Intel® Open Source HD Graphics and Intel Iris™ Graphics

Programmer's Reference Manual

For the 2014-2015 Intel Core™ Processors, Celeron™ Processors and Pentium™ Processors based on the "Broadwell" Platform

Volume 5: Memory Views

May 2015, Revision 1.0

Table of Contents

Introduction

For BDW, the hardware supports three engines:

- The Render command streamer interfaces to 3D/IE and display streams.
- The Media command streamer interfaces to the fixed function media.
- The Blitter command streamer interfaces to the blit commands.

Software interfaces of all three engines are very similar and should only differ on engine-specific functionality.

Memory Views Glossary

Term	Definition
BDW	Broadwell CPU/GFX platform. 8th generation processor graphics (Gen8).
IOMMU	I/O Memory Mapping Unit
SVM	Shared Virtual Memory, implies the same virtual memory view between the IA cores and processor graphics.
Page Walker (GAM)	GFX page walker which handles page level translations between GFX virtual memory to physical memory domain.

GPU Memory Interface

GPU memory interface functions are divided into 4 different major sections:

- Global Arbitration
- Memory Interface Functions
- Page Translations (GFX Page Walker)
- Ring Interface Functions (GTI)

GT Interface functions are covered at a different chapter/HAS and not part of this documentation. The following documentation is meant for GFX arbitration paths in accessing to memory/cache interfaces and page translations and page walker functions.

Global Arbitration

The global memory arbitration fabric is meant to be a hierarchal memory fabric where memory accesses from different stages of the pipeline are consolidated to a single interface towards GT’s connection to CPU’s ring interface.

The arbitration on the fabric is programmable via a simple per pipeline stage priority levels.

The final arbitration takes places in GAM between parallel compute engines. Each engine (in some cases major pipeline stages are also separated, i.e. Z vs Color vs L3 vs Fixed Functions) gets a count in a grace period where its accesses are counted against a global pool. If a particular engine (or pipeline stage) exhausts its max allowed, it is dropped to a lower priority and goes to fixed pipeline based prioritization. Once all counts are expired, the grace period completes and resets.

The count values are programmable via MMIO (i.e. *_MAX_REQ_COUNT) registers with defaults favoring the pipeline order.

GFX MMIO – MCHBAR Aperture

Address:	140000h – 147FFFh
Default Value:	Same as MCHBAR
Access:	Aligned Word, Dword or Qword Read/Write

This range defined in the graphics MMIO range is an alias with which graphics driver can read and write registers defined in the MCHBAR MMIO space claimed through Device #0. Attributes for registers defined within the MCHBAR space are preserved when the same registers are accessed via this space. Registers that the graphics driver requires access to are Rank Throttling, GMCH Throttling, Thermal Sensor, etc.

The Alias functions works for MMIO access from the CPU only. A command stream load register immediate will drop the data and store register immediate will return all Zeros.

Graphics MMIO registers can be accessed through MMIO BARs in function #0 and function #1 in Device #2. The aliasing mechanism is turned off if memory access to the corresponding function is turned off via software or in certain power states.

Graphics Memory Interface Functions

The major role of an integrated graphics device's Memory Interface (MI) function is to provide various client functions access to "graphics" memory used to store commands, surfaces, and other information used by the graphics device. This chapter describes the basic mechanisms and paths by which graphics memory is accessed.

Information not presented in this chapter includes:

- Microarchitectural and implementation-dependent features (e.g., internal buffering, caching, and arbitration policies).
- MI functions and paths specific to the operation of external (discrete) devices attached via external connections.
- MI functions essentially unrelated to the operation of the internal graphics devices, .e.g., traditional "chipset functions"
- GFX Page Walker and GT interface functions are covered in different chapters.

Graphics Memory Clients

The MI function provides memory access functionality to a number of external and internal graphics memory *clients*, as described in the table below.

Graphics Memory Clients

MI Client	Access Modes
Host Processor	Read/Write of Graphics Operands located in Main Memory. Graphics Memory is accessed using Device 2 Graphics Memory Range Addresses
External PEG Graphics Device	Write-Only of Graphics Operands located in Main Memory via the Graphics Aperture. (This client is not described in this chapter).
Peer PCI Device	Write-Only of Graphics Operands located in Main Memory. Graphics Memory is accessed using Device 2 Graphics Memory Range Addresses (i.e., mapped by GTT). Note that DMI access to Graphics registers is not supported.
Coherent Read/Write (internal)	Internally-generated snooped reads/writes.
Command Stream (internal)	DMA Read of graphics commands and related graphics data.
Vertex Stream (internal)	DMA Read of indexed vertex data from Vertex Buffers by the 3D Vertex Fetch (VF) Fixed Function.
Instruction/State Cache (internal)	Read of pipelined 3D rendering state used by the 3D/Media Functions and instructions executed by the EUs.
Render Cache (internal)	Read/Write of graphics data operated upon by the graphics rendering engines (Blitter, 3D, MPEG, etc.) Read of render surface state.
Sampler Cache (internal)	Read of texture (and other sampled surface) data stored in graphics memory.
Display/Overlay Engines (internal)	Read of display, overlay, cursor, and VGA data.
Media Engines	Read and write of media content and media processing.

Graphics Memory Addressing Overview

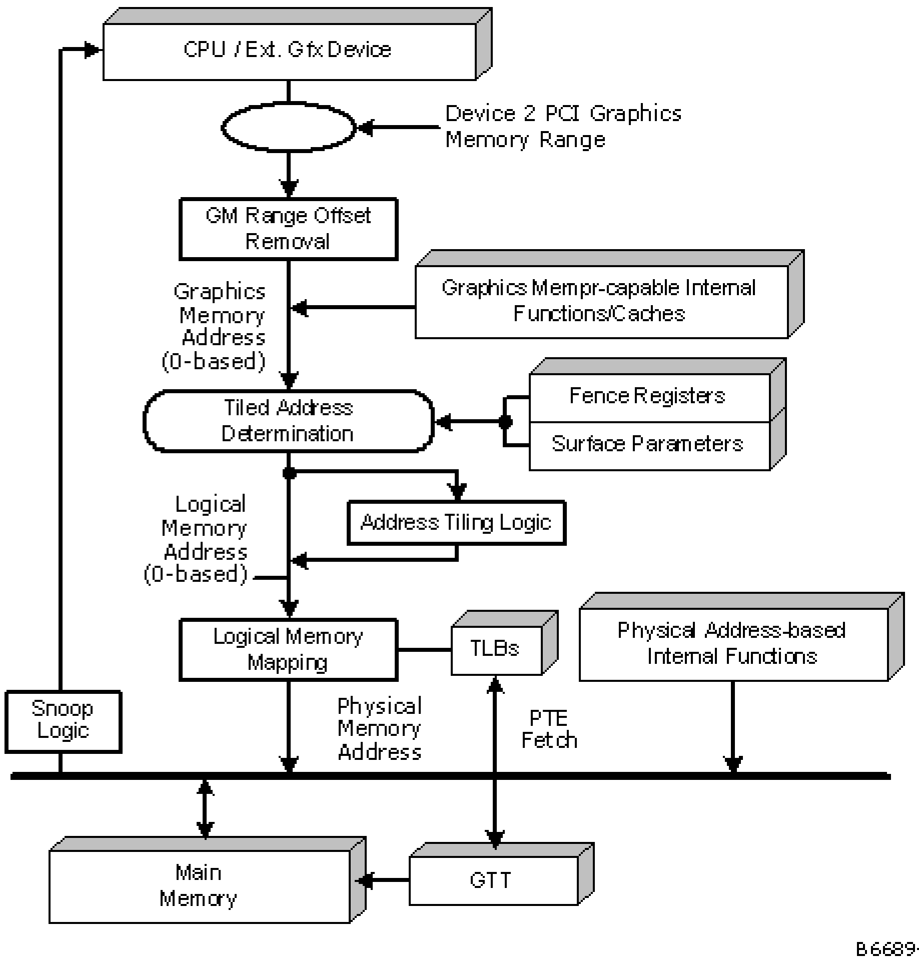
The Memory Interface function provides access to graphics memory (GM) clients. It accepts memory addresses of various types, performs a number of optional operations along *address paths*, and eventually performs reads and writes of graphics memory data using the resultant addresses. The remainder of this subsection will provide an overview of the graphics memory clients and address operations.

Graphics Address Path

Graphics Address Path shows the internal graphics memory address path, connection points, and optional operations performed on addresses. Externally-supplied addresses are normalized to zero-based *Graphics Memory* (*GM) addresses* (GM_Address). If the GM address is determined to be a tiled address (based on inclusion in a fenced region or via explicit surface parameters), *address tiling* is performed. At this point the address is considered a *Logical Memory address*, and is translated into a *Physical Memory address* via the GTT and associated TLBs. The physical memory location is then accessed.

CPU accesses to graphics memory are sent back on the ring to snoop. Hence pages that are mapped cacheable in the GTT will be coherent with the CPU cache if accessed through graphics memory aperture.

Graphics Memory Paths


CPU / Ext. Gfx Device
Device 2 PCI Graphics Memory Range
GM Range Offset Removal
Graphics Memory Address (0-based)
Graphics Mempr-capable Internal Functions/Caches
Tiled Address Determination
Fence Registers
Surface Parameters
Logical Memory Address (0-based)
Address Tiling Logic
Logical Memory Mapping
TLBs
Physical Address-based Internal Functions
Snoop Logic
Physical Memory Address
PTE Fetch
Main Memory
GTT
B6689-01

The remainder of this chapter describes the basic features of the graphics memory address pipeline, namely Address Tiling, Logical Address Mapping, and Physical Memory types and allocation considerations.

Graphics Memory Address Spaces

The Graphics Memory Address Types table lists the five supported Graphics Memory Address Spaces. Note that the Graphics Memory Range Removal function is automatically performed to transform system addresses to internal, zero-based Graphics Addresses.

Graphics Memory Address Types

Address Type	Description	Range	Gen8(BDW)
GMADR	Address range allocated via the Device 2 (integrated graphics device) GMADR register. The processor and other peer (DMI) devices utilize this address space to read/write graphics data that resides in Main Memory. This address is internally converted to a GM_Address.	This is a 4 GB BAR above physical memory.	128 MB, 256 MB, 512 MB, 1GB, 2GB, 4GB
GTTMMADR	The combined Graphics Translation Table Modification Range and Memory Mapped Range. The range requires 16 MB combined for MMIO and Global GTT aperture, with 8 MB of that used by MMIO and 8 MB used by GTT. GTTADR begins at GTTMMADR 8 MB while the MMIO base address is the same as GTTMMADR. For the Global GTT, this range is defined as a memory BAR in graphics device config space It is an alias into which software is required to write Page Table Entry values (PTEs). Software may read PTE values from the global Graphics Translation Table GTT. PTEs cannot be written directly into the global GTT memory area.	This is a 16MB BAR above physical memory.	16 MB (2 MB MMIO + 6 MB reserved + 8 MB GGTT)
GSM	GTT Stolen Memory. It is an 8 MB (max) region taken out of physical memory to store the Global GTT entries for page translations specific to GFX driver use. It is accessible via GTTMMADR from the CPU path however GPU/DE can access the same region directly.	This is an 8 MB region in physical memory not visible to OS.	1 MB, 2 MB, 4 MB, 8 MB
DSM	Data stolen memory, the size is determined with GMS filed (8 bits) with MAX size of 4 GB. This is a stolen memory which can be accessed via GMADR for CPU and directly for GPU/DE. Size is programmable with 32 MB multiplier. Due to a workaround, first 4KB of DSM has to be reserved for GFX hardware use during render engine execution.	This is a max of 4 GB stolen physical memory for GFX data structures.	0 MB, 32 MB, 64 MB, 96 MB, ...4096MB

Address Tiling Function Introduction

When dealing with memory operands (e.g., graphics surfaces) that are inherently rectangular in nature, certain functions within the graphics device support the storage/access of the operands using alternative (tiled) memory formats to increase performance. This section describes these memory storage formats, why and when they should be used, and the behavioral mechanisms within the device to support them.

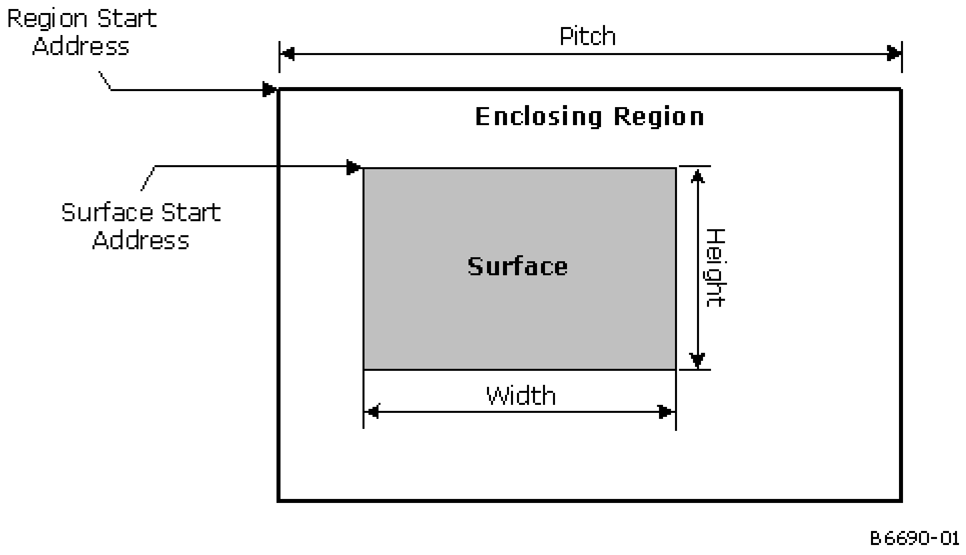
Legacy Tiling Modes:

- **TileY:** Used for most tiled surfaces when **TR_MODE**=TR_NONE.
- **TileX:** Used primarily for display surfaces.
- **TileW:** Used for Stencil surfaces.

