Intel® Iris® Xe and UHD Graphics Open Source

Programmer's Reference Manual

For the 2020-2021 11th Generation Intel Xeon®, Core™, Celeron®, Pentium® Gold Processors based on the "Tiger Lake" Platform

Volume 13: General Assets

December 2021, Revision 1.0

Notices and Disclaimers

Table of Contents

General Assets

This is the General Assets section.

MMIO

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	2	subsliceid[0..1]
00000C00	00000DFF	512	AON	No	-	1	-
00000E00	00000FFF	512	AON	No	-	1	-
00001000	00001FFF	4096	AON	Yes	SQIDI	2	subsliceid[0..1]
00002000	000026FF	1792	RENDER	No	-	1	-
00002700	000027FF	256	GT	No	-	1	-
00002800	00002AFF	768	RENDER	No	-	1	-
00002B00	00002FFF	1280	GT	No	-	1	-
00003000	00003FFF	4096	RENDER	No	-	1	-
00004000	000041FF	512	GT	No	-	1	-
00004200	000043FF	512	GT	No	-	1	-
00004400	000048FF	1280	GT	No	-	1	-
00004900	00004FFF	1792					
00005000	000051FF	512					
00005200	000052FF	256	RENDER	No	-	1	-
00005300	000053FF	256	RENDER	No	-	1	-
00005400	000054FF	256					
00005500	00005FFF	2816	RENDER	No	-	1	-
00006000	00006FFF	4096	RENDER	No	-	1	-
00007000	00007FFF	4096	RENDER	No	-	1	-
00008000	000080FF	256	GT	No	-	1	-
00008100	0000813F	64	GT	No	-	1	-
00008140	0000814F	16	RENDER	No	-	1	-
00008150	0000815F	16	RENDER	Yes	DSS	6	subsliceid[0..5]
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	-
00008700	000087FF	256	GT	Yes	SQIDI	2	subsliceid[0..1]
00008800	00008FFF	2048					
00009000	000093FF	1024	GT	No	-	1	-
00009400	0000947F	128	GT	No	-	1	-
00009480	000094CF	80					

000094D0	0000951F	80	RENDER	No	-	1	-
00009520	0000955F	64	RENDER	Yes	DSS	6	subsliceid[0..5]
00009560	000095FF	160	AON	No	-	1	-
00009600	000097FF	512					
00009800	00009FFF	2048	GT	No	-	1	-
0000A000	0000AFFF	4096	GT	No	-	1	-
0000B000	0000B0FF	256	RENDER	No	-	1	-
0000B100	0000B3FF	768	RENDER	Yes	L3BANK	8	subsliceid[0..7]
0000B400	0000B47F	128	GT	No	-	1	-
0000B480	0000BFFF	2944					
0000C000	0000C7FF	2048	GT	No	-	1	-
0000C800	0000CFFF	2048	GT	No	-	1	-
0000D000	0000D3FF	1024	AON	No	-	1	-
0000D400	0000D7FF	1024	AON	No	-	1	-
0000D800	0000D8FF	256	RENDER	No	-	1	-
0000D900	0000DBFF	768	GT	No	-	1	-
0000DC00	0000DDFF	512	RENDER	No	-	1	-
0000DE00	0000DE7F	128					
0000DE80	0000DEFF	128	RENDER	Yes	DSS	6	subsliceid[0..5]
0000DF00	0000DFFF	256	RENDER	Yes	DSS	6	subsliceid[0..5]
0000E000	0000E0FF	256	RENDER	Yes	DSS	6	subsliceid[0..5]
0000E100	0000E1FF	256	RENDER	Yes	DSS	6	subsliceid[0..5]
0000E200	0000E3FF	512	RENDER	Yes	DSS	6	subsliceid[0..5]
0000E400	0000E7FF	1024	RENDER	Yes	DSS	6	subsliceid[0..5]
0000E800	0000E8FF	256	RENDER	Yes	DSS	6	subsliceid[0..5]
0000E900	0000EFFF	1792					
0000F000	0000F0FF	256	GT	No	-	1	-
0000F100	0000FFFF	3840	GT	No	-	1	-
00010000	000147FF	18432					
00014800	00014FFF	2048	RENDER	No	-	1	-
00015000	00016DFF	7680					
00016E00	00016FFF	512	RENDER	No	-	1	-
00017000	00017FFF	4096	RENDER	No	-	1	-
00018000	00019FFF	8192	RENDER	No	-	1	-
0001A000	0001BFFF	8192	RENDER	No	-	1	-
0001C000	0001DFFF	8192					
0001E000	0001FFFF	8192					
00020000	00020FFF	4096	VD0	No	-	1	-
00021000	00021FFF	4096	VD2	No	-	1	-
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	6	subsliceid[0..5]
00024A80	000251FF	1920					
00025200	0002527F	128	GT	No	-	1	-
00025280	000252FF	128	GT	No	-	1	-
00025300	000255FF	768					
00025600	0002567F	128	VD0	No	-	1	-
00025680	000256FF	128	VD2	No	-	1	-
00025700	000259FF	768					
00025A00	00025A7F	128	VD0	No	-	1	-
00025A80	00025AFF	128	VD2	No	-	1	-
00025B00	00025FFF	1280					
00026000	00027FFF	8192					
00028000	0002FFFF	32768					
00030000	0003FFFF	65536	GT	No	-	1	-

MMIO Range Start	MMIO Range End	# Bytes	Wake Target	Replicated / Multicast ?	Replication Group Type	Inst. Count	Steering
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	001C3EFF	4352					
001C3F00	001C3FFF	256	VD0	No	-	1	-
001C4000	001C47FF	2048					
001C4800	001C4FFF	2048					
001C5000	001C5FFF	4096					
001C6000	001C67FF	2048					
001C6800	001C6AFF	768					
001C6B00	001C6BFF	256					
001C6C00	001C6CFF	256					
001C6D00	001C6DFF	256					
001C6E00	001C7EFF	4352					
001C7F00	001C7FFF	256					
001C8000	001C9FFF	8192	VE0	No	-	1	-
001CA000	001CA0FF	256	VE0	No	-	1	-
001CA100	001CBEFF	7680					
001CBF00	001CBFFF	256	VE0	No	-	1	-
001CC000	001CCFFF	4096	VD0	No	-	1	-

001CD000	001CDFFF	4096					
001CE000	001CEFFF	4096					
001CF000	001CFFFF	4096					
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	001D3EFF	4352					
001D3F00	001D3FFF	256	VD2	No	-	1	-
001D4000	001D47FF	2048					
001D4800	001D4FFF	2048					
001D5000	001D5FFF	4096					
001D6000	001D67FF	2048					
001D6800	001D6AFF	768					
001D6B00	001D6BFF	256					
001D6C00	001D6CFF	256					
001D6D00	001D6DFF	256					
001D6E00	001D7EFF	4352					
001D7F00	001D7FFF	256					
001D8000	001D9FFF	8192					
001DA000	001DA0FF	256					
001DA100	001DBEFF	7680					
001DBF00	001DBFFF	256					
001DC000	001DFFFF	16384					
001E0000	001E07FF	2048					
001E0800	001E0FFF	2048					
001E1000	001E1FFF	4096					
001E2000	001E27FF	2048					
001E2800	001E2AFF	768					
001E2B00	001E2BFF	256					
001E2C00	001E2CFF	256					
001E2D00	001E2DFF	256					
001E2E00	001E3EFF	4352					
001E3F00	001E3FFF	256					
001E4000	001E47FF	2048					
001E4800	001E4FFF	2048					
001E5000	001E5FFF	4096					

