Intel® Open Source HD Graphics

Programmer's Reference Manual

For the 2016 Intel Atom™ Processors, Celeron™ Processors, and Pentium™ Processors based on the "Apollo Lake" Platform (Broxton Graphics)

Volume 6: 3D-Media-GPGPU

May 2017, Revision 1.0

Creative Commons License

You are free to Share - to copy, distribute, display, and perform the work under the following conditions:

- **Attribution.** You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
- **No Derivative Works.** You may not alter, transform, or build upon this work.

Notices and Disclaimers

Table of Contents

Shared Functions

3D Sampler

State

The 3D sampler uses both surface state objects (RENDER_SURFACE_STATE) as well as sampler state objects (SAMPLER_STATE). These objects are cached locally in the sampler state cache for improved performance as it is assumed that many sampler messages will utilize the same surface and sampler states.

Surface State Fetch

Surface state is fetched from system memory using a Binding Table Pointer (**BTP**). The **BTP** is a 16-bit value provided by the command stream (not directly by the shader) which determines the binding-table to be used. An 8-bit Binding Table Index (**BTI**) is then provided by the shader via the message descriptor, which indicates the offset into the Binding Table. The BTP and BTI are relative to the **Surface State Base Address** and the binding table itself resides in system memory. The contents of the Binding Table is a list of pointers to surface state objects. The pointer from the Binding Table is also relative to the **Sampler State Base Address,** and points directly to a 256-bit **RENDER_SURFACE_STATE** object which sampler will fetch and store in its internal state cache.

Sampler State Fetch

SAMPLER_STATE objects are fetched independently of surface state and cached locally in the 3D sampler independently (there may one or more SAMPLER_STATE objects associates with one or more RENDER_SURFACE_STATE objects). The sampler state is fetched using the **Sampler State Pointer (SSP)** which is provided either in the message header or directly from the command stream (message headers are not required). The **SSP** is an offset relative to the **Dynamic_State_Base_Address** and selects a table of 16 sampler states. The 4-bit **Sampler Index** (**SI**) in the message descriptor is used to select the specific **SAMPLER_STATE** object to be fetched from system memory and cached locally in the 3D sampler.

Bindless Sampler State

The sampler supports a "Bindless" sampler model. Bindless in this case does not actually refer to the lack of a Binding table since the legacy Sampler State model also did not have a Binding Table. However, the mechanism is similar to bindless surfaces in that the pointer provided directly selects a SAMPLER_STATE object. The sampler uses the same **Sampler State Pointer (SSP)**, but it is relative to the **Bindless_Surface_State_Base_Address** rather than the Dynamic_State_Base_Addr. Bindless Samplers can only be used in conjunction with Bindless Surfaces. The Sampler Index (SI) in the message descriptor is not used, and can be set to 0. The SAMPLER_STATE object is cached locally in the 3D sampler.

State Caching

As mentioned above, the 3D Sampler allows for automatic caching of **RENDER_SURFACE_STATE** objects and **SAMPLER_STATE** objects to provide higher performance. Coherency with system memory in the state cache, like the texture cache is handled partially by software. It is expected that the command stream or shader will issue Cache Flush operation or Cache_Flush sampler message to ensure that the L1 cache remains coherent with system memory.

Programming Note	
Context:	State Cache Coherency
Whenever the value of the Dynamic_State_Base_Addr, Surface_State_Base_Addr are altered, the L1 state cache must be invalidated to ensure the new surface or sampler state is fetched from system memory. Whenever the RENDER_SURFACE_STATE object in memory pointed to by the **Binding Table Pointer (BTP)** and **Binding Table Index (BTI)** is modified or SAMPLER_STATE object pointed to by the **Sampler State Pointer (SSP)** and **Sampler Index (SI)** is modified, the L1 state cache must be invalidated to ensure the new surface or sampler state is fetched from system memory.	

Messages

Message Header

The message header for the sampling engine is the same regardless of the message type. The messasge header is optional. If the header is not present, the behavior is as if the message was sent with all fields in the header set to zero and the write channel masks are all enabled and offsets are zero. However, if the header is not included in the message, the Sampler State Pointer will be obtained from the command stream input for the given thread.

When Response length is 0 for sample_8x8 message then the data from sampler is directly written out to memory using media write message.

DWord	Bits	Description
M0.5	31:5	Reserved
M0.5	4:0	Output format. Only for Sample_8x8 message with direct HDC write: 0 YCRCB_NORMAL 1 YCRCB_SWAPUVY 2 YCRCB_SWAPUV 3 YCRCB_SWAPY 4 PLANAR_420_8 (NV12 only) 5 Y8_UNORM 6 Y16_SNORM 7-16 Reserved

DWord	Bits	Description
		17 Y32_UNORM 18 Reserved
M0.4	31:16	Destination Y Address (u16)
M0.4	15:0	Destination X Address in bytes (u16) X Address should always be DWord-aligned.
M0.3	31:4	**Sampler State Pointer:** Specifies the 16-byte aligned pointer to the sampler state table. This field is ignored for *ld* and *resinfo* message types. This pointer is relative to the **Dynamic State Base Address** or **Bindless Surface State Base Address** depending on the setting of Bindless Surface Base Address Select bit in the GT_Mode Register. Format = StateOffset[31:4] The Sampler State Pointer does not have to be defined by the Message Header (many messages do not require a message header). The Sampler State Pointer may be delivered from the Command Streamer without the need for a Message Header.
M0.3	3:0	Ignored
M0.2	31:24	**Render Target or Destination Binding Table Index** Specifies the index into the binding table for the render target or HDC for messages with response length of zero (the binding table index for the sampler surface is in the message descriptor). Format = U8 Range = [0,255]
M0.2	23	**Pixel Null Mask Enable** Specifies whether the writeback message includes an extra phase indicating the pixel null mask. Refer to the *Writeback Message* section for details on format. This field must be disabled for sample+killpix and all SIMD32/64 messages. Format = Enable Ignored for Sample_8x8 message
M0.2	22	**SIMD Mode Extension** If SIMD Mode in the message descriptor is set to SIMD8D/SIMD4x2, this field specifies which mode is used. For other SIMD Modes, this field is ignored. 0: SIMD8D 1: SIMD4x2
M0.2	21	Reserved
M0.2	20	Reserved

<table>
<tr><th>DWord</th><th>Bits</th><th>Description</th></tr>
<tr><td>M0.2</td><td>19:18</td><td>SIMD32/64 Output Format Control
Specifies the output format of SIMD32/64 messages (sample_unorm* and sample_8x8). Ignored for other message types. Refer to the writeback message formats for details on how this field affects returned data.
This field is ignored for sample_8x8 messages if the Function is not AVS and MinMaxFilter. For MinMaxFilter only 16 bit Full and 8 bit Full modes are supported.
This field is ignored and not used for HDC write message.
0: 16 bit Full
1: 16 bit Chrominance Downsampled
2: 8 bit Full
3: 8 bit Chrominance Downsampled
This feature should be programmed to 0h because non-0 values may cause data corruption in returned values.</td></tr>
<tr><td>M0.2</td><td>17:16</td><td>Gather4 Source Channel Select: Selects the source channel to be sampled in the gather4* messages. Ignored for other message types.
0: Red channel
1: Green channel
2: Blue channel
3: Alpha channel
For gather4*_c messages, this field must be set to 0 (Red channel).</td></tr>
<tr><td>M0.2</td><td>15</td><td>Alpha Write Channel Mask: Enables the alpha channel to be written back to the originating thread.
0: Alpha channel is written back.
1: Alpha channel is not written back.
<table><tr><th colspan="2">Programming Note</th></tr><tr><td>Context:</td><td>3D Sampler Messages</td></tr><tr><td colspan="2">A message with all four channels masked is not allowed.
This field must be set to zero for sample_8x8 in VSA mode.
For Sample_8x8 messages, Alpha/Blue/Red channels should be always masked (set to 1) and only Green channel is enabled (set to 0).
This field must be set to zero for all gather4* messages.</td></tr></table></td></tr>
<tr><td>M0.2</td><td>14</td><td>Blue Write Channel Mask: See Alpha Write Channel Mask.</td></tr>
<tr><td>M0.2</td><td>13</td><td>Green Write Channel Mask: See Alpha Write Channel Mask.</td></tr>
</table>