Linear vs Tiled Storage

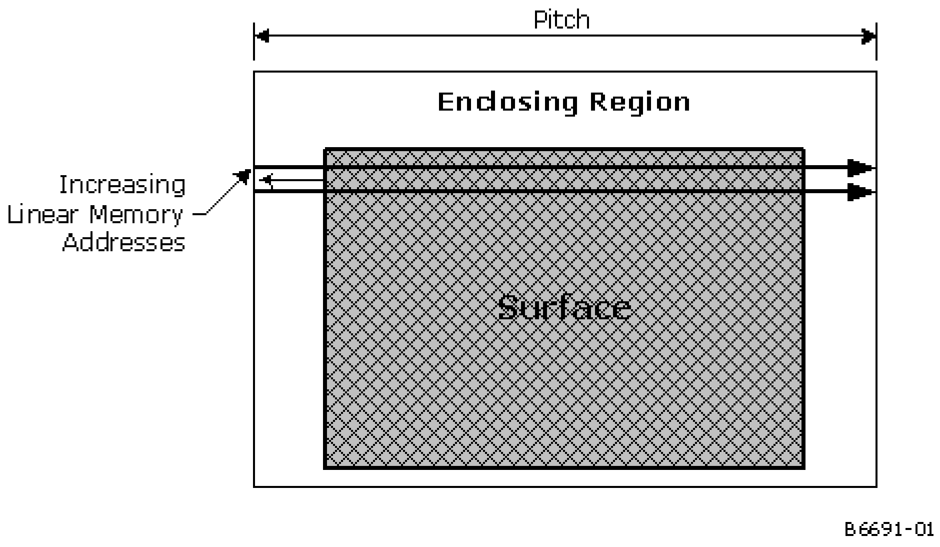
Regardless of the memory storage format, "rectangular" memory operands have a specific *width* and *height*, and are considered as residing within an enclosing rectangular region whose width is considered the *pitch* of the region and surfaces contained within. Surfaces stored within an enclosing region must have widths less than or equal to the region pitch (indeed the enclosing region may coincide exactly with the surface). *Rectangular Memory Operand Parameters* shows these parameters.

Rectangular Memory Operand Parameters



The simplest storage format is the *linear* format (see *Linear Surface Layout*), where each row of the operand is stored in sequentially increasing memory locations. If the surface width is less than the enclosing region's pitch, there will be additional memory storage between rows to accommodate the region's pitch. The pitch of the enclosing region determines the distance (in the memory address space) between vertically-adjacent operand elements (e.g., pixels, texels).

Linear Surface Layout

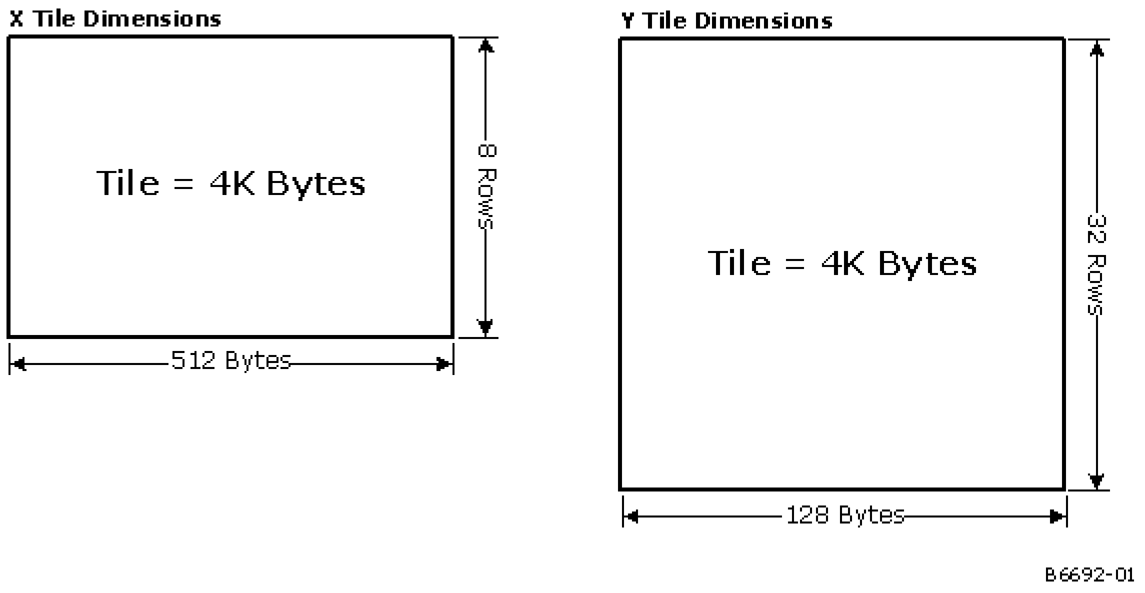


The linear format is best suited for 1-dimensional row-sequential access patterns (e.g., a display surface where each scanline is read sequentially). Here the fact that one object element may reside in a different memory page than its vertically-adjacent neighbors is not significant; all that matters is that horizontally-adjacent elements are stored contiguously. However, when a device function needs to access a 2D subregion within an operand (e.g., a read or write of a 4x4 pixel span by the 3D renderer, a read of a 2x2 texel block for bilinear filtering), having vertically-adjacent elements fall within different memory pages is to be avoided, as the page crossings required to complete the access typically incur increased memory latencies (and therefore lower performance).

One solution to this problem is to divide the enclosing region into an array of smaller rectangular regions, called memory *tiles*. Surface elements falling within a given tile will all be stored in the same physical memory page, thus eliminating page-crossing penalties for 2D subregion accesses within a tile and thereby increasing performance.

Tiles have a fixed 4KB size and are aligned to physical DRAM page boundaries. They are either 8 rows high by 512 bytes wide or 32 rows high by 128 bytes wide (see *Memory Tile Dimensions*). Note that the dimensions of tiles are irrespective of the data contained within – e.g., a tile can hold twice as many 16-bit pixels (256 pixels/row x 8 rows = 2K pixels) than 32-bit pixels (128 pixels/row x 8 rows = 1K pixels).

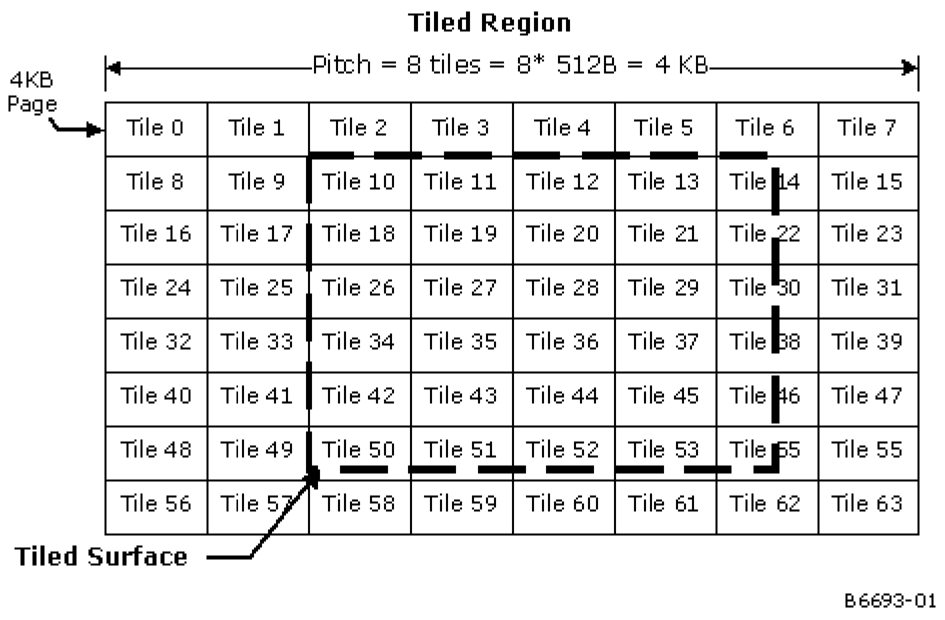
Memory Tile Dimensions



The pitch of a tiled enclosing region must be an integral number of tile widths. The 4KB tiles within a tiled region are stored sequentially in memory in row-major order.

The *Tiled Surface Layout* figure shows an example of a tiled surface located within a tiled region with a pitch of 8 tile widths (512 bytes * 8 = 4KB). Note that it is the *enclosing region* that is divided into tiles – the surface is not necessarily aligned or dimensioned to tile boundaries.

Tiled Surface Layout



Tile Formats

Multiple tile formats are supported by the Gen Core. The following sections define and describe these formats.

Tiling formats are controlled by programming the fields Tile_Mode and Tiled_Resource_Mode in the RENDER_SURFACE_STATE.

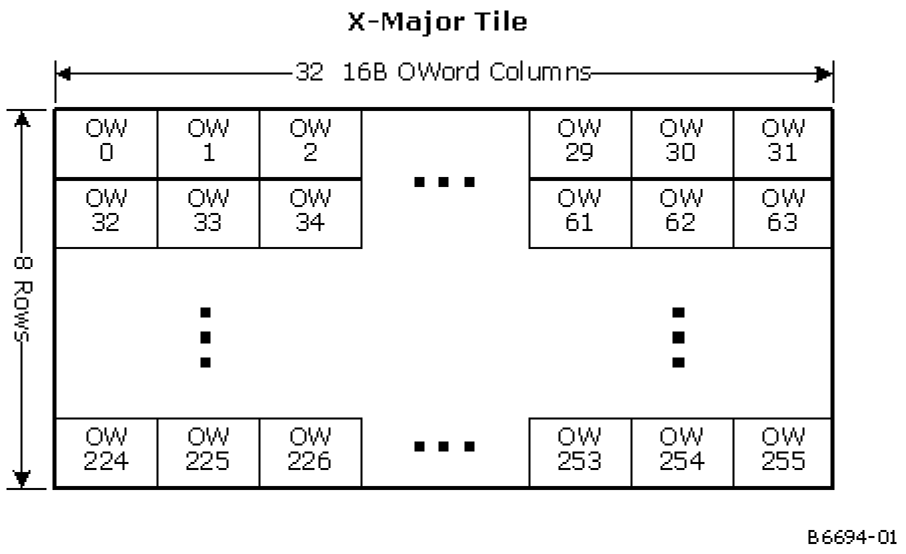
Tile-X Legacy Format

The legacy format Tile-X is a *X-Major* (row-major) storage of tile data units, as shown in the following figure. It is a 4KB tile which is subdivided into an 8-high by 32-wide array of 16-byte OWords. The selection of tile direction only impacts the internal organization of tile data, and does not affect how surfaces map onto tiles. Note that an X-major tiled region with a tile pitch of 1 tile is actually stored in a linear fashion.

Tile-X format is selected for a surface by programming the Tiled_Mode field in RENDER_SURFACE_STATE to XMAJOR.

For 3D sampling operation, a surface using Tile-X layout is generally lower performance the organization of texels in memory.

Tile X-Tile (X-Major) Layout



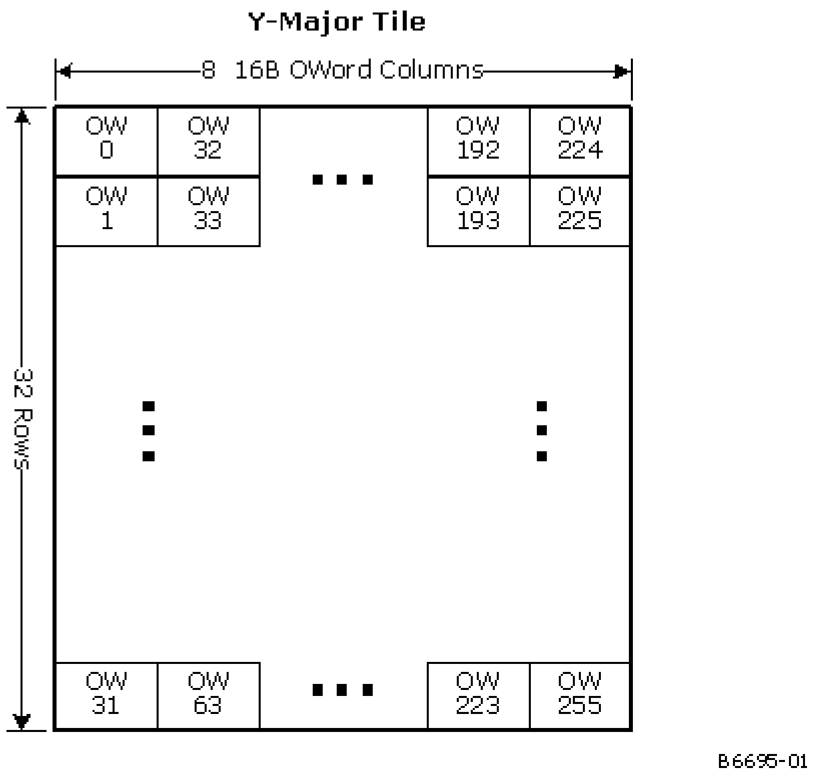
Tile-Y Legacy Format

The device supports Tile-Y legacy format which is *Y-Major* (column major) storage of tile data units, as shown in the following figure. A 4KB tile is subdivided 32-high by 8-wide array of OWords. The selection of tile direction only impacts the internal organization of tile data, and does not affect how surfaces map onto tiles.

Tile-Y surface format is selected by programming the Tile_Mode field in RENDER_SURFACE_STATE to YMAJOR.

Note that 3D sampling of a surface in Tile-Y format is usually has higher performance due to the layout of pixels.

