytes of A19-A16	High bytes of A15-A12	High bytes of A11-A8	High bytes of A7-A4	High bytes of A3-A0
B-Cntr 7	B-Cntr 6	B-Cntr 5	B-Cntr 4	B-Cntr 3	B-Cntr 2	B-Cntr 1	B-Cntr 0
C-Cntr 7	C-Cntr 6	C-Cntr 5	C-Cntr 4	C-Cntr 3	C-Cntr 2	C-Cntr 1	C-Cntr 0

OAG Report Format (Counter Select = 0b101)

A-Cntr 3 (low dword)	A-Cntr 2 (low dword)	A-Cntr 1 (low dword)	A-Cntr 0 (low dword)	GPU_TICKS	CTX ID	TIME_STAMP	RPT_ID
A-Cntr 11 (low dword)	A-Cntr 10 (low dword)	A-Cntr 9 (low dword)	A-Cntr 8 (low dword)	A-Cntr 7 (low dword)	A-Cntr 6 (low dword)	A-Cntr 5 (low dword)	A-Cntr 4 (low dword)
A-Cntr 19 (low dword)	A-Cntr 18 (low dword)	A-Cntr 17 (low dword)	A-Cntr 16 (low dword)	A-Cntr 15 (low dword)	A-Cntr 14 (low dword)	A-Cntr 13 (low dword)	A-Cntr 12 (low dword)
A-Cntr 27 (low dword)	A-Cntr 26 (low dword)	A-Cntr 25 (low dword)	A-Cntr 24 (low dword)	A-Cntr 23 (low dword)	A-Cntr 22 (low dword)	A-Cntr 21 (low dword)	A-Cntr 20 (low dword)
A-Cntr 35 (low dword)	A-Cntr 34 (low dword)	A-Cntr 33 (low dword)	A-Cntr 32 (low dword)	A-Cntr 31 (low dword)	A-Cntr 30 (low dword)	A-Cntr 29 (low dword)	A-Cntr 28 (low dword)
High bytes of A31-A28	A-Cntr 37 (low dword)	High bytes of A23-A20	High bytes of A19-A16	High bytes of A15-A12	High bytes of A11-A8	High bytes of A7-A4	A-Cntr 36 (low dword)
B-Cntr 7	B-Cntr 6	B-Cntr 5	B-Cntr 4	B-Cntr 3	B-Cntr 2	B-Cntr 1	B-Cntr 0
C-Cntr 7	C-Cntr 6	C-Cntr 5	C-Cntr 4	C-Cntr 3	C-Cntr 2	C-Cntr 1	C-Cntr 0

OAM Report Format (Counter Select = 0b101):

A-Cntr 37		A-Cntr 36		GPU_TICKS	CTX ID	TIME_STAMP	RPT_ID
B-Cntr 7	B-Cntr 6	B-Cntr 5	B-Cntr 4	B-Cntr 3	B-Cntr 2	B-Cntr 1	B-Cntr 0
C-Cntr 7	C-Cntr 6	C-Cntr 5	C-Cntr 4	C-Cntr 3	C-Cntr 2	C-Cntr 1	C-Cntr 0

OAR Report Format (Counter Select = 0b001):

GPU_TICKS (high dword)	GPU_TICKS (low dword)	CTX ID (high dword) reserved	CTX ID (low dword)	TIME_STAMP (high dword)	TIME_STAMP (low dword)	RPT_ID (high dword)	RPT_ID (low dword)
A-Cntr 3 (high dword)	A-Cntr 3 (low dword)	A-Cntr 2 (high dword)	A-Cntr 2 (low dword)	A-Cntr 1 (high dword)	A-Cntr 1 (low dword)	A-Cntr 0 (high dword)	A-Cntr 0 (low dword)
A-Cntr 7 (high dword)	A-Cntr 7 (low dword)	A-Cntr 6 (high dword)	A-Cntr 6 (low dword)	A-Cntr 5 (high dword)	A-Cntr 5 (low dword)	A-Cntr 4 (high dword)	A-Cntr 4 (low dword)
A-Cntr 11 (high dword)	A-Cntr 11 (low dword)	A-Cntr 10 (high dword)	A-Cntr 10 (low dword)	A-Cntr 9 (high dword)	A-Cntr 9 (low dword)	A-Cntr 8 (high dword)	A-Cntr 8 (low dword)
A-Cntr 15 (high dword)	A-Cntr 15 (low dword)	A-Cntr 14 (high dword)	A-Cntr 14 (low dword)	A-Cntr 13 (high dword)	A-Cntr 13 (low dword)	A-Cntr 12 (high dword)	A-Cntr 12 (low dword)
A-Cntr 19 (high dword)	A-Cntr 19 (low dword)	A-Cntr 18 (high dword)	A-Cntr 18 (low dword)	A-Cntr 17 (high dword)	A-Cntr 17 (low dword)	A-Cntr 16 (high dword)	A-Cntr 16 (low dword)
A-Cntr 23 (high dword)	A-Cntr 23 (low dword)	A-Cntr 22 (high dword)	A-Cntr 22 (low dword)	A-Cntr 21 (high dword)	A-Cntr 21 (low dword)	A-Cntr 20 (high dword)	A-Cntr 20 (low dword)
A-Cntr 27 (high dword)	A-Cntr 27 (low dword)	A-Cntr 26 (high dword)	A-Cntr 26 (low dword)	A-Cntr 25 (high dword)	A-Cntr 25 (low dword)	A-Cntr 24 (high dword)	A-Cntr 24 (low dword)
A-Cntr 31 (high dword)	A-Cntr 31 (low dword)	A-Cntr 30 (high dword)	A-Cntr 30 (low dword)	A-Cntr 29 (high dword)	A-Cntr 29 (low dword)	A-Cntr 28 (high dword)	A-Cntr 28 (low dword)
A-Cntr 35 (high dword)	A-Cntr 35 (low dword)	A-Cntr 34 (high dword)	A-Cntr 34 (low dword)	A-Cntr 33 (high dword)	A-Cntr 33 (low dword)	A-Cntr 32 (high dword)	A-Cntr 32 (low dword)
B-Cntr 7	B-Cntr 6	B-Cntr 5	B-Cntr 4	B-Cntr 3	B-Cntr 2	B-Cntr 1	B-Cntr 0
C-Cntr 7	C-Cntr 6	C-Cntr 5	C-Cntr 4	C-Cntr 3	C-Cntr 2	C-Cntr 1	C-Cntr 0

OAC Report Format (Counter Select = 0b001):

GPU_TICKS (high dword)	GPU_TICKS (low dword)	CTX ID (high dword) reserved	CTX ID (low dword)	TIME_STAMP (high dword)	TIME_STAMP (low dword)	RPT_ID (high dword)	RPT_ID (low dword)
A-Cntr 8 (high dword)	A-Cntr 8 (low dword)	A-Cntr 7 (high dword)	A-Cntr 7 (low dword)	A-Cntr 4 (high dword)	A-Cntr 4 (low dword)	A-Cntr 0 (high dword)	A-Cntr 0 (low dword)
A-Cntr 12 (high dword)	A-Cntr 12 (low dword)	A-Cntr 11 (high dword)	A-Cntr 11 (low dword)	A-Cntr 10 (high dword)	A-Cntr 10 (low dword)	A-Cntr 9 (high dword)	A-Cntr 9 (low dword)
A-Cntr 16 (high dword)	A-Cntr 16 (low dword)	A-Cntr 15 (high dword)	A-Cntr 15 (low dword)	A-Cntr 14 (high dword)	A-Cntr 14 (low dword)	A-Cntr 13 (high dword)	A-Cntr 13 (low dword)
A-Cntr 20 (high dword)	A-Cntr 20 (low dword)	A-Cntr 19 (high dword)	A-Cntr 19 (low dword)	A-Cntr 18 (high dword)	A-Cntr 18 (low dword)	A-Cntr 17 (high dword)	A-Cntr 17 (low dword)
A-Cntr 31 (high dword)	A-Cntr 31 (low dword)	A-Cntr 30 (high dword)	A-Cntr 30 (low dword)	A-Cntr 29 (high dword)	A-Cntr 29 (low dword)	A-Cntr 28 (high dword)	A-Cntr 28 (low dword)
A-Cntr 35 (high dword)	A-Cntr 35 (low dword)	A-Cntr 34 (high dword)	A-Cntr 34 (low dword)	0	0	A-Cntr 32 (high dword)	A-Cntr 32 (low dword)
B-Cntr 7	B-Cntr 6	B-Cntr 5	B-Cntr 4	B-Cntr 3	B-Cntr 2	B-Cntr 1	B-Cntr 0
C-Cntr 7	C-Cntr 6	C-Cntr 5	C-Cntr 4	C-Cntr 3	C-Cntr 2	C-Cntr 1	C-Cntr 0

OAG Report Format (Counter Select = 0b001):

GPU_TICKS (high dword)	GPU_TICKS (low dword)	CTX ID (high dword) reserved	CTX ID (low dword)	TIME_STAMP (high dword)	TIME_STAMP (low dword)	RPT_ID (high dword)	RPT_ID (low dword)
A-Cntr 3 (high dword)	A-Cntr 3 (low dword)	A-Cntr 2 (high dword)	A-Cntr 2 (low dword)	A-Cntr 1 (high dword)	A-Cntr 1 (low dword)	A-Cntr 0 (high dword)	A-Cntr 0 (low dword)
A-Cntr 7 (high dword)	A-Cntr 7 (low dword)	A-Cntr 6 (high dword)	A-Cntr 6 (low dword)	A-Cntr 5 (high dword)	A-Cntr 5 (low dword)	A-Cntr 4 (high dword)	A-Cntr 4 (low dword)
A-Cntr 11 (high dword)	A-Cntr 11 (low dword)	A-Cntr 10 (high dword)	A-Cntr 10 (low dword)	A-Cntr 9 (high dword)	A-Cntr 9 (low dword)	A-Cntr 8 (high dword)	A-Cntr 8 (low dword)
A-Cntr 15 (high dword)	A-Cntr 15 (low dword)	A-Cntr 14 (high dword)	A-Cntr 14 (low dword)	A-Cntr 13 (high dword)	A-Cntr 13 (low dword)	A-Cntr 12 (high dword)	A-Cntr 12 (low dword)
A-Cntr 19 (high dword)	A-Cntr 19 (low dword)	A-Cntr 18 (high dword)	A-Cntr 18 (low dword)	A-Cntr 17 (high dword)	A-Cntr 17 (low dword)	A-Cntr 16 (high dword)	A-Cntr 16 (low dword)
A-Cntr 23 (high dword)	A-Cntr 23 (low dword)	A-Cntr 22 (high dword)	A-Cntr 22 (low dword)	A-Cntr 21 (high dword)	A-Cntr 21 (low dword)	A-Cntr 20 (high dword)	A-Cntr 20 (low dword)

A-Cntr 27 (high dword)	A-Cntr 27 (low dword)	A-Cntr 26 (high dword)	A-Cntr 26 (low dword)	A-Cntr 25 (high dword)	A-Cntr 25 (low dword)	A-Cntr 24 (high dword)	A-Cntr 24 (low dword)
A-Cntr 31 (high dword)	A-Cntr 31 (low dword)	A-Cntr 30 (high dword)	A-Cntr 30 (low dword)	A-Cntr 29 (high dword)	A-Cntr 29 (low dword)	A-Cntr 28 (high dword)	A-Cntr 28 (low dword)
A-Cntr 35 (high dword)	A-Cntr 35 (low dword)	A-Cntr 34 (high dword)	A-Cntr 34 (low dword)	A-Cntr 33 (high dword)	A-Cntr 33 (low dword)	A-Cntr 32 (high dword)	A-Cntr 32 (low dword)
				A-Cntr 37 (high dword)	A-Cntr 37 (low dword)	A-Cntr 36 (high dword)	A-Cntr 36 (low dword)
B-Cntr 7	B-Cntr 6	B-Cntr 5	B-Cntr 4	B-Cntr 3	B-Cntr 2	B-Cntr 1	B-Cntr 0
C-Cntr 7	C-Cntr 6	C-Cntr 5	C-Cntr 4	C-Cntr 3	C-Cntr 2	C-Cntr 1	C-Cntr 0

OAM Report Format (Counter Select = 0b001):

GPU_TICKS (high dword)	GPU_TICKS (low dword)	CTX ID (high dword) reserved	CTX ID (low dword)	TIME_STAMP (high dword)	TIME_STAMP (low dword)	RPT_ID (high dword)	RPT_ID (low dword)
				A-Cntr 37 (high dword)	A-Cntr 37 (low dword)	A-Cntr 36 (high dword)	A-Cntr 36 (low dword)
B-Cntr 7	B-Cntr 6	B-Cntr 5	B-Cntr 4	B-Cntr 3	B-Cntr 2	B-Cntr 1	B-Cntr 0
C-Cntr 7	C-Cntr 6	C-Cntr 5	C-Cntr 4	C-Cntr 3	C-Cntr 2	C-Cntr 1	C-Cntr 0

OAM Report Format (Counter Select = 0b001):

GPU_TICKS (high dword)	GPU_TICKS (low dword)	CTX ID (high dword) reserved	CTX ID (low dword)	TIME_STAMP (high dword)	TIME_STAMP (low dword)	RPT_ID (high dword)	RPT_ID (low dword)
				A-Cntr 37 (high dword)	A-Cntr 37 (low dword)	A-Cntr 36 (high dword)	A-Cntr 36 (low dword)
B-Cntr 7	B-Cntr 6	B-Cntr 5	B-Cntr 4	B-Cntr 3	B-Cntr 2	B-Cntr 1	B-Cntr 0
C-Cntr 7	C-Cntr 6	C-Cntr 5	C-Cntr 4	C-Cntr 3	C-Cntr 2	C-Cntr 1	C-Cntr 0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

OAC Report Format (Counter Select = 0b010):

GPU_TICKS (high dword)	GPU_TICKS (low dword)	CTX ID (high dword) reserved	CTX ID (low dword)	TIME_STAMP (high dword)	TIME_STAMP (low dword)	RPT_ID (high dword)	RPT_ID (low dword)
A-Cntr 12 (low dword)	A-Cntr 11 (low dword)	A-Cntr 10 (low dword)	A-Cntr 9 (low dword)	A-Cntr 8 (low dword)	A-Cntr 7 (low dword)	A-Cntr 4 (low dword)	A-Cntr 0 (low dword)
A-Cntr 20 (low dword)	A-Cntr 19 (low dword)	A-Cntr 18 (low dword)	A-Cntr 17 (low dword)	A-Cntr 16 (low dword)	A-Cntr 15 (low dword)	A-Cntr 14 (low dword)	A-Cntr 13 (low dword)
reserved	reserved	A-Cntr 35 (low dword)	A-Cntr 34 (low dword)	0	A-Cntr 32 (low dword)	A-Cntr 31 (low dword)	A-Cntr 30 (low dword)