001E6000	001E67FF	2048					
001E6800	001E6AFF	768					
001E6B00	001E6BFF	256					
001E6C00	001E6CFF	256					
001E6D00	001E6DFF	256					
001E6E00	001E7EFF	4352					
001E7F00	001E7FFF	256					
001E8000	001E9FFF	8192					
001EA000	001EA0FF	256					
001EA100	001EBEFF	7680					
001EBF00	001EBFFF	256					
001EC000	001EFFFF	16384					
001F0000	001F07FF	2048					
001F0800	001F0FFF	2048					
001F1000	001F1FFF	4096					
001F2000	001F27FF	2048					
001F2800	001F2AFF	768					
001F2B00	001F2BFF	256					
001F2C00	001F2CFF	256					
001F2D00	001F2DFF	256					
001F2E00	001F3EFF	4352					
001F3F00	001F3FFF	256					
001F4000	001F47FF	2048					
001F4800	001F4FFF	2048					
001F5000	001F5FFF	4096					
001F6000	001F67FF	2048					
001F6800	001F6AFF	768					
001F6B00	001F6BFF	256					
001F6C00	001F6CFF	256					
001F6D00	001F6DFF	256					
001F6E00	001F7EFF	4352					
001F7F00	001F7FFF	256					
001F8000	001F9FFF	8192					
001FA000	001FA0FF	256					
001FA100	001FBEFF	7680					
001FBF00	001FBFFF	256					
001FC000	001FFFFF	16384					
00200000	0023FFFF	262144					

- 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 MDRB endpoints only
3	0xFD8	Access steering towards SF endpoints only
4	0xFDC	Access steering towards all other endpoints (all but above)

- 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:

Replication Group Type	Description / Notes
SQIDI	• 2 instances • subsliceid: 0..1 • all instances are always present.
DSS	• LP has max 6 DSS • subsliceid: 0..5 • Terminated/disabled when the corresponding dss_enable bit is '0'
L3BANK	• 8 instances • subsliceid 0..7 to access • Terminated/disabled when corresponding fuse_gt_l3disable bit is 'disable'

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

Fuse	Register reflection
fuse_gt_dssen[5:0]	0x913C[5:0]
fuse_gt_l3_disable[3:0]	0x9118[7:0] (fuse is replicated into [3:0] and [7:4])

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
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 inaccessable 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)

Address	Register Name (Read)	Register Name (Write)
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		
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
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

Index	Sym	Register Name
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
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

Index	Sym	Register Name
22h	CR22	Memory Read Latch Data

GUC

GUC is part of the System Interfaces Volume.

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 can take place. This is used to ensure atomic updates of several registers.	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.

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.

Arming Memory Based Doorbells

Doorbells (As described in the GPU doorbells section in the "Memory View" chapter) are used to submit work to the hardware. Each doorbell monitors a memory address (cacheline) that detects a write to the cacheline and generates an interrupt to the MinIA core.

A distributed doorbells infrastructure is provided. Distributed doorbell units save doorbells in parallel so there are some performance benefits if SW distributes doorbells evenly across Doorbell Units. Hence the infrastructure allows SW to explicitly pick the IDI (ring interface) on which the doorbell needs to be installed.

The memory address programmed into the doorbell register needs to be Physical address. Since addresses are hashed on IDI (physical address based hashing), this address needs to have specific attributes to land on a specific IDI. To enable this Graphics KMD picks the doorbell Page Address, GuC Shim HW computes the cacheline address within the page.

When virtualization is enabled, the memory address provided by the Graphics Kernel Mode Driver needs to be translated from Guest Physical Address(GPA) to Host Physical Address(HPA) before getting programmed into the doorbell register. To conform to the security requirements of virtualization where only Virtual Machine Manager is aware of the HPA values, the GPA --> HPA translation is automatically done by GuC Shim HW as described below. The Guc Shim computes the cacheline address based on the HPA of the doorbell page address and this is returned to the KMD so that it has a fully formed Doorbell cacheline address.

Note that the hashing algorithm is different depending on how many LLC/CBO slices are present in the part. GT receives a fuse value at manufacturing that indicates which hashing rule should be used. That fuse value is not communicated directly to GUC hardware; instead, the Graphics Kernel Mode Driver is required to copy information about the hashing mode fuse from a register in the GT perimeter (SNOOP_FILTER_Q_STATUS - offset 0xB00[30]) to the GUC shim (SHIM_CTRL_GUC0_REG - offset 0xC068[10]). Driver must do this prior to arming any doorbells. Note that this value is retained across RC6 cycles but is lost on FLR, warm/cold reset, etc. Thus driver must program this value on every driver start/load as well as after any driver initiated FLR. Stated differently, any time driver loads GUC firmware, it should also perform this B00[30]-> C068[10] copy operation.

Doorbell Arming Sequence

- The memory page address to be monitored is picked by the Graphics Kernel Mode Driver and provided to the MinIA FW.
- Graphics KMD communicates the doorbell details to MinIA FW: Doorbell memory Page-address to be monitored, IDI#, Doorbell ID

- MinIA FW computes Doorbell MMIO
- MinIA FW programs the memory address to be monitored to the doorbell range: 0x1000 - 0x17FF range.
- GuC Shim HW intercepts the Doorbell MMIO targeted cycles to provide a GPA --> HPA translation and installs the translated address into the doorbell. Using the returned HPA, HW also computes the appropriate cacheline address that is returned to the FW for propagaton back up to the KMD. Detailed HW sequence is described below.
- MinIA FW monitors register 0xC090 and 0xC094 waiting for HW to signal completion of the doorbell install step and then checks status to determine if this was completed successfully.

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.

Context:	The following table provides a view of the GuC address space

Address Top	Address Bottom	Space	Description
0xFFFF_FFFF	0xFFE0_0000	Graphics MMIO	2 MB off the top of the 4GB space
0xFFDF_FFFF	0xFEE0_1000	Hole	
0xFEE0_0FFF	0xFEE0_0000	LAPIC	4KB that houses the Local APIC registers. The accesses to this region are redirected to the LAPIC before they get to the shim decoder.
0xFEDF_FFFF	WOPCM_TOP	DRAM - Graphics Memory	Section should be decoded as: **Upper bound:** 4GB - 2MB(Gfx mmio) - 16MB **Lower bound:** 80KB Lower bound accounts for 16KB + 64KB
WOPCM_TOP-1	0x0006_8000	DRAM - WOPCM	Write Only Protected Content Memory (WO-PCM) This allows for code to straddle SRAM and memory (as described later)
0x0006_7FFF	0x0000_8000	SRAM space	384 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.

Context:	MinuteIA L1 Cacheability

By default, the Shim uses the following rules for caching in the MinuteIA L1 cache.

- 0xFEE0_0000 - 0xFFFF_FFFF: Un-Cached (Covers the 2MB Gfx address space + 16MB hole)
- 0x0000_0000 - 0xFEDF_FFFF: Cached (Everything below)
- By default, MinIA treats all WOPCM cycles as WB while Gfx-GTT cycles are WT. If C064[16] is set, all cycles (below xFFE0_0000) from MinIA are WB.

Graphics driver can create an Un-cached region window in the lower range described above, by programming the Non_Cacheable_Region_Base and Non_Cacheable_Region_Limit registers.

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
UOS_FULL_HASH_LO - uOS Full Hash Low
UOS_FULL_HASH_HI - uOS Full Hash High
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 HuC micro-kernel. To load a third party HuC ukernel, the third party GuC ukernel must be loaded first. An authenticated GuC ukernel can then be invoked to load a HuC ukernel. (If a third party HuC 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 Group Registers
Doorbell Group Registers
Engine Interrupt Regiters
GuC Timer Registers
GuC DMA Interrupt Registers
GuC Host Registers

Interrupt Group Registers

GUCINT uses two registers to map all incoming interrupts to the 256-bit interrupt vector sent to the LAPIC. Internal to the LAPIC, the 256-bit interrupt vector is organized as 16 groups of 16 vectors. GUCINT maps each group of interrupts to a single group in the vector sent to LAPIC. Note that the higher the group number, the higher the priority of the group of interrupts to which it is assigned.