<table>
<tr><th>DWord</th><th>Bits</th><th>Description</th></tr>
<tr><td>M0.2</td><td>12</td><td>Red Write Channel Mask: See Alpha Write Channel Mask.</td></tr>
<tr><td>M0.2</td><td>11.8</td><td>U Offset: The u offset from the _aoffimmi modifier on the sample or ld instruction in DX10. Must be zero if the Surface Type is SURFTYPE_CUBE or SURFTYPE_BUFFER. Must be set to zero if _aoffimmi is not specified. Format is S3 2's complement.
<table><tr><th colspan="2">Programming Note</th></tr><tr><th>Context:</th><td>3D Sampler Messages</td></tr><tr><td colspan="2">This field is ignored for the sample_unorm*, sample_8x8, and deinterlace messages.
This field is ignored if the offu parameter is included in the gather4* messages.</td></tr></table></td></tr>
<tr><td>M0.2</td><td>7:4</td><td>V Offset: The v offset from the _aoffimmi modifier on the sample or ld instruction in DX10. Must be zero if the Surface Type is SURFTYPE_CUBE or SURFTYPE_BUFFER. Must be set to zero if _aoffimmi is not specified. Format is S3 2's complement.
<table><tr><th colspan="2">Programming Note</th></tr><tr><th>Context:</th><td>3DSampler Messages</td></tr><tr><td colspan="2">This field is ignored for the sample_unorm*, sample_8x8, and deinterlace messages.
This field is ignored if the offu parameter is included in the gather4* messages.</td></tr></table>
<table><tr><th colspan="2">Programming Note</th></tr><tr><th>Context:</th><td>Non-Normalized Floating-Point Coordinates</td></tr><tr><td colspan="2">Texel offsets can only be applied to messages with floating-point normalized coordinates or integer non-normalized coordinates.</td></tr></table></td></tr>
<tr><td>M0.2</td><td>3:0</td><td>R Offset: The r offset from the _aoffimmi modifier on the sample or ld instruction in DX10. Must be zero if the Surface Type is SURFTYPE_CUBE or SURFTYPE_BUFFER. Must be set to zero if _aoffimmi is not specified. Format is S3 2's complement.
<table><tr><th colspan="2">Programming Note</th></tr><tr><th>Context:</th><td>3D Sampler Messages</td></tr><tr><td colspan="2">This field is ignored for the sample_unorm*, sample_8x8, and deinterlace messages.</td></tr></table></td></tr>
<tr><td>M0.1</td><td>31:0</td><td>Reserved</td></tr>
<tr><td>M0.0</td><td>31:0</td><td>Reserved</td></tr>
</table>

Message Gateway

Message Payload

DWord	Bits	Description
M0.5	31:0	**Ignored**
M0.4	31:0	**Ignored**
M0.3	31:0	**Predicate Mask.** This field has a bit set per SIMD channel that passes the predicate. For SIMD8 and SIMD16 the rest of the bits must be 0. This field is ignored for non-predicated barriers.
M0.2	31	**Barrier ID MSB.** This field is bit[4] of the BarrierID, for full 5-bit barrier ID it should be combined with Barrier ID[3:0]. Format: U1
M0.2	30	**Ignored**
	27:24	**BarrierID.** This field indicates which one from the 16 Barrier States is updated. There is a fifth bit for the BarrierID, in bit 31. Format: U4 Note: This field location matches with that of R0 header.
	23:16	**Ignored**
	15	**Barrier Count Enable.** Allows the message to reprogram the terminating barrier count. If set, the stored value of the terminating barrier count is set to the value of Barrier Count field (below), and used for this barrier operation. If clear, the stored value of the terminating barrier count is not modified and the stored value is used for this barrier operation. **Programming Note:** This control is intended only for Hull Shader threads. Do not use this control if the barrier is linked with other barriers in other subslices (i.e. pooled EU thread groups in BXT). Format: Enable
	14:9	**Barrier Count**. If Barrier Count Enable is set, this field specifies the terminating barrier count. Otherwise this field is ignored. All threads that belong to a single barrier must deliver the same value for this field for a particular barrier iteration.
	8:0	**Ignored**
M0.1	31:0	**Ignored**
M0.0	31:4	**Ignored**

Media GPGPU Pipeline

This section discusses Programming the GPGPU Pipeline, Thread Group Tracking, Generic Media, and other related topics.

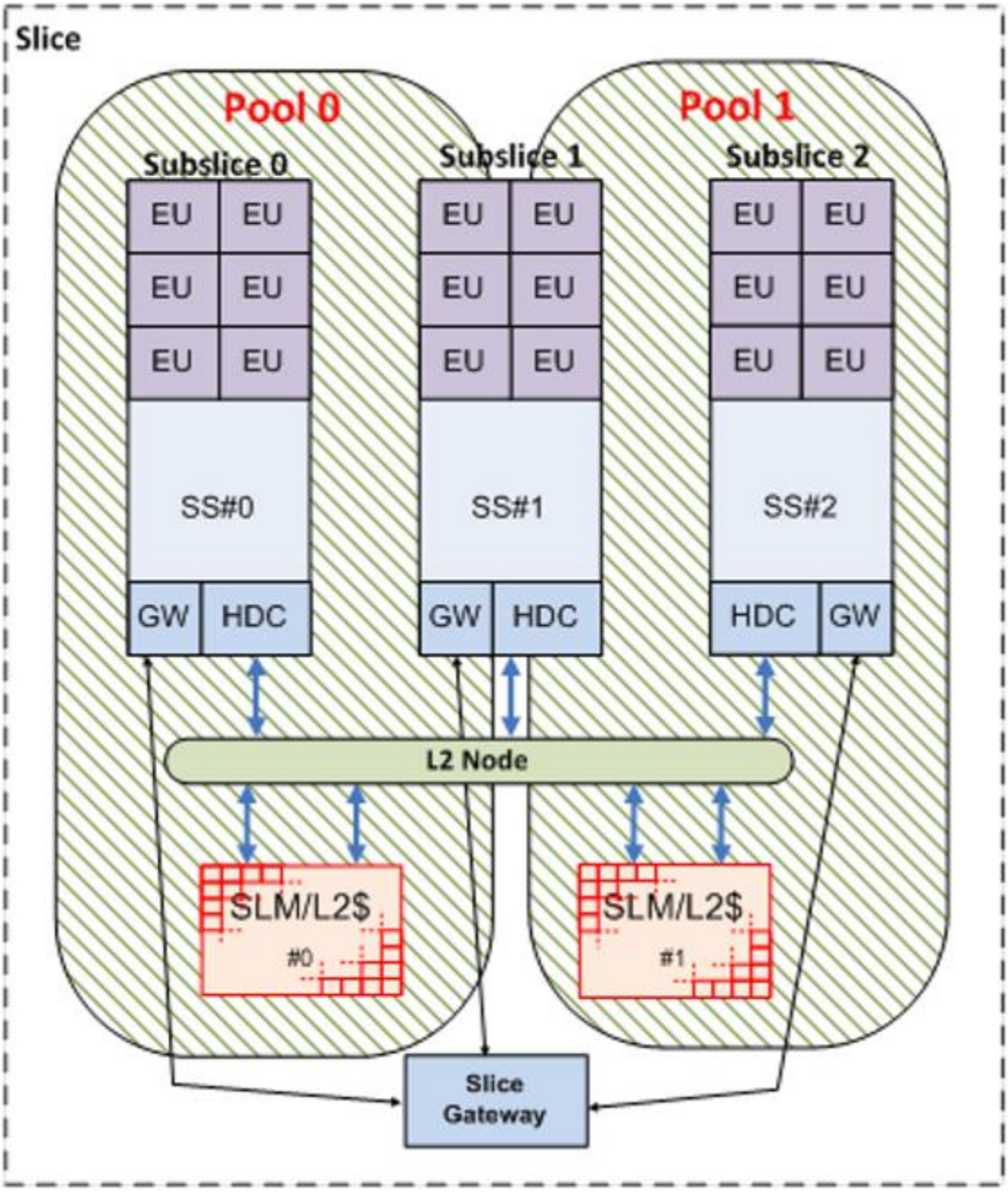
Thread Pools

Overview

Earlier project generations imposed a hardware-enforced restriction on the dispatching of workgroups to the EU array. Specifically a workgroup, broken down into threads, was dispatched in whole, entirely to a single subslice; no spanning of subslices was allowed. As long as an implementation's subslice size (EU * thread count) was sufficiently large, multiple workgroups of reasonably large sizes could be loaded onto a single subslice, filling the subslice's threads, providing good latency coverage leading to good performance. But for configurations with smaller subslice sizes, either in thread count per EU and/or EU count, and depending on workgroup size , this workgroup-per-subslice restriction could pose non-optimal thread loading and non-optimal performance,.

The BXT project improves this situation by relaxing the workgroup-to-subslice coupling by aggregating EUs from different subslices into one or more 'virtual pools' to which a workgroup can be dispatched. For example, a configuration which has 3 subslices, each with 6 EUs, may create two virtual pools with 9 and 9 EUs each (see figure below). In terms of threads, if each EU has 6 threads, then the two pools have 9*6=54 threads each, vs. the three pools of 6*6=36 threads in previous architectural generations. Depending on the workgroup size, the pooled and larger aggregated pools allow for a larger integer number of workgroups to be active in the EU array at any instance, and minimize quantization problems in thread occupancy.