B-Cntr 7	B-Cntr 6	B-Cntr 5	B-Cntr 4	B-Cntr 3	B-Cntr 2	B-Cntr 1	B-Cntr 0
C-Cntr 7	C-Cntr 6	C-Cntr 5	C-Cntr 4	C-Cntr 3	C-Cntr 2	C-Cntr 1	C-Cntr 0

Description of RPT_ID and other important fields of the layout:

Field	Description
GPU TICKS[31:0]	GPU_TICKS is simply a free-running count of render clocks elapsed used for normalizing other counters (e.g., EU active time), it is expected that the rate that this value advances will vary with frequency and freeze (but not lose its value) when all GT clocks are gated, GT is in RC6, and so on.
Context ID[31:0]	This field carries the Context ID of the active context in render engine. [31:0]: Context ID in Execlist mode of scheduling.
TIME_STAMP[55:32]	Upper bits 55:32 of TIME_STAMP. This field provides an elapsed real-time value that can be used as a timestamp for GPU events over short periods of time.
TIME_STAMP[31:0]	This field provides an elapsed real-time value that can be used as a timestamp for GPU events over short periods of time.
RPT_ID[47]	Status bit (once set) to indicate delayed report.
RPT_ID[46:38]	Reserved (for future use)
RPT_ID[37:36]	For OAC: CCS Selected ID for perf mon For all other OA units: Reserved
RPT_ID[35:34]	Reserved (for future Tile IDs)
RPT_ID[33:32]	Tile ID (Field applies to multi-tile configs only)
RPT_ID[31:0]	This field has several sub fields as defined below: 31:26 **SourceID[5:0]** Encoded value to identify various sources like any CS or Shader unit from which the Report was requested. **Programming note:** 25:19 **Report Reason[6:0]** Report_reason[0]: When set indicates current report is due to "Timer Triggered". Report_reason[1]: When set indicates current report is due to "Internal report trigger 1". Report_reason[2]: When set indicates current report is due to "Internal report trigger 2". Report_reason[3]: When set indicates current report is due to "Context switch". Report_reason[4]: When set indicates current report is due to "GO transition from '1' to '0' ". Report_reason[5]: When set indicates current report is due to a change in unslice/slice

Field	Description	
		ratio Report_reason[6]: When set indicates current report is due to a MMIO Trigger **Programming note:**
	18	**Start Trigger Event:** This bit is multiplexed from "Start Trigger Event-1" or "Start Trigger Event-2" based on the "Internal Report Trigger-1" or "Internal Report Trigger-2" asserted in the Report Reason respectively. "Internal Report Trigger-1" is given priority over "Internal Report Trigger-2". By default Start Trigger Event-1 is outputted.
	17	**Threshold Enable:** This bit is multiplexed from "Report Trigger Threshold Enable-1" or "Report Trigger Threshold Enable-2" based on the "Internal Report Trigger-1" or "Internal Report Trigger-2" asserted in the Report Reason respectively. "Internal Report Trigger-1" is given priority over "Internal Report Trigger-2". By default "Report Trigger Threshold Enable-1" is outputted.
	16	**Context Valid**
	15:0	Reserved

Encoding of SourceID field in RPT_ID is as follows:

SourceID[5] **Non-Media/ Media**	**SourceID[4:3]**	**SourceID[2:0]**	**Description**
0	00	000	Sources other than CPU, all CS and all Shaders listed below
0	00	001	CPU
0	00	010 to 111	Reserved
0	01	000 to 010 (Reserved 011 to 111)	All CS excluding Media related CS namely RCS, CCS, BCS In case of CCS source : RPT_ID [46:36] to indicate CCS id in addition.
0	10	000 to 110 (Reserved 111)	Shaders: VS, VSR, HS, TDS, GS, PS, TS
0	11	000 to 111	Reserved

1	00/01/10/11 (Media SliceN =0/1/2/3)	000 to 010 (Reserved 011 to 111)	VCS0/VCS1/VECS of Media SliceN

Performance Counting Register Interface

Global Registers
OACTXID - Observation Architecture Control Context ID
OA_IMR - OA Interrupt Mask Register
OASTATUS - Observation Architecture Status Register
OAHEADPTR - Observation Architecture Head Pointer
OATAILPTR - Observation Architecture Tail Pointer
OABUFFER - Observation Architecture Buffer
OASTARTTRIG_COUNTER - Observation Architecture Start Trigger Counter
OARPTTRIG_COUNTER - Observation Architecture Report Trigger Counter
OAREPORTTRIG2 - Observation Architecture Report Trigger 2
OAREPORTTRIG6 - Observation Architecture Report Trigger 6
CEC0-0 - Customizable Event Creation 0-0
CEC1-0 - Customizable Event Creation 1-0
CEC1-1 - Customizable Event Creation 1-1
CEC2-0 - Customizable Event Creation 2-0
CEC2-1 - Customizable Event Creation 2-1
CEC3-0 - Customizable Event Creation 3-0
CEC3-1 - Customizable Event Creation 3-1
CEC4-0 - Customizable Event Creation 4-0
CEC5-0 - Customizable Event Creation 5-0
CEC5-1 - Customizable Event Creation 5-1
CEC6-0 - Customizable Event Creation 6-0
CEC6-1 - Customizable Event Creation 6-1
CEC7-0 - Customizable Event Creation 7-0
CEC7-1 - Customizable Event Creation 7-1
EU_PERF_CNT_CTL0 - Flexible EU Event Control 0
EU_PERF_CNT_CTL1 - Flexible EU Event Control 1
EU_PERF_CNT_CTL2 - Flexible EU Event Control 2
EU_PERF_CNT_CTL3 - Flexible EU Event Control 3
EU_PERF_CNT_CTL4 - Flexible EU Event Control 4
EU_PERF_CNT_CTL5 - Flexible EU Event Control 5
EU_PERF_CNT_CTL6 - Flexible EU Event Control 6

Symmetrical Registers
OAPERF_A0 - Aggregate Perf Counter A0
OAPERF_A0_UPPER - Aggregate Perf Counter A0 Upper DWord
OAPERF_A1 - Aggregate Perf Counter A1
OAPERF_A1_UPPER - Aggregate Perf Counter A1 Upper DWord
OAPERF_A2 - Aggregate Perf Counter 2
OAPERF_A2_UPPER - Aggregate Perf Counter A2 Upper DWord
OAPERF_A3 - Aggregate Perf Counter A3
OAPERF_A3_UPPER - Aggregate Perf Counter A3 Upper DWord
OAPERF_A4 - Aggregate Perf Counter A4
OAPERF_A4_UPPER - Aggregate Perf Counter A4 Upper DWord
OAPERF_A4_LOWER_FREE - Aggregate Perf Counter A4 Lower DWord Free
OAPERF_A4_UPPER_FREE - Aggregate Perf Counter A4 Upper DWord Free
OAPERF_A5 - Aggregate Perf Counter A5
OAPERF_A5_UPPER - Aggregate Perf Counter A5 Upper DWord
OAPERF_A6 - Aggregate Perf Counter A6
OAPERF_A6_UPPER - Aggregate Perf Counter A6 Upper DWord
OAPERF_A6_LOWER_FREE - Aggregate Perf Counter A6 Lower DWord Free
OAPERF_A6_UPPER_FREE - Aggregate Perf Counter A6 Upper DWord Free
OAPERF_A7 - Aggregate Perf Counter A7
OAPERF_A7_- Upper Aggregate Perf Counter A7 Upper DWord
OAPERF_A8 - Aggregate Perf Counter A8
OAPERF_A8_UPPER - Aggregate Perf Counter A8 Upper DWord
OAPERF_A9 - Aggregate Perf Counter A9
OAPERF_A9_UPPER - Aggregate Perf Counter A9 Upper DWord
OAPERF_A10 - Aggregate Perf Counter A10
OAPERF_A10_UPPER - Aggregate Perf Counter A10 Upper DWord
OAPERF_A11 - Aggregate Perf Counter A11
OAPERF_A11_UPPER - Aggregate Perf Counter A11 Upper DWord
OAPERF_A12 - Aggregate Perf Counter A12
OAPERF_A12_UPPER - Aggregate Perf Counter A12 Upper DWord
OAPERF_A13 - Aggregate Perf Counter A13
OAPERF_A13_UPPER - Aggregate Perf Counter A13 Upper DWord
OAPERF_A14 - Aggregate Perf Counter A14
OAPERF_A14_UPPER - Aggregate Perf Counter A14 Upper DWord
OAPERF_A15 - Aggregate Perf Counter A15
OAPERF_A15_UPPER - Aggregate Perf Counter A15 Upper DWord
OAPERF_A16 - Aggregate Perf Counter A16

Symmetrical Registers
OAPERF_A16_UPPER - Aggregate Perf Counter A16 Upper DWord
OAPERF_A17 - Aggregate Perf Counter A17
OAPERF_A17_UPPER - Aggregate Perf Counter A17 Upper DWord
OAPERF_A18 - Aggregate Perf Counter A18
OAPERF_A18_UPPER - Aggregate Perf Counter A18 Upper DWord
OAPERF_A19 - Aggregate Perf Counter A19
OAPERF_A19_UPPER - Aggregate Perf Counter A19 Upper DWord
OAPERF_A19_LOWER_FREE - Aggregate Perf Counter A19 Lower DWord Free
OAPERF_A19_UPPER_FREE - Aggregate Perf Counter A19 Upper DWord Free
OAPERF_A20 - Aggregate Perf Counter A20
OAPERF_A20_UPPER - Aggregate Perf Counter A20 Upper DWord
OAPERF_A20_UPPER_FREE - Aggregate Perf Counter A20 Upper DWord Free
OAPERF_A20_LOWER_FREE - Aggregate Perf Counter A20 Lower DWord Free
OAPERF_A21 - Aggregate Perf Counter A21
OAPERF_A21_UPPER - Aggregate Perf Counter A21 Upper DWord
OAPERF_A22 - Aggregate Perf Counter A22
OAPERF_A22_UPPER - Aggregate Perf Counter A22 Upper DWord
OAPERF_A23 - Aggregate Perf Counter A23
OAPERF_A23_UPPER - Aggregate Perf Counter A23 Upper DWord
OAPERF_A24 - Aggregate Perf Counter A24
OAPERF_A24_UPPER - Aggregate Perf Counter A24 Upper DWord
OAPERF_A25 - Aggregate Perf Counter A25
OAPERF_A25_UPPER - Aggregate Perf Counter A25 Upper DWord
OAPERF_A26 - Aggregate Perf Counter A26
OAPERF_A26_UPPER - Aggregate Perf Counter A26 Upper DWord
OAPERF_A27 - Aggregate Perf Counter A27
OAPERF_A27_UPPER - Aggregate Perf Counter A27 Upper DWord
OAPERF_A28 - Aggregate Perf Counter A28
OAPERF_A28_UPPER - Aggregate Perf Counter A28 Upper DWord
OAPERF_A29 - Aggregate Perf Counter A29
OAPERF_A29_UPPER - Aggregate Perf Counter A29 Upper DWord
OAPERF_A30 - Aggregate Perf Counter A30
OAPERF_A30_UPPER - Aggregate Perf Counter A30 Upper DWord
OAPERF_A31 - Aggregate_Perf_Counter_A31
OAPERF_A31_UPPER - Aggregate Perf Counter A31 Upper DWord
OAPERF_A32 - Aggregate_Perf_Counter_A32
OAPERF_A32 _UPPER- Aggregate_Perf_Counter_A32 Upper DWord

Symmetrical Registers
OAPERF_A33 - Aggregate_Perf_Counter_A33
OAPERF_A33_UPPER - Aggregate_Perf_Counter_A33 Upper DWord
OAPERF_A34 - Aggregate_Perf_Counter_A34
OAPERF_A34_UPPER - Aggregate_Perf_Counter_A34 Upper DWord
OAPERF_A35 - Aggregate_Perf_Counter_A35
OAPERF_A35_UPPER - Aggregate_Perf_Counter_A35 Upper DWord
OAPERF_A36 - Aggregate_Perf_Counter_A36
OAPERF_A36_UPPER - Aggregate_Perf_Counter_A36 Upper DWord
OAPERF_A37 - Aggregate_Perf_Counter_A37
OAPERF_A37_UPPER - Aggregate_Perf_Counter_A37 Upper DWord
GPU_TICKS - GPU_Ticks_Counter

OA Interrupt Control Registers

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

Bit	Description
31:29	**Reserved. MBZ:** These bits may be assigned to interrupts on future products/steppings.
28	**Performance Monitoring Buffer Half-Full Interrupt:** For internal trigger (timer based) reporting, if the report buffer crosses the half full limit, this interrupt is generated.
27:0	**Reserved: MBZ (These bits must be never set by OA, these bit could be allocated to some other unit)**

- **WDBoxOAInterrupt Vector**
- IMR
- Bit Definition for Interrupt Control Registers

Performance Counter Reporting

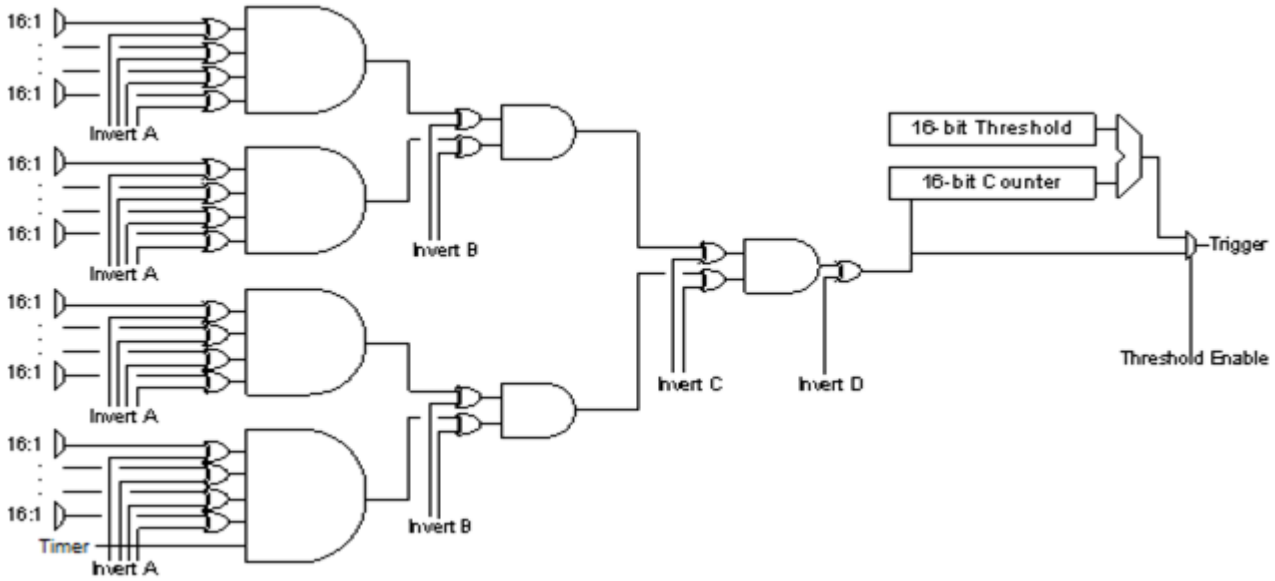
When either the MI_REPORT_PERF_COUNT command is received or the internal report trigger logic fires, a snapshot of the performance counter values is written to memory. The format used by HW for such reports is selected using the Counter Select field within the register. The organization and number of report formats vary per project and are detailed in Performance Counter Report Formats.