Interrupt Group Registers
INTR_GROUP_1 - Interrupt Group 1
INTR_GROUP_0 - Interrupt Group 0

The following interrupt groups are supported:

Interrupt Group
Doorbells (counts as 8 groups)
Semaphores
Engines
IOMMU
DMA/TIMER/FLR
HOST

Each group of interrupts needs 4 bits to map to the correct group in the 256-bit vector.

Interrupt Group 0 Register maps the 8 groups of Doorbells. **Interrupt Group 1 Register** maps the remaining 5 groups listed above.

Programming Note	
Context:	Interrupt Group Registers

- All groups must be unique. Only Doorbell groups may share a value with another doorbell group.
- The group value cannot be 4'h1. That is illegal. This restriction applies to both GROUP registers.
- Group values cannot dynamically change from one nonzero value to a different nonzero value.

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 Enhancment ..).

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, its 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 FIFO's 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 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 part of the BSpec 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 pin point 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.

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:

Counters layout for various values of the "Counter Select" from the register:

Counter Select = 000

A-Cntr 10 (low dword)	A-Cntr 9 (low dword)	A-Cntr 8 (low dword)	A-Cntr 7 (low dword)	GPU_TICKS	CTX ID	TIME_STAMP	RPT_ID
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 11 (low dword)

Counter Select = 010

A-Cntr 10 (low dword)	A-Cntr 9 (low dword)	A-Cntr 8 (low dword)	A-Cntr 7 (low dword)	GPU_TICKS	CTX ID	TIME_STAMP	RPT_ID
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 11 (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

Counter Select = 111

C-Cntr 3	C-Cntr 2	C-Cntr 1	C-Cntr 0	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

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	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

Counter Select = 111

C-Cntr 3	C-Cntr 2	C-Cntr 1	C-Cntr 0	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

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[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. This field has the same format at TIMESTAMP register defined in Vol1C.4 Render Command Streamer BSpec.
RPT_ID[46:38]	Reserved (for future use)
RPT_ID[35:34]	Reserved (for future Tile IDs)
RPT_ID[31:0]	This field has several sub fields as defined below: 31:26 **SourceID[5:0]**

Field	Description	
		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 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	**Timer Enabled**
	15:0	Reserved

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	OAG Offset	OAR Offset
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		

Symmetrical Registers	OAG Offset	OAR Offset
OAPERF_A16 - Aggregate Perf Counter A16		
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		

Symmetrical Registers	OAG Offset	OAR Offset
OAPERF_A33 - Aggregate_Perf_Counter_A33		
OAPERF_A34 - Aggregate_Perf_Counter_A34		
OAPERF_A35 - Aggregate_Perf_Counter_A35		
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 Here.

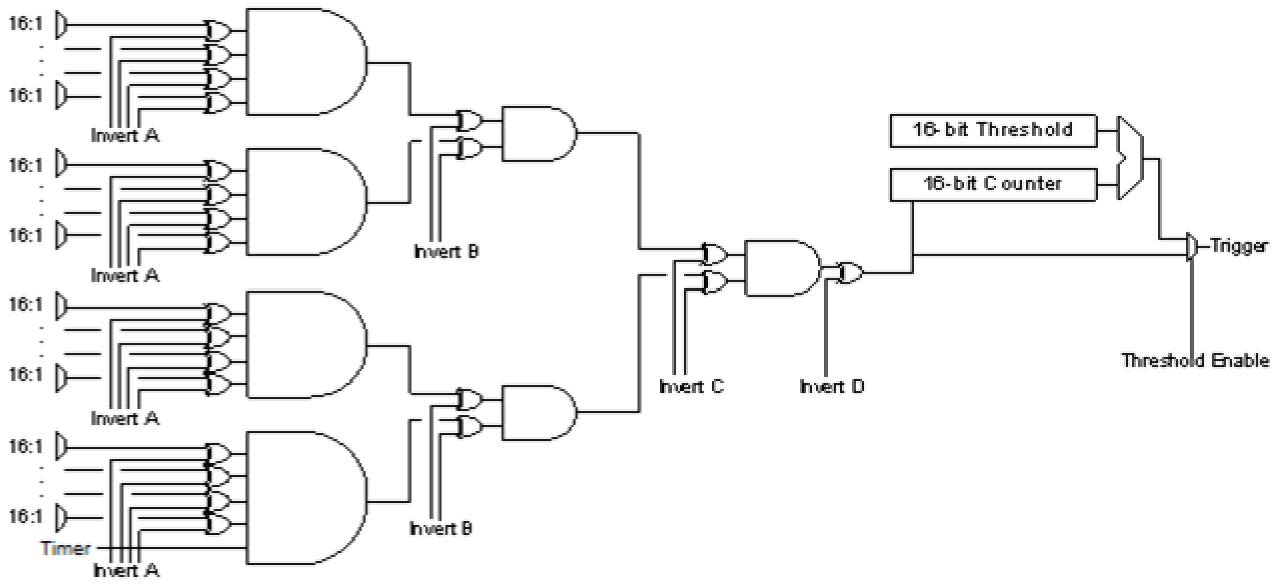
Details of Start Trigger Behavior

- 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.

While architectural intent was that Start Trigger logic would control all qualified counter types (A/B/C), there is a long-standing implementation bug whereby start trigger logic only affects B/C counters.

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.



Note that counters which are 40 bits wide are split in the report format into low DWORD and high byte chunks for simplicity of HW implementation as well as SW-friendly alignment of report data. The performance counter read logically done before writing out report data for these 40-bit counters is guaranteed to be an atomic operation, the counter data is simply swizzled as it is being packed into the report.

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 OACONTROL (). 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.

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 **OARCONTROL**. 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 OACONTROL (). 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.

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 **OARCONTROL**. 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).
A1	# of Vertex Shader Threads Dispatched	Count of VS fused threads dispatched to EUs
A2	# of Hull Shader Threads Dispatched	Count of HS fused threads dispatched to EUs
A3	# of Domain Shader Threads Dispatched	Count of DS fused threads dispatched to EUs
A4	# of GPGPU Threads Dispatched	Count of GPGPU fused threads dispatched to EUs. Available on both qualified and free-running counters.
A5	# of Geometry Shader Threads Dispatched	Count of GS fused threads dispatched to EUs
A6	# of Pixel Shader Threads Dispatched	Count of PS fused threads dispatched to EUs. Available on both qualified and free-running counters.
A7	Aggregating EU counter 0	User-defined (details in Flexible EU Event Counters section)
A8	Aggregating EU counter 1	User-defined (details in Flexible EU Event Counters section)
A9	Aggregating EU counter 2	User-defined (details in Flexible EU Event Counters section)
A10	Aggregating EU counter 3	User-defined (details in Flexible EU Event Counters section)
A11	Aggregating EU counter 4	User-defined (details in Flexible EU Event Counters section)
A12	Aggregating EU counter 5	User-defined (details in Flexible EU Event Counters section)
A13	Aggregating EU counter 6	User-defined (details in Flexible EU Event Counters section)
A14	Aggregating EU counter 7	User-defined (details in Flexible EU Event Counters section)
A15	Aggregating EU counter 8	User-defined (details in Flexible EU Event Counters section
A16	Aggregating EU counter 9	User-defined (details in Flexible EU Event Counters section)
A17	Aggregating EU counter 10	User-defined (details in Flexible EU Event Counters section)
A18	Aggregating EU counter 11	User-defined (details in Flexible EU Event Counters section)
A19	Aggregating EU counter 12	Available on both qualified and free-running counters

Counter #	Event	Description
		User-defined (details in Flexible EU Event Counters section)
A20	Aggregating EU counter 13	Available on both qualified and free-running counters User-defined (details in Flexible EU Event Counters section)
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.)
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.)
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.)
A24	2x2s Killed in PS	Number of samples entirely killed in the pixel shader as a result of explicit instructions in the kernel (counted in 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.)
A25	2x2s Failing post-PS Tests "POSTPS_DEPTH_STENCIL_ALPHA_FAIL"	Number of samples that entirely fail "late" tests (i.e. tests that can only be performed after pixel shader execution). 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.)
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.
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.