The creation of pools, and mapping of EUs into them, is done through the new MEDIA_POOL_STATE command. This command should be issued as part of context initialization. The setting is maintained as part of context state and preserved across context switch operations.



JIT Impact

Note that other than possibly providing more flexibility in the simd-ness of the JIT, this feature is kernel-transparent. The kernel itself is oblivious of thread-pools.

Pool Configuration Rules

The restrictions on pool configurations are described in MEDIA_POOL_STATE command.

Legacy-Mode

Primarily for the purposes of backward compatibility of drivers, a 'LegacyMode' is defined where the new thread-pools feature is disabled. In LegacyMode the workgroups are dispatched such that they are contained to only a single subslice. Additionally the subslice is coupled to a single bank of SLM, and all barriers of that thread-group remain local to the subslice.

LegacyMode may be set through a field in the MEDIA_POOL_STATE command. By default this mode is enabled. Only a single 'enable' bit is defined for the entire GT device, thus selection of PooledMode or LegacyMode is global setting across all slices in the implementation.

The performance while in LegacyMode may be significantly less than the architectural capability of the device. For example it may be that there is insufficient SLM banks to guarantee peak performance of kernels which stress Local Memory. Or it may be that there is insufficient total thread count in the subslice to handle large thread groups at the more desirable simd-8 or simd-16 execution width, and that simd-32 is required to handle the workgroup, and/or load/store operations are required and thereby introduce instruction overhead. Thus LegacyMode should be used intelligently with the understanding of its device-specific limitations.

GPGPU Commands

This section contains various commands for GPGPU, including a number of them shared with media mode.

MEDIA_VFE_STATE with varying definitions for different generations and projects:

MEDIA_CURBE_LOAD

MEDIA_INTERFACE_DESCRIPTOR_LOAD [

Interface Descriptor Data payload as pointed to by the Interface Descriptor Data Start Address, with varying definitions for different generations and projects:

MEDIA_STATE_FLUSH

MEDIA_POOL_STATE

SubslicePool

GPGPU Context Switch

Context switch allows the switch to take place in the middle of a thread group to provide better response time.

The command sequence has been simplified – MEDIA_STATE_FLUSH and MI_ARB_CHECK are now optional between commands for preemption to occur. MEDIA_STATE_FLUSH is now only needed before MEDIA_LOAD_CURBE commands to ensure CURBE is done being read before reloading it. The watermark bit in MEDIA_STATE_FLUSH is not needed, since the check is done automatically before a thread group is started.

Preemption can occur on commands listed here:

- **MI_ARB_CHECK**
- **MEDIA_STATE_FLUSH**
- **PIPE_CONTROL**
- **MI_WAIT_FOR_EVENT**
- **MI_SEMAPHORE_WAIT**
- **GPGPU_WALKER** – Preemption can occur at any time, with thread groups partially complete; the system state is saved/restored for context save and restore
- **MI_WAIT_FOR_EVENT**
- **MI_SEAMPHORE_WAIT**

Messages to the Sampler must use headers (controlled by bit 19 of the Message Descriptor) when pre-emption is enabled.

Note that command preemption is not supported for MEDIA_OBJECT_* commands for MI_ARB_ON/OFF should be used to prevent preemption except at frame boundaries, where an MI_ARB_CHECK should be inserted.

The memory map of the context image that is saved from GPGPU pipeline includes SLM and EU State. A contiguous space is allocated to save the SLM and EU State starting from the address provided in GPGPU_CSR_BASE_ADDRESS.

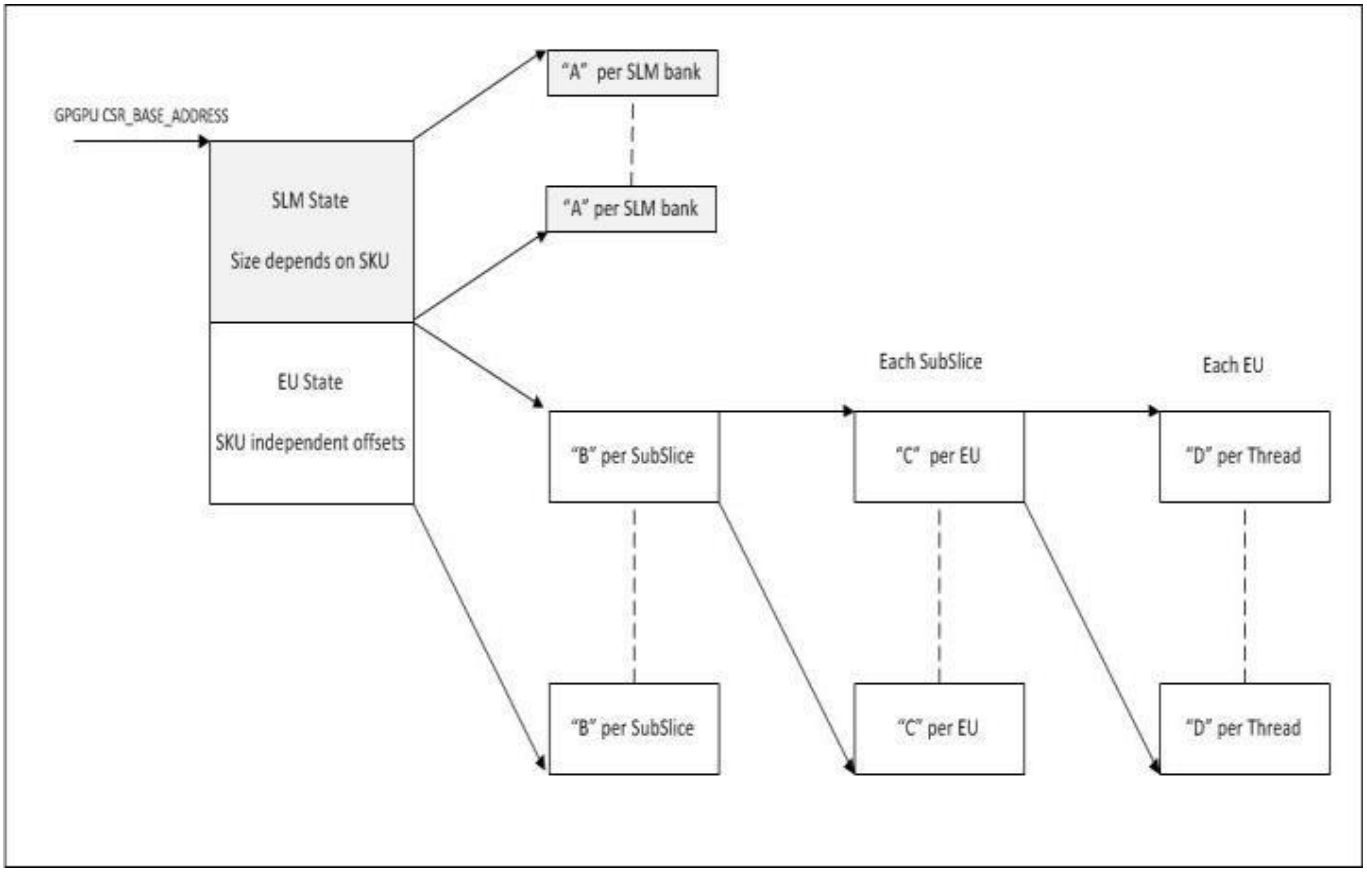
Maximum Upper Bound

Description
The maximum upper bound is set to 2MB for BXT-C.

In GPGPU context save/restore mode, hardware writes to this location and does NOT use the surface state or scratch space.

Memory Map of the GPGPU Context Image

The address offsets computed by hardware depend on the number of slices, subslices, EUs, and threads.



Base Address Calculation

Base Address = CSR_Base_Address + A* [Config.NumSlmBanks] + B*SubSliceId + C*EuId + D*ThreadId + Message_Offset

A = 0x10000 // 64K SLM per SubSlice

B = C * [Config.NumEusPerSubSlice]

C = D * [Config.NumThreadsPerEu]

D = 0x2000 // 8KB

Note: The slicenumber and EUIDs may require re-mapping such that a contiguous space is used with no gaps inbetween.