Details of Start Trigger Behavior

- All counters not explicitly defined as free running will advance after the start trigger conditions are met.
- Counting will continue after the start trigger has fired until OA is disabled, or device is reset.
- Multiple start triggering blocks (where implemented) are OR'd together in order to allow specification of multiple trigger conditions.
- Bit 18 in the report format reflects whether the start trigger has fired or not.

Configuration of Trigger Logic

OA contains logic to control when performance counter values are reported to memory. This functionality is controlled using the OA report trigger and OA start trigger registers. More detailed register descriptions are included in the Hardware Programming interface. The block diagram below illustrates the logic these registers control.



Context Switch Triggered Reports

A context load/switch on RCS will cause a performance counter snapshot to be written to memory at the next location in the OA circular report buffer using the perf counter format selected in **OAGCONTROL**. This functionality can be leveraged when preemption is enabled to re-construct the contribution of a specific context to a performance counter delta, requires SW to consider both the delta reported by MI_REPORT_PERF and the reports that may have been issued to OABUFFER by intervening contexts.

Frequency Change Triggered Reports

A GFX frequency change will cause a performance counter snapshot to be written to memory at the next location in the OA circular report buffer using the perf counter format selected in **OAGCONTROL**. Please note that a change back to the same frequency can occur and that such changes will still cause a performance counter report to occur.

Aggregating Counters

The table below described the desired high-level functionality from each of the aggregating counters.

Note that there is no counter of 2x2s sent to pixel shader, this is based on the assumption that the pixel shader invocation pipeline statistics counter increments for partially lit 2x2s as well and hence does not require a duplicate performance counter.

Please also note that some of the information provided by A-counters is useful for GFX/system load-balancing and is hence made available via free-running counters which do not require initial setup and count irrespective of OA enable/disable or freeze.

Counter #	Event	Description
A0	GPU Busy	GPU is not idle (includes all GPU engines). Link to detailed register definition: **Dual Context Considerations** Please note that the value of GPU Busy count reported to OABUFFER includes activities from all contexts.
A1	# of Vertex Shader Threads Dispatched	Count of VS fused threads dispatched to EUs Link to detailed register definition:
A2	# of Hull Shader Threads Dispatched	Count of HS fused threads dispatched to EUs Link to detailed register definition:
A3	# of Domain Shader Threads Dispatched	Count of DS fused threads dispatched to EUs Link to detailed register definition:
A4	# of GPGPU Threads Dispatched	Count of GPGPU fused threads dispatched to EUs. Available on both qualified and free-running counters. Link to detailed register definition: **Dual Context Considerations** Please note that the value of this event sampled by an MI_REPORT_PERF command reflects only the count associated with the context issuing the MI_REPORT_PERF command whereas the value of this event reported to OABUFFER includes activity from all contexts.
A5	# of Geometry Shader Threads Dispatched	Count of GS fused threads dispatched to EUs Link to detailed register definition:
A6	# of Pixel Shader Threads Dispatched	Count of PS fused threads dispatched to EUs. Available on both qualified and free-running counters. Link to detailed register definition:
A7	Aggregating EU counter 0	User-defined (details in Flexible EU Event Counters section) Link to detailed register definition:

Counter #	Event	Description
A8	Aggregating EU counter 1	User-defined (details in Flexible EU Event Counters section) Link to detailed register definition:
A9	Aggregating EU counter 2	User-defined (details in Flexible EU Event Counters section) Link to detailed register definition:
A10	Aggregating EU counter 3	User-defined (details in Flexible EU Event Counters section) Link to detailed register definition:
A11	Aggregating EU counter 4	User-defined (details in Flexible EU Event Counters section) Link to detailed register definition:
A12	Aggregating EU counter 5	User-defined (details in Flexible EU Event Counters section) Link to detailed register definition:
A13	Aggregating EU counter 6	User-defined (details in Flexible EU Event Counters section) Link to detailed register definition:
A14	Aggregating EU counter 7	User-defined (details in Flexible EU Event Counters section) Link to detailed register definition:
A15	Aggregating EU counter 8	User-defined (details in Flexible EU Event Counters section) Link to detailed register definition:
A16	Aggregating EU counter 9	User-defined (details in Flexible EU Event Counters section) Link to detailed register definition:
A17	Aggregating EU counter 10	User-defined (details in Flexible EU Event Counters section) Link to detailed register definition:
A18	Aggregating EU counter 11	User-defined (details in Flexible EU Event Counters section) Link to detailed register definition:
A19	Aggregating EU counter 12	Available on both qualified and free-running counters User-defined (details in Flexible EU Event Counters section) Link to detailed register definition:
A20	Aggregating EU counter 13	Available on both qualified and free-running counters User-defined (details in Flexible EU Event Counters section) Link to detailed register definition:

Counter #	Event	Description
A21	2x2s Rasterized	Count of the number of samples of 2x2 pixel blocks generated from the input geometry before any pixel-level tests have been applied. (Please note that 2x2s may be in terms of pixels or in terms of samples depending on project but are consistent between A21-A27.) Link to detailed register definition:
A22	2x2s Failing Fast pre-PS Tests	Count of the number of samples failing fast "early" (i.e., before pixel shader execution) tests (counted at 2x2 granularity). (Please note that 2x2s may be in terms of pixels or in terms of samples depending on project but are consistent between A21-A27.) Link to detailed register definition:
A23	2x2s Failing Slow pre-PS Tests	Count of the number of samples of failing slow "early" (i.e. before pixel shader execution) tests (counted at 2x2 granularity). (Please note that 2x2s may be in terms of pixels or in terms of samples depending on project but are consistent between A21-A27.) Link to detailed register definition:
A26	2x2s Written To Render Target	Number of samples that are written to render target.(counted at 2x2 granularity). MRT case will report multiple writes per 2x2 processed by the pixel shader. (Please note that 2x2s may be in terms of pixels or in terms of samples depending on project but are consistent between A21-A27.) Please note that this counter will not advance if a render target update does not occur and that pixel masking operations performed by the fixed function HW or shader may not be reflected in counters A22-A25 which only track their specific defined operations. This can lead to an apparent discrepancy between A21 vs. A22-A25 vs. A26/A27. Link to detailed register definition:
A27	Blended 2x2s Written to Render Target	Number of samples of blendable that are written to render target.(counted at 2x2 granularity). MRT case will report multiple writes per 2x2 processed by the pixel shader. (Please note that 2x2s may be in terms of pixels or in terms of samples depending on project but are consistent between A21-A27.) Please note that this counter will not advance if a render target update does not occur and that pixel masking operations performed by the fixed function HW or shader may not be reflected in counters A22-A25 which only track their specific defined operations. This can lead to an apparent discrepancy between A21 vs. A22-A25 vs. A26/A27. Link to detailed register definition:
A28	2x2s Requested from Sampler	Aggregated total 2x2 texel blocks requested from all EUs to all instances of sampler logic.

Counter #	Event	Description
		Link to detailed register definition: **Dual Context Considerations** Please note that the value of this event sampled by an MI_REPORT_PERF command reflects only the count associated with the context issuing the MI_REPORT_PERF command whereas the value of this event reported to OABUFFER includes activity from all contexts.
A29	Sampler L1 Misses	Aggregated misses from all sampler L1 caches. Please note that the number of L1 accesses varies with requested filtering mode and in other implementation specific ways. Hence it is not possible in general to draw a direct relationship between A28 and A29. However, a high number of sampler L1 misses relative to texel 2x2s requested frequently degrades sampler performance. Link to detailed register definition: **Dual Context Considerations** Please note that the value of this event sampled by an MI_REPORT_PERF command reflects only the count associated with the context issuing the MI_REPORT_PERF command whereas the value of this event reported to OABUFFER includes activity from all contexts.
A30	SLM Reads	Total read requests from an EU to SLM (including reads generated by atomic operations). Link to detailed register definition: **Dual Context Considerations** Please note that the value of this event sampled by an MI_REPORT_PERF command and reported to OABUFFER includes activity from all contexts.
A31	SLM Writes	Total write requests from an EU to SLM (including writes generated by atomic operations). Link to detailed register definition: **Dual Context Considerations** Please note that the value of this event sampled by an MI_REPORT_PERF command and reported to OABUFFER includes activity from all contexts.
A34	Atomic Accesses	Aggregated total atomic accesses from all EUs. This counter increments on atomic accesses to both SLM and URB. Link to detailed register definition: **Dual Context Considerations** Please note that the value of this event sampled by an MI_REPORT_PERF command

Counter #	Event	Description
		reflects only the count associated with the context issuing the MI_REPORT_PERF command whereas the value of this event reported to OABUFFER includes activity from all contexts. **Workaround** SLM atomics are not included in prior releases by this OA event (only global memory atomics are counted), a workaround using B/C counters is possible.
A35	Barrier Messages	Aggregated total kernel barrier messages from all Eus (one per thread in barrier). Link to detailed register definition: **Dual Context Considerations** Please note that the value of this event sampled by an MI_REPORT_PERF command reflects only the count associated with the context issuing the MI_REPORT_PERF command whereas the value of this event reported to OABUFFER includes activity from all contexts.
A36	GT Read Requests	For integrated GPU - Counts the number of memory read requests GT makes toward the SoC memory hierarchy. For discrete GPU this counter reflects only Device memory read requests. Note: It counts either the number of read transactions/4 or data transferred (in 128B units) depending on the programming of the field "OA byte per clock reporting vs commands per clock reporting" in the Register"Super Queue Internal Cnt Register I"
A37	GT Write Requests	For integrated GPU - Counts the number of memory write requests GT makes toward the SoC memory hierarchy. For discrete GPU this counter reflects only Device memory write requests. Note: It counts either the number of write transactions/4 or data transferred (in 128B units) depending on the programming of the field "OA byte per clock reporting vs commands per clock reporting" in the Register"Super Queue Internal Cnt Register I"

SPM Counters

Counter #	Event	Description
SPM0	EU Stall	Event reflects the condition where an EU is not idle but also not processing an ISA instruction. Each increment of the event reflects (2x (number of EUs per subslice)) clocks where this condition is met.

Counter #	Event	Description
SPM1	EU IPC	Event counts the number of execution slots among all compute pipes (e.g., FPU/EM etc.) that GT processes. It comprehends cases where multiple instructions are processed in a single clock. Each increment of the event reflects (2x (number of EUs per subslice)) clocks where this condition is met.
SPM2	Threads loaded	Event counts the total number of threads that have been fully loaded onto an EU in a given clock. This event DOES NOT include the time where the thread header is being sent to the EU. Each increment of the event reflects (4x (number of EUs per subslice)) clocks where this condition is met.
SPM3	EU Not Idle	Event reflects the condition where an EU is not idle. Each increment of the event reflects (2x (number of EUs per subslice)) clocks where this condition is met.
SPM4	Sampler Not Idle	Event counts sampler activity.
SPM5	EU Stalled & Sampler Not Idle	Event reflects the condition where the EU has sent a request(s) to sampler and all threads on the EU are stalled. Please note that the EU could be stalled for reasons other than sampler as well. Each increment of the event reflects (4x (number of EUs per subslice)) clocks where this condition is met.

External Events

External Architectural Event List for Performance Tuning

Architecture Event Name	Event Definition
SAMPLER_MEMORY_LATENCY_STALL	Number of cycles Sampler stalled due to latency hiding structure full
SAMPLER_2X2_READ	Number of 2x2 texel block requested from Sampler
SAMPLER_INPUT_AVAILABLE	Number of cycles when Sampler input is available
SAMPLER_OUTPUT_READY	Number of cycles when Sampler output is ready
SAMPLER_TEXTURE_CACHE_MISS	Number of Sampler L1 misses
SAMPLER_TEXTURE_CACHE_ACCESS	Number of Sampler L1 requests
SAMPLER_BUSY_CYCLES	Number of cycles Sampler pipeline active
AMFS_4X4_SHADING_REQUEST	Number of Shading Request (evaluate) messages processed by AMFS
AMFS_CACHE_HIT	Number of hits in the AMFS cache
AMFS_CACHE_MISS	Number of cache misses in AMFS

Architecture Event Name	Event Definition
AMFS_STALL_ANY_INPUT	Number of cycles AMFS stalls at any of the color pipe inputs
AMFS_STALL_ALL_INPUT	Number of cycles AMFS stalls at both of the color pipe inputs
AMFS_L3_ACCESS	Number of AMFS accesses to L3
AMFS_L3_WRITE	Number of AMFS writes to L3
AMFS_L3_ATOMIC	Number of AMFS atomics to L3
PIXEL_WRITE	Number of 2x2 pixels written to all render targets
PIXEL_BLEND	Number of blended 2x2 pixels written to all render targets
COLOR_PIPE_CACHE_LATENCY2_STALL	Number of cycles Color Pipeline stalled due to Render Cache latency hiding structure full
RENDER_CACHE_ALLOC	Number of Render Cache allocations
RENDER_CACHE_HIT	Number of Render Cache hits
RENDER_CACHE_READ	Number of Render Cache reads
RENDER_CACHE_WRITE	Number of Render Cache writes
RENDER_CACHE_INPUT_AVAILABLE	Number of cycles Render Cache input available
RENDER_CACHE_OUTPUT_READY	Number of cycles Render Cache output ready
PIXEL_POST_PROCESS_OUTPUT_READY	Number of cycles Color Pipeline pixel output ready
COLOR_PIPE_CACHE_LATENCY1_STALL	Number of cycles Color Pipeline stalled due to MultiSampling Cache latency hiding structure full
PIXEL_POST_PROCESS_INPUT_AVAILABLE	Number of cycles Color Pipeline input ready
PS_OUTPUT_AVAILABLE	Number of cycles Pixel Shader data is available
EU_SHARED_FUNCTION_ACCESS_HOLD	Number of cycles EU requests stalled by Shared Function units
EU_INST_EXECUTED_SEND_ALL	Number of instruction (GRF) dispatches executed on SEND Pipe
EU_INST_EXECUTED_CONTROL_ALL	Number of instructions executed on JEU Pipe
EU_INST_EXECUTED_ALU1_ALL	Number of execution slots taken by instructions executed on ALU1 pipe
EU_THREADS_OCCUPANCY_ALL	Number of thread slots occupied
EU_ACTIVE_CYCLES	Number of cycles at least one pipe is active in EU