Counter #	Event	Description
A28	2x2s Requested from Sampler	Aggregated total 2x2 texel blocks requested from all EUs to all instances of sampler logic.
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.
A30	SLM Reads	Total read requests from an EU to SLM (including reads generated by atomic operations).
A31	SLM Writes	Total write requests from an EU to SLM (including writes generated by atomic operations).
A34	Atomic Accesses	Aggregated total atomic accesses from all EUs. This counter increments on atomic accesses to both SLM and URB. **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).

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 4 clocks where a single EU met this condition.
SPM1	EU IPC	Event counts the number of ISA FPU/EM instructions that GT processes. It comprehends cases where multiple instructions are processed in a single clock. Each increment of this event reflects 8 instructions processed.
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. The per-clock increment is added to an accumulator each clock, a single increment of this event reflects that the accumulator has reached 8.
SPM3	EU Not Idle	Event reflects the condition where an EU is not idle. Each increment of the event reflects 8 clocks where a single EU met this condition.

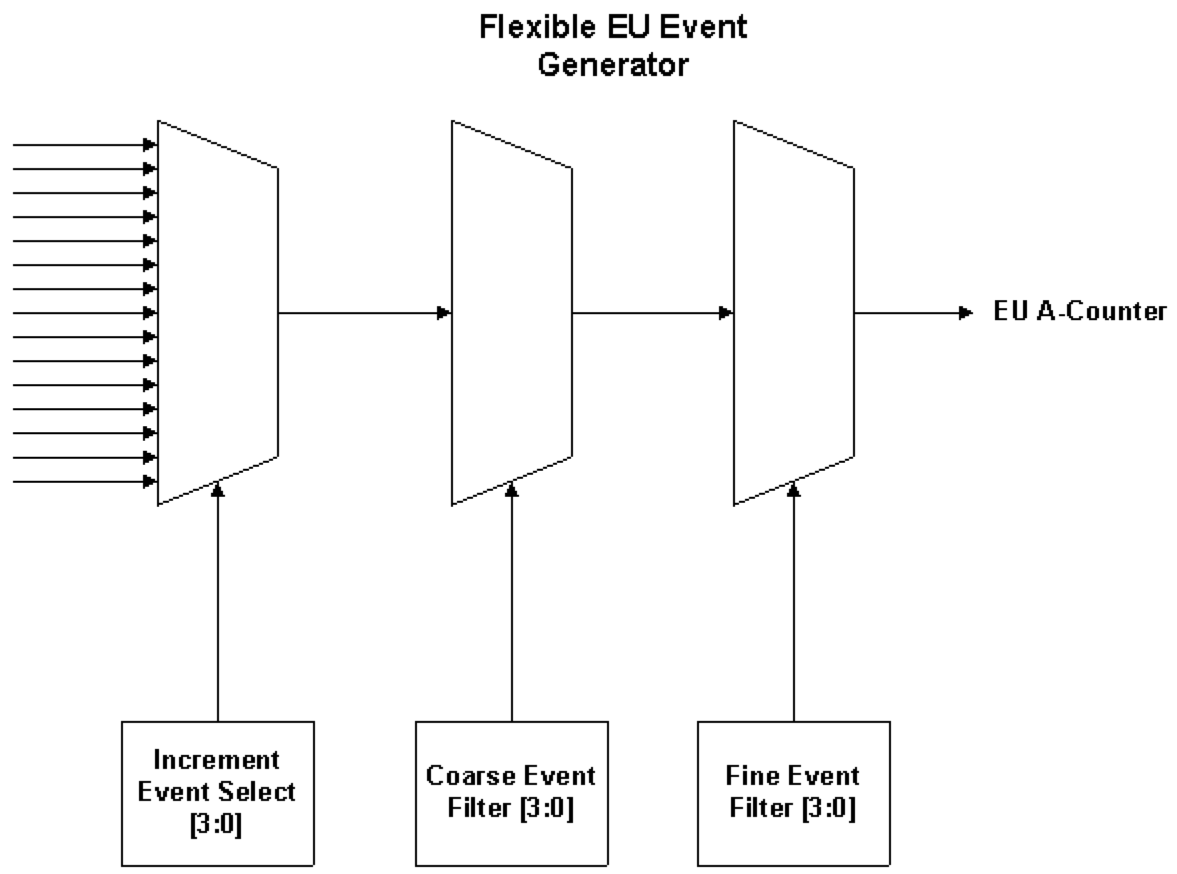
Counter #	Event	Description
SPM4	Sampler Not Idle	Event counts sampler activity.
SPM5	EU Stalled & Sampler Not Idle	Event reflects the condition where the EU has sent a request(s) to sampler and all threads on the EU are stalled. Please note that the EU could be stalled for reasons other than sampler as well. Each increment of the event reflects 8 clocks where a single EU met this condition.

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).

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.


Flexible EU Event Generator
EU A-Counter
Increment Event Select [3:0]
Coarse Event Filter [3:0]
Fine Event Filter [3:0]

Supported Increment Events

Increment Event	Encoding	Notes
EU_INST_EXECUTED_ALU0_ALL	0b00000	Signal that is high on every EU clock where the EU FPU pipeline is actively executing an ISA instruction.
EU_INST_EXECUTED_ALU1_ALL	0b00001	Signal that is high on every EU clock where the EU EM pipeline is actively executing an ISA instruction.
		Only fine event filters 0b0000, 0b0111, 0b1000, 0b1001, and 0b1010 are supported with this increment event.
EU_INST_EXECUTED_SEND_ALL	0b00010	Number of instructions 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	Signal that is high on every EU clock where the EU FPU and EM 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.
	0b01111	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.
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.
TS Thread Filter	0b0110	For increment events 0b00000/0b00001/0b00010, masks increment event unless the FFID which dispatched the currently executing thread equals FFID of TS.
		For increment events 0b00100/0b00101, masks increment event unless at least one of the loaded threads was dispatched by TS.
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).