Generic Media

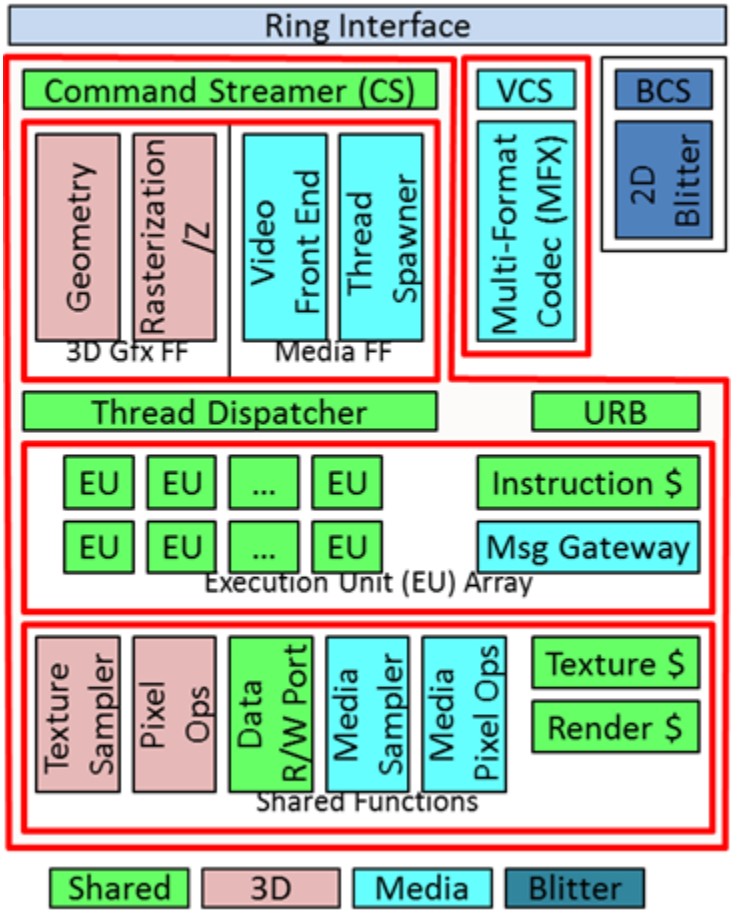
This introduction provides a brief overview of the Media product features. It includes Media functions, feature benefits, and how the features fit into graphics products as part of a whole solution.

Media product features include:

- Multi-format codec engine
- Video front end
- Media fixed functions
- Video encoding
- Video decoding
- Sampling

These product features support specific applications, such as interactive gaming, videogames, social media, virtual reality, and augmented reality.

The following block diagram shows the Main Render Engine, unified for 3D graphics and Media.



- **Fixed Function (FF) pipelines:** Provide thread generation and control.

- **3D graphics or Media FF:** Controls EU array at a given time. The EU (Execution Unit) array is shared between 3D and Media; ISA is optimized for both.
- **Shared functions:** Are accelerators to run filtered load, scatter, gather, and filter/blended store operations.
- **MFX:** Is a parallel codec engine that runs in a separate context.

Media State and Primitive Commands

This section contains various commands for media, all with the RenderCS source.

MEDIA_CURBE_LOAD

MEDIA_INTERFACE_DESCRIPTOR_LOAD

Interface Descriptor Data payload as pointed to by the Interface Descriptor Data Start Address:

The MEDIA_OBJECT command is the basic media primitive command for the media pipeline. It supports loading of inline data as well as indirect data. At least one form of payload (either inline, indirect, or CURBE) must be sent with the MEDIA_OBJECT.

MEDIA_OBJECT

MEDIA_OBJECT_PRT

The MEDIA_OBJECT_WALKER command uses the hardware walker in VFE for generating threads associated with a rectangular shaped object. It only supports loading of inline data or CURBE but not indirect data. At least one form of payload must be sent. Control of scoreboards (up to 8) is implicit based on the (X, Y) address of the generated thread and the scoreboard control state.

The command can be used only in Generic modes.

When **Use Scoreboard** field is set, the (X, Y) address and the Color field of the generated thread are used in the hardware scoreboard and the thread dependencies are set by states from the MEDIA_VFE_STATE command.

One or more threads may be generated by this command. This command does not support indirect object load. When inline data is present, it is repeated for all threads it generates. Unlike CURBE, which requires pipeline flush for change, continued change of this kind of 'global' (in the sense of shared by multiple threads from this command) data is supported when MEDIA_OBJECT_WALKER commands are issued without a pipeline flush in between.

MEDIA_POOL_STATE

SubslicePool

Media State and Primitive Command Workarounds

Media State and Primitive Commands have some subtle programming restrictions and workarounds, as listed below.

Programming Note
When a media walker/media object group id with either a local or global barrier is used, then a stalling PIPE_CONTROL is required before the next MEDIA_VFE_STATE (instead of the usual MEDIA_STATE_FLUSH).
Global barriers should not be used at the same time as linked barriers. A PIPE_CONTROL with CS Stall should be placed between work that uses linked barriers and work that uses global barriers. Global and local barriers can be used together only if the Thread Dispatch Selection Policy in MEDIA_VFE_STATE is set to Legacy.

L3 Cache and URB

This is the volume for BXT L3/URB/SLM. Much of the content is identical to Gen8, with important changes to enhance performance.

Overview

The overall organization of the L3 and theory of operation have not changed. It retains all the performance enhancements from previous generations with some additional improvements.

To provide the bandwidth needed L3 is still organized into independent banks which can be accessed concurrently. Clocking remains 2X in the arrays to balance bandwidth with incoming requests. The L3$ and URB data for all L3 banks are shared as a contiguous memory space for all products regardless of how many banks or slices.

- Each Bank 192KB in size.
- Each logical bank consists of:
    - Data Array
    - Tag Array
    - LRU Array (implements a Pseudo Least Recently Used algorithm)
    - State Array
    - SuperQ Buffer
    - Atomic Processing Units
- The rest of the support logic around L3 consists of:
    - SuperQ (main scheduler).
    - Ingress/Egress queues to L3/SQ (L3 arbiter).
    - CAM structures to maintain coherency.
    - Crossbars for data routing.
- Use of 2x/1x clocking.
- L3 can operate concurrently in GFX and IA coherent domain.
- A portion of L3 can be allocated as highly banked memory and/or unified buffer (URB).

L3 Bank Configuration

Each L3 bank is identical as described below. In products where there is only a single L3 bank supported, the bank will be different to support a larger data array. The Multi-bank describes the L3 Bank configuration for all other products.

Multi-Bank: 192KB data-array and 96 logical ways:

- Up to 64 ways, up to 128KB, tagged for L3$, remaining is treated as memory.
- 64KB or 96KB allocated for URB (does not use tag).
- Shared local memory (SLM) capable (64KB)
    - Uses Highly-banked memory to allow up to 16 32-bit accesses in parallel within a 64KB space per sub-slice.
    - Highly-banked memory formed from 16x4KB arrays

Single-Bank: 320KB data-array and 160 logical ways:

- Up to 96 ways, up to 192KB, tagged for L3$, remaining is treated as memory.
- 64KB or 128KB allocated for URB (does not use tag).
- Shared local memory (SLM) capable (64KB)
    - Uses Highly-banked memory to allow up to 16 32-bit accesses in parallel within a 64KB space per sub-slice.
    - Highly-banked memory formed from 16x4KB arrays.

All Bank Implementations:

- 64B Cacheline.
- Interface 64B to SQDB for the fill/write path, 64B Read/Evict path to SQDB. Additional 64B read and 64B write capability for SLM.
- Data protection via ECC.
- TAG/LRU/STATE (using gen-ram via RLS flows):
    - 39/48-bit addressing support in TAG.
    - 6-bit state (2-bits of MESI, 2-bits of Surface type, 1 bit Phys/Virtual, 1-bit Global/Local).
    - Intel pseudo-LRU implementation for selecting the line to be replaced or 1b LRU (added for Gen9).

Bandwidth and Throughput Capability

L3 Blocks Overview

L3 is formed via some number of logical banks that are identical to each other. The major blocks in each logical bank are:

- L3 Cache Arrays & Controller

- Super Q and related data buffer
- Ingress queues and related CAMs with arbitration
- Atomics Block/SLM pipeline & crossbar for data routing

Our vision is to build a compute scalable cache where with each additional compute both the size and bandwidth are scaled while maintaining the functional single cache concept. Each added bank becomes an additional cache rather than an independent content. The concept is to be able to keep a single copy of a line and service all requesters via distributing their accesses over many physical caches.

Size of L3 Bank and Allocations

Multi-Bank Allocation Options

The table below shows the size of the L3 bank and the ranges of allocation for L3$, URB, and SLM. Below is a table of all validated configurations supported.

No Shared Local Memory Mode (KBytes)						
			L3			
Config	SLM	URB	Rest	DC	RO(I/S/C/T)	Sum
0	0	96	96	0	0	192
1	0	96	0	32	64	192
2	0	64	0	128	0	192
3	0	64	0	0	128	192
4	0	64	128	0	0	192
Shared Local Memory Mode (KBytes)						
			L3			
Config	SLM	URB	Rest	DC	RO(I/S/C/T)	Sum
1	64	32	96	0	0	192
2	64	32	0	32	64	192
3	64	32	0	64	32	192
4	64	16	112	0	0	192

Essentially, the L3 block can either support SLM or not. When SLM is supported, each slice provides 64KB of its 192KB data array for SLM. When SLM is not supported, the 64KB is allocated to URB or split equally between URB and L3$.