Y-Major Tile Layout



Tiling Algorithm

The following pseudo-code describes the algorithm for translating a tiled memory surface in graphics memory to an address in logical space.

```
 Inputs:
     LinearAddress (offset into regular or LT aperture in terms of bytes)
     Pitch (in terms of tiles)
     WalkY (1 for Y and 0 for X)
     WalkW (1 for W and 0 for the rest)
Static Parameters:
     TileH (Height of tile, 8 for X, 32 for Y, and 64 for W),
     TileW (Width of Tile in bytes, 512 for X, 128 for Y, and 64 for W)

TileSize = TileH * TileW;
RowSize = Pitch * TileSize;
If ( Fenced ) {
     LinearAddress = LinearAddress – FenceBaseAddress;
     LinearAddrInTileW = LinearAddress div TileW;
     Xoffset_inTile = LinearAddress mod TileW;
     Y = LinearAddrInTileW div Pitch;
     X = LinearAddrInTileW mod Pitch + Xoffset_inTile;
}

// Internal graphics clients that access tiled memory already have the X, Y coordinates and
can start here.
YOff_Within_Tile = Y mod TileH;
XOff_Within_Tile = X mod TileW;
TileNumber_InY = Y div TileH;
TileNumber_InX = X div TileW;
TiledOffsetY = RowSize * TileNumber_InY + TileSize * TileNumber_InX +
         TileH * 16 * (XOff_Within_Tile div 16) + YOff_Within_Tile * 16 + (XOff_Within_Tile
mod 16);
TiledOffsetW = RowSize * TileNumber_InY + TileSize * TileNumber_InX +
         TileH * 8 * (XOff_Within_Tile div 8) +
         64 * (YOff_Within_Tile div 8) +
         32 * ((YOff_Within_Tile div 4) mod 2) +
         16 * ((XOff_Within_Tile div 4) mod 2) +
          8 * ((YOff_Within_Tile div 2) mod 2) +
          4 * ((XOff_Within_Tile div 2) mod 2) +
          2 * (YOff_Within_Tile mod 2) +
              (XOff_Within_Tile mod 2);
TiledOffsetX = RowSize * TileNumber_InY + TileSize * TileNumber_InX + TileW *
YOff_Within_Tile + XOff_Within_Tile;
TiledOffset = WalkW ? TiledOffsetW : (WalkY ? TiledOffsetY : TiledOffsetX);
TiledAddress = Tiled ? (BaseAddress + TiledOffset) : (BaseAddress + Y*LinearPitch + X);
TiledAddress = (Tiled &&
         (Address Swizzling for Tiled-Surfaces == 01)) ?
         (WalkW || WalkY) ?
         (TiledAddress div 128) * 128 +
         (((TiledAddress div 64) mod 2) ^
         ((TiledAddress div 512) mod 2)) +
         (TiledAddress mod 32)
         :
         (TiledAddress div 128) * 128 +
         (((TiledAddress div 64) mod 2) ^
         ((TiledAddress div 512) mod 2)
         ((TiledAddress Div 1024) mod2) +
         (TiledAddress mod 32)
         :
         TiledAddress;
```

Address Swizzling for Tiled-Surfaces is no longer used because the main memory controller has a more effective address swizzling algorithm.

For Address Swizzling for Tiled-Surfaces see ARB_MODE – Arbiter Mode Control register, ARB_CTL—Display Arbitration Control 1, and TILECTL - Tile Control register.

The Y-Major tile formats have the characteristic that a surface element in an even row is located in the same aligned 64-byte cacheline as the surface element immediately below it (in the odd row). This spatial locality can be exploited to increase performance when reading 2x2 texel squares for bilinear texture filtering, or reading and writing aligned 4x4 pixel spans from the 3D Render pipeline.

On the other hand, the X-Major tile format has the characteristic that horizontally-adjacent elements are stored in sequential memory addresses. This spatial locality is advantageous when the surface is scanned in row-major order for operations like display refresh. For this reason, the Display and Overlay memory streams only support linear or X-Major tiled surfaces. (Y-Major tiling is not supported by these functions.) This has the side effect that 2D- or 3D-rendered surfaces must be stored in linear or X-Major tiled formats if they are to be displayed. Non-displayed surfaces, e.g., “rendered textures”, can also be stored in Y-Major order.

Tiled Channel Select Decision

Before Gen8, there was a historical configuration control field to swizzle address bit[6] for in X/Y tiling modes. This was set in three different places: TILECTL[1:0], ARB_MODE[5:4], and DISP_ARB_CTL[14:13].

For Gen8 and subsequent generations, the swizzle fields are all reserved, and the CPU's memory controller performs all address swizzling modifications.

Tiling Support

The rearrangement of the surface elements in memory must be accounted for in device functions operating upon tiled surfaces. (Note that not all device functions that access memory support tiled formats). This requires either the modification of an element's linear memory address or an alternate formula to convert an element's X,Y coordinates into a tiled memory address.

However, before tiled-address generation can take place, some mechanism must be used to determine whether the surface elements accessed fall in a linear or tiled region of memory, and if tiled, what the tile region pitch is, and whether the tiled region uses X-Major or Y-Major format. There are two mechanisms by which this detection takes place: (a) an implicit method by detecting that the pre-tiled (linear) address falls within a "fenced" tiled region, or (b) by an explicit specification of tiling parameters for surface operands (i.e., parameters included in surface-defining instructions).

The following table identifies the tiling-detection mechanisms that are supported by the various memory streams.

Access Path	Tiling-Detection Mechanisms Supported
Processor access through the Graphics Memory Aperture	Fenced Regions
3D Render (Color/Depth Buffer access)	Explicit Surface Parameters
Sampled Surfaces	Explicit Surface Parameters
Blt operands	Explicit Surface Parameters
Display and Overlay Surfaces	Explicit Surface Parameters

Tiled (Fenced) Regions

The only mechanism to support the access of surfaces in tiled format by the host or external graphics client is to place them within “fenced” tiled regions within Graphics Memory. A fenced region is a block of Graphics Memory specified using one of the sixteen FENCE device registers. (See *Memory Interface Registers* for details). Surfaces contained within a fenced region are considered tiled from an external access point of view. Note that fences cannot be used to untile surfaces in the PGM_Address space since external devices cannot access PGM_Address space. Even if these surfaces (or any surfaces accessed by an internal graphics client) fall within a region covered by an enabled fence register, that enable will be effectively masked during the internal graphics client access. Only the explicit surface parameters described in the next section can be used to tile surfaces being accessed by the internal graphics clients.

Restriction: Each FENCE register (if its Fence Valid bit is set) defines a Graphics Memory region ranging from 4KB to the aperture size. The region is considered rectangular, with a pitch in tile widths from 1 tile width (128B or 512B) to 512 tile X widths (512 * 512B = 256KB) and 2048 tile Y widths (2048 * 128B = 256KB). Note that fenced regions must not overlap, or operation is UNDEFINED.
Context: Tiled (Fenced) Regions

Restriction: Also included in the FENCE register is a Tile Walk field that specifies which tile format applies to the fenced region.
Context: Tiled (Fenced) Regions

Tiled Surface Parameters

Internal device functions require explicit specification of surface tiling parameters via information passed in commands and state. This capability is provided to limit the reliance on the fixed number of fence regions.

The following table lists the surface tiling parameters that can be specified for 3D Render surfaces (Color Buffer, Depth Buffer, Textures, etc.) via SURFACE_STATE.

Surface Parameter	Description
Tiled Surface	If ENABLED, the surface is stored in a tiled format. If DISABLED, the surface is stored in a linear format.
Tile Walk	If Tiled Surface is ENABLED, this parameter specifies whether the tiled surface is stored in Y-Major or X-Major tile format.
Base Address	Additional restrictions apply to the base address of a Tiled Surface vs. that of a linear surface.
Pitch	Pitch of the surface. Note that, if the surface is tiled, this pitch must be a multiple of the tile width.

Tiled Surface Restrictions

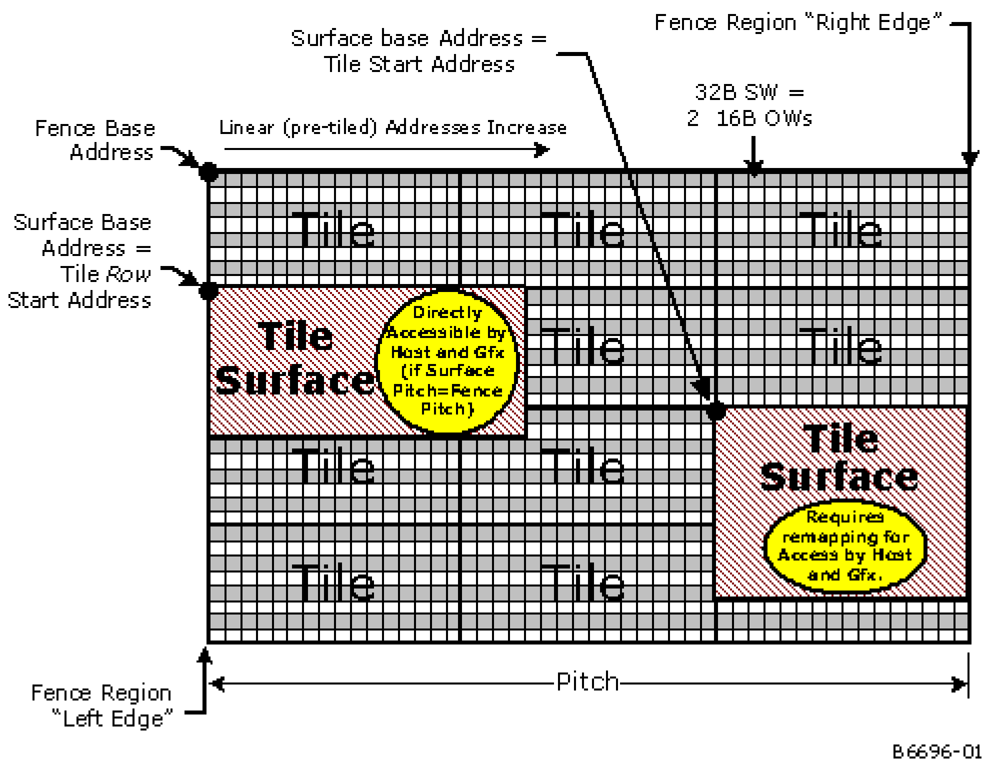
Additional restrictions apply to the Base Address and Pitch of a surface that is tiled. In addition, restrictions for tiling via SURFACE_STATE are subtly different from those for tiling via fence regions. The most restricted surfaces are those that will be accessed both by the host (via fence) and by internal device functions. An example of such a surface is a tiled texture that is initialized by the CPU and then sampled by the device.

The tiling algorithm for internal device functions is different from that of fence regions. Internal device functions always specify tiling in terms of a surface. The surface must have a base address, and this base address is not subject to the tiling algorithm. Only *offsets* from the base address (as calculated by X, Y addressing within the surface) are transformed through tiling. The base address of the surface must therefore be 4KB-aligned. This forces the 4KB tiles of the tiling algorithm to exactly align with 4KB device pages once the tiling algorithm has been applied to the offset. The width of a surface must be less than or equal to the surface pitch. There are additional considerations for surfaces that are also accessed by the host (via a fence region).

Fence regions have no base address per se. Host linear addresses that fall in a fence region are translated in their entirety by the tiling algorithm. It is as if the surface being tiled by the fence region has a base address in graphics memory equal to the fence base address, and all accesses of the surfaces are (possibly quite large) offsets from the fence base address. Fence regions have a virtual “left edge” aligned with the fence base address, and a “right edge” that results from adding the fence pitch to the “left edge”. Surfaces in the fence region must not straddle these boundaries.

Base addresses of surfaces that are to be accessed both by an internal graphics client and by the host have the tightest restrictions. In order for the surface to be accessed without GTT re-mapping, the surface base address (as set in SURFACE_STATE) must be a “Tile Row Start Address” (TRSA). The first address in each tile row of the fence region is a Tile Row Start Address. The first TRSA is the fence base address. Each TRSA can be generated by adding an integral multiple of the row size to the fence base address. The row size is simply the fence pitch in tiles multiplied by 4KB (the size of a tile.)

Tiled Surface Placement



The pitch in SURFACE_STATE must be set equal to the pitch of the fence that will be used by the host to access the surface if the same GTT mapping will be used for each access. If the pitches differ, a different GTT mapping must be used to eliminate the "extra" tiles (4KB memory pages) that exist in the excess rows at the right side of the larger pitch. Obviously no part of the surface that will be accessed can lie in pages that exist only in one mapping but not the other. The new GTT mapping can be done manually by SW between the time the host writes the surface and the device reads it, or it can be accomplished by arranging for the client to use a different GTT than the host (the PPGTT -- see *Logical Memory Mapping* below).

The width of the surface (as set in SURFACE_STATE) must be less than or equal to both the surface pitch and the fence pitch in any scenario where a surface will be accessed by both the host and an internal graphics client. Changing the GTT mapping will not help if this restriction is violated.