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 executed	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 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

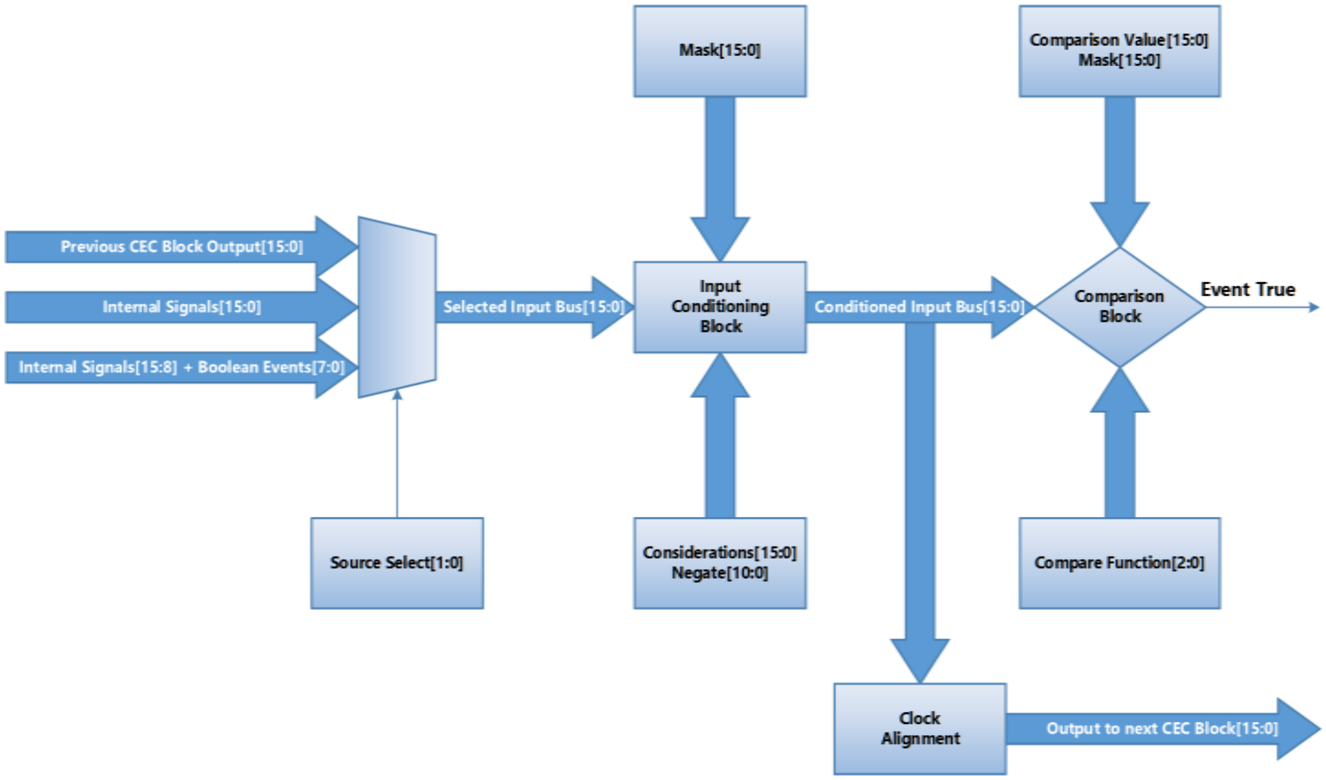
EU_PERF_CNT_CTL5 - Flexible EU Event Control 5

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

GPU Doorbells

On current platforms, graphics workload is submitted to the graphics hardware through multiple SW layers: Application, user mode driver (UMD), Graphics Runtime, kernel mode driver (KMD) and then finally down to hardware. This submission path incurs the penalty of traversing the various SW layers and then a ring3-to-ring0 switch when the kernel mode driver finally submits the workload by writing to some graphics register. With the advent of the GPGPU computing, applications like to use the parallelism provided by the graphics hardware for graphics and non-graphics workloads. However, due to the large latency introduced by the legacy submission mechanism, an App has to determine if the computation cycles required for a workload exceeds a threshold to justify incurring the latency that will be incurred to get this workload running on the graphics hardware.

The proposed GFX doorbell solution provides a mechanism for submission of workload to the graphics hardware by a ring3 application - without the penalty of ring transition for each workload submission. The structure has the following components:

- Assignment of a work queue page by Kernel Mode driver into an Apps address space
- Infrastructure in GTI/BGF to detect submission of work into a N work queue pages; GTI detects a write to a specific cacheline in the doorbell page.

- Message from GTI/BGF to a micro-controller in the graphics engine indicating new work has arrived in a queue.
- Microcontroller inspects queue and determines the specific graphics engine to submit the workload to (render engine,...).

Essentially, the hardware will support monitoring of 256 cache-line addresses that are on separate 4KB pages. Each GPGPU App/thread negotiates with KMD on the address of this 4KB page (in app's space) as a "doorbell" page. This negotiation happens during the app's call to the driver (KMD) for open GPGPU. The page is assigned for doorbell operation which is basically a way to MONITOR accesses to the specified cacheline within the doorbell page.

This portion of the document details the interactions needed in GT BGF Doorbell Block as well as the HW pieces.

The Concept and Usage Flow

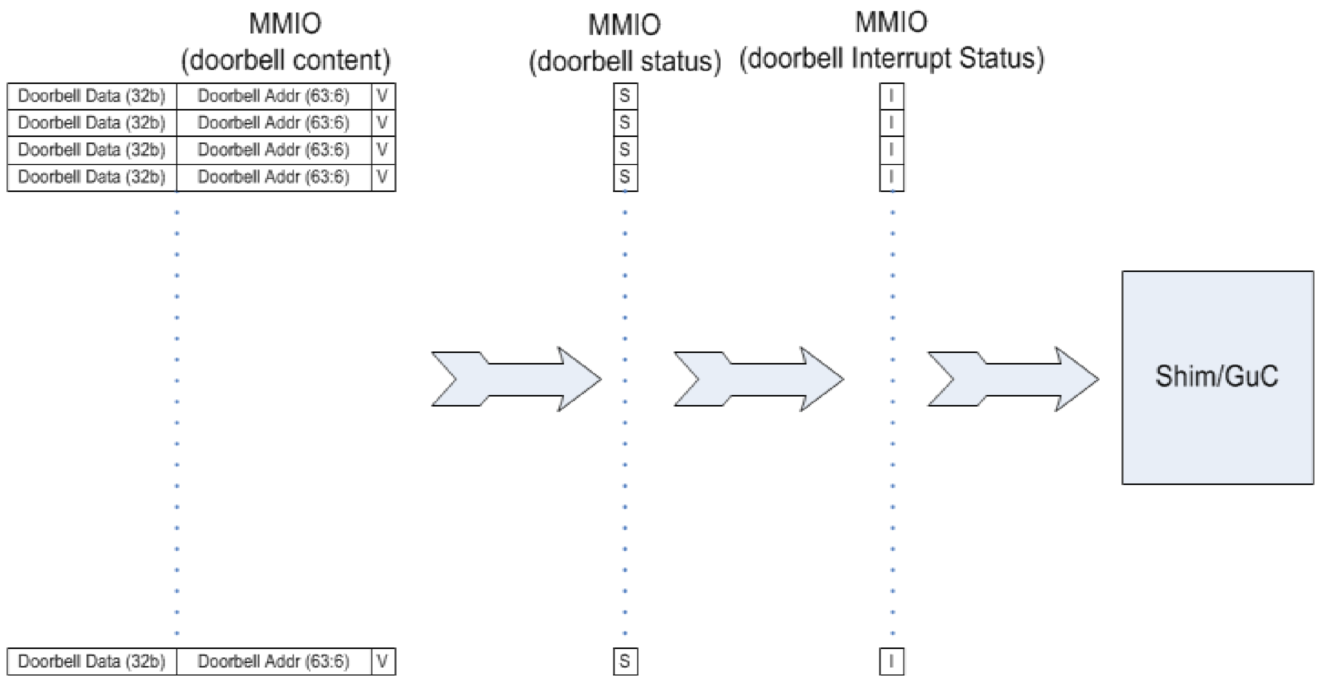
The interactions between the doorbell hardware and software flows described as viewed from the doorbell controller:

Doorbell Structures

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.

Doorbell Page and Cacheline

As part of the "*openGPGPU*" call between the App and GFX driver a doorbell page is assigned. The page is allocated within the WB space (requirement) and pinned to prevent any faulting while exchanging semaphores (requirement). The doorbell monitoring hardware operates on each doorbell at a 64B granularity.