The number of L3 Banks will vary for different products and SKUs. The number of banks supported for each product is defined in the Configurations section of the BSPEC. The total amount of L3$, URB, and SLM supported by a product can be calculated by multiplying the number of banks by the values in the above tables.

Single-Bank Allocation Options

The table below shows the size of the L3 bank and the ranges of allocation for L3$, URB, and SLM in the case of a product with a single bank (e.g. BXT:*:C).

No Shared Local Memory Mode (KBytes)						
			L3			
Config	SLM	URB	Rest	DC	RO(I/S/C/T)	Sum
0	0	128	192	0	0	320
1	0	128	0	32	160	320
2	0	128	0	128	64	320
Shared Local Memory Mode (KBytes)						
			L3			
Config	SLM	URB	Rest	DC	RO(I/S/C/T)	Sum
1	64	64	192	0	0	320
2	64	64	0	160	32	320
3	64	64	0	64	128	320

The table above describes the allocation of the data array for a single-bank BXT:*:C.

Shared Local Memory

Shared local memory (SLM, also known as highly-banked memory) is a portion of L3 which will be dedicated to EUs as a local memory when enabled.

SLM Behavior and Software Usage

For Gen9LP (BXT derivatives), the SLM in an L3 is not dedicated for use by a single sub-slice. Because there may be more sub-slices than there are L3 banks, the SLM is shared across all EUs in all sub-slices within a slice. Each SLM bank is shared by some sub-set of the EUs as determined in the HDC by software. See the volume on HDC for more details on how SLM is allocated to different EUs/threads.

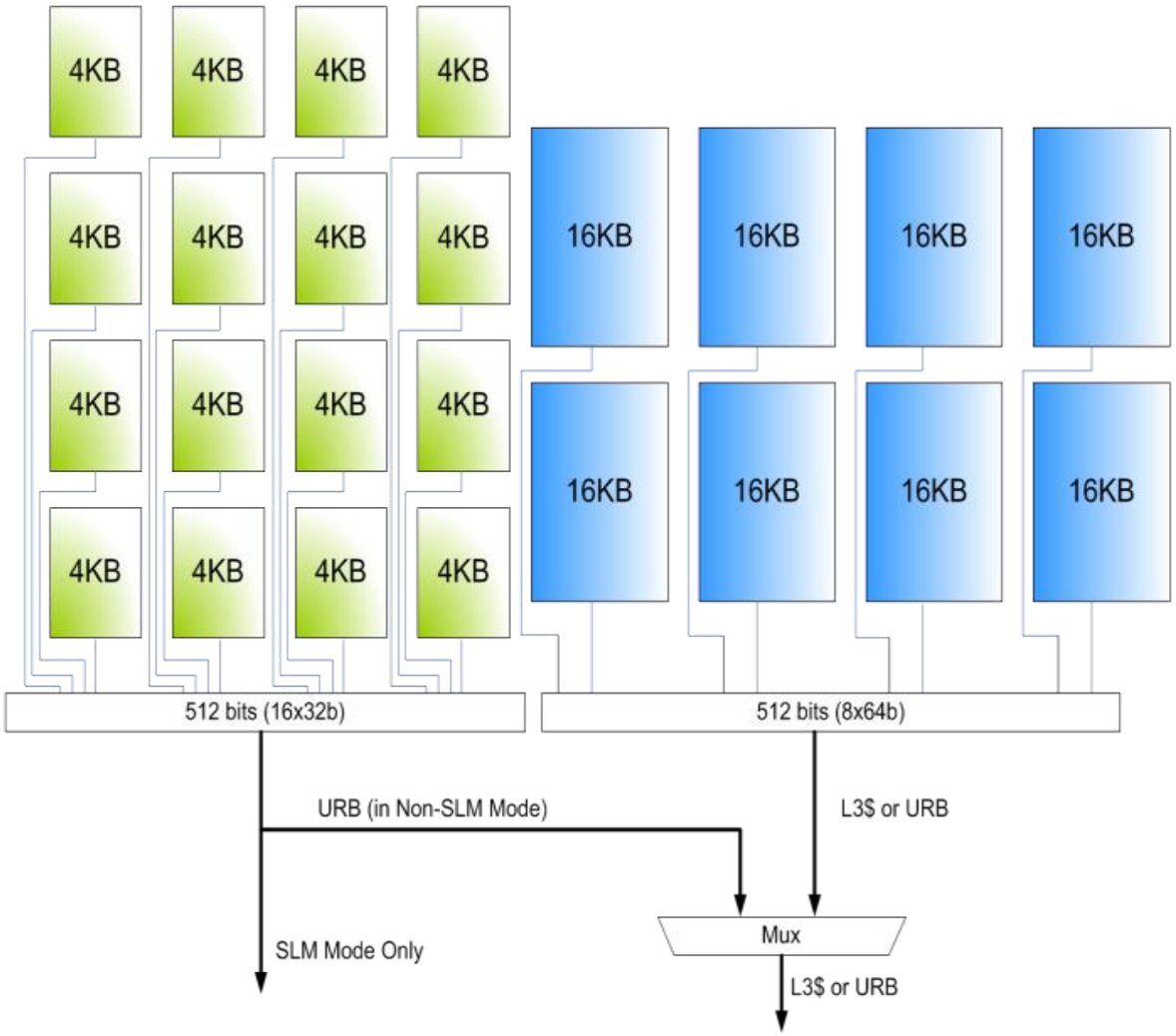
SLM is enabled statically by the application. When enabled, it uses a fixed 64K block in each L3 Bank. When SLM is enabled, URB will not use this 64K block of memory. URB accesses will automatically be moved to a different location in the L3 bank.

SLM is implemented to allow for highly parallel applications. 16 DWord-accesses (all read or all write) can be done simultaneously to different locations in the 64K space. Atomic operations as described in the Atomic Operation section of this volume can be performed on the output of the SLM and written back in one operation.

SLM Bank Hardware Implementation

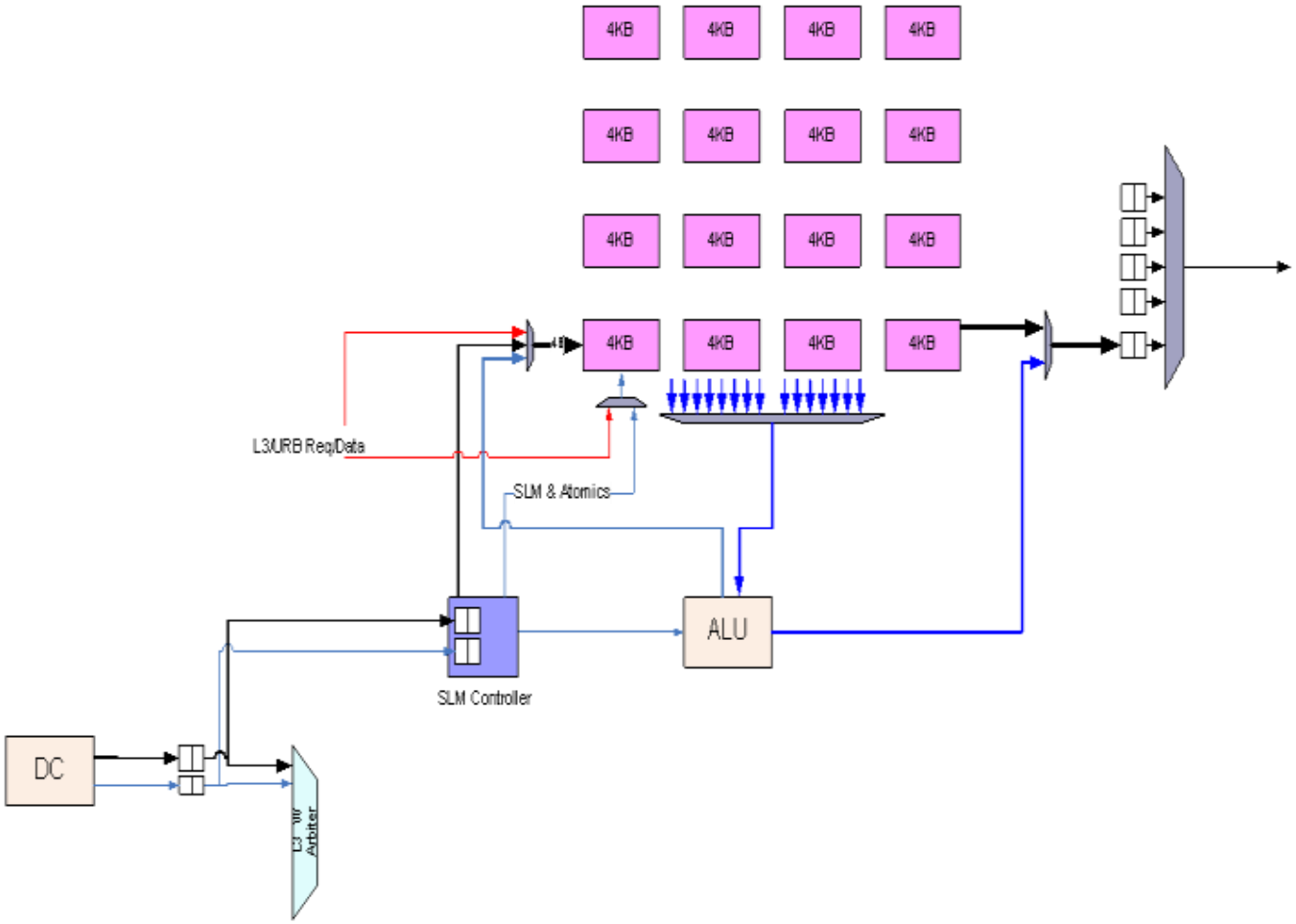
The accesses are only possible through data cluster with the destination flag set as SLM. To support a highly banked design, each of the L3 banks are structured to have 16x4KB portion which could be accessed independently per clock. This part of the L3 can support 16 DW size accesses (per SLM) in a