Surface Access	Base Address	Pitch	Width	Tile "Walk"
Host only	No restriction	Integral multiple of tile size <= 256KB	Must be <= Fence Pitch	No restriction
Client only	4KB-aligned	Integral multiple of tile size <= 256KB	Must be <= Surface Pitch	Restrictions imposed by the client (see Per Stream Tile Format Support)
Host and Client, No GTT Remapping	Must be TRSA	Fence Pitch = Surface Pitch = integral multiple of tile size <= 256KB	Width <= Pitch	Surface Walk must meet client restriction, Fence Walk = Surface Walk
Host and Client, GTT Remapping	4KB-aligned for client (will be tile-aligned for host)	Both must be Integral multiple of tile size <=128KB, but not necessarily the same	Width <= Min(Surface Pitch, Fence Pitch)	Surface Walk must meet client restriction, Fence Walk = Surface Walk

Per-Stream Tile Format Support

MI Client	Tile Formats Supported	
CPU Read/Write	All	
Display/Overlay	Y-Major not supported. X-Major required for Async Flips	
Blt	Linear and X-Major only No Y-Major support	
3D Sampler	All Combinations of TileY, TileX and Linear are supported. TileY is the fastest, Linear is the slowest.	
3D Color,Depth	**Rendering Mode Color-vs-Depth bpp**	**Buffer Tiling Supported**
	Classical Same Bpp	Both Linear Both TileX Both TileY Linear & TileX Linear & TileY TileX & TileY
	Classical Mixed Bpp	Both Linear Both TileX Both TileY Linear & TileX Linear & TileY TileX & TileY

Main Memory

The integrated graphics device is capable of using 4KB pages of physical main (system) memory for graphics functions. Some of this main memory can be “stolen” from the top of system memory during initialization (e.g., for a VGA buffer). However, most graphics operands are dynamically allocated to satisfy application demands. To this end the graphics driver frequently needs to allocate locked-down (i.e., non-swappable) physical system memory pages – typically from a cacheable non-paged pool. The locked pages required to back large surfaces are typically non-contiguous. Therefore a means to support “logically-contiguous” surfaces backed by discontiguous physical pages is required. The Graphics Translation Table (GTT) described in previous sections provides the means.

Optimizing Main Memory Allocation

This section includes information for software developers on how to allocate SDRAM Main Memory (SM) for optimal performance in certain configurations. The general idea is that these memories are divided into some number of page types, and careful arrangement of page types both within and between surfaces (e.g., between color and depth surfaces) results in fewer page crossings and therefore yields somewhat higher performance.

The algorithm for allocating physical SDRAM Main Memory pages to logical graphics surfaces is somewhat complicated by (1) permutations of memory device technologies (which determine page sizes and therefore the number of pages per device row), (2) memory device row population options, and (3) limitations on the allocation of physical memory (as imposed by the OS).

However, the theory to optimize allocation by limiting page crossing penalties is simple: (a) Switching between open pages is optimal (again, the pages do not need to be sequential), (b) Switching between memory device rows does not in itself incur a penalty, and (c) Switching between pages within a particular bank of a row incurs a page miss and should therefore be avoided.

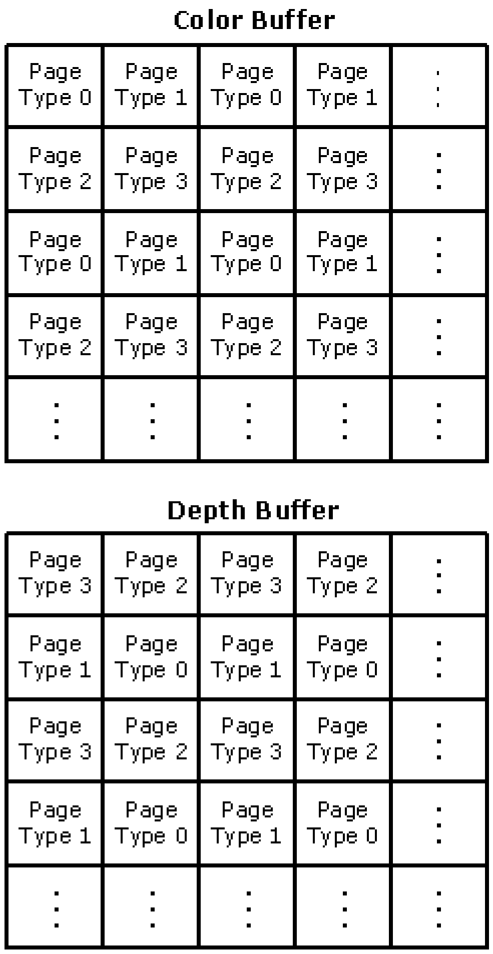
Application of the Theory (Page Coloring)

This section provides some scenarios of how Main Memory page allocation can be optimized.

3D Color and Depth Buffers

Here we want to minimize the impact of page crossings (a) between corresponding pages (1-4 tiles) in the Color and Depth buffers, and (b) when moving from a page to a neighboring page within a Color or Depth buffer. Therefore corresponding pages in the Color and Depth Buffers, and adjacent pages within a Color or Depth Buffer should be mapped to different page types (where a page's "type" or "color" refers to the row and bank it's in).

Memory Pages Backing Color and Depth Buffers



Color Buffer

Page Type 0	Page Type 1	Page Type 0	Page Type 1	⋮
Page Type 2	Page Type 3	Page Type 2	Page Type 3	⋮
Page Type 0	Page Type 1	Page Type 0	Page Type 1	⋮
Page Type 2	Page Type 3	Page Type 2	Page Type 3	⋮
⋮	⋮	⋮	⋮	⋮

Depth Buffer

Page Type 3	Page Type 2	Page Type 3	Page Type 2	⋮
Page Type 1	Page Type 0	Page Type 1	Page Type 0	⋮
Page Type 3	Page Type 2	Page Type 3	Page Type 2	⋮
Page Type 1	Page Type 0	Page Type 1	Page Type 0	⋮
⋮	⋮	⋮	⋮	⋮

B6701-01

For higher performance, the Color and Depth Buffers could be allocated from different memory device rows.

Media/Video

The Y surfaces can be allocated using 4 page types in a similar fashion to the Color Buffer diagram. The U and V surfaces would split the same 4 page types as used in the Y surface.

Physical Graphics Address Types

The Physical Memory Address Types table lists the various *physical* address types supported by the integrated graphics device. Physical Graphics Addresses are either generated by Logical Memory mappings or are directly specified by graphics device functions. *These physical addresses are not subject to tiling or GTT re-mappings*.

Physical Memory Address Types

Address Type	Description	Range
MM_Address	Main Memory Address. Offset into physical, *unsnooped* Main Memory.	[0,TopOfMemory-1]
SM_Address	System Memory Address. Accesses are snooped in processor cache, allowing shared graphics/ processor access to (locked) cacheable memory data.	[0,512GB]

Graphics Translation Tables

The Graphics Translation Tables GTT (Graphics Translation Table, sometimes known as the global GTT) and PPGTT (Per-Process Graphics Translation Table) are memory-resident page tables containing an array of DWord Page Translation Entries (PTEs) used in mapping logical Graphics Memory addresses to physical memory addresses, and sometimes snooped system memory "PCI" addresses.

The base address (MM offset) of the GTT and the PPGTT are programmed via the PGTBL_CTL and PGTBL_CTL2 MI registers, respectively. The translation table base addresses must be 4KB aligned. The GTT size can be either 128KB, 256KB, or 512KB (mapping to 128MB, 256MB, and 512MB aperture sizes respectively) and is physically contiguous. The global GTT should only be programmed via the range defined by GTTMMADR. The PPGTT is programmed directly in memory. The per-process GTT (PPGTT) size is controlled by the PGTBL_CTL2 register. The PPGTT can, in addition to the above sizes, also be 64KB in size (corresponding to a 64MB aperture). Refer to the GTT Range chapter for a bit definition of the PTE entries.

Virtual Memory

GT supports standard virtual memory models as defined by the IA programmer's guide. This section describes the different paging models, their behaviors, and the page table formats.

GFX Page Tables

GPU supports the following page table mechanisms:

- IA32e compatible GTT
- PPGTT – per process GTT (private GFX)
- GGTT- global GTT

All page tables have the same PTE format, the difference is how to reach the final physical page and which fields with PTE are used. The page walks have been extended to cover guest physical address to host physical address translation. But conceptually, the GFX page tables remain intact.

The pre-gen8 page tables are formed via a 32-bit format which underwent small adjustments in various generations, but nothing fundamental. For gen8, the approach for page walks are going modified to allow larger page table entries with increased capability and similarity to IA32e page tables to simplify the transition.

Gen8 Page Table Modes

For gen8, the GFX Aperture and Display accesses are always mapped thru Global GTT. This is done to keep the walk simple (i.e. 1-level), however GT accesses to memory can be mapped via Global GTT and/or ppGTT with various addressing modes.

The walk modes are listed as following:

Walk Mode
Global GTT with 32b virtual addressing: Global GTT usage is similar to pre-gen8 behavior with extended capability to increase the VA to 4GB (from 2GB) and use a similar 64b PTE as ppGTT. The breakdown of the PTE for global GTT is given in later sections but fundamentally allows 1-level pagewalk where the 20b index is used to select the 64b PTE from stolen memory.
Legacy 32b VA with ppGTT: This is a mode where ppGTT page tables are managed via GFX s/w (driver) and context is tagged as Legacy 32b VA. Given each page walk is managed via 9b of the virtual address, 20b index is broken into 3 parts. However to optimize the walks and make it look like pre-gen8, s/w provides 4 pointers to page tables (called 4 PDP entries) – GPA. GFX h/w uses the four pointers and fetches the 4x4KB into h/w (for render and media) before the context execution starts. The optimization limits the dynamic (on demand) page walks to 1-level only.
Legacy 48b VA with ppGTT: Going forward to allow GFX address expansion beyond 4GB, the capability is added address space. 48b VA requires 36b indexing (4x9b) which means for 4-levels of page walk. To prevent the overhead slightly, h/w will cache the entire content of PML4 (4kB) to limit the on-demand walks to 3 levels in the worst case.
Advanced 48b VA with IA32e support via IOMMU: 48b addressing in advanced mode is managed via IOMMU settings where the base of the page table can be found after the root / context tables based on bus/device/function numbers. As final step the PASID# is used as an index in PASID table to find page table pointer to start the 4-level page walk. Similar to previous 48b VA mode, h/w will read the entire content of PML4 and limit the dynamic page walks to 3-level (worst case)

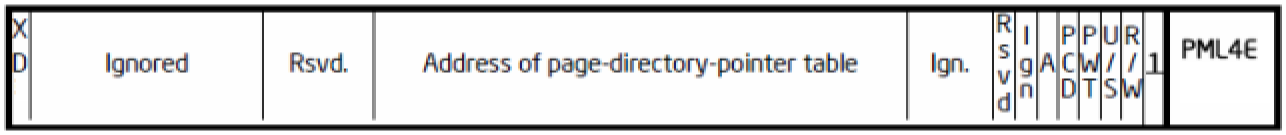
Gen8 Per Process GTT

Gen8 per process GTT mechanism has multiple hooks and mechanisms for s/w to prepare the page walks on hardware. The listed mechanisms here are selectable per-context and descriptors are delivered to hardware as part of context descriptor.

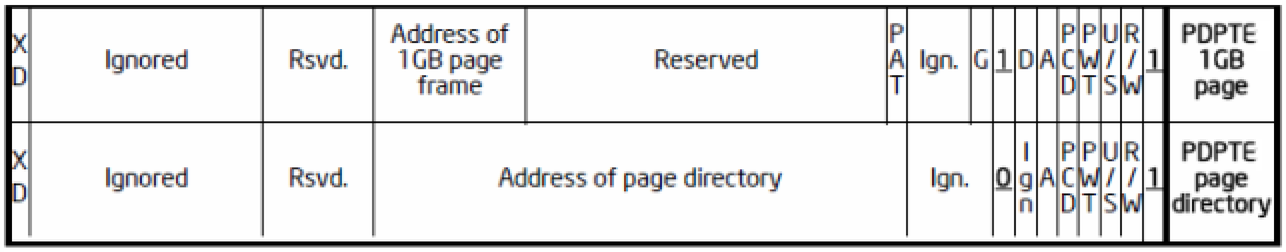
The entry contents are also modified to match the same format as IA32e page tables allowing future expansion for sharable page tables as well as higher order virtual addressing.