Architecture Event Name	Event Definition
EU_STALL_CYCLES	Number of cycles any thread loaded but not even a single pipe is active in EU
EU_INST_ISSUED_ALL	Number of instructions issued (decoded) to any pipe
EU_INST_EXECUTED_INT16	Number of execution slots taken by INT16 ALU instructions
EU_INST_EXECUTED_INT32	Number of execution slots taken by INT32 ALU instructions
EU_INST_EXECUTED_MATH	Number of execution slots taken by extended math instructions
EU_INST_EXECUTED_ALU0_ALL	Number of execution slots taken by instructions executed on ALU0 pipe
EU_INST_EXECUTED_FP32	Number of execution slots taken by FP32 ALU instructions
EU_INST_EXECUTED_FP16	Number of execution slots taken by FP16 ALU instructions
EU_INST_EXECUTED_SYSTOLIC_ALL	Number of execution slots taken by instructions executed in Systolic pipe
EU_INST_EXECUTED_SYSTOLIC_FP16	Number of execution slots taken by FP16 DPAS instructions
EU_INST_EXECUTED_SYSTOLIC_BF16	Number of execution slots taken by BF16 DPAS instructions
EU_INST_EXECUTED_SYSTOLIC_INT8	Number of execution slots taken by INT8 DPAS instructions
EU_STALL_INSTFETCH_CYCLES	Number of cycles EU stalled, with at least one thread waiting for Instruction Fetch
EU_STALL_BARRIER_CYCLES	Number of cycles EU stalled, with at least one thread waiting for Gateway to write Notify register
EU_STALL_SBID_CYCLES	Number of cycles EU stalled, with at least one thread waiting for Scoreboard token to be available
EU_STALL_ALUWR_CYCLES	Number of cycles EU stalled, with at least one thread waiting for ALU to write GRF/ACC register
EU_STALL_SENDWR_CYCLES	Number of cycles EU stalled, with at least one thread waiting for SEND to write GRF register
EU_STALL_OTHER_CYCLES	Number of cycles EU stalled, with at least one thread waiting on any other dependency (Flag/EoT etc)
ICACHE_HIT	Number of Instruction Cache Hits

Architecture Event Name	Event Definition
ICACHE_MISS	Number of Instruction Cache Misses
EU_MULTIPLE_PIPE_ACTIVE_CYCLES	Number of cycles at least two pipes are actively executing a Gen ISA instruction among ALU0, ALU1 and SYSTOLIC pipes
EU_PIPE_ALU0_AND_ALU1_ACTIVE_CYCLES	Number of cycles ALU0 and ALU1 pipes are both actively executing a Gen ISA instruction
EU_PIPE_ALU0_AND_SYSTOLIC_ACTIVE_CYCLES	Number of cycles ALU0 and SYSTOLIC pipes are both actively executing a Gen ISA instruction
STREAMOUT_PRIMITIVE_WRITE_COUNT	Number of objects data written to memory, read from pipe 0 only
STREAMOUT_PRIMITIVE_STORAGE_NEED	Number of objects data that needed beyond the current allocated space, read from pipe 0 only
STREAMOUT_OUTPUT_VERTEX_COUNT	Number of vertices written by SOL
VS_OUTPUT_READY	Number of cycles Vertex Shader output is ready
IA_PRIMITIVE	Number of primitives in a draw
IA_VERTEX	Number of vertices in a draw
VERTEX_FETCH_OUTPUT_READY	Number of cycles Vertex Fetch output is ready
VERTEX_FETCH_INPUT_AVAILABLE	Number of cycles Vertex Fetch input is available
COMMAND_PARSER_COPY_ENGINE_BUSY	Number of cycles there is a context loaded and active on the copy queue.
COMMAND_PARSER_COMPUTE_ENGINE_DISPATCH_KERNEL_COUNT	Number of compute walker commands parsed on the compute engine
COMMAND_PARSER_COMPUTE_ENGINE_BUSY	Number of cycles there is a context loaded and active on the compute queue
COMMAND_PARSER_FLUSH_COUNT	Number of Stalling flushes at the top of the pipe
COMMAND_PARSER_RENDER_ENGINE_DRAW_COUNT	Number of DRAW and MESH commands parsed on the 3D engine
COMMAND_PARSER_RENDER_ENGINE_DISPATCH_KERNEL_COUNT	Number of compute walker commands parsed on the 3D engine
COMMAND_PARSER_RENDER_ENGINE_BUSY	Number of cycles there is a context loaded and active on the 3D queue
COMMAND_PARSER_VIDEO_ENGINE_BUSY	Number of cycles there is a context loaded and active on the media queue
COMMAND_PARSER_VIDEO_ENHANCEMENT_ENGINE_BUSY	Number of cycles there is a context loaded and active on the video enhancement queue

Architecture Event Name	Event Definition
GPU_MEMORY_L3_READ	Number of GTI memory reads from L3 caused by L3 Cache misses
GPU_MEMORY_L3_WRITE	Number of GTI memory writes from L3 caused by L3 invalidations
GPU_MEMORY_READ	Number of GTI memory reads
GPU_MEMORY_WRITE	Number of GTI memory writes
GPU_MEMORY_REQUEST_QUEUE_FULL	Number of cycles when SQ is filled above a threshold (usually 48 entries)
GPU_MEMORY_CYCLES_ACTIVE	Number of cycles device local memory (HBM, GDDR, LPDDR, etc.) is active
GPU_MEMORY_64B_TRANSACTION_READ	Number of device local memory (HBM, GDDR, LPDDR, etc.) reads (64B)
GPU_MEMORY_64B_TRANSACTION_WRITE	Number of device local memory (HBM, GDDR, LPDDR, etc.) writes (64B)
GPU_MEMORY_32B_TRANSACTION_READ	Number of device local memory (HBM, GDDR, LPDDR, etc.) reads (32B)
GPU_MEMORY_32B_TRANSACTION_WRITE	Number of device local memory (HBM, GDDR, LPDDR, etc.) writes (32B)
GPU_MEMORY_BYTE_READ	Number of device local memory (HBM, GDDR, LPDDR, etc.) read bytes
GPU_MEMORY_BYTE_WRITE	Number of device local memory (HBM, GDDR, LPDDR, etc.) write bytes
EU_DATAPORT_READ_MESSAGE_COUNT	Number of read messages sent by EUs to the Dataport
EU_DATAPORT_WRITE_MESSAGE_COUNT	Number of write messages sent by EUs to the Dataport
EU_DATAPORT_ATOMIC_MESSAGE_COUNT	Number of atomic messages sent by EUs to the Dataport
EU_DATAPORT_FENCE_MESSAGE_COUNT	Number of fence messages sent by EUs to the Dataport
DATAPORT_TEXTURE_CACHE_ACCESS	Number of cacheline requests from the Dataport to the texture cache not including uncached accesses
DATAPORT_TEXTURE_CACHE_HIT	Number of cache requests from the Dataport to the texture cache that resulted in a cache hit
EU_DATAPORT_REGISTER_REQUEST_COUNT	Number of message payload transactions sent from EUs to the Dataport
EU_DATAPORT_REGISTER_RESPONSE_COUNT	Number of return message payload transactions sent from the Dataport to EUs

Architecture Event Name	Event Definition
DATAPORT_INPUT_AVAILABLE	Number of cycles EUs have requests to the Dataport
DATAPORT_OUTPUT_READY	Number of cycles the Dataport has data to return to EUs
DATAPORT_BYTE_READ	Number of bytes read through the Dataport
DATAPORT_BYTE_WRITE	Number of bytes written through the Dataport
L3_INPUT_AVAILABLE	Number of cycles L3 bank has input requests
L3_OUTPUT_READY	Number of cycles L3 bank has output ready
L3_ATOMIC_ACCESS	Number of atomic accesses to L3 bank
DATAPORT_L3_READ	Number of L3 read requests coming from EU via Dataport
DATAPORT_L3_WRITE	Number of L3 write requests coming from EU via Dataport
DATAPORT_L3_HIT	Number of L3 hits for requests coming from the Dataport
ICACHE_L3_READ	Number of L3 read requests coming from EU Instruction Cache
ICACHE_L3_HIT	Number of EU instruction cache requests that hit the L3
SAMPLER_L3_READ	Number of L3 read requests resulting from sampler local cache miss
SAMPLER_L3_HIT	Number of L3 hit requests resulting from sampler local cache miss which hits in L3
COLOR_L3_HIT	Number of L3 read requests resulting from color local cache miss which hits in L3
Z_L3_HIT	Number of L3 read requests resulting from Z local cache miss which hits in L3
L3_READ	Number of L3 read requests
L3_WRITE	Number of L3 write requests
L3_MISS	Number of L3 accesses which miss in the L3 cache
L3_HIT	Number of L3 accesses which hits in the L3 cache
L3_SUPERQ_FULL	Number of cycles all slots in L3 request queue are waiting for data return / response
L3_BUSY	Number of cycles L3 request queue has one or more requests pending
L3_STALL	Number of cycles L3 bank stalled

Architecture Event Name	Event Definition
EU_LOAD_STORE_CACHE_READ_MESSAGE_COUNT	Number of read messages sent by EUs to the Load Store Cache
EU_LOAD_STORE_CACHE_WRITE_MESSAGE_COUNT	Number of write messages sent by EUs to the Load Store Cache
EU_LOAD_STORE_CACHE_ATOMIC_MESSAGE_COUNT	Number of atomic operations sent by EUs to the Load Store Cache
EU_LOAD_STORE_CACHE_FENCE_MESSAGE_COUNT	Number of fence messages sent by EUs to the Load Store Cache
EU_SLM_READ_MESSAGE_COUNT	Number of SLM read messages sent by EUs
EU_SLM_WRITE_MESSAGE_COUNT	Number of SLM write messages sent by EUs
EU_SLM_ATOMIC_MESSAGE_COUNT	Number of SLM atomic operations sent by EUs
EU_SLM_FENCE_MESSAGE_COUNT	Number of SLM fence operations sent by EUs
RT_LOAD_STORE_CACHE_READ_MESSAGE_COUNT	Number of read messages sent from Ray Tracing unit to the Load Store Cache
RT_LOAD_STORE_CACHE_WRITE_FROM_MESSAGE_COUNT	Number of write messages sent from Ray Tracing unit to the Load Store Cache
LOAD_STORE_CACHE_HIT	Number of Load Store Cache hits.
LOAD_STORE_CACHE_ACCESS	Number of Load Store Cache accesses.
LOAD_STORE_CACHE_NUMBER_OF_BANK_ACCESS_COUNT	Number of Load Store Cache banks accessed in a clock.
SLM_BANK_CONFLICT_COUNT	Number of SLM accesses resulting in a bank conflict.
LOAD_STORE_CACHE_L3_READ	Number of cacheline read requests from the Load Store Cache to L3
LOAD_STORE_CACHE_L3_WRITE	Number of cacheline write requests from the Load Store Cache to L3
LOAD_STORE_CACHE_PARTIAL_WRITE_COUNT	Number of writes to the Load Store Cache that don't fill a subsector
EU_LOAD_STORE_CACHE_REGISTER_REQUEST_COUNT	Number of message payload transactions sent by EUs to the Load Store Cache
EU_LOAD_STORE_CACHE_REGISTER_RESPONSE_COUNT	Number of message payload transactions sent from the Load Store Cache back to EUs
LOAD_STORE_CACHE_INPUT_AVAILABLE	Number of cycles the Load Store Cache has input available
LOAD_STORE_CACHE_OUTPUT_READY	Number of cycles the Load Store Cache has output ready
LOAD_STORE_CACHE_BYTE_READ	Number of bytes read out of the Load Store Cache, excluding SLM accesses.

Architecture Event Name	Event Definition
LOAD_STORE_CACHE_BYTE_WRITE	Number of bytes written to the Load Store Cache, excluding SLM accesses.
SLM_BYTE_READ	Number of bytes read from SLM
SLM_BYTE_WRITE	Number of bytes written to SLM
SLM_ACCESS_COUNT	Number of SLM accesses.
HOST_TO_GPUMEM_TRANSACTION_READ	Number of host reads to GPU local (HBM) memory (downstream)
HOST_TO_GPUMEM_TRANSACTION_WRITE	Number of host writes to GPU local (HBM) memory (downstream)
SYSMEM_TRANSACTION_READ	Number of system memory reads (upstream)
SYSMEM_TRANSACTION_WRITE	Number of system memory writes (upstream)
CLIPPER_INPUT_AVAILABLE	Number of cycles Clipper has input available (from Vertex Shader or SOL)
CLIPPER_PRIMITIVE_CULL	Number of Clipper early cull primitives
CLIPPER_TRANSACTION_OUTPUT	Number of elements pushed by Clipper into Stripsfan stage
CLIPPER_INPUT_VERTEX	Number of Clipper input vertices
CLIPPER_OUTPUT_READY	Number of cycles Clipper output ready
CLIPPER_PRIMITIVE_OUTPUT	Number of primitives going out of Clipper, must clip plus the trivial accept
CLIPPER_PRIMITIVE_FAR_NEAR_CLIP	Number of primitives clipped by Clipper due to near/far planes
STRIPSFAN_OUTPUT_READY	Number of cycles in which geometry pipeline output is ready
STRIPSFAN_OBJECT_COUNT	Number of objects exiting Stripsfan stage
STRIPSFAN_OBJECTS_CULL	Number of simple culled objects in Stripsfan stage
RASTERIZER_INPUT_AVAILABLE	Number of cycles Rasterizer input is available
PIXEL_2x2_LIT_POST_RASTERIZER_EARLY_DEPTH	Number of promoted 2x2 that are lit from Rasterizer
PIXEL_2x2_LIT_POST_RASTERIZER_LATE_DEPTH	Number of non-promoted 2x2 that are lit from Rasterizer
RASTERIZER_OUTPUT_READY	Number of cycles Rasterizer output is ready
RASTERIZER_TRANSACTION_OUTPUT	Number of transactions pushed from Rasterizer to the Z pipe
RASTERIZER_SAMPLE_OUTPUT	Number of lit samples emitted by Rasterizer
RT_CLOSEST_HIT_THREAD_RAY_DISPATCH	Number of closest hit threads dispatched for RayQuery only