given clock cycle. Each bank's SLM is dedicated to a single sub-slice. The diagram below shows the organization of data for a 192KB L3 Bank.



These 16 banks can either be used as URB or used as shared local memory with parallel accesses to all banks. The choice of enabling SLM mode is done through MMIO programming

Bits	Access	Default Value	Description
0	RW/C	0	Enable Shared Local Memory: When set, it enables the use of 2 banks of L3 as shared local memory which allows 64KB of L3 to be banked as 16x4KB and allows independent accesses to all banks within the same clock cycle. **Note:** This mode can only be enabled once L3 content is completely flushed.


4KB
L3/URB Req/Data
SLM & Atomics
ALU
SLM Controller
DC
L3 - W Arbiter

Shared Local Memory

Shared local memory (SLM, also known as highly-banked memory) is a portion of L3 which will be dedicated to EUs as a local memory when enabled.

SLM Behavior and Software Usage

For BXT:*:C, the SLM in an L3 is not dedicated for use by a single sub-slice. Because there may be more sub-slices than there are L3 banks, the SLM is shared across all EUs in all sub-slices within a slice. Each SLM bank is shared by some sub-set of the EUs as determined in the HDC by software. See the volume on HDC for more details on how SLM is allocated to different EUs/threads.

SLM is enabled statically by the application. When enabled, it uses a fixed 64K block in each L3 Bank. When SLM is enabled, URB will not use this 64K block of memory. URB accesses will automatically be moved to a different location in the L3 bank.

SLM is implemented to allow for highly parallel applications. 16 DWord-accesses (all read or all write) can be done simultaneously to different locations in the 64K space. Atomic operations as described in the Atomic Operation section of this volume can be performed on the output of the SLM and written back in one operation.

SLM Bank Hardware Implementation

The accesses are only possible through data cluster with the destination flag set as SLM. To support a highly banked design, each of the L3 banks are structured to have 16x4KB portion which could be accessed independently per clock. This part of the L3 can support 16 DW size accesses (per SLM) in a given clock cycle. Each bank's SLM is dedicated to a single sub-slice.

These 16 banks can either be used as URB or used as shared local memory with parallel accesses to all banks. The choice of enabling SLM mode is done through MMIO programming

Bits	Access	Default Value	Description
0	RW/C	0	Enable Shared Local Memory: When set, it enables the use of 2 banks of L3 as shared local memory which allows 64KB of L3 to be banked as 16x4KB and allows independent accesses to all banks within the same clock cycle. **Note:** This mode can only be enabled once L3 content is completely flushed.

EU Overview

The GEN instruction set is a general-purpose data-parallel instruction set optimized for graphics and media computations. Support for 3D graphics API (Application Programming Interface) Shader instructions is mostly native, meaning that GEN efficiently executes Shader programs. Depending on Shader program operation modes (for example, a Vertex Shader may be executed on a base of a vertex pair, while a Pixel Shader may be executed on a base of a 16-pixel group), translation from 3D graphics API Shader instruction streams into GEN native instructions may be required. In addition, there are many specific capabilities that accelerate media applications. The following feature list summarizes the GEN instruction set architecture:

- SIMD (single instruction multiple data) instructions. The maximum number of data elements per instruction depends on the data type.
- SIMD parallel arithmetic, vector arithmetic, logical, and SIMD control/branch instructions.
- Instruction level variable-width SIMD execution.
- Conditional SIMD execution via destination mask, predication, and execution mask.
- Instruction compaction.
- An instruction may executed in multiple cycles over a SIMD execution pipeline.
- Most GEN instructions have three operands. Some instructions have additional implied source or destination operands. Some instructions have explicit dual destinations.
- Region-based register addressing.
- Direct or indirect (indexed) register addressing.
- Scalar or vector immediate source operand.
- Higher precision accumulator registers are architecturally visible.
- Self-modifying code is not allowed (instruction streams, including instruction caches, are read-only).

Accumulator Registers

Accumulator Registers Summary

Attribute	Value
ARF Register Type Encoding (RegNum[7:4]):	0010b
Number of Registers:	10
Default Value:	None
Normal Access:	RW

Accumulator registers can be accessed either as explicit or implied source and/or destination registers. To a programmer, each accumulator register may contain either 8 DWords or 16 Words of data elements. However, as described in the Implementation Precision Restriction notes below, each data element may have higher precision with added guard bits not indicated by the numeric data type.

Accumulator capabilities vary by data type, not just data size, as described in the Accumulator Channel Precision table below. For example, D and F are both 32-bit data types, but differ in accumulator support.

See the Accumulator Restrictions section for information about additional general accumulator restrictions and also accumulator restrictions for specific instructions.

Accumulator Registers
There are 10 accumulator registers. The accumulator registers are of two types.

Register and Subregister Numbers for Accumulator Registers

RegNum[3:0]	SubRegNum[4]	SubRegNum[3:0]
0000b-1001b = acc0-acc9 All other encodings are reserved.	0 : Lower half 1 : Upper half	Reserved: MBZ

- Accumulators are updated implicitly only if the AccWrCtrl bit is set in the instruction. The Accumulator Disable bit in control register cr0.0 allows software to disable the use of AccWrCtrl for implicit accumulator updates. The write enable in word granularity for the instruction is used to update the accumulator. Data in disabled channels is not updated.
- When an accumulator register is an implicit source or destination operand, hardware always uses acc0 by default and also uses acc1 if the execution size exceeds the number of elements in acc0. When implicit access to acc1 is required, QtrCtrl is used. Note that QtrCtrl can be used only if acc1 is accessible for a given data type. If acc1 is not accessible for a given data type, QtrCtrl defaults to acc0.

Description
acc0 and acc1 are supported for half-precision (HF, Half Float) and single-precision (F, Float). Use QtrCtrl of Q2 or Q4 to access acc1 for Float. use QtrCtrl of H2 to access acc1 for Half Float.

Examples:

```
// Updates acc0 and acc1 because it is SIMD16:
add (16) r10:f r11:f r12:f {AccWrEn}
// Updates acc0 because it is SIMD8:
add (8) r10:f r11:f r12:f {AccWrEn}
// Updates acc1. Implicit access to acc1 using QtrCtrl:
add (8) r10:f r11:f r12:f {AccWrEn, Q2}
// Updates acc1 for Half Floats using QtrCtrl:
add (16) r10:hf r11:hf r12:hf {AccWrEn, H2}
```

- It is illegal to specify different accumulator registers for source and destination operands in an instruction (e.g. "*add (8) acc1:f acc0:f*"). The result of such an instruction is unpredictable.

Limits on SIMD16 Float Operations
Accumulator registers may be accessed explicitly as src0 operands only.

- Swizzling is not allowed when an accumulator is used as an implicit source or an explicit source in an instruction.

- Reading accumulator content with a datatype different from the previous write will result in undeterministic values.
- Word datatype write to accumulator is not allowed when destination is odd offset strided by 2.
- For any DWord operation, including DWord multiply, accumulator can store up to 8 channels of data, with only acc0 supported.
- When an accumulator register is an explicit destination, it follows the rules of a destination register. If an accumulator is an explicit source operand, its register region must match that of the destination register with the exception(s) described below.