Page Tables Entry (PTE) Formats

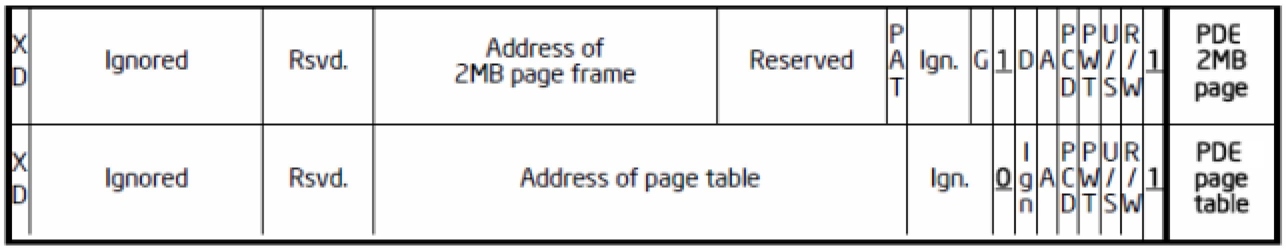
Page Table Entry (PTE) formats will follow the IA32e layout as given below:



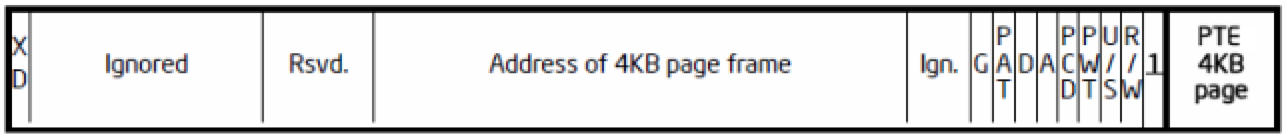
XD	Ignored	Rsvd.	Address of page-directory-pointer table	Ign.	Rsvd	Ign	A	PCD	PWT	U/S	R/W	1	PML4E



XD	Ignored	Rsvd.	Address of 1GB page frame	Reserved	PAT	Ign.	G	1	D	A	PCD	PWT	U/S	R/W	1	PDPTE 1GB page
XD	Ignored	Rsvd.	Address of page directory			Ign.		0	Ign	A	PCD	PWT	U/S	R/W	1	PDPTE page directory



XD	Ignored	Rsvd.	Address of 2MB page frame	Reserved	PAT	Ign.	G	1	D	A	PCD	PWT	U/S	R/W	1	PDE 2MB page
XD	Ignored	Rsvd.	Address of page table			Ign.		0	Ign	A	PCD	PWT	U/S	R/W	1	PDE page table

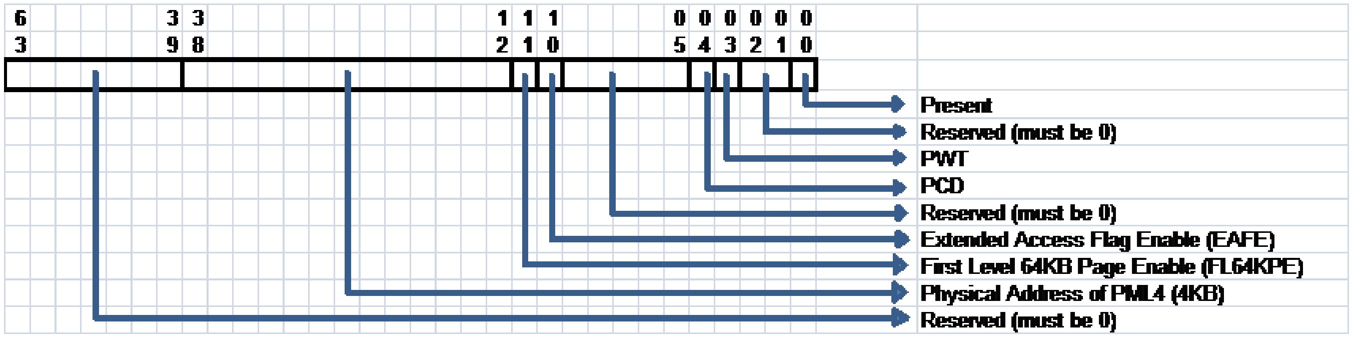


XD	Ignored	Rsvd.	Address of 4KB page frame	Ign.	G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE 4KB page

Each table entry is further broken down along with the required functions. GFX has a 4 level page table which is pointed out by context descriptor starting with the 4th level of PML4. The next levels have slightly different formats depending on the size of the page supported. 1GB and 2MB page formats are required for support.

Pointer to PML4 Table

Page table pointer is the starting address where the PML4 table starts. The contents of pointer are provided by PASID table entry in case of advanced context, else it is provided by software as part of the legacy context with 48b addressing.


63
39
38
12
11
10
05
04
03
02
01
00
Present
Reserved (must be 0)
PWT
PCD
Reserved (must be 0)
Extended Access Flag Enable (EAFE)
First Level 64KB Page Enable (FL64KPE)
Physical Address of PML4 (4KB)
Reserved (must be 0)

PWT/PCD bits are used as indexes into a PAT register which defines the cache attributes for the next level page table access. The description of their use is listed later in the document.

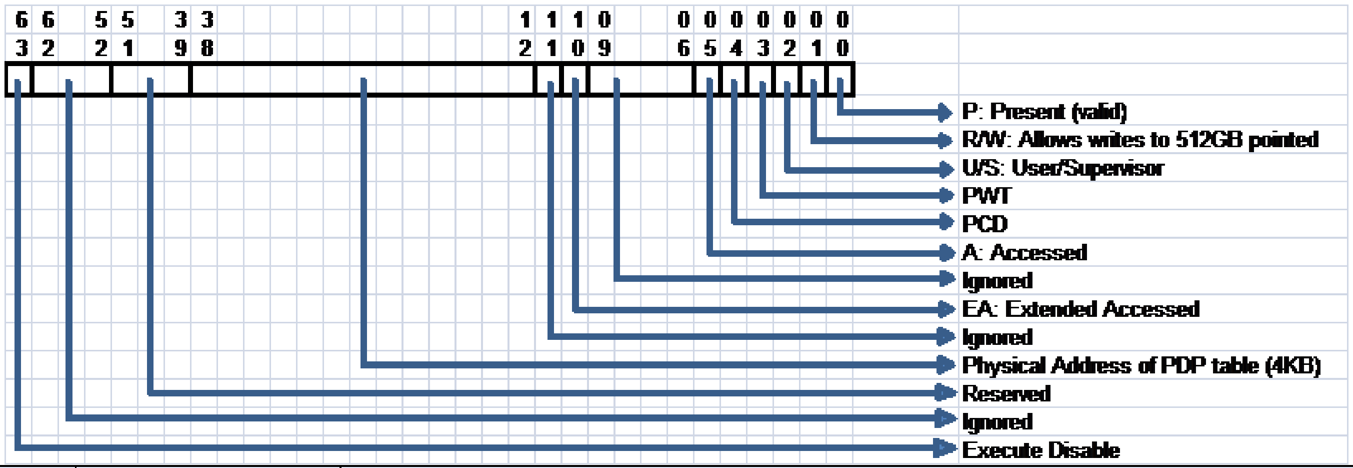
GPU architecture does not follow memory typing for page table accesses. Page table accesses are handled as WB.

Physical address of PML4 is a physical address pointer to a 4KB page where the PML4 table would reside.

PML4E: Pointer to PDP Table

PML4 is used to locate the page directory pointer tables distributed in physical memory. For Gen8, PML4 is used for advanced GPGPU context scheduled via PASID table as well as legacy context with 48b VA.

For 32b VA scheduling, there is no use of PML4.



Bits	Field	Description
63	XD: Execute Disable	If NXE=1 in the relevant extended-context-entry, execute permission is not granted for requests to the 512-GByte region controlled by this entry when XD=1. (**Note:** Gen8 (BDW) does not support execute privilege.) *Advanced mode only.*
62:52	Ignored/Reserved	Ignored/not used by hardware.
51:39	Ignored/Reserved	Ignored/not used by hardware.
38:12	ADDR: Address	Physical address of PDP Table which is a pointer to a 4KB region in memory where the corresponding page directory pointer table is.
11	Ignored/Reserved	Ignored/not used by hardware.
10	EA: Extended Access	Extended Access bit is added for devices to separate accesses from IA cores. If EAFE=1 in the relevant PASID-entry, this bit indicates whether this entry has been used for address translation by device. It is the device's responsibility to set this bit. If EAFE=0 in the relevant PASID-entry, this bit is ignored. This bit applies to GPU. *Advanced mode only.*
9:6	Ignored/Reserved	Ignored/not used by hardware.
5	A: Accessed	A-bit needs to be managed as the PDP table being accessed. Hardware needs to write this bit for the first access to the 512GB region defined with this PML4 entry. See later sections for A/D-bit management. *Advanced mode only.*
4	PCD: Page level Cache Disable	For devices operating in the processor coherency domain, this field indirectly determines the memory type used to access the page directory-pointer table referenced by this entry. *GPU does not support any memory type but WB when accessing paging structures.*

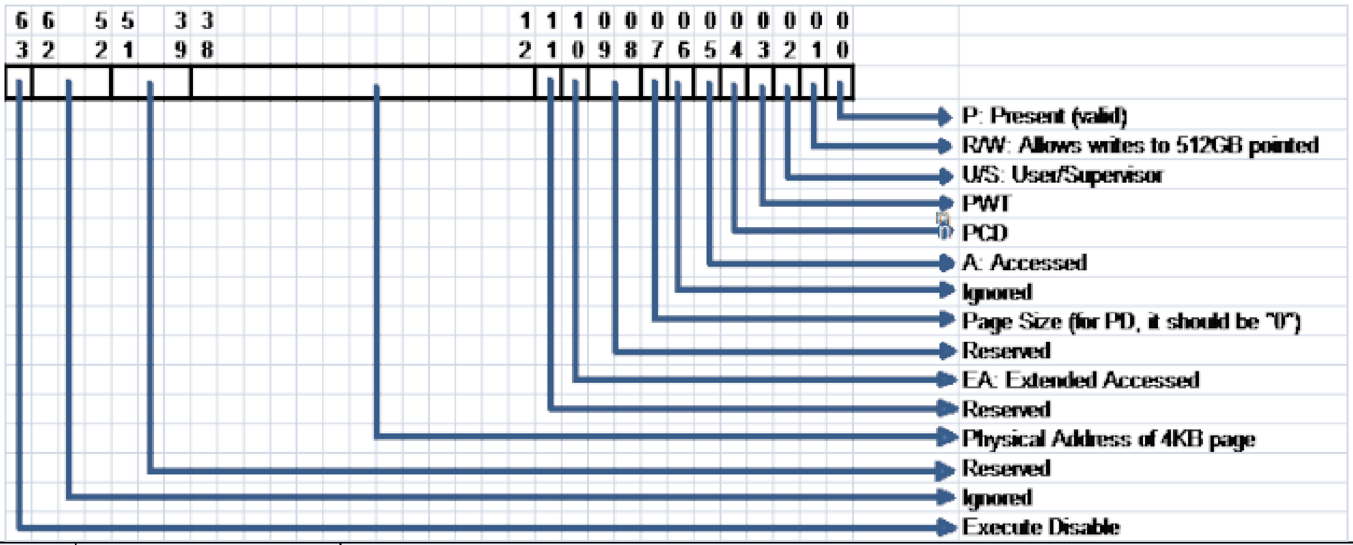
Bits	Field	Description
3	PWT: Page level Write-Through	For devices operating in the processor coherency domain, this field indirectly determines the memory type used to access the page directory-pointer table referenced by this entry. *GPU does not support any memory type but WB when accessing paging structures.*
2	U/S: User/Supervisor	User vs. supervisor access rights. If 0, requests with user-level privilege are not allowed to the 512-GByte region controlled by this entry. See a later section for access rights. (**Note:** Gen8 (BDW) does not support privilege.) *Advanced mode only.*
1	R/W: Read/Write	Write permission rights. If 0, write permission not granted for requests with user-level privilege (and requests with supervisor-level privilege, if WPE=1 in the relevant extended-context-entry) to the 512-GByte region controlled by this entry. See a later section for access rights. (**Note:** Gen8 (BDW) does not support privilege.) *Advanced mode only.*
0	P: Present	PML4 Entry is present. It must be 1 to point to a page directory pointer table.

PDPE: Pointer to PD Table

PDP is used to locate the page directory. PDP table is used by GFX in case of standard context, however the entries are used regardless of context type.

Given IA32e supports 1GB pages, the PDPE has a mechanism to identify a way to say whether this PDPE represents a pointer to page directory or to a contiguous 1GB physical memory.

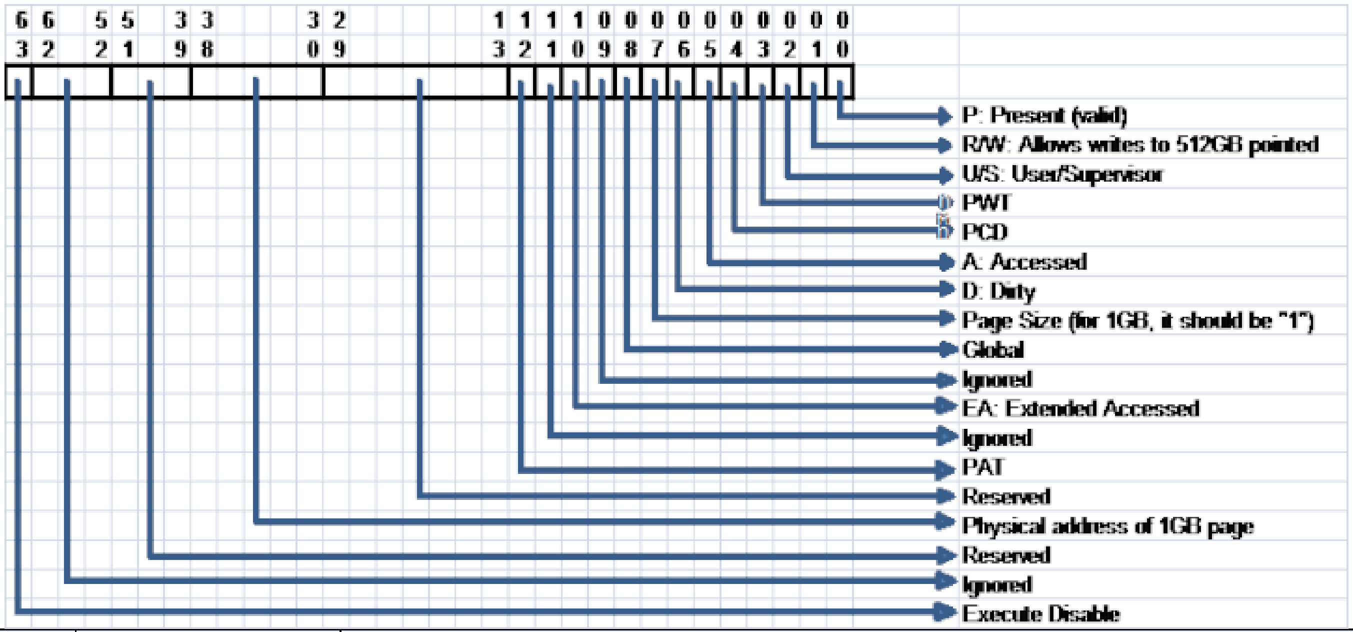
PDPE for PD



Bits	Field	Description
63	XD: Execute Disable	If NXE=1 in the relevant extended-context-entry, execute permission is not granted for requests to the 1-GByte region controlled by this entry when XD=1. (**Note:** Gen8 (BDW) does not support execute privilege.) *Advanced mode only.*
62:52	Ignored/Reserved	Ignored/not used by hardware.
51:39	Ignored/Reserved	Ignored/not used by hardware.
38:12	ADDR: Address	Physical address of PD Table which is a pointer to a 4KB region in memory where the corresponding page directory pointer table is.
11	Ignored/Reserved	Ignored/not used by hardware.
10	EA: Extended Access	Extended Access bit is added for devices to separate accesses from IA cores. If EAFE=1 in the relevant PASID-entry, this bit indicates whether this entry has been used for address translation by device. It is the device's responsibility to set this bit. If EAFE=0 in the relevant PASID-entry, this bit is ignored. This bit applies to GPU. *Advanced mode only.*
9:8	Ignored/Reserved	Ignored/not used by hardware.
7	PS: Page Size	As part of IA32e, there is an option to be able to support 1GB pages. Whether to use this PDP entry as a pointer to Page Directory vs. pointer to a 1GB page is defined as part of the Page Size. i.e. "0" means this entry points to a page directory.
6	Ignored/Reserved	Ignored/not used by hardware.