App and KMD negotiates which cacheline (64Byte) entity within the doorbell page should be picked as doorbell line. Even though this could be simply the first 64B of the page, it is highly recommended to pick a rotating 64B entity to maximize the hardware performance. The suggested algorithm should be:

Doorbell CL# = Bell# mod 64

This simply means Bell0,64,128,192 goes to CL0 of the page, Bell1,65,129,193 goes to CL1 of the page, Bell2,66,130,194 goes to CL2 of the page, and so on.

Only the first 2 DWords of the Doorbell CL is relevant, the rest is not used by the doorbell hardware. Same is also applicable for the rest of page which could be utilized for some other means.

1st DWord (4 bytes) contains 1-bit for doorbell status: Active vs. Inactive:

- Active - Bit0=1: Doorbell is Active for Application.
- Inactive - Bit0=0: Doorbell is suspended for the Application, hence Application should re-negotiate the doorbell.

Note that as part of the negotiation a different cacheline can be picked within the doorbell page, different than what was originally agreed between GFX Driver and Application.

GFX driver control the initial setting of bit[0] of DW[0] to enable and disable doorbell for a given application. However HW setup of doorbell can be done only via GuC and disabling in HW is self-detected via snooping driver's clearing of DW0[0].

2nd DWord (4 bytes) carries a cookie value. It is reset to "0h" by the GFX driver prior to being assigned and acquired and incremented by the App. Note that value "0h" is a reserved value for cookie which App should never write. A roll-over increment should go from "FFFF_FFFFh" to "0000_0001h". Details on how cookie and flag value should be processed by application and hardware are given in later sections.

Work Queue and DoorBell Creation

App calls the driver for "OpenGPU()API", UMD calls the KMD and KMD sets up the queues. KMD maps the queue addresses to application space. KMD assigns a doorbell if one available:

- **If available:** Doorbell page gets allocated and assigned to Applications space. Doorbell cacheline is decided and QW0 is initialized as:
    - DW0 = "0000_0001h"
    - DW1 = "0000_0000h"
- **If not-available:** Driver has to victimize doorbell from an existing application and wait for it to clear from HW before assigning to new application requesting one.

KMD assigns a unique ID (i.e. Context ID) which is a 32b value to define the context. KMD allocates HW context memory for "State" - up to 3MB.

Note that the SW flows here are a high level description; the actual flows may contain further steps and will be detailed in SAS.

KMD communicates the contextID as a handle to memory (work queue) to GuC. If there is a doorbell assignment, GuC also receives the doorbell number as well as the physical address of the cacheline decided.

GuC has to ensure there are no pending doorbell events before updating the doorbell register for use. To ensure this, GuC should read "doorbell status vector". If there are any pending events to the same doorbell that is about to be assigned, GuC expedites the service of the matching vector and waits for status to clear. If status is cleared, GuC needs to check the "doorbell interrupt status" register in the GuC/Shim to ensure there are no pending interrupts. Once both stages are clear, GuC can proceed with re-assignment of the doorbell.

The doorbell address is GPA. Rest of the flow is between the GuC Shim/GAM and Doorbell control block:

- GuC the address (GPA) of the doorbell page/cacheline. Given the address is GPA, a 2nd level page table translation is required before the monitor address can be written to doorbell block.
- GuC sends the following message pair to GAM asking for a GPA to HPA translation.
- Doorbell Address programming has to be routed thru GuC/Shim - driver (KMD) is not allowed to update the doorbell addresses directly.

Doorbell Flows

Doorbell Controller has a single primary flow of acquiring the ownership of a doorbell cacheline and detecting and reporting doorbell rings. Besides the main flow, there is various side flows to handle power states.

There are two cases where the main (ownership) flow can be invoked:

1. VALID bit in one of the doorbell registers transitions from 0=>1 indicating a new assignment of doorbell. This is considered initial flow.
2. A snoop from LLC/Cbo removing the ownership of a doorbell cacheline that was already being monitored. This is considered subsequent flow.

Both flows are handled same, as they overlap whether this is the initial ownership flow or subsequent case.

Doorbell controller collects the results of the RFO where data gets processed:

- If DW0[0] - flag - is seen as "0", that means s/w already victimized the doorbell and in the process of re-assigning it. However it is possible that application had a chance to sneak in work at the very last opportunity. And the doorbell register is still valid.
- If DW0[0] - flag - is seen as "1", the doorbell is still active.

Both cases are handled similarly in the doorbell controller.

- The value of cookie is checked:
  - If cookie is "0000_0000h", this is the initialized value of the cookie by GFX driver. Application did not use the doorbell yet. Doorbell block initializes the cookie value stored in h/w and does not ring the bell.
  - If cookie is same as what was stored in the cookie register this means either we are still at the initialized value or the ownership was lost due to LLC victimization and there is no RING event.
  - If cookie value is different than the previous value and it is not all "0's", this is a VALID RING event.
- In case of a valid RING event,

    - Doorbell controller sets up the corresponding bit flag in the "Active Doorbell Status Vector".
    - If in RC6, generate a wake-event to GT-Shim pm block.
        - Doorbell Block has to be aware of the RC6 status and need to be able to communicate the wake up request to Shim-PM block.
        - The message to Shim has to be hold-off until we wake up GT.
    - If not in RC6 or woken up from RC6, generate a message to GuC unless the 32b group of active vector is non-Zero. Meaning there has been already a message to Guc and GuC did not have a chance to read the corresponding "Active Doorbell Vector" piece. The communication between the doorbell monitoring hardware and GuC is handled via internal messaging.
- Once a group's message is sent (via message channel) subsequent rings to the same group does not cause a message to GuC. The only time a message is sent if the previous value of the doorbell vector for the corresponding group was "0000_0000h".
- At some point, GuC will read the vector from the doorbell block to see which doorbells have rung. Same read event should clear the corresponding doorbell vector register allowing any subsequent rings to generate a new message back to GuC.

Assigning Doorbells

Doorbells assignments are straight forward - see doorbell creation section

In addition s/w shall not assign the same cacheline to more than one doorbell. Such assignments will create unexpected behavior when the doorbell is rung.

Removing Doorbells

Doorbells are floating hardware resources, and they can be temporarily attached to Apps. GFX Driver can remove an application's doorbell without the need to notify or negotiate with that App. The removal process described requires KMD to poll doorbell status locations in HW to ensure doorbell is freed before re-assigning. And re-assignment process is done making sure all previous interrupts from this doorbell are processed.

An application losing a doorbell simply needs to request one as it realizes that its doorbell is lost. This is considered as a doorbell virtualization process where GFX driver and GuC can move a limited number of doorbells between a practically unbounded (2^31) number of applications.

The removal processes for doorbells are managed by the driver (KMD). However, it is completely asynchronous to what applications are doing with the doorbells and where HW is in processing these doorbells. The following flow is used to de-activate a doorbell.

As part of the de-activation:

1. GFX driver decides which doorbell to victimize and clears the DW0[0] of the doorbell cacheline in WB space. This is an indication to the GPU doorbell controller that the corresponding doorbell is removed. The doorbell block in GPU will see the update from IA core as part of the snoop monitoring (see later sections on how application checks the flag via compare&exchange).

From this point on, Application will not be able to ring that doorbell, and it needs to request a new one once it realizes its doorbell is lost.

2. The GFX driver clearing DW0[0] of the doorbell is seen by the GFX doorbell hardware as de-activation. However, there is a possibility that the App had a chance to use it to increment the cookie just before doorbell got de-activated. The doorbell block compares the cookie value new vs old to make a decision to set the interrupt status bit or not.