Exceptions
Half floats can be written to either the lower or the upper word of the accumulator. However, this is not supported for integer word operations.
acc1 must not be used with half-float and word datatypes

Implementation Precision Restriction: As there are only 64 bits per channel in DWord mode (D and UD), it is sufficient to store the multiplication result of two DWord operands as long as the post source modified sources are still within 32 bits. If any one source is type UD and is negated, the negated result becomes 33 bits. The DWord multiplication result is then 65 bits, bigger than the storage capacity of accumulators. Consequently, the results are unpredictable.

Implementation Precision Restriction: As there are only 33 bits per channel in Word mode (W and UW), it is sufficient to store the multiplication result of two Word operands with and without source modifier as the result is up to 33 bits. Integers are stored in accumulator in 2's complement form with bit 32 as the sign bit. As there is no guard bit left, the accumulator can only be sourced once before running into a risk of overflowing. When overflow occurs, only modular addition can generate a correct result. But in this case, conditional flags may be incorrect. When saturation is used, the output is unpredictable. This is also true for other operations that may result in more than 33 bits of data. For example, adding UD (FFFFFFFFh) with D (FFFFFFFFh) results in 1FFFFFFFEh. The sign bit is now at bit 34 and is lost when stored in the accumulator. When it is read out later from the accumulator, it becomes a negative number as bit 32 now becomes the sign bit.

Accumulator Channel Precision

Data Type	Accumulator Number	Number of Channels	Bits Per Channel	Description
DF	acc0	4	64	When accumulator is used for Double Float, it has the exact same precision as any GRF register.
F	acc0/acc1	8	32	When accumulator is used for Float, it has the exact same precision as any GRF register.
Q	N/A	N/A	N/A	Not supported data type.
D (UD)	acc0	8	33/64	When the internal execution data type is doubleword integer, each accumulator register contains 8 channels of (extended) doubleword

Data Type	Accumulator Number	Number of Channels	Bits Per Channel	Description
				integer values. The data are always stored in accumulator in 2's complement form with 64 bits total regardless of the source data type. This is sufficient to construct the 64-bit D or UD multiplication results using an instruction macro sequence consisting of *mul*, *mach*, and *shr* (or *mov*).
W (UW)	acc0	16	33	When the internal execution data type is word integer, each accumulator register contains 16 channels of (extended) word integer values. The data are always stored in accumulator in 2's complement form with 33 bits total. This supports single instruction multiplication of two word sources in W and/or UW format.
B (UB)	N/A	N/A	N/A	Not supported data type.

Accumulators
Accumulators acc2-acc9

These are accumulator registers defined for a special purpose. They are used to emulate IEEE-compliant fdiv and sqrt macro operations. The access is different from acc0 and acc1. Each of these accumulator registers are defined as 256-bit registers having 8 DWords. These may be accessed explicitly or implicitly.

- These registers may be accessed explicitly only by a *mov* operation, with no source modifiers, condition modifiers, or saturation. When accessed explicitly, the datatype must be D. On reads, the low 2 bits of each DWord are valid data. The other bits are undefined. On writes, the low two bits are updated and other bits are dropped.

  Example:

```
// Move 256 bits from acc2 to r10. Just low two bits of each DWord are valid:

mov (8) r10:ud acc2:ud

// Move 256 bits from r10 to acc2. Just low two bits of each DWord are updated:
mov (8) acc2:ud r10:ud
```

- These registers are accessed implicitly by three opcodes defined for the macro operations. **Note:** These macro operations are defined under the *math* opcode section. The macro descriptions also define the restrictive implicit uses of these registers.

Description
Implicit access across accumulator registers is required for each source operand for these macro instructions. These opcodes are accessed in Align16 mode only. The Channel Select bits in the instruction are used to implicitly address

Description
the different accumulators for each source. Similarly the Channel Enable bits are used to implicitly address the accumulators for destination. The noacc value is specified when no write to accumulator is required; think of it as a null.

Encoding	Accumulator Register
00000001b	acc3
00000010b	acc4
00000011b	acc5
00000100b	acc6
00000101b	acc7
00000110b	acc8
00000111b	acc9
00001000b	noacc

Register Region Restrictions

A register region is described as *packed* if its elements are adjacent in memory, with no intervening space, no overlap, and no replicated values. If there is more than one element in a row, elements must be adjacent. If there is more than one row, rows must be adjacent. When two registers are used, the registers must be adjacent and both must exist.

The following register region rules apply to the GEN implementation.

1. **General Restrictions Based on Operand Types**

   There are these general restrictions based on operand types:

   1. Where *n* is the largest element size in bytes for any source or destination operand type, *ExecSize* * *n* must be <= 64.
   2. When the Execution Data Type is wider than the destination data type, the destination must be aligned as required by the wider execution data type and specify a *HorzStride* equal to the ratio in sizes of the two data types. For example, a *mov* with a D source and B destination must use a 4-byte aligned destination and a *Dst.HorzStride* of 4.

2. **General Restrictions on Regioning Parameters**

   The mapping of data elements within the region of a source operand is in row-major order and is determined by the region description of the source operand, the destination operand, and the *ExecSize*, with these restrictions:

   1. *ExecSize* must be greater than or equal to *Width*.
   2. If *ExecSize* = *Width* and *HorzStride* ≠ 0, *VertStride* must be set to *Width* * *HorzStride*.
   3. If *ExecSize* = *Width* and *HorzStride* = 0, there is no restriction on *VertStride*.
   4. If *Width* = 1, *HorzStride* must be 0 regardless of the values of *ExecSize* and *VertStride*.
   5. If *ExecSize* = *Width* = 1, both *VertStride* and *HorzStride* must be 0.

6. If *VertStride* = *HorzStride* = 0, *Width* must be 1 regardless of the value of *ExecSize*.
7. *Dst.HorzStride* must not be 0.
8. *VertStride* must be used to cross GRF register boundaries. This rule implies that elements within a '*Width*' cannot cross GRF boundaries.

3. **Region Alignment Rules for Direct Register Addressing**
    1. In Direct Addressing mode, a source cannot span more than 2 adjacent GRF registers.
    2. A destination cannot span more than 2 adjacent GRF registers.
    3. When a source or destination spans two registers, there are restrictions that vary by project, described in the following table. If you are viewing a version of the BSpec limited to other particular projects, the table may appear with no data rows.
4. **Special Cases for Byte Operations**
    1. When the destination type is byte (UB or B) only a 'raw move' using the *mov* instruction supports a packed byte destination register region: *Dst.HorzStride* = 1 and *Dst.DstType* = (UB or B). This packed byte destination register region is not allowed for any other instructions, including a 'raw move' using the *sel*instruction, because the *sel* instruction is based on Word or DWord wide execution channels.
    2. There is a relaxed alignment rule for byte destinations. When the destination type is byte (UB or B), destination data types can be aligned to either the lowest byte or the second lowest byte of the execution channel. For example, if one of the source operands is in word mode (a signed or unsigned word integer), the execution data type will be signed word integer. In this case the destination data bytes can be either all in the even byte locations or all in the odd byte locations.

        This rule has two implications illustrated by this example:

```
// Example:
mov (8) r10.0<2>:b r11.0<8;8,1>:w
mov (8) r10.1<2>:b r11.0<8;8,1>:w

// Dst.HorzStride must be 2 in the above example so that the destination
// subregisters are aligned to the execution data type, which is :w.
// However, the offset may be .0 or .1.
// This special handling applies to byte destinations ONLY.
```

5. **Special Cases for Word Operations**

    There are some special cases for word operations for specific projects, described in the following table. If you are viewing a version of the BSpec limited to other particular projects, the table may not show and there are no special cases in this category.

    There is a relaxed alignment rule for word destinations. When the destination type is word (UW, W, HF), destination data types can be aligned to either the lowest word or the second lowest word of the execution channel. This means the destination data words can be either all in the even word locations or all in the odd word locations.

```
// Example:
add (8)  r10.0<2>:hf r11.0<8;8,1>:f r12.0<8;8,1>:hf
```

```
add (8)  r10.1<2>:hf r11.0<8;8,1>:f r12.0<8;8,1>:hf

// Note: The destination offset may be .0 or .1 although the destination subregister
// is required to be aligned to execution datatype.
```

6. **Special Requirements for Handling Double Precision Data Types**

   There are special requirements for handling double precision data types that vary by project, described in the following table. If you are viewing a version of the BSpec limited to other particular projects, the table may appear with no data rows.

Special Requirements for Handling Double Precision Data Types

Requirement

When source or destination datatype is 64b or operation is integer DWord multiply, regioning in Align1 must follow these rules:
1. Source and Destination horizontal stride must be aligned to the same qword.
Example:
`// mov (4) r10.0:df r11.0<16;8,2>:f // Source stride must be 2 since datatype is smaller.`
`// mov (4) r10.0<2>:f r11.0<4;4,1>:df // Destination stride must be 2 since datatype is smaller.`
`// mul (4) r10.0<2>:d r11.0<8;4,2>:d r12.0<8;4,2>:d // Source and Destination stride must be 2 since the execution type is Qword.`
2. Regioning must ensure Src.Vstride = Src.Width * Src.Hstride.
3. Source and Destination offset must be the same, except the case of scalar source.