Bits	Field	Description
5	A: Accessed	A-bit needs to be managed as the PDP table being accessed. Hardware needs to write this bit for the first access to the 1GB region defined with this PDP entry. See later sections for A/D-bit management. *Advanced mode only.*
4	PCD: Page level Cache Disable	For devices operating in the processor coherency domain, this field indirectly determines the memory type used to access the page directory-pointer table referenced by this entry. *GPU does not support any memory type but WB when accessing paging structures.*
3	PWT: Page level Write-Through	For devices operating in the processor coherency domain, this field indirectly determines the memory type used to access the page directory-pointer table referenced by this entry. *GPU does not support any memory type but WB when accessing paging structures.*
2	U/S: User/Supervisor	User vs supervisor access rights. If 0, requests with user-level privilege are not allowed to the 1-GByte region controlled by this entry. See a later section for access rights. (**Note:** Gen8 (BDW) does not support privilege.) *Advanced mode only.*
1	R/W: Read/Write	Write permission rights. If 0, write permission not granted for requests with user-level privilege (and requests with supervisor-level privilege, if WPE=1 in the relevant extended-context-entry) to the 1-GByte region controlled by this entry. See a later section for access rights. (**Note:** Gen8 (BDW) does not support privilege.) *Advanced mode only.*
0	P: Present	PDP Entry is present. It must be 1 to point to a page directory table.

PDPE for 1GB Page



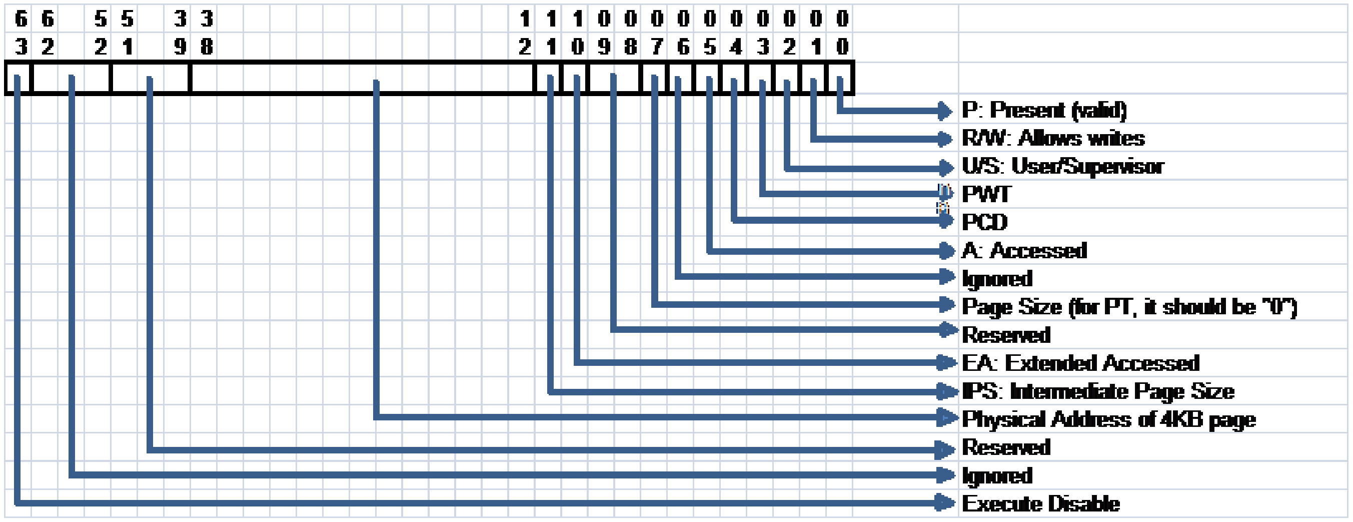
Bits	Field	Description
63	XD: Execute Disable	If NXE=1 in the relevant extended-context-entry, execute permission is not granted for requests to the 1GB page referenced by this entry when XD=1. (**Note:** Gen8 (BDW) does not support execute privilege.) *Advanced mode only.*
62:52	Ignored/Reserved	Ignored/not used by hardware.
51:39	Ignored/Reserved	Ignored/not used by hardware.
38:30	ADDR: Address	Physical address of the 1GB page referenced by this entry. This address is the GPA and may need to be further translated with 2nd level translations.
29:13	Ignored/Reserved	Ignored/not used by hardware.
12	PAT: Page Attribute	For devices operating in the processor coherency domain, this field indirectly determines the memory type used to access the 1GB page referenced by this entry. *See later sections for memory type determination.*
11	Ignored/Reserved	Ignored/not used by hardware.
10	EA: Extended Access	Extended Access bit is added for devices to separate accesses from IA cores. If EAFE=1 in the relevant PASID-entry, this bit indicates whether this entry has been used for address translation by device. It is the device's responsibility to set this bit. If EAFE=0 in the relevant PASID-entry, this bit is ignored. This bit applies to GPU. *Advanced mode only.*
9	Ignored/Reserved	Ignored/not used by hardware.
8	G: Global	Global Page which can be used across contexts and does not need to be invalidated with context/PASID level invalidations. If PGE=1 in the relevant extended-context-entry, this field can be Set by software to indicate the 1-GByte page translation is global. *G-bit is not used by GPU.*

Bits	Field	Description
7	PS: Page Size	As part of IA32e, there is an option to be able to support 1GB pages. Whether to use this PDP entry as a pointer to Page Directory vs pointer to a 1GB page is defined as part of the Page Size. i.e. 1 means this entry points to a 1GB page and no further translation is required.
6	D: Dirty	D-bit needs to be managed as the 1GB page being written. Hardware needs to write this bit for the first write access to 1GB defined with PDP entry. See later sections for A/D-bit management. *Advanced mode only.*
5	A: Accessed	A-bit needs to be managed as the 1GB page being accessed. Hardware needs to write this bit for the first access to 1GB defined with PDP entry. See later sections for A/D-bit management. *Advanced mode only.*
4	PCD: Page level Cache Disable	For devices operating in the processor coherency domain, this field indirectly determines the memory type used to access the 1GB page referenced by this entry. See later sections for memory type determination.
3	PWT: Page level Write-Through	For devices operating in the processor coherency domain, this field indirectly determines the memory type used to access the 1GB page referenced by this entry. See later sections for memory type determination.
2	U/S: User/Supervisor	User vs. supervisor access rights. If 0, requests with user-level privilege are not allowed to the 1GB page referenced by this entry. See a later section for access rights. (**Note:** Gen8 (BDW) does not support privilege.) *Advanced mode only.*
1	R/W: Read/Write	Write permission rights. If 0, write permission not granted for requests with user-level privilege (and requests with supervisor-level privilege, if WPE=1 in the relevant extended-context-entry) to the 1GB page referenced by this entry. See a later section for access rights. (**Note:** Gen8 (BDW) does not support privilege.) *Advanced mode only.*
0	P: Present	PDP Entry is present. It must be 1 to map to a 1GB page.

PDE for Page Table

Page directory has the same concept as existing GFX page table definition (prior to gen8) with different entry format. PDE is used to point to page tables as well as to define large pages (2MB).

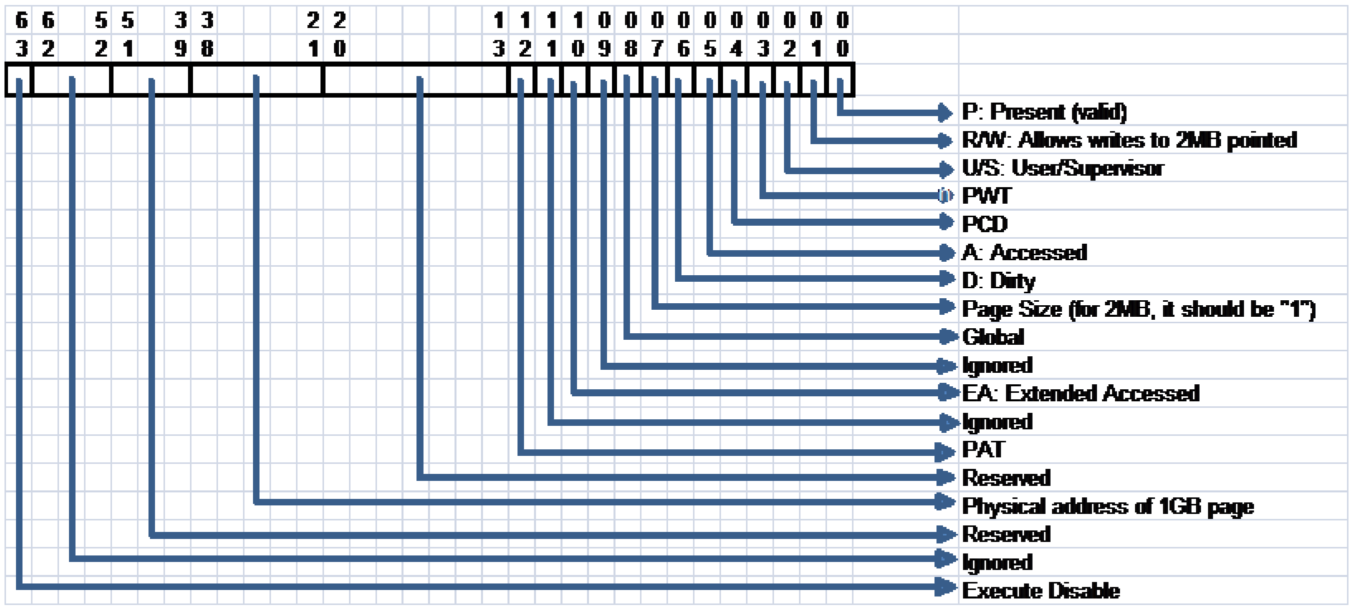
PDE for Page Table



Bits	Field	Description
63	XD: Execute Disable	If NXE=1 in the relevant extended-context-entry, execute permission is not granted for requests to the 2MByte region controlled by this entry when XD=1. *(Note: Gen8 (BDW) does not support execute privilege.)* *Advanced mode only*
62:52	*Ignored/Reserved*	*Ignored/not used by hardware*
51:39	*Ignored/Reserved*	*Ignored/not used by hardware*
38:12	ADDR: Address	Physical address of Page Table which is a pointer to a 4KB region in memory where the corresponding page directory pointer table is.
11	IPS: Intermediate Page Size	If FL64KPE=1 in the relevant PASID-entry, the page-table referenced by PDE with IPS=1, translates to 64-KByte pages. If IPS=0 in the PDE, the page-table referenced by the PDE translates to 4-KByte pages. If FL64KPE=0 in the relevant PASID-entry, this bit is ignored. Note that 64KB pages are for the Page table under the PD entry. *Note: Gen8 implementation does not use FL64KPE.* *Note: In BDW:G1 this bit is ignored.*
10	EA: Extended Access	Extended Access bit is added for devices to separate accesses from IA cores. If EAFE=1 in the relevant PASID-entry, this bit indicates whether this entry has been used for address translation by device. It is the devices responsibility to set this bit. If EAFE=0 in the relevant PASID-entry, this bit is ignored. This bit applies to GPU. *Advanced mode only*
9:8	*Ignored/Reserved*	*Ignored/not used by hardware*
7	PS: Page Size	As part of IA32e, there is an option to be able to support 2MB pages. Whether to use

Bits	Field	Description
		the this PD entry as a pointer to Page table vs pointer to a 2MB page is defined as part of the Page Size. i.e. “0” means this entry points to a page table.
6	*Ignored/Reserved*	*Ignored/not used by hardware*
5	A: Accessed	A-bit needs to be managed as the PDP table being accessed. Hardware needs to write this bit for the first access to the 2MB region defined with this PD entry. See later sections for A/D-bit management. *Advanced mode only*
4	PCD: Page level cache disable	For devices operating in the processor coherency domain, this field indirectly determines the memory type used to access the page directory-pointer table referenced by this entry. *GPU does not support any memory type but WB when accessing paging structures.*
3	PWT: Page level Write-through	For devices operating in the processor coherency domain, this field indirectly determines the memory type used to access the page directory- pointer table referenced by this entry. *GPU does not support any memory type but WB when accessing paging structures.*
2	U/S: User/Supervisor	User vs supervisor access rights. If 0, requests with user-level privilege are not allowed to the 2MByte region controlled by this entry. See a later section for access rights. *(Note: Gen8 (BDW) does not support privilege.)* *Advanced mode only*
1	R/W: Read/Write	Write permission rights. If 0, write permission not granted for requests with user-level privilege (and requests with supervisor-level privilege, if WPE=1 in the relevant extended-context-entry) to the 2MByte region controlled by this entry. See a later section for access rights. *(Note: Gen8 (BDW) does not support privilege.)* *Advanced mode only*
0	P: Present	PD Entry is present. It must be “1” to point to a page table