Architecture Event Name	Event Definition
RT_MISS_THREAD_RAY_DISPATCH	Number of miss threads dispatched for RayQuery only
RT_INTERSECTION_THREAD_RAY_DISPATCH	Number of intersection threads dispatched for RayQuery only
RT_ANY_HIT_THREAD_RAY_DISPATCH	Number of any hit threads dispatched for RayQuery only
RT_CALLABLE_THREAD_RAY_DISPATCH	Number of callable threads dispatched for RayQuery only
RT_BVH_CACHE_MISS	Number of BVH cache misses for RayQuery only
RT_INPUT_MESSAGE_RAY_COUNT	Number of valid SIMD lanes in the TraceRay message
RT_INPUT_AVAILABLE_COUNT	Number of cycles new message is accepted by Ray Tracing Frontend
RT_MESSAGE_STALL_COUNT	Number of cycles when Ray Tracing message input is stalled while accepting input
RT_REQUEST_COLLISION	Number of cycles when Ray Tracing Leaf has two Traversal inputs valid
RT_TRAVERSAL_INPUT_RAY_COUNT	Number of Ray Tracing Traversal input rays
RT_TRAVERSAL_OUTPUT_RAY_COUNT	Number of Ray Tracing Traversal output rays
RT_TRAVERSAL_STEP_RAY_COUNT	Number of BVH nodes processed
RT_QUAD_TEST_RAY_COUNT	Number of nodes processed that use ray-quad intersection pipeline
RT_TRANSFORM_RAY_COUNT	Number of only HW transform accesses
RT_INTERNAL_NODE_RAY_COUNT	Number of internal BVH nodes processed by the traversal function
RT_PROCEDURAL_NODE_RAY_COUNT	Number of procedural BVH nodes processed by the traversal function
RT_INSTANCE_NODE_RAY_COUNT	Number of instance BVH nodes processed by the traversal function
RT_QUAD_LEAF_RAY_COUNT	Number of triangle BVH nodes processed by the traversal function
THREAD_DISPATCH_QUEUE0_ACTIVE_CYCLES	Number of cycles non-Pixel Shader threads are ready for dispatch in a particular subslice
THREAD_DISPATCH_QUEUE1_ACTIVE_CYCLES	Number of cycles Async GPGPU threads are ready for dispatch in a particular subslice
THREAD_DISPATCH_PS_ACTIVE_CYCLES	Number of cycles Pixel Shader threads are ready for dispatch in a particular subslice
TASK_THREADGROUP_COUNT	Number of Task Shader threadgroups dispatched
GPGPU_THREADGROUP_COUNT	Number of GPGPU threadgroups dispatched

Architecture Event Name	Event Definition
ASYNC_GPGPU_THREADGROUP_COUNT	Number of Async GPGPU threadgroups dispatched
TASK_THREAD_EXIT_COUNT	Number of Task Shader EOT messages received
ASYNC_GPGPU_THREAD_EXIT_COUNT	Number of Async GPGPU EOT messages received
THREADGROUP_DISPATCH_QUEUE0_INPUT_AVAILABLE	Number of cycles Thread Spawner has input available on queue 0
THREADGROUP_DISPATCH_QUEUE1_INPUT_AVAILABLE	Number of cycles Thread Spawner has input available on queue 1
THREADGROUP_DISPATCH_RESOURCE_STALL_CYCLES	Number of cycles Thread Spawner is stalled waiting for resources to be available (SLM, Barrier, BTD stack)
URB_READ	Number of URB reads
URB_WRITE	Number of URB writes
URB_CROSS_SLICE_READ	Number of URB reads occurred from cross slices
HIZ_SUBSPAN_LATENCY_FIFOFULL	Number of cycles for which HiZ latency hiding structure full
HIZ_DEPTH_TEST_PASS_P	Number of promoted 2x2 passed by hierarchical depth test
HIZ_DEPTH_TEST_AMBIG_P	Number of promoted 2x2 that are ambiguous by hierarchical depth test
HIZ_DEPTH_TEST_FAIL_P	Number of promoted 2x2 that failed the hierarchical depth test
HIZ_DEPTH_TEST_AMBIG_NP	Number of non-promoted 2x2 that are ambiguous by hierarchical depth test
HIZ_DEPTH_TEST_FAIL_NP	Number of non-promoted 2x2 that failed the hierarchical depth test
IZ_SUBSPAN_LATENCY_FIFOFULL	Number of cycles for which IZ latency hiding structure full
POSTPS_DEPTH_STENCIL_ALPHA_TEST_FAIL	Number of 2x2 that were lit from Rasterizer but failed the depth stencil test or alpha test
EARLY_DEPTH_STENCIL_TEST_FAIL_P	Number of promoted 2x2 that failed Depth / Stencil that were previously ambiguous at HiZ
EARLY_DEPTH_STENCIL_TEST_FAIL_NP	Number of non-promoted 2x2 that failed Depth / Stencil before the Pixel Shader that were previously ambiguous at HiZ
IZ_OUTPUT_READY	Number of cycles IZ has requests to different clients

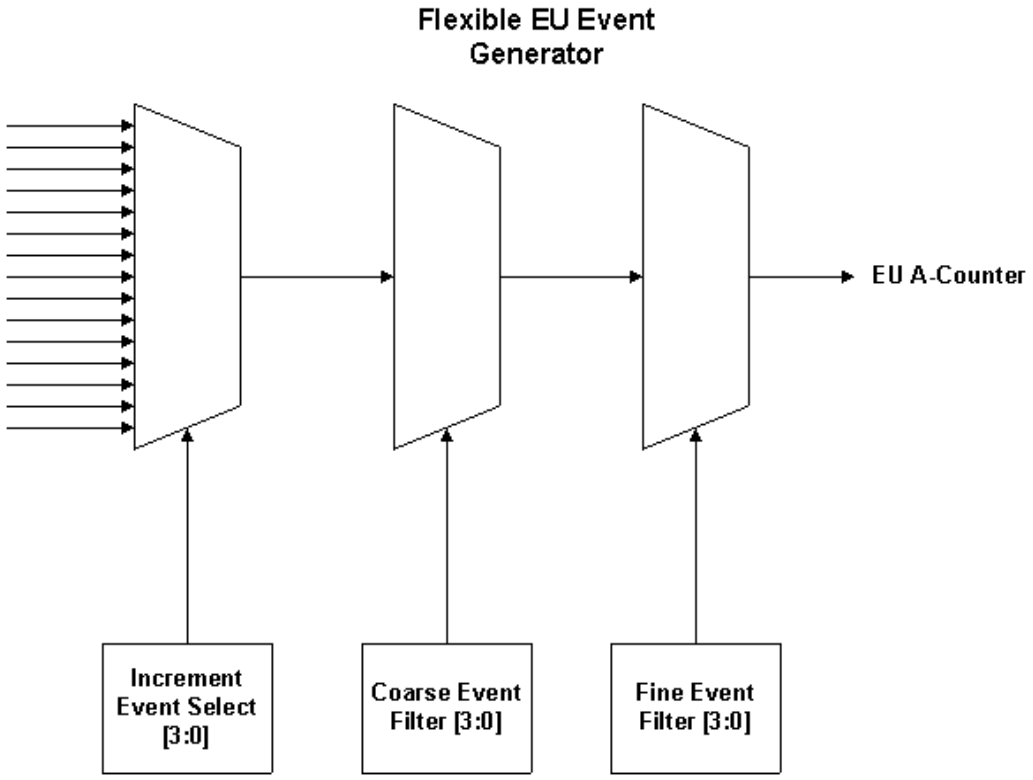
Note: Flex EU Coarse and Fine filters to apply as appropriately on base flex EU events on the external event list above. Please refer to Flex EU section for more info.

Flexible EU Event Counters

Since EU performance events are most interesting in many cases when aggregated across all EUs and many interesting EU performance events are limited to certain APIs (e.g., hull shader kernel stats only applicable when running a DX11+ workload), additional flexibility has been added to the aggregated counters coming from the EU array.

The following block diagram shows the high-level flow that generates each flexible EU event.

Note that no support is provided for differences between flexible EU event programming between EUs because the resulting output from each EU is eventually merged into a single OA counter anyway.



Supported Increment Events

Increment Event	Encoding	Notes
EU_INST_EXECUTED_ALU0_ALL	0b00000	Number of execution slots taken by instructions executed on ALU0 pipe
EU_INST_EXECUTED_ALU1_ALL	0b00001	Number of execution slots taken by instructions executed on ALU1 pipe
		Only fine event filters 0b0000, 0b0111, 0b1000, 0b1001, and 0b1010 are supported with this increment event.
EU_INST_EXECUTED_SYSTOLIC_ALL	0b01001	Number of execution slots taken by instructions executed in Systolic pipe
EU_INST_EXECUTED_SEND_ALL	0b00010	Number of instruction (GRF) dispatches executed on SEND Pipe. Only fine event filters 0b0000,0b0101, 0b0111, 0b1000, 0b1001, and 0b1010 are supported with this increment event.
EU_PIPE_ALU0_AND_ALU1_ACTIVE_CYCLES	0b00011	Number of occurrences of signal that is high on every EU clock where the EU ALU0 and ALU1 pipelines are both actively executing an ISA instruction. Only coarse event filters 0b0000, 0b0111, and 0b1000 are supported with this increment event. Only fine event filters 0b0000, 0b0111, 0b1000, 0b1001, and 0b1010 are supported with this increment event.
EU_PIPE_ALU0_AND_SYSTOLIC_ACTIVE_CYCLES	0b01010	Number of occurrences of signal that is high on every EU clock where the EU ALU0 and Systolic pipelines are both actively executing an ISA instruction. Only coarse event filters 0b0000, 0b0111, and 0b1000 are supported with this increment event. Only fine event filters 0b0000, 0b0111, 0b1000, 0b1001, and 0b1010 are supported with this increment event.
EU_ACTIVE_CYCLES	0b00100	Number of occurrences of signal that is high on every EU clock where at least one EU pipeline is actively executing an ISA instruction. All coarse event filters are supported. Only fine event filters 0b0000,0b0101, 0b0111, 0b1000, 0b1001, and 0b1010 are supported with this increment event.
EU_STALL_CYCLES	0b00101	Number of occurrences of signal that is high on every EU clock where at least one thread is loaded but no EU pipeline is actively executing an ISA instruction. All coarse event filters are supported. Only fine event filters 0b0000, 0b0111, 0b1000, 0b1001, and 0b1010 are supported with this increment event.

EU_THREADS_OCCUPANCY_ALL	0b01000	Number of Thread slots occupied. Accumulated every clock. Implies an accumulator which increases every EU clock by the number of loaded threads, signal pulses high for one clock when the accumulator exceeds a multiple of the number of thread slots (e.g. for a 8-thread EU, signal pulses high every clock where the increment causes a 3-bit accumulator to overflow). Only coarse event filters 0b0000, 0b0111, and 0b1000 are supported with this increment event. Only fine event filters 0b0000, 0b0111, 0b1000, 0b1001, and 0b1010 are supported with this increment event.
EU_INST_EXECUTED_FP16	0b00110	Number of execution slots taken by FP16 ALU instructions
EU_INST_EXECUTED_FP32	0b00111	Number of execution slots taken by FP32 ALU instructions
EU_INST_EXECUTED_INT16	0b01100	Number of execution slots taken by INT16 ALU instructions
EU_INST_EXECUTED_INT32	0b01101	Number of execution slots taken by INT32 ALU instructions
EU_INST_EXECUTED_SYSTOLIC_BF16	0b01110	Number of execution slots taken by BF16 DPAS instructions
EU_INST_EXECUTED_SYSTOLIC_INT8	0b01111	Number of execution slots taken by INT8 DPAS instructions
EU_INST_EXECUTED_MATH	0b10000	Number of execution slots taken by extended math instructions
EU_INST_EXECUTED_CONTROL_ALL	0b10001	Number of instructions executed on JEU Pipe. Only coarse event filters 0b0000, 0b0111, and 0b1000 are supported with this increment event.
EU_INST_EXECUTED_SYSTOLIC_FP16	0b10010	Number of execution slots taken by FP16 DPAS instructions
	0b11111	Expected HW default, allows logic to be power-optimized.

Supported Coarse Event Filters

Coarse Event Filter	Encoding	Notes
No mask	0b0000	Never masks increment event.
VS Thread Filter	0b0001	For increment events 0b00000/0b00001/0b00010, masks increment events unless the FFID which dispatched the currently executing thread equals FFID of VS.
		For increment events 0b00100/0b00101, masks increment event unless at least one of the loaded threads was dispatched by VS.

Coarse Event Filter	Encoding	Notes
HS Thread Filter	0b0010	For increment events 0b00000/0b00001/0b00010, masks increment event unless the FFID which dispatched the currently executing thread equals FFID of HS.
		For increment events 0b00100/0b00101, masks increment event unless at least one of the loaded threads was dispatched by HS
DS Thread Filter	0b0011	For increment events 0b00000/0b00001/0b00010, masks increment event unless the FFID which dispatched the currently executing thread equals FFID of DS.
		For increment events 0b00100/0b00101, masks increment event unless at least one of the loaded threads was dispatched by DS.
GS Thread Filter	0b0100	For increment events 0b00000/0b00001/0b00010, masks increment event unless the FFID which dispatched the currently executing thread equals FFID of GS.
		For increment events 0b00100/0b00101, masks increment event unless at least one of the loaded threads was dispatched by GS.
PS Thread Filter	0b0101	For increment events 0b00000/0b00001/0b00010, masks increment event unless the FFID which dispatched the currently executing thread equals FFID of PS.
		For increment events 0b00100/0b00101, masks increment event unless at least one of the loaded threads was dispatched by PS.
GPTS Thread Filter	0b0110	For increment events 0b00000/0b00001/0b00010, masks increment event unless the FFID which dispatched the currently executing thread equals FFID of GPTS.
		For increment events 0b00100/0b00101, masks increment event unless at least one of the loaded threads was dispatched by GPTS.
Row = 0	0b0111	Masks increment event unless the row ID for this EU is 0 (control register is in TDL so only have to check within quarter-slice).
GP1 Thread Filter	0b1001	For increment events 0b00000/0b00001/0b00010, masks increment event unless the FFID which dispatched the currently executing thread equals FFID associated with GP1/Async Compute. For increment events 0b00100/0b00101, masks increment event unless at least one of the loaded threads was dispatched by GP1/Async Compute.
TASK Thread Filter	0b1010	For increment events 0b00000/0b00001/0b00010, masks increment event unless the FFID which dispatched the currently executing thread equals FFID of TASK.
		For increment events 0b00100/0b00101, masks increment event unless at least one of the loaded threads was dispatched by TASK.