When source or destination datatype is 64b or operation is integer DWord multiply, indirect addressing must not be used.
ARF registers must never be used with 64b datatype or when operation is integer DWord multiply.
When source or destination datatype is 64b or operation is integer DWord multiply, DepCtrl must not be used.

7. **Special Requirements for Handling Mixed Mode Float Operations**

   There are some special requirements for handling mixed mode float operations for specific projects, described in the following table. If you are viewing a version of the BSpec limited to other particular projects, the table may appear with no data rows.

Requirement
In Align16 mode, when half float and float data types are mixed between source operands OR between source and destination operands, the register content are assumed to be packed. In such cases the execution size reflects the number of float elements. Since a stride of 1 is assumed, source is selected in packed form and 16 bit packed data is updated on the destination operand, if the datatype is half-float.
For Align16 mixed mode, both input and output packed f16 data must be oword aligned, no oword crossing in packed f16.

Requirement

Examples:
Case (a)
mad (8) r10.0.xy:hf r11.0.xxxx:f r12.xyzw:hf r13.yyyy:hf
// The 16b of each word (r12.0, r12.1, r12.2, r12.3.. and so on) forms the source operand.
// r13.1 and r13.5 is replicated for source operand.
// The lower 16b of a Dword is updated for destination. With channel enables .xy , r10.0, r10.1, r10.4 and r10.5 are updated.
 	
Case (b)
mad (8) r10.0.xy:f r11.0.xxxx:f r12.xyzw:hf r13.yyyy:hf
// The example is similar to Case(a), except that entire DWord is updated on the destination.

In Align16 mode, replicate is supported and is coissueable. mad(8) r20.xyzw:hf r3.0.r:f r6.0.xyzw:hf r6.0.xyzw:hf {Q1}
No SIMD16 in mixed mode when destination is packed f16 for both Align1 and Align16. mad(8) r3.xyzw:hf r4.xyzw:f r6.xyzw:hf r7.xyzw:hf add(8) r20.0<1>:hf r3<8;8,1>:f r6.0<8;8,1>:hf {Q1}
No accumulator read access for Align16 mixed float.
When source is float or half float from accumulator register and destination is half float with a stride of 1, the source must register aligned. i.e., source must have offset zero.
No swizzle is allowed when an accumulator is used as an implicit source or an explicit source in an instruction. i.e. when destination is half float with an implicit accumulator source, destination stride needs to be 2. mac(8) r3<2>:hf r4.0<8;8,1>:f r6.0<8;4,2>:hf mov(8) r3<1>:f acc0.0<8;4,2>:hf
In Align16, vertical stride can never be zero for f16 add(8) r3.xyzw:hf r4.0<4>xyzw:f r6.0<0>.xyzw:hf
Math operations for mixed mode: - In Align16, only packed format is supported math(8) r3.xyzw:hf r4.0.<4>xyzw:f r6.0<0>.xyzw:hf 0x09 - In Align1, f16 inputs need to be strided math(8) r3<1>:hf r4.0<8;8,1>:f r6.0<8;4,2>:hf

Requirement
In Align1, destination stride can be smaller than execution type. When destination is stride of 1, 16 bit packed data is updated on the destination. However, output packed f16 data must be oword aligned, no oword crossing in packed f16. `add(8) r3<1>:hf r4.0<8;8,1>:f r6.0<8;4,2>:hf`

8. **Regioning Rules for Register Indirect Addressing**

   Regioning rules for register indirect addressing vary for specific projects, described in the following table. If you are viewing a version of the BSpec limited to other particular projects, the table may appear with no data rows.

Rules

1. When the execution size and destination regioning parameters require two adjacent registers, these registers are accessed using one index register ONLY.

```
// Example:
mov (16) r[a0.0]:f r10:f
// The above instruction behaves the same as the following two instructions:
mov (8) r[a0.0]:f r10:f
mov (8) r[a0.0, 8*4]:f r11:f
```

2. When the destination requires two registers and the sources are 1x1 indirect mode, the sources must be assembled from two GRF registers accessed by a single index register. The data for each destination GRF register is entirely derived from one source register. This is ensured by appropriate use of regioning parameters. The exception to this is the use of indirect scalar sources, where the same element is used across the execution size.

```
// Example:
// Case (a)
add (16) r[a0.0]:f r[a0.2]:f r[a0.4]:f
// The above instruction behaves the same as the following two instructions:
add (8) r[a0.0]:f r[a0.2]:f r[a0.4]:f
add (8) r[a0.0, 8*4]:f r[a0.2, 8*4]:f r[a0.4, 8*4]:f
// Note that the immediate for the second instruction is based on regioning.
// In this case, it is 8 DWs.

// Case (b)
add (16) r[a0.0]:ud r[a0.2]<4;8,1>:w r10<8;8,1>:ud
// The above instruction behaves the same as the following two instructions:
add (8) r[a0.0]:f r[a0.2]<4;8,1>:w r10<8;8,1>:ud
add (8) r[a0.0, 8*4]:f r[a0.2, 4*2]<4;8,1>:w r11<8;8,1>:ud
// Note that the immediate for the second instruction is based on regioning.
// VertStride of 4 with data type of word.

// Case (c):
add (16) r[a0.0]:f r[a0.2]:f r[a0.4]<0;1,0>:f
// The above instruction behaves the same as the following two instructions:
add (8) r[a0.0]:f r[a0.2]:f r[a0.4]<0;1,0>:f
add (8) r[a0.0, 8*4]:f r[a0.2, 8*4]:f r[a0.4]<0;1,0>:f
// Note that the src1 indirect address does not change.
```

3. Indirect addressing on src1 must be a 1x1 indexed region mode.

Rules

4. When a Vx1 or a VxH addressing mode is used on src0, the destination may use one or two registers.
`// Example:`
`// Case (a)`
`add (16) r[a0.0]<1>:d r[a0.0]<4,1>:ud r16.0<8;8,1>:ud`
`// The above instruction behaves the same as the following two instructions:`
`add (8) r[a0.0]<1>:d r[a0.0]<4,1>:ud r16.0<8;8,1>:ud`
`add (8) r[a0.0, 8*4]<1>:d r[a0.2]<4,1>:ud r17.0<8;8,1>:ud`
`// Since the pointer (index register) is incremented every 4 elements`
`// (width), the second instruction moves from a0.0 to a0.2.`
 	
`// Case (b)`
`add (16) r10.0<2>:uw r[a0.0, 0]<1,0>:uw r16.0<8;8,1>:uw`
`// The above instruction behaves the same as the following two instructions:`
`add (8) r10.0<2>:uw r[a0.0, 0]<1,0>:uw r16.0<8;8,1>:uw`
`add (8) r11.0<2>:uw r[a0.8, 0]<1,0>:uw r17.0<8;8,1>:uw`
`// Since the pointer (index register) is incremented every 1 element`
`// (width), the second instruction moves from a0.0 to a0.8.`
 	
5. Indirect addressing on the destination must be a 1x1 indexed region mode.

Execution size of 32 is NOT supported in Vx1 or VxH modes.

1. **Special Restrictions**

   There are some special restrictions on register region access for specific projects, described in the following table. If you are viewing a version of the BSpec limited to other particular projects, the table may appear with no data rows.

Restriction
All flow control (branching) instructions must use the Align1 access mode.
When using Align16 mode for conversion of data elements of different sizes, both source and destination must be one register each.

In Align16 mode, each destination register gets all data from one source register. This means, the data for one destination register is never scattered across two source registers.
 	
`// Example:`
`// Allowed – all sources are contained within one register.`
`mul (8) r10.0:f r11.0:f r12.4<0>:f`
 	
`// NOT Allowed – src1 (r14) is scattered across two registers.`
`mad (8) r10.0:f r12.0<0>:f r14.4:f r16.0:f`

Restriction

Conversion between Integer and HF (Half Float) must be DWord-aligned and strided by a DWord on the destination.
 	
`// Example:`
`add (8) r10.0<2>:hf r11.0<8;8,1>:w r12.0<8;8,1>:w`
`// Destination stride must be 2.`
`mov (8) r10.0<2>:w r11.0<8;8,1>:hf`
`// Destination stride must be 2.`

The src, dst overlapping behavior with the second half src and the first half destination to the same register must not be used with any compressed instruction.
In mid-thread pre-emptible contexts, if a compressed instruction uses vx1 or vxh addressing modes, then the second half of the compressed instruction must use more than one index register to access source0.