PDE for 2MB Page

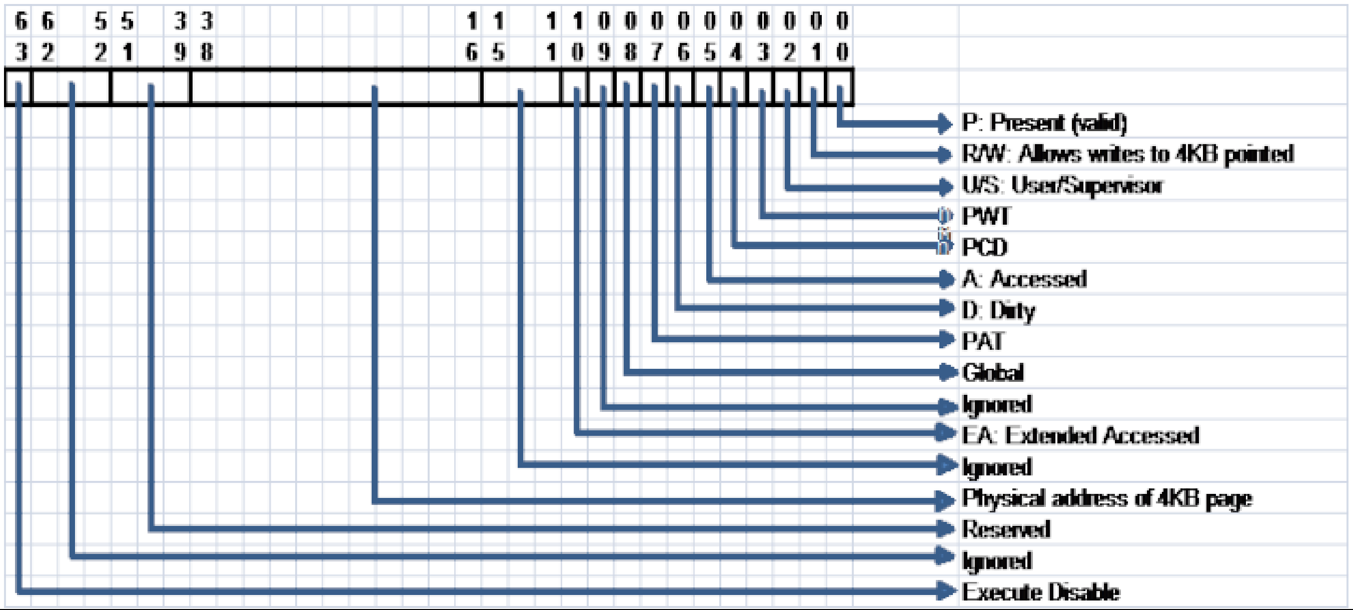


Bits	Field	Description
63	XD: Execute Disable	If NXE=1 in the relevant extended-context-entry, execute permission is not granted for requests to the 2MB page referenced by this entry when XD=1. *(Note: Gen8 (BDW) does not support execute privilege.)* *Advanced mode only*
62:52	*Ignored/Reserved*	*Ignored/not used by hardware*
51:39	*Ignored/Reserved*	*Ignored/not used by hardware*
38:21	ADDR: Address	Physical address of the 2MB page referenced by this entry. This address is the GPA and may need to be further translated with 2nd level translations.
20:13	*Ignored/Reserved*	*Ignored/not used by hardware*
12	PAT: Page Attribute	For devices operating in the processor coherency domain, this field indirectly determines the memory type used to access the 2MB page referenced by this entry. *See later sections for memory type determination.*
11	Ignored/Reserved	Ignored/not used by hardware
10	EA: Extended Access	Extended Access bit is added for devices to separate accesses from IA cores. If EAFE=1 in the relevant PASID-entry, this bit indicates whether this entry has been used for address translation by device. It is the devices responsibility to set this bit. If EAFE=0 in the relevant PASID-entry, this bit is ignored. This bit applies to GPU. *Advanced mode only*
9	*Ignored/Reserved*	*Ignored/not used by hardware*
8	G: Global	Global Page which can be used across contexts and does not need to be invalidated with context/PASID level invalidations. If PGE=1 in the relevant extended-context-entry, this field can be Set by software to indicate the 2MByte page translation is global. *G-bit is not used by GPU*

Bits	Field	Description
7	PS: Page Size	As part of IA32e, there is an option to be able to support 2MB pages. Whether to use the this PDP entry as a pointer to Page Directory vs pointer to a 2MB page is defined as part of the Page Size. i.e. "1" means this entry points to a 2MB page, no further translation is required.
6	D: Dirty	D-bit needs to be managed as the 2MB page being written. Hardware needs to write this bit for the first write access to 2MB defined with PDP entry. See later sections for A/D-bit management. *Advanced mode only*
5	A: Accessed	A-bit needs to be managed as the 2MB page being accessed. Hardware needs to write this bit for the first access to 2MB defined with PDP entry. See later sections for A/D-bit management. *Advanced mode only*
4	PCD: Page level cache disable	For devices operating in the processor coherency domain, this field indirectly determines the memory type used to access the 2MB page referenced by this entry. *See later sections for memory type determination.*
3	PWT: Page level Write-through	For devices operating in the processor coherency domain, this field indirectly determines the memory type used to access the 2MB page referenced by this entry. *See later sections for memory type determination.*
2	U/S: User/Supervisor	User vs supervisor access rights. If 0, requests with user-level privilege are not allowed to the 2MB page referenced by this entry. See a later section for access rights. *(Note: Gen8 (BDW) does not support privilege.)* *Advanced mode only*
1	R/W: Read/Write	Write permission rights. If 0, write permission not granted for requests with user-level privilege (and requests with supervisor-level privilege, if WPE=1 in the relevant extended-context-entry) to the 2MB page referenced by this entry. See a later section for access rights. *(Note: Gen8 (BDW) does not support privilege.)* *Advanced mode only*
0	P: Present	PDP Entry is present. It must be "1" to map to a 2MB page.

PTE: Page Table Entry for 64KB Page

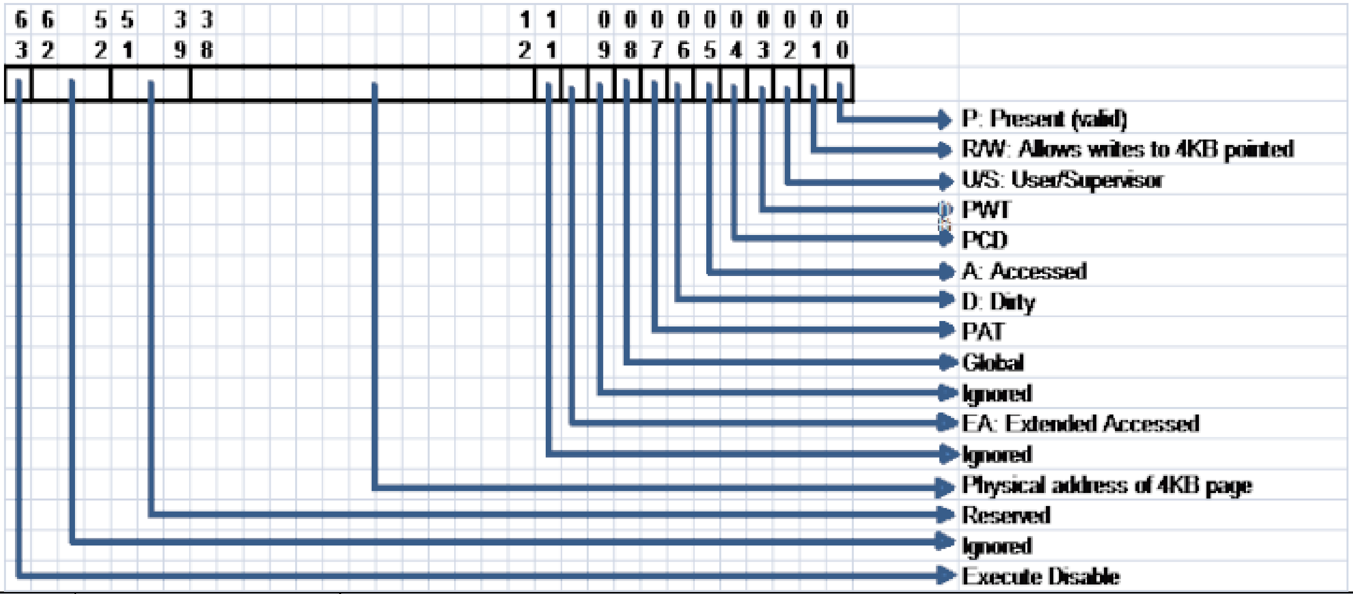
Note:BDW A-step does not support 64KB page formats.



Bits	Field	Description
63	XD: Execute Disable	If NXE=1 in the relevant extended-context-entry, execute permission is not granted for requests to the 64KB page referenced by this entry when XD=1. (**Note:** Gen8 (BDW) does not support execute privilege.) *Advanced mode only.*
62:52	Ignored/Reserved	Ignored/not used by hardware.
51:39	Ignored/Reserved	Ignored/not used by hardware.
38:16	ADDR: Address	Physical address of the 64KB page referenced by this entry. This address is the GPA and may need to be further translated with 2nd level translations.
15:11	Ignored/Reserved	Ignored/not used by hardware.
10	EA: Extended Access	Extended Access bit is added for devices to separate accesses from IA cores. If EAFE=1 in the relevant PASID-entry, this bit indicates whether this entry has been used for address translation by device. It is the device's responsibility to set this bit. If EAFE=0 in the relevant PASID-entry, this bit is ignored. This bit applies to GPU. *Advanced mode only.*
9	Ignored/Reserved	Ignored/not used by hardware.
8	G: Global	Global Page which can be used across contexts and does not need to be invalidated with context/PASID level invalidations. If PGE=1 in the relevant extended-context-entry, this field can be Set by software to indicate that the 64KByte page translation is global. *G-bit is not used by GPU.*
7	PAT: Page Attribute	For devices operating in the processor coherency domain, this field indirectly determines the memory type used to access the 64KB page referenced by this entry. *See later sections for memory type determination.*

Bits	Field	Description
6	D: Dirty	D-bit needs to be managed as the 64KB page being written. Hardware needs to write this bit for the first write access to 64KB defined with PDP entry. See later sections for A/D-bit management. *Advanced mode only.*
5	A: Accessed	A-bit needs to be managed as the 64KB page being accessed. Hardware needs to write this bit for the first access to 64KB defined with PDP entry. See later sections for A/D-bit management. *Advanced mode only.*
4	PCD: Page level Cache Disable	For devices operating in the processor coherency domain, this field indirectly determines the memory type used to access the 64KB page referenced by this entry. *See later sections for memory type determination.*
3	PWT: Page level Write-Through	For devices operating in the processor coherency domain, this field indirectly determines the memory type used to access the 64KB page referenced by this entry. *See later sections for memory type determination.*
2	U/S: User/Supervisor	User vs. supervisor access rights. If 0, requests with user-level privilege are not allowed to the 64KB page referenced by this entry. See a later section for access rights. (**Note:** Gen8 (BDW) does not support privilege.) *Advanced mode only.*
1	R/W: Read/Write	Write permission rights. If 0, write permission is not granted for requests with user-level privilege (and requests with supervisor-level privilege, if WPE=1 in the relevant extended-context-entry) to the 64KB page referenced by this entry. See a later section for access rights.
0	P: Present	PDP Entry is present. It must be 1 to map to a 64KB page.