Coarse Event Filter	Encoding	Notes
MESH Thread Filter	0b1011	For increment events 0b00000/0b00001/0b00010, masks increment event unless the FFID which dispatched the currently executing thread equals FFID of MESH.
		For increment events 0b00100/0b00101, masks increment event unless at least one of the loaded threads was dispatched by MESH.
Ray Tracing Shader via RT mechanism: Any Hit Shader	0b1100	For increment events 0b00000/0b00001/0b00010, masks increment event unless the FFID which dispatched the currently executing thread equals FFID of RT or RT1 and header has the matching encoding. For increment events 0b00100/0b00101, masks increment event unless at least one of the loaded threads was dispatched by RT.
Ray Tracing Shader via RT mechanism: Closest Hit Shader	0b1101	For increment events 0b00000/0b00001/0b00010, masks increment event unless the FFID which dispatched the currently executing thread equals FFID of RT or RT1 and header has the matching encoding. For increment events 0b00100/0b00101, masks increment event unless at least one of the loaded threads was dispatched by RT.
Ray Tracing Shader via RT mechanism: Miss Shader	0b1110	For increment events 0b00000/0b00001/0b00010, masks increment event unless the FFID which dispatched the currently executing thread equals FFID of RT or RT1 and header has the matching encoding. For increment events 0b00100/0b00101, masks increment event unless at least one of the loaded threads was dispatched by RT.

Fine Event Filters

Fine Event Filter	Encoding	Notes
None	0b0000	Never mask increment event.
Cycles where hybrid instructions are being executed	0b0001	Masks increment event unless the instruction(s) being executed on the pipeline(s) selected by the increment event are hybrid instructions.
Cycles where ternary instructions are being executed	0b0010	Masks increment event unless the instruction(s) being executed on the pipeline(s) selected by the increment event are ternary instructions.
Cycles where binary instructions are being executed	0b0011	Masks increment event unless the instruction(s) being executed on the pipeline(s) selected by the increment event are binary instructions.
Cycles where mov instructions are being executed	0b0100	Masks increment event unless the instruction(s) being executed on the pipeline(s) selected by the increment event are mov instructions.
Cycles where sends start being	0b0101	Masks increment event unless the instruction(s) being executed on the pipeline(s) selected by the increment event are send start of dispatch. Note that if this fine

Fine Event Filter	Encoding	Notes
executed		event filter is used in combination with increment events not related to the EU send pipeline (e.g., FPU0 active), the associated flexible event counter will increment in an implementation-specific manner.
EU# = 0b00	0b0111	Masks increment event unless the EU number for this EU is 0b00.
EU# = 0b01	0b1000	Masks increment event unless the EU number for this EU is 0b01.
EU# = 0b10	0b1001	Masks increment event unless the EU number for this EU is 0b10.
EU# = 0b11	0b1010	Masks increment event unless the EU number for this EU is 0b11.

Flexible EU Event Config Registers

EU_PERF_CNT_CTL0 - Flexible EU Event Control 0

EU_PERF_CNT_CTL1 - Flexible EU Event Control 1

EU_PERF_CNT_CTL2 - Flexible EU Event Control 2

EU_PERF_CNT_CTL3 - Flexible EU Event Control 3

EU_PERF_CNT_CTL4 - Flexible EU Event Control 4

EU_PERF_CNT_CTL5 - Flexible EU Event Control 5

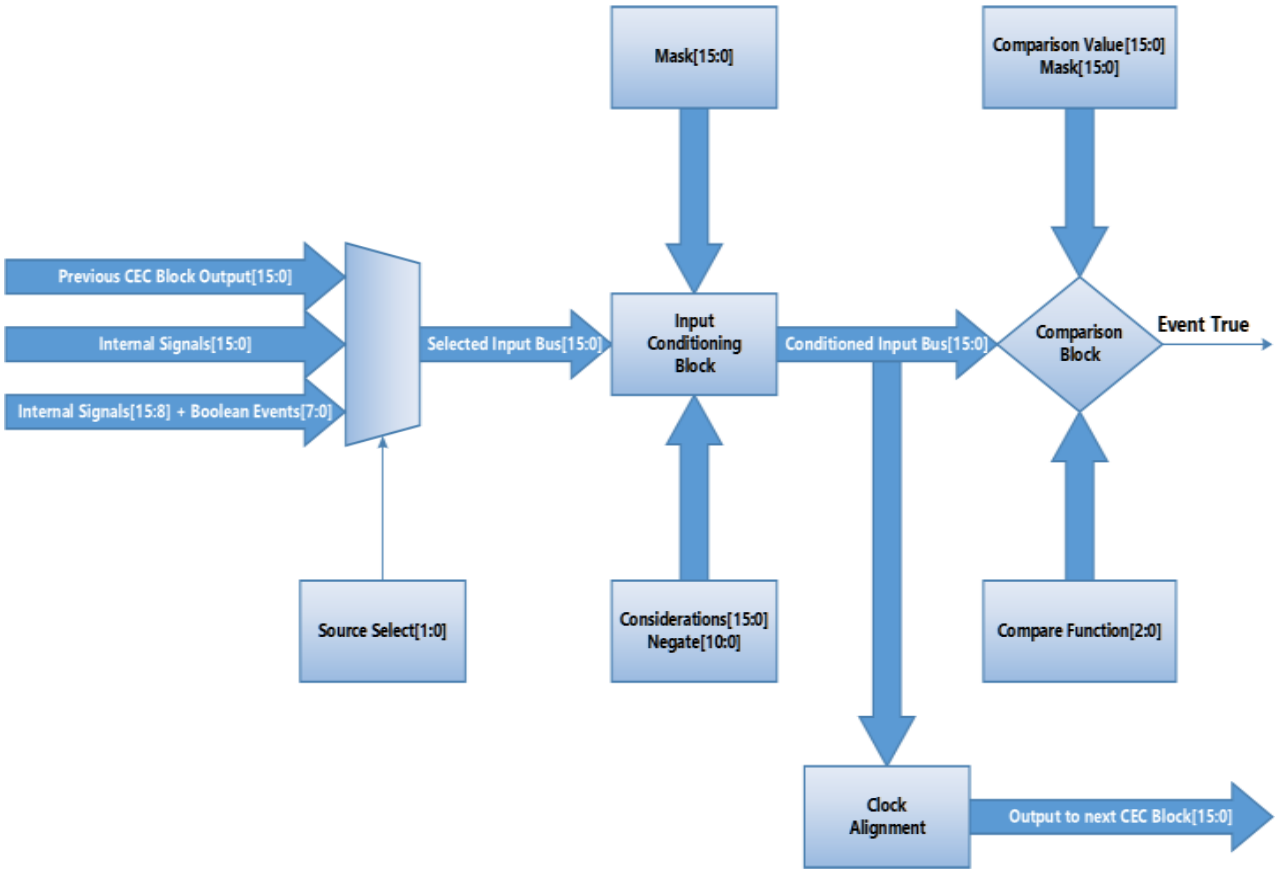
EU_PERF_CNT_CTL6 - Flexible EU Event Control 6

Custom Event Counters

Also known as B-counters, the events counted in these counters are defined from Boolean combinations of

input signals using the custom event creation logic built into OA.

The following diagram(s) illustrate(s) the structure used to create a custom event. Every B-counter has such a block.



MI_REPORT_PERF_COUNT

MI_REPORT_PERF_COUNT

Interrupts

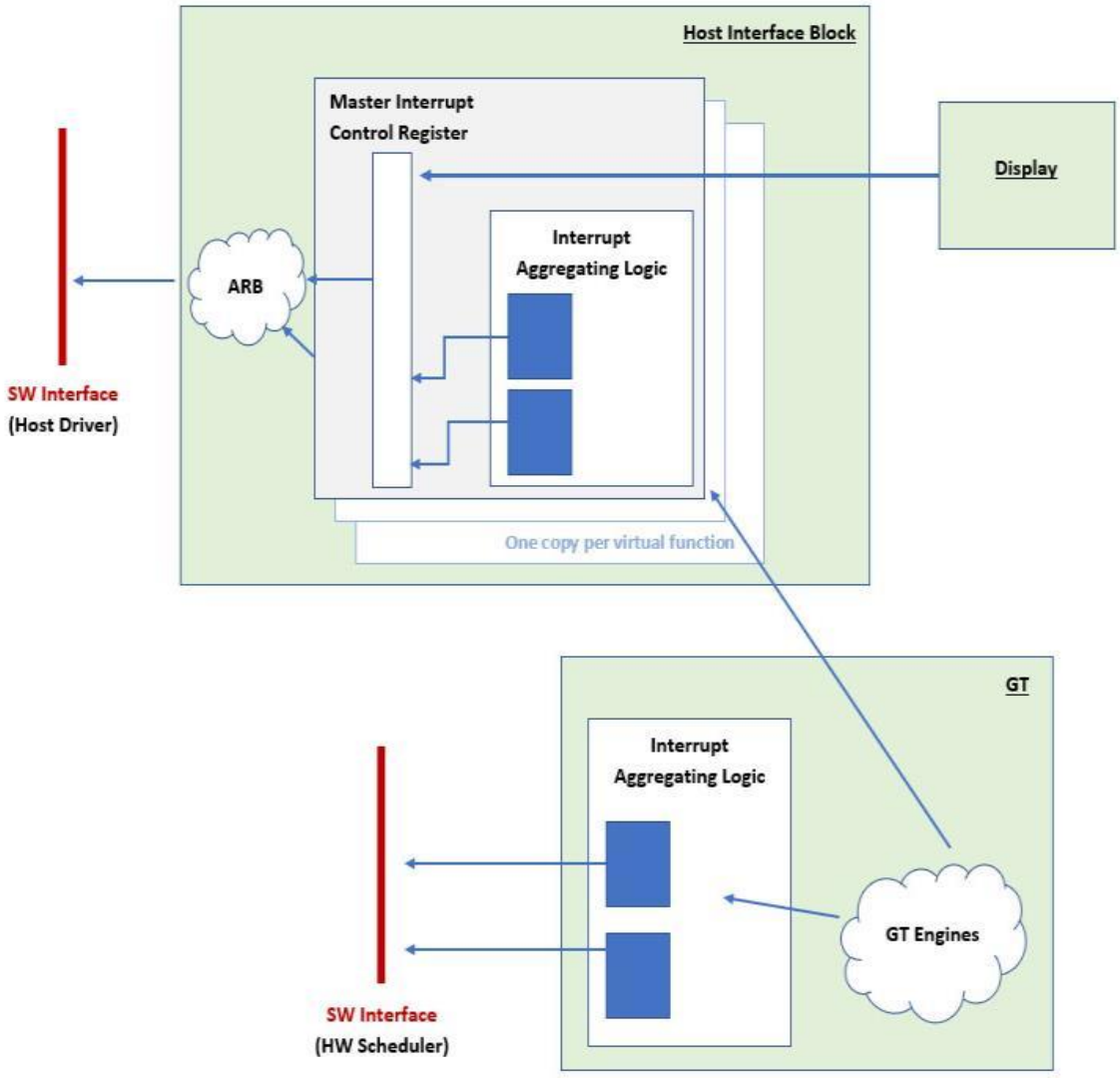
Introduction

Overview:

The Graphics device is comprised of a number of independent engines that can be invoked to execute workloads. Engines communicate status primarily through interrupts. The Graphics device supports two models of scheduling and handling of interrupts:

- Host SW schedules and manages all interrupts
- Scheduling and related interrupts are managed by hardware scheduler (MinIA micro-controller) and host SW manages interrupts not related to scheduling.

The hardware can be configured to work in either of these models. HW scheduling is the preferred mode because it provides best utilization of resources. The figure below shows the high-level overview of the interrupt infrastructure.



The interrupt infrastructure is designed to support both of these models. Each engine is allocated a set of interrupt bits that it can set to report events (the number of bits allotted to each engine varies -- most engines are allocated 16bits, some engines which have more events are allocated 32bits). Interrupt messages sent by engines result in interrupt bits being recorded in MMIO registers and an interrupt being generated to the servicing agent (MinIA scheduler or Host SW). The interrupt handler determines the source of the interrupt (by reading registers) and then processes the interrupts. Processing interrupts involves reading the interrupt status register, performing the operations for handling the interrupt and indicating completion of handling by writing to registers (clear).

When using the HW scheduler, the scheduling related interrupts are directed to the MinIA scheduler.

GT Engine Interrupts:

Within GT, engines are categorized into different engine classes and instances. An engine class is used to differentiate between engines that perform different functions (Copy, Render, VideoDecode,

VideoEncode, etc). A product may have a number of instances of a specific engine class e.g.: GT2 has 2 instances of VD, GT3 has 4 instances of VD, etc. The following table lists various engine classes as well as instances within each class.

Engine Class	Engine Instance Name	ClassID[2:0]	InstanceID[5:0]
Render	RCS	0	0
Video Decode	VCS0-N	1	0-N
Video Enhancement Engine	VECS0-N/2	2	0-N/2
Copy Engine	BCS	3	0
Other	GuC	4	0
	GTPM	4	1
	WDOAPerf	4	2
	SCTRG	4	3
	KCR	4	4
	Gunit	4	5
	CSME	4	6
Compute Engine	CCS0-N	5	0-N
Reserved		6-7	

Each engine reports up to 16 interrupts to interrupt handling logic. Source identification data is included in interrupt messages to interrupt aggregating logic, i.e. when reporting an interrupt to either host or graphics firmware, the generating engine must identify itself. 16 bits of identification is sent along with interrupt data, and comprises Engine Class ID, Instance ID and Virtual Function Number. Interrupt bit definition varies per engine class, these are listed in the Bspec in the Global/ section.