   From this point on we know that the particular doorbell cannot be used and HW will not match incoming snoops even if there may be snoops to the line (Valid bit is managed by the HW when getting cleared).
3. After GFX driver clears the DW0[0] of the doorbell that is getting victimized, it polls the doorbell register VALID bit in HW which indicates when HW got to see the victimized doorbell and cleared the VALID bit state. This also ensures that all previous updates of the victim doorbell are now registered in the HW.
4. Once the GFX driver sees the VALID bit cleared in the doorbell register, it is free to re-allocate the same doorbell for another APP. Note that assignment of the doorbell requires GuC to clear all previous usages of the previous owner of that doorbell.

Now the doorbell is ready to be re-assigned to any application.

Application Flow

The application use of doorbell is described in this document for completeness and to give the design owners what to expect.

Application can start using the doorbell once agreed with the GFX driver, from this point the main part of the flow is to be able cache the value of the cookie (if initial assignment application knows cookie is "0" and next value is "1"). The flow relies on the fact that application simply increments the cookie by "1" to indicate a doorbell event. And application has the value of cookie and the next value that it wants to update to. To understand how the rest of the sequence works, one needs to understand the inner works of the LOCK_CMPXCHG8B.

1. Assumption thread knows the value of the cookie - if this is the first time using it, it knows KMD initialized to "0". Thread prepares ECX = 1
2. Thread does compare exchange 8B (LOCK_CMPXCHG8B)

a. Outcome#1: **Success** - thread gets "ZF == 1", rings the doorbell successfully and resumes execution
b. Outcome#2: **Fail:** Bell is present (EAX =1) but EDX is not the expected value - *means some other thread is sharing the same bell.*
   i. Load ECX with EDX, and INC ECX - *prepare the next value for destination*
   ii. And loop to compare exchange (step#2) - *at this point this is a RACE to the bell between multiple threads. The thread that succeeds updates the bell*
c. Outcome#3: **Fail** : Bell is not there (EAX = 0) - ask for a bell and loop to step#1

In both cases algorithm is self correcting. For the case of one doorbell to one application thread, Outcome#2 is never possible given the application thread exactly knows what the value of the cookie is (it is the only thread writing to it). Outcome#3 is possible if GFX driver decides to remove the doorbell.

For the case of one doorbell to many application threads, each application thread has an idea of what the value of the cookie could be. If this is the first time the thread is writing the cookie value, it assumes "0"; if not the first time, it assumes whatever the value it wrote last time. In this case Outcome#2 becomes possible given that other threads are doing LOCK_CMPXCHG8B on the same data structure as well. However the good part of the LOCK_CMPXCHG8B is that, operation is atomic and a fail case (i.e. mismatch of the compare) returns the most up to date value from memory and does not change the contents of the memory. For multiple application-thread case, all threads are going to be racing to the flag while making successful updates and any doorbell ring is going to get uC to check and process all application-threads' work queues.

Atomic operation, "CMPXCHNG8B", is a required instruction to use if multiple application threads are mapped to same doorbell or same doorbell is virtualized via KMD between multiple applications. The latter part refers to a case where KMD removes a doorbell without letting the application know about it. If there is no multiple thread or virtualization case for doorbells, simple memory operations are suffice to operate with doorbells.

Doorbells and Lock

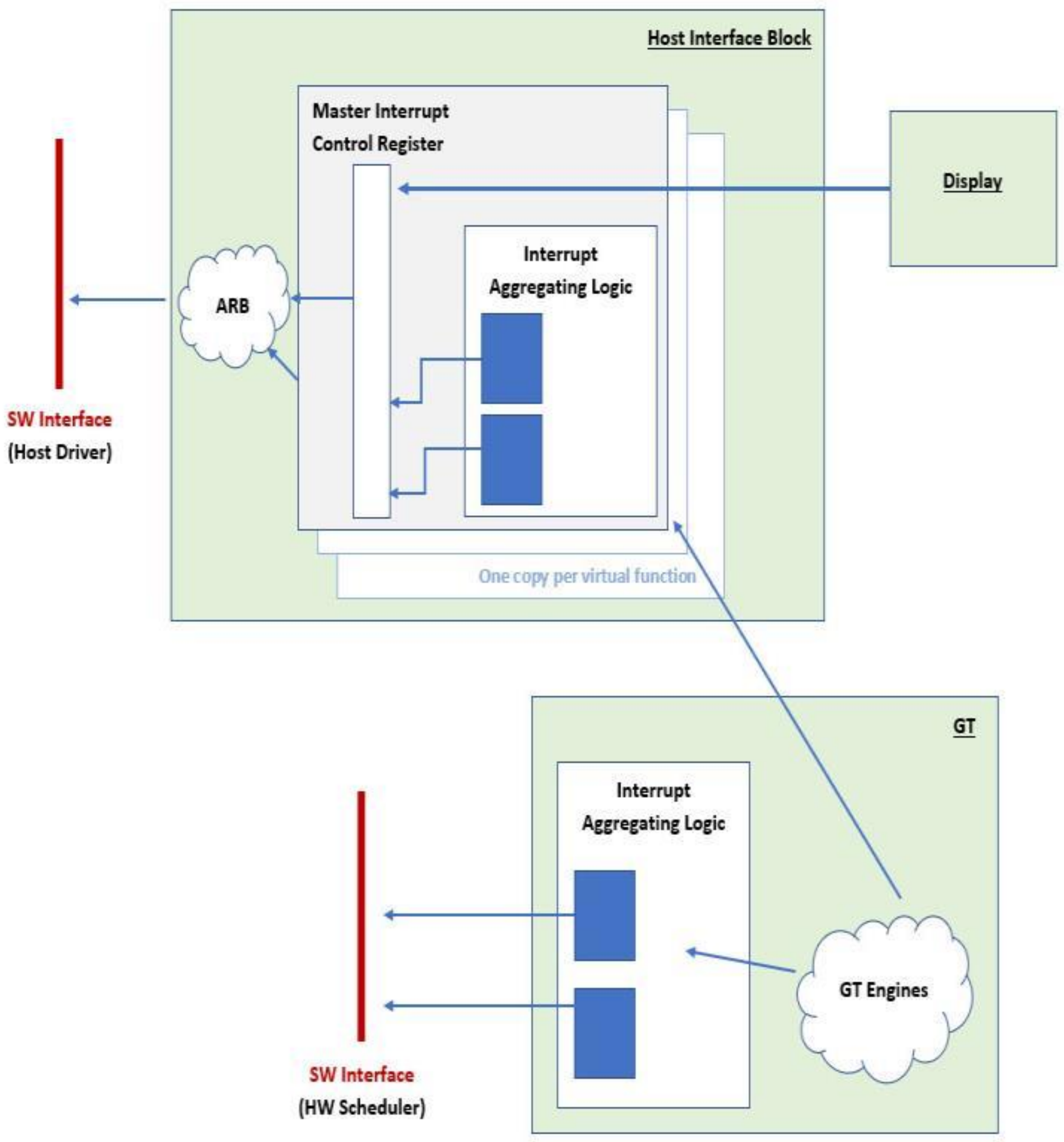
The fact that doorbell block is capable of generating accesses to IDI, it will have to be involved in LOCK process. The impact has to be looked in two different operational modes:

Interrupts 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

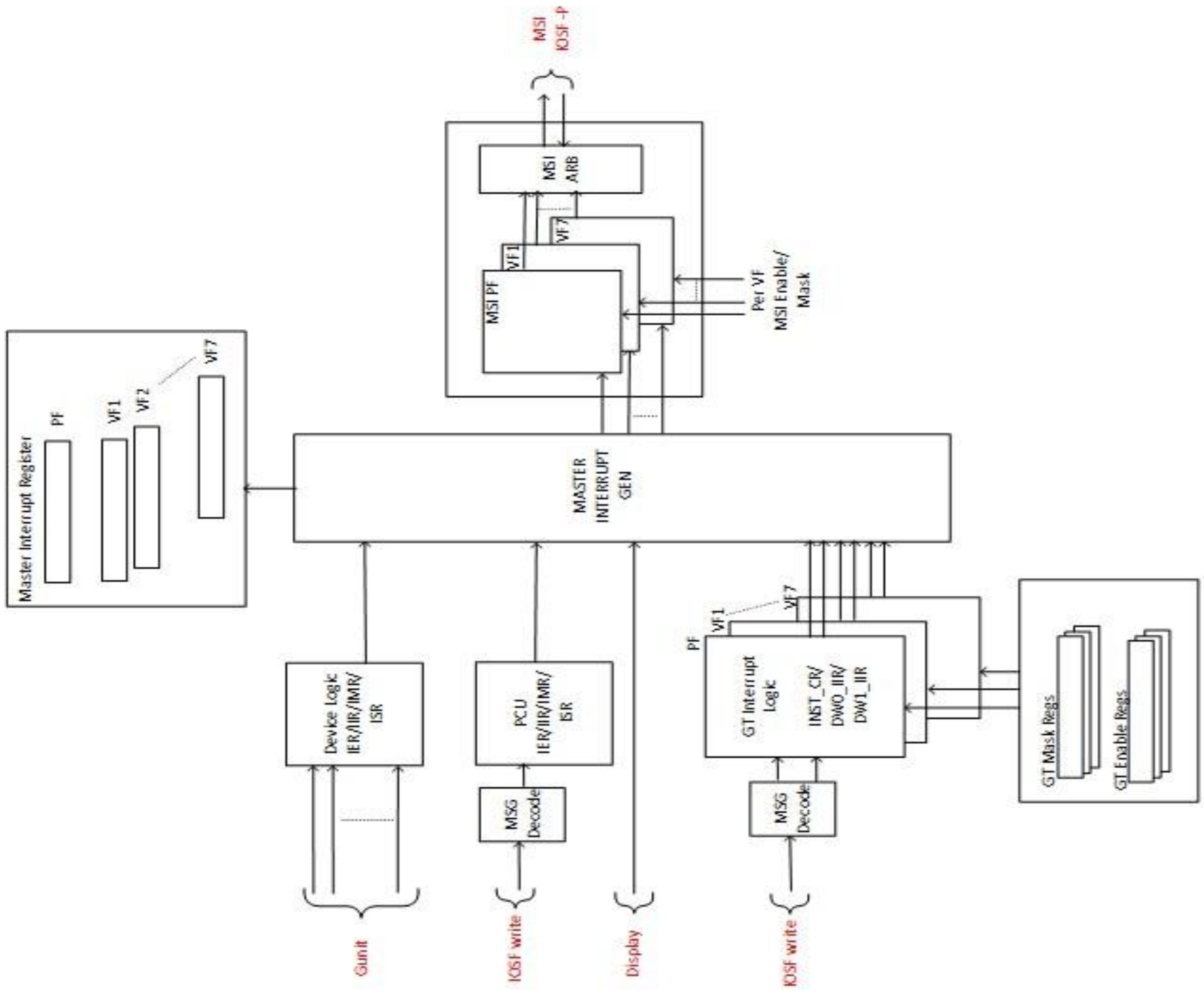
[19]	Reserved
[18:16]	Engine Class ID
[15:0]	Interrupt data

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.

Host SW Interface



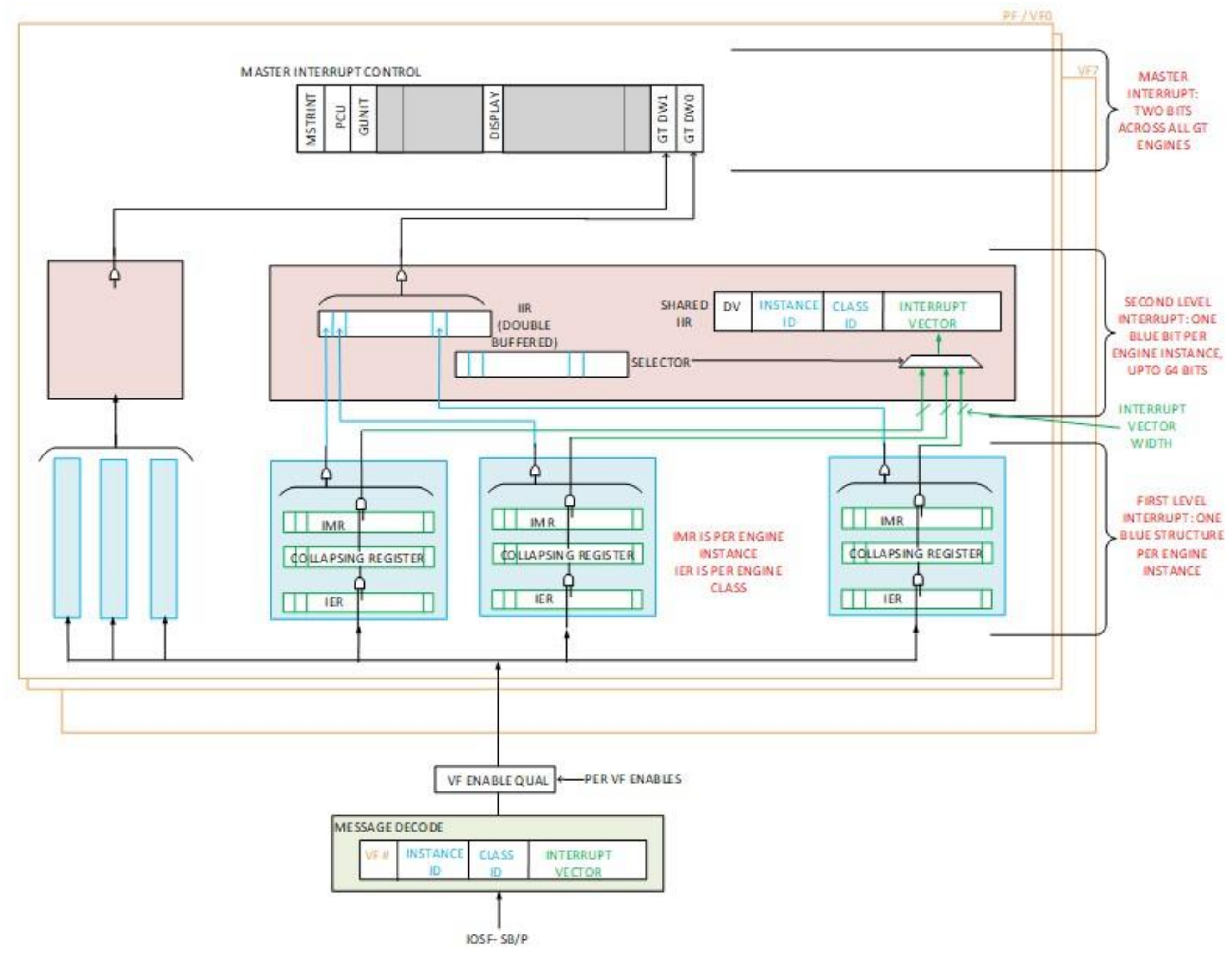
Interrupts to Host are delivered via a Primary Interrupt Control Register. Graphics interrupts use 2 bits in the Primary Interrupt Control Register. In addition, interrupt events from Display are also represented in the Primary Interrupt Control Register. Multiple copies of Primary Interrupt Control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