Format of interrupt message:

Bit Fied	Purpose
[31:30]	Reserved
[29:27]	VF ID
[26]	Reserved
[25:20]	Instance ID
[19]	Reserved
[18:16]	Engine Class ID
[15:0]	Interrupt data

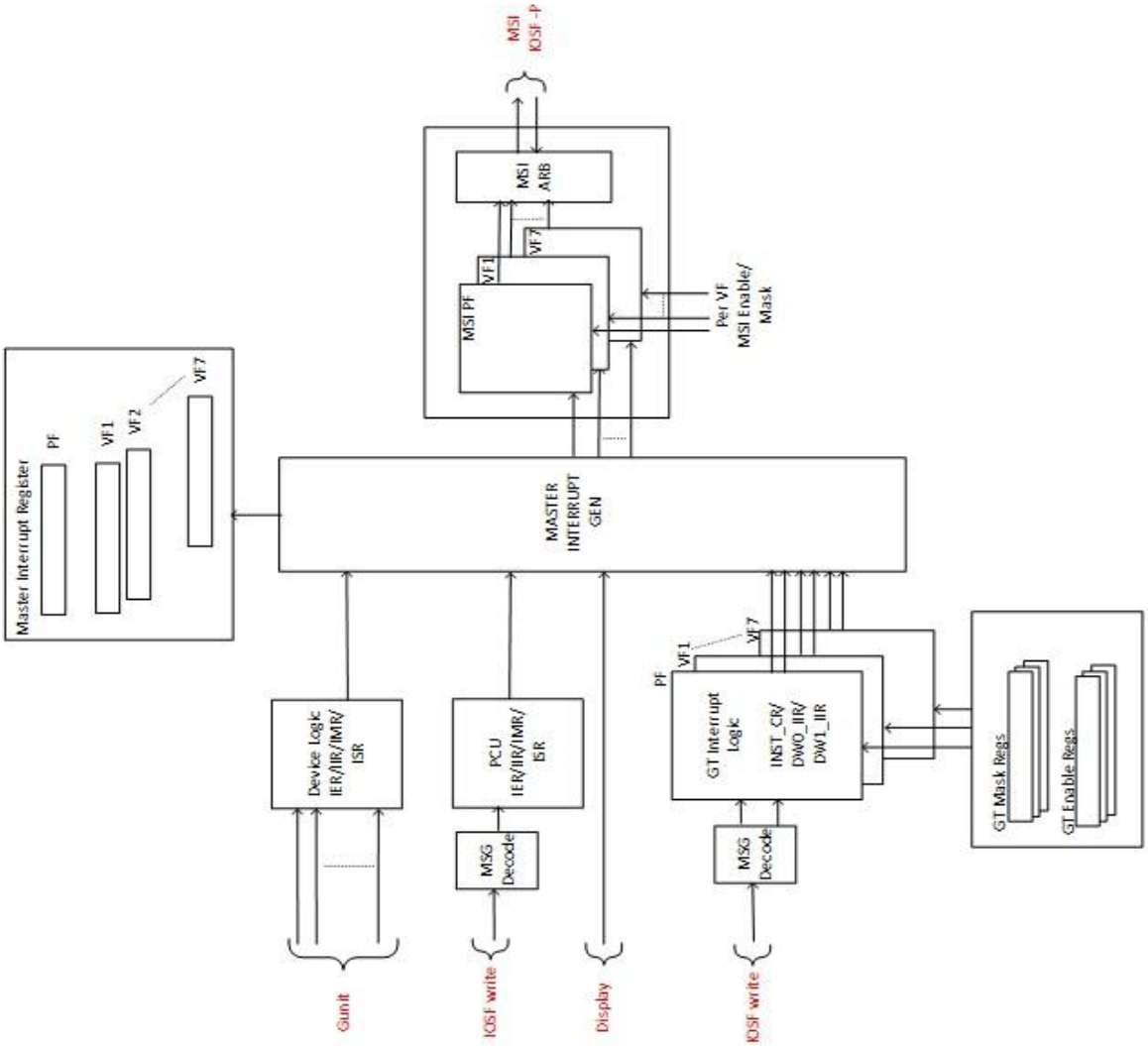
Hardware Scheduler/MinIA SW Interface

Graphics interrupts to scheduling firmware are delivered as two unique vector values. Each vector accounts for 32 graphics engines. Firmware processes each of two groups of graphics engines independently.

Service routines are independent for the two interrupt vectors presented to the MinIA firmware.

The diagrams below assume SRIOV-8 implementation which does not include memory-based interrupt support.

Host SW Interface



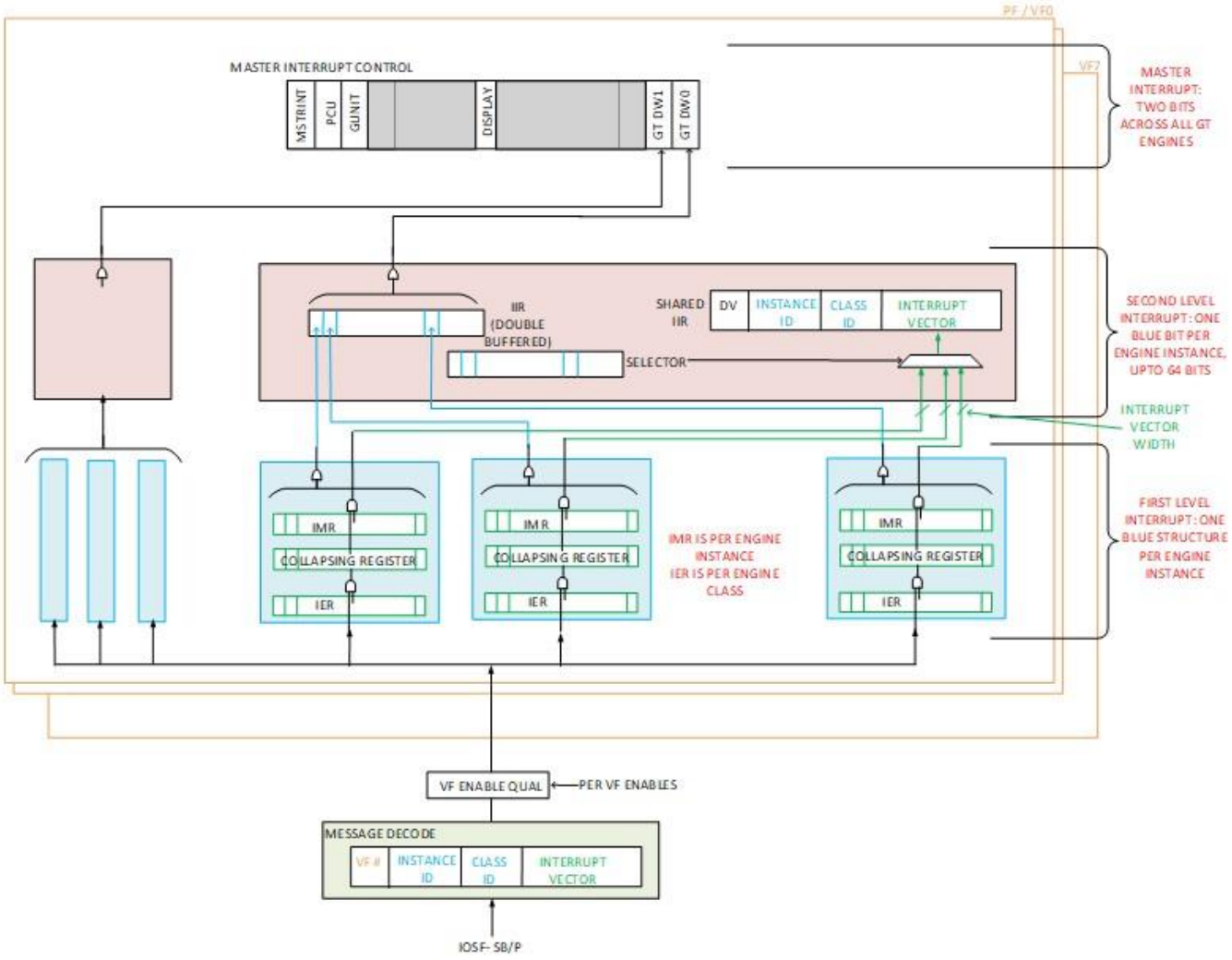
Interrupts to Host are delivered via a Primary Interrupt Control Register. Graphics interrupts use 2 bits in the Primary Interrupt Control Register. In addition, interrupt events from Display are also represented in the Primary Interrupt Control Register. Multiple copies of Primary Interrupt Control Register exist, one for every virtual machine in the system.

Interrupt bits in the Primary Interrupt Control Register are Read-Only bits, and are level indications that a second level interrupt is present (As seen earlier, second level interrupts per client are OR-ed together. When the second level IIR is cleared, the bit represented will be 0.). An interrupt is sent to driver whenever bits are set in the Primary Interrupt Control Register and the Enable bit is also set.

As a result of this interrupt, SW first resets the Primary Control Enable bit. SW then reads the Primary Interrupt Control register into a local variable, and works off this local variable to service interrupts. Once all lower-level interrupts have been serviced, SW writes the Primary Interrupt Control register to set the Primary Control Enable bit.

Interrupt Aggregating Logic

A hierarchical interrupt status infrastructure is provided to efficiently determine the source of the interrupt. The first level of interrupts is generated by GT Engines. Interrupt handling logic accumulates these interrupts from the various engines, and organizes it as a single bit per engine in a second level. 32 bits of second level interrupts are OR-ed together to generate a DW-level interrupt event for up to 32 engines. Two such events are used to provide support for up to 64 GT engines. When communicating with the MinIA, these events are mapped to two unique interrupt vectors in the MinIA LAPIC. When communicating with host driver, these events form two bits of the Primary Interrupt Control Register as marked in the picture.



First Level Interrupt Bits:

When an interrupt event comes into the interrupt handling logic, it is AND-ed with a per-Engine Enable register (IER). Only enabled events make forward progress. Disabled events are simply dropped by the interrupt handling logic. [Note that multiple instances of the same engine type (except those in the 'Other' Engine Class) share the same Enable register.]

Enabled interrupts are logged in a per-instance, non-SW readable Collapsing Register. These events are AND-ed with (the inverse of) a per-Instance Mask Register (IMR). Only unmasked events make forward progress. Masked events remain in the per-Instance Collapsing Register until they are unmasked. [Note that every instance (even of the same engine type) has its own Mask Register.]

Unmasked events in the per-Instance Collapsing Register are OR-ed together to produce a single second level interrupt event.

Second Level Interrupt Bits:

Second level interrupt events are stored in a double buffered IIR structure. A snapshot of events is taken when SW reads the IIR. From the time of read to the time of SW completely clearing the second-level

IIR (to indicate end of service), all incoming interrupts are logged in a secondary storage structure. This guarantees that the record of interrupts SW is servicing will not change while under service.

Bits in the second-level IIR are OR-ed together to generate a DW-level event. The IIR is cleared by writing 1s. If events exist in the secondary storage at the time that the IIR is completely cleared, a second DW-level event will be generated.

Shared IIR, Selector:

Shared IIR and Selector registers are used when SW is in the process of handling reported interrupts. As a result of a GT interrupt (DW-level interrupt), SW reads the second-level IIR register. The read provides an indication of engines needing service. SW must then service engines one at a time by writing a one-hot selection into the Selector Register.

When a selection is made by writing the Selector, interrupt handling logic presents all the unmasked interrupt bits (first level interrupt events) for the selected engine in the Shared IIR, and sets the Data-Valid bit (MSB). SW can then read the Shared IIR and take action for the reported events. SW must clear the Shared IIR by writing 1 to the Data-Valid bit to indicate end of service for the selected engine. This clearing of the Shared IIR Data-Valid bit clears both the Shared IIR as well as the Selector. Note that the Selector data must be one-hot. Selector must not have a bit set that is not set in the second-level IIR at the time of SW read.

SW then repeats the above steps for each bit set in the second-level IIR. Multiple rounds of Selector write-Shared IIR clear may be required to service a DW level interrupt a single time.

Second-level IIR bits are cleared only after individual engines are serviced via the Selector write -Shared IIR clear routine. This clearing can be done after each iteration through the Selector write-Shared IIR clear routine (i.e. one second-level bit cleared after each iteration), or all at once after all engines have been serviced. Second-level IIR bits must not be cleared without first servicing that engine's interrupts via the Selector and Shared IIR registers.

Enable and Mask Registers:

Interrupt aggregating logic includes Enable registers(IER) per Engine Class. Different instances of the same engine class use the same Enable register, except for engines in the 'Other' class. Each instance in the 'Other' class has its own Enable register.

Interrupt aggregating logic also includes Mask registers (IMR). Each engine instance, even within the same Engine Class, has a unique Mask Register.

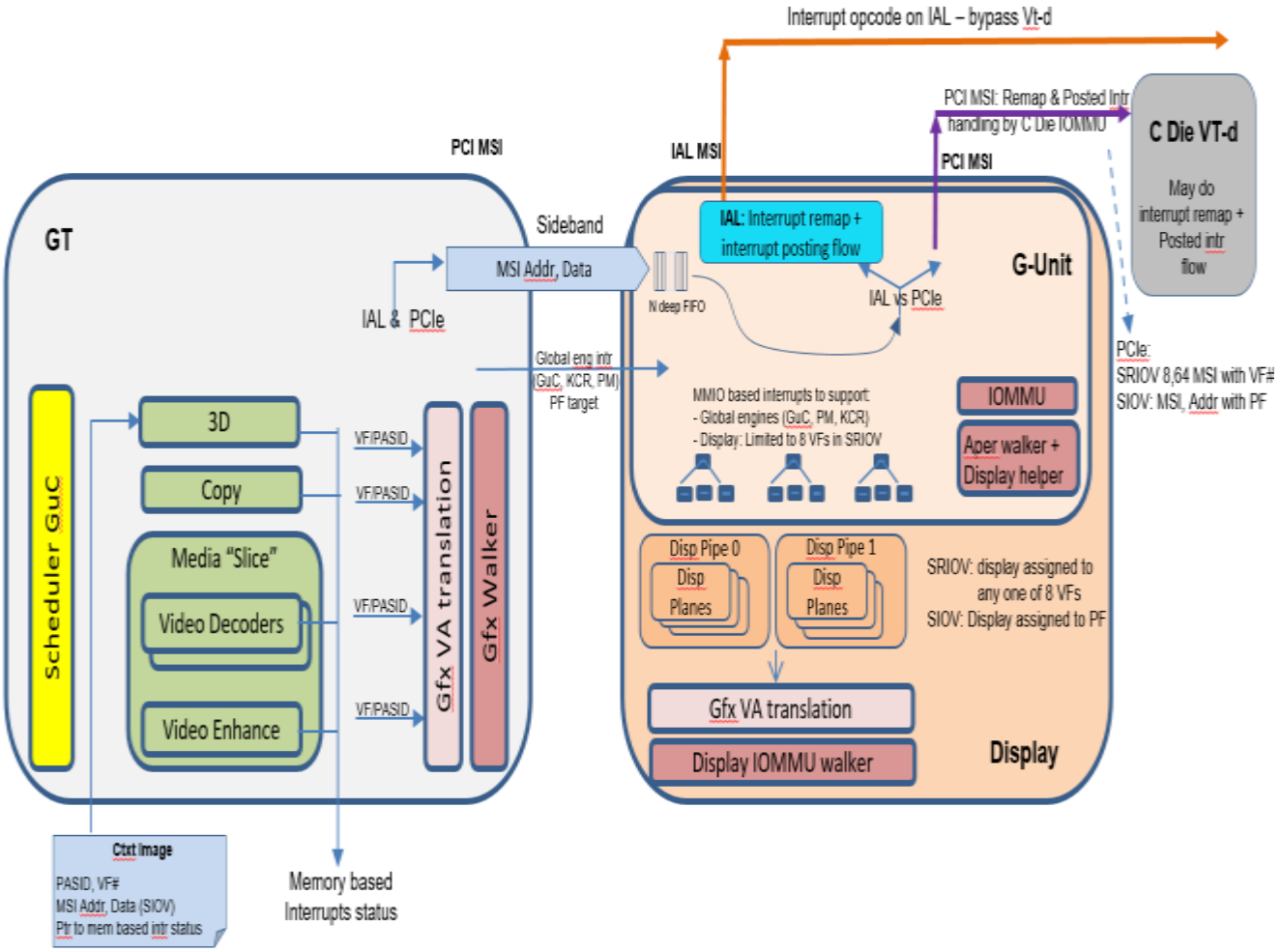
Enables for Engine classes at the two software interfaces are typically complements of each other.

Memory Based Interrupt Status Reporting

SRIOV add support for executing workloads from a large numbers of Virtual Machine or containers.

Schedulable engines (engines with Command Streamers) need a mechanism to signal events to the Host SW through interrupts. MMIO register based interrupts infrastructure used for non-virtualized mode or SRIOV-8 (which supports 8 Virtual Functions) does not scale efficiently to allow delivering interrupts to a large number of Virtual machines or containers. Memory based interrupt status reporting provides an efficient and scalable infrastructure.

Memory based interrupt infrastructure is used by all engines (CS-es and GuC) that need to communicate with a VF/container when running in SRIOV mode. The overall flow is shown in the figure below.



For memory based interrupt status reporting hardware sequence is:

- Engine writes the interrupt event to memory - pointer to memory location is provided by SW. This memory surface must be mapped to system memory and must be marked as un-cacheable (UC) on Graphics IP Caches.
- Engine sends a request (message to G-unit) to trigger an interrupt to host

Interrupt Status Format

MMIO register based interrupt status reporting uses 16 bits per engine where each bit represents an interrupt event.

Memory based interrupt status uses a byte per interrupt event (byte granularity allows engines to update memory status using byte enables instead of requring a read-modify-write).

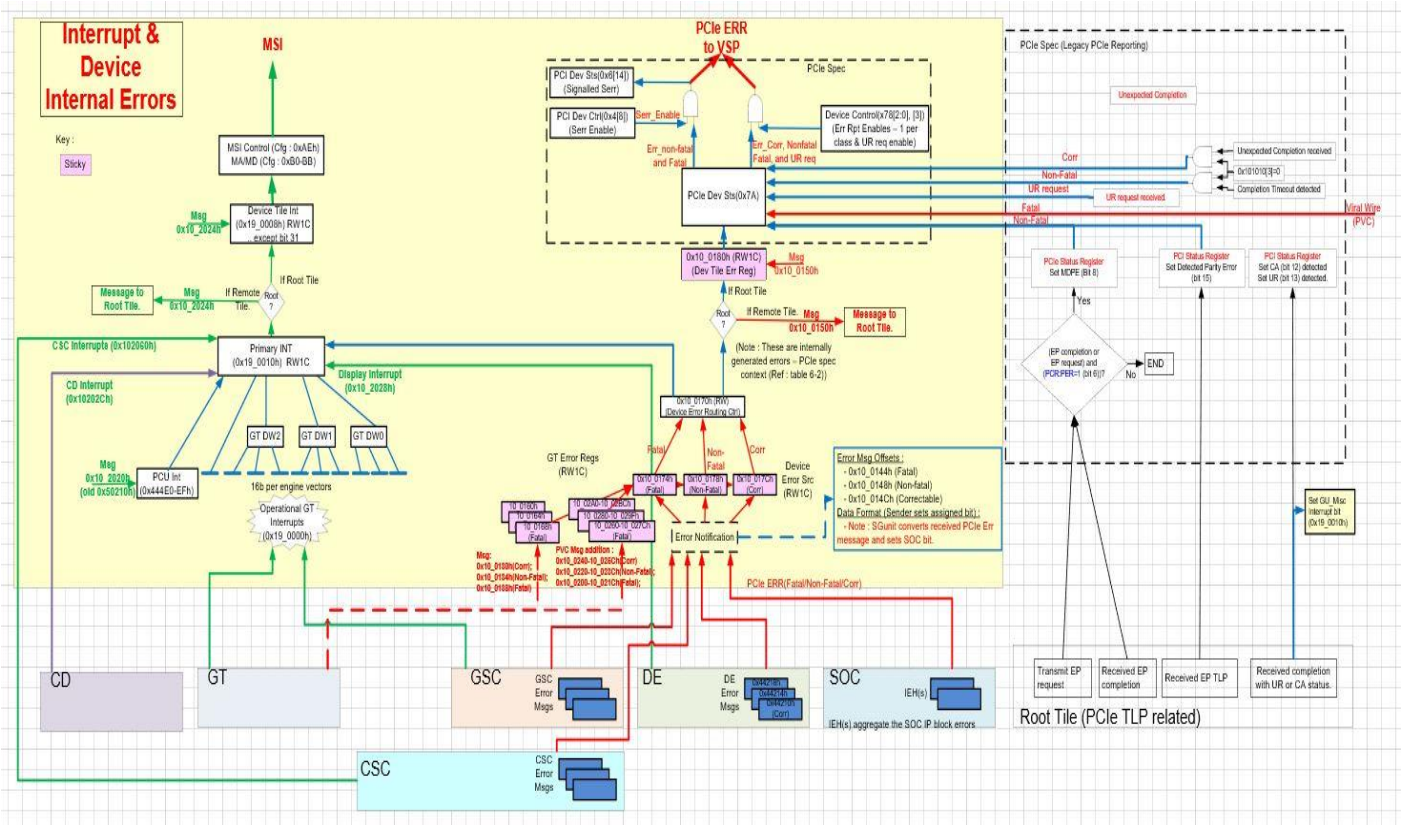
The byte interrupts use the same ordering as bits in interrupt register (i.e Render CS event signalled by bit 0 is represented by byte 0).

Following table summarizes the format for convenience. Please refer to the bitwise definition of the interrupt vectors for each engine for latest field definition.

Byte # in 16B Engine status	**RCS**	**CCS**	**VCS**	**VECS**	**BCS**	**GuC**
15	Catastrophic Error	Catastrophic Error	Catastrophic Error	Catastrophic Error	Catastrophic Error	GuC Interrupt to Host
14	EU Restart	EU Restart				
13	Context Stall	Context Stall				
12	EU ECC					
11	Wait on Semaphore	Wait on Semaphore	Wait on Semaphore	Wait on Semaphore	Wait on Semaphore	
10	Reserved					
9	TR Invalid Tile Detected	TR Invalid Tile Detected				
8	Context Switch	Context Switch	Context Switch	Context Switch	Context Switch	
7	Legacy Context Per Proc Page Fault	Legacy Context Per Proc Page Fault	Legacy Context Per Proc Page Fault	Legacy Context Per Proc Page Fault	Legacy Context Per Proc Page Fault	
6	Watchdog Expired	Watchdog Expired	Watchdog Expired	Watchdog Expired	Watchdog Expired	
5	L3 Parity Error					
4	Pipe Control Notify	Pipe Control Notify	MI Flush DW Notify	MI Flush DW Notify	MI Flush DW Notify	
3	Error Interrupt	Error Interrupt	Error Interrupt	Error Interrupt	Error Interrupt	
2						
1						
0	User Interrupt	User Interrupt	User Interrupt	User Interrupt	User Interrupt	

Gdie Interrupt and Errors

Interrupt and error handling will be supported via a common scale-able structure.



For projects that support Advanced Error Reporting, reference the diagram below.

Visible differences vs. prior integrated graphics products for interrupt :

- Extra level of register hierarchy for interrupt processing
    - o Addition of Tile interrupt register (in Primary Tile)
- Software must RW1C top level registers
    - o Tile interrupt register (in Root tile)
    - o Per Tile Primary Interrupt register
- Addition of SoC and Error bits in the Per Tile Primary Interrupt Register

Message address offsets are used for hardware related communication. The Gfx driver does not use the message addresses.

The Gfx driver (via the CPU) accesses addresses are used by the interrupt handler.

Below is an error/interrupt hardware message table, along with the associated software visible offsets for reference.

Register	Software Accessible Addresses
GT Error Correctable	10_0160h (RW1C)
GT Error Non-Fatal	10_0164h (RW1C)
GT Error Fatal	10_0168h (RW1C)
Reserved	10_016Ch
Device Error Routing Ctrl	10_0170h (RW)
Device Error Source (Fatal)	10_0174h (RW1C)
Device Error Source (Non-Fatal)	10_0178h (RW1C)
Device Error Source (Corr)	10_017Ch (RW1C)
PCIe Err (Corr/NonFatal/Fatal) IOSF SB message	10_0174/8/Ch
PCU Interrupt Register	444E0-EFh
Device Tile Interrupt Register	19_0008h (RW1C)
Primary Interrupt Register	19_0010h (Mostly RW1C .. Except bit 31)
Display Interrupt Message	19_0010h
CD (aka ANR) Interrupt Message	19_0010h
VF Memory Based Interrupt Trigger	

Interrupt Service Routine Pseudocode

This interrupt service routine pseudocode:

ISR routine

{

KMD writes reg 190008h(TILE_INT_REG) to clear bit 31 (Primary INT on Tile_INT_REG) to disable further interrupts while processing ISR

KMD reads Primary Interrupt MMIO reg 190008h = Save that in local variable specially Tile Only

KMD writes MMIO 190008h to clear bits local variable (i.e. TILE_BITS)

// Start per Tile INT service

Per Tile(Loop based on local variable)

KMD reads Per Tile 190010h

KMD Writes the same data that is read in previous step to Clear the (Per Tile 190010h)

KMD finds any of GT bits set DW0 or DW1 INT bits

//Handle GT Engine INT (DW 0). .Example RCS has pipe_Control INT

KMD reads MMIO Per Tile 190018h (GT INTR DW0) and finds bit 0 set (RCS bit).

KMD writes MMIO Per Tile 190070h (GT INTR IIR DW0 Selector) with bit 0 set (to get RCS interrupts)

KMD repeatedly reads Per Tile MMIO 190060h (GT INTR Identity Reg 0) until DataValid (bit 31) is set

KMD stores bits 0:15 of value returned from reg Per Tile 190060h read to local variable holding RCS interrupts

KMD writes Per Tile MMIO 190060h (GT INTR Identity Reg 0) with same bits set to clear those interrupts

KMD writes with the same data that is read to per tile 190018h to clear DW0 interrupt bits

//Handle GT Engine INT (DW 1).. Example VCS0 and VCS3 has STDW_INT

KMD reads Per Tile MMIO 19001Ch (GT INTR DW1) and finds bits 0 and 3 set (VCS0 and VCS3)

KMD writes Per Tile MMIO 190074h (GT INTR IIR DW1 Selector) with bit 0 set (to get VCS0 interrupts)

KMD repeatedly read Per Tile MMIO 190064h (GT INTR Identity 1) until DataValid (bit 31) is set

KMD stores bits 0:15 of value returned from Per Tile reg 190064h read to local variable holding VCS0 interrupts

KMD writes Per Tile MMIO 190064h (GT INTR Identity 1) with same bits set to clear those interrupts

KMD writes Per Tile MMIO 190074h (GT INTR IIR Selector 1) with bit 3 set (to get VCS3 interrupts)

KMD repeatedly Per Tile reads MMIO 190064h (GT INTR Identity 1) until DataValid (bit 31) is set

KMD stores Per Tile bits 0:15 of value returned from reg 190064h read to local variable holding VCS3 interrupts

KMD writes MMIO 190064h (GT INTR Identity 1) with same bits set to clear those interrupts

KMD writes with the same data that is read to per tile 19001Ch to clear DW1 interrupt bits

KMD finds Display Interrupt bit (via 19_0010h) read above

1. Disable Display Interrupt Control (can be done here, or anywhere before step 5)
   - Clear bit 31 of DISPLAY_INT_CTL
   - This is required to prevent missing any interrupts occurring back to back or during the service routine
2. Find the category of interrupt that is pending
   - Read DISPLAY_INT_CTL and record which interrupt pending category bits are set
3. Find the source(s) of the interrupt and clear the Interrupt Identity bits (IIR)
   - Read the IIR associated with each pending interrupt category, record which bits are set, then write back 1s to clear the bits that are set.
   - There can be up to 2 interrupts recorded per source, requiring multiple writes to the IIR to fully clear.
4. Process the interrupt(s) that had bits set in the IIRs
5. Re-enable Display Interrupt Control
   - Set bit 31 of DISPLAY_INT_CTL
   - If interrupts were not fully cleared in step 3, then the display interrupt will re-assert and there will be a new display interrupt in GFX_MSTR_INTR

// For any new interrupts set after the 190010h read, new interrupts are re-generated after setting 31 of 190008h.

End Per Tile LOOP

KMD writes reg 190008h to Set bit 31 // Re-enable INT

//Any pending INT from tile setting in Tile_INT_REG will re-trigger INT and re-Service request in KMD

// Based on local variable (Tiles and engines).. Schedule DPC and call Actual Handler to ***some useful work***

// Example

// Call KMD_PIPE_CONTROL_HANDLER (Tile_ID, Engine ID) {// Fence/Work processing for RCS on particular Tile/GT 0..3}

// Call KMD_STDW_HANDLER (Tile_ID, Engine ID) {//Fence/Work processing for VCS instance 0..8 on particular Tile/GT 0..3}

}

Error Service Routine Pseudocode

Correctable and Non-fatal errors are expected to route as interrupts (and the Gfx driver would then service). Fatal errors are expected to route as PCIe Error messages. In today's platforms, these PCIe error messages result in the OS issuing a Gfx device FLR.

However, if a error handler were to be written for PCIe messaged errors, below is a pseudo-code example.

Red denotes the SW(Error handler's) part.

1) A fatal error is logged in the Error source register

2) Hardware initiates a PCIe Fatal Error message which flows to the CPU.

3) Error handler reads the Device tile error status register and determines which tile.

    a. For example, tile 0.

4) Error handler writes 1 to clear the appropriate Device tile error status bit.

    a. Ie. the Tile 0 bit in this example.

5) Error handler reads the Tile Fatal register and determines the error source.

    a. Ie. Tile 0 read determines SoC fatal error bit.

6) Error handler does what it needs to do for SoC and writes 1 to clear the SoC bit.