PTE: Page Table Entry for 4KB Page

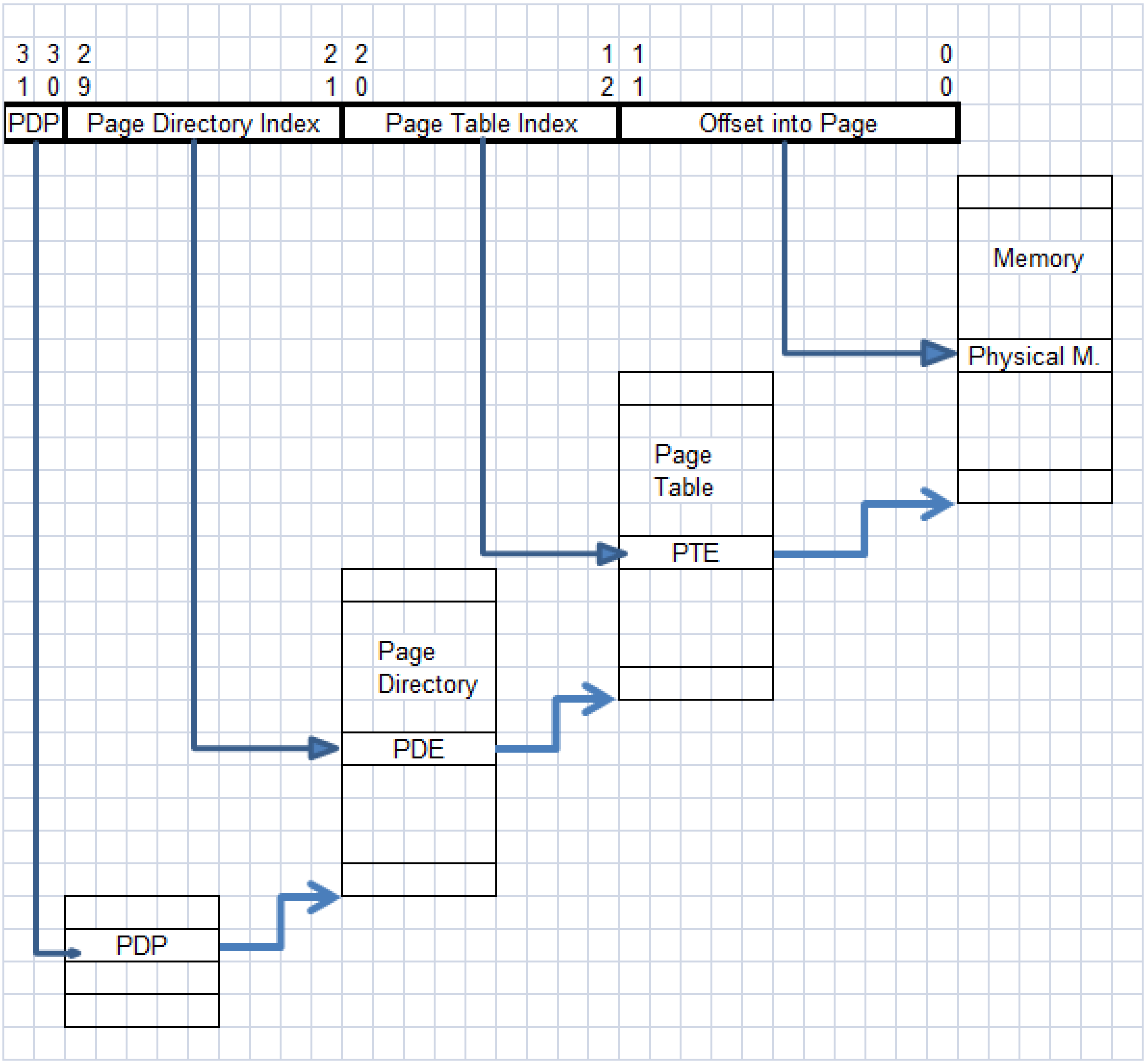


Bits	Field	Description
63	XD: Execute Disable	If NXE=1 in the relevant extended-context-entry, execute permission is not granted for requests to the 4KB page referenced by this entry when XD=1. (**Note:** Gen8 (BDW) does not support execute privilege.) *Advanced mode only.*
62:52	Ignored/Reserved	Ignored/not used by hardware.
51:39	Ignored/Reserved	Ignored/not used by hardware.
38:16	ADDR: Address	Physical address of the 4KB page referenced by this entry. This address is the GPA and may need to be further translated with 2nd level translations.
15:11	Ignored/Reserved	Ignored/not used by hardware.
10	EA: Extended Access	Extended Access bit is added for devices to separate accesses from IA cores. If EAFE=1 in the relevant PASID-entry, this bit indicates whether this entry has been used for address translation by device. It is the device's responsibility to set this bit. If EAFE=0 in the relevant PASID-entry, this bit is ignored. This bit applies to GPU. *Advanced mode only.*
9	Ignored/Reserved	Ignored/not used by hardware.
8	G: Global	Global Page which can be used across contexts and does not need to be invalidated with context/PASID level invalidations. If PGE=1 in the relevant extended-context-entry, this field can be Set by software to indicate that the 4KByte page translation is global. *G-bit is not used by GPU.*
7	PAT: Page Attribute	For devices operating in the processor coherency domain, this field indirectly determines the memory type used to access the 4KB page referenced by this entry. *See later sections for memory type determination.*
6	D: Dirty	D-bit needs to be managed as the 4KB page being written. Hardware needs to write this bit for the first write access to 4KB defined with PDP entry. See later sections for

Bits	Field	Description
		A/D-bit management. *Advanced mode only.*
5	A: Accessed	A-bit needs to be managed as the 4KB page being accessed. Hardware needs to write this bit for the first access to 4KB defined with PDP entry. See later sections for A/D-bit management. *Advanced mode only.*
4	PCD: Page level Cache Disable	For devices operating in the processor coherency domain, this field indirectly determines the memory type used to access the 4KB page referenced by this entry. *See later sections for memory type determination.*
3	PWT: Page level Write-Through	For devices operating in the processor coherency domain, this field indirectly determines the memory type used to access the 4KB page referenced by this entry. *See later sections for memory type determination.*
2	U/S: User/Supervisor	User vs. supervisor access rights. If 0, requests with user-level privilege are not allowed to the 4KB page referenced by this entry. See a later section for access rights. (**Note:** Gen8 (BDW) does not support privilege.) *Advanced mode only.*
1	R/W: Read/Write	Write permission rights. If 0, write permission is not granted for requests with user-level privilege (and requests with supervisor-level privilege, if WPE=1 in the relevant extended-context-entry) to the 64KB page referenced by this entry. See a later section for access rights.
0	P: Present	PDP Entry is present. It must be 1 to map to a 4KB page.

PPGTT for 32b Virtual Address

This page walk mechanism is used for traditional 3D, Media type context. There is going to be a descriptor in the context header which defines the per process GTT walk that is required. For the standard context with 32bit virtual addressing, there is a possibility to take short cuts to reduce the overhead of the walk.



With 32-bit addressing the only entries that are needed for page directory pointers are 4x64bit locations (PDPE). For any standard context scheduling, it is required for SW to provide 4 PDPEs as part of the context which would prevent HW to do additional walks.

Hardware does the remaining walks for PD and PTE similar to legacy behavior. To reduce the overhead of walks, hardware implements large caches for PDs:

- 4x4KB for 3D context
- 2x4x4KB for Media Context
- 4KB for Blitter

For Media and 3D context, the 16KB caches are preloaded for the entire page directory set up which limits the walk to 1-level before the final access. For remaining clients the PD cache is loaded on demand and can contain up to 512 entries.

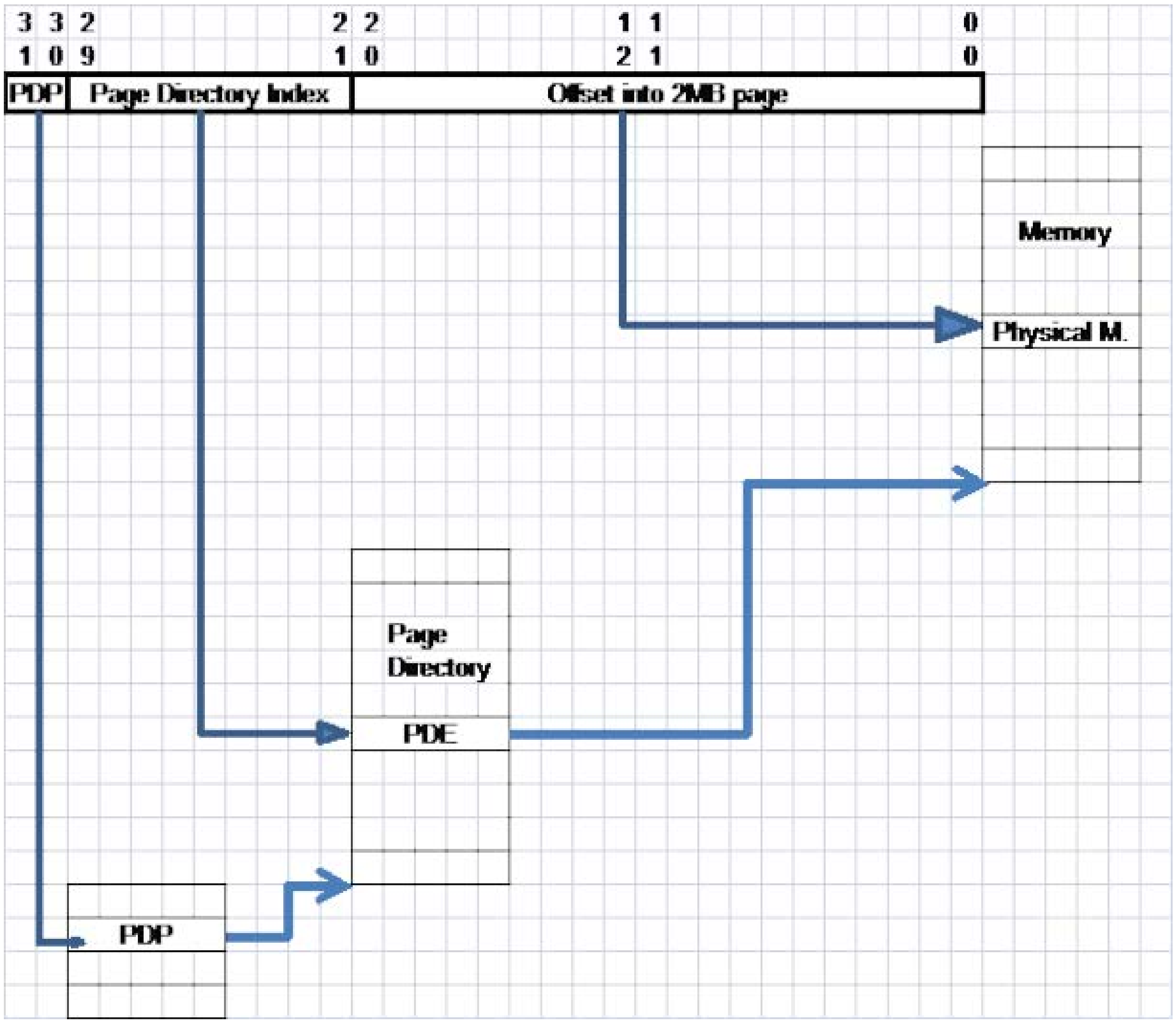
Walk with 64 KB Page

64 KB Page size has a slightly different usage for how PTEs are selected for the corresponding 64KB page. In page table every 16th entry (PTE#0, PTE#16, PTE#32, ... PTE#496) should be used to index. This is calculated using address[21:16] & “0000”. Note that hardware should not make any assumptions for any other PTEs.

Programming Note	
Context:	Walk with 64 KB Page
BDW does not support 64KB page size.	

Walk with 2MB Page

There is an option in the page walk to work with bigger page sizes, one of those sizes is 2MB pages. If allocated the page directory entry indicates the page size and walk can be shortened as following:

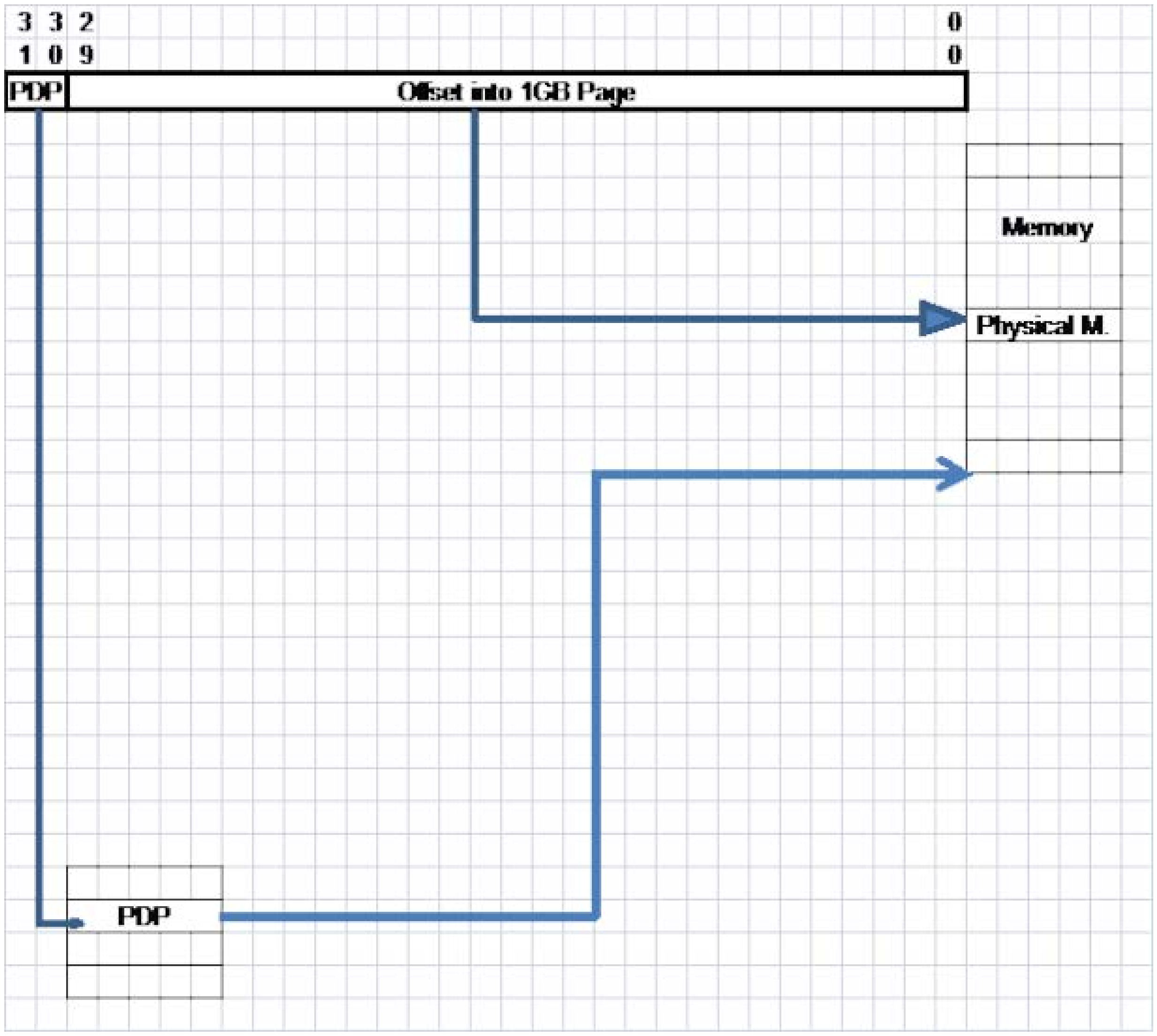


In this case there is no need to walk the page table after directory. And page directory has a pointer to 2MB range is physical memory.

Programming Note	
Context:	Walking the Page Table with 2MBPage
PPGTT32 is not going to support 2MB pages.	

Walk with 1GB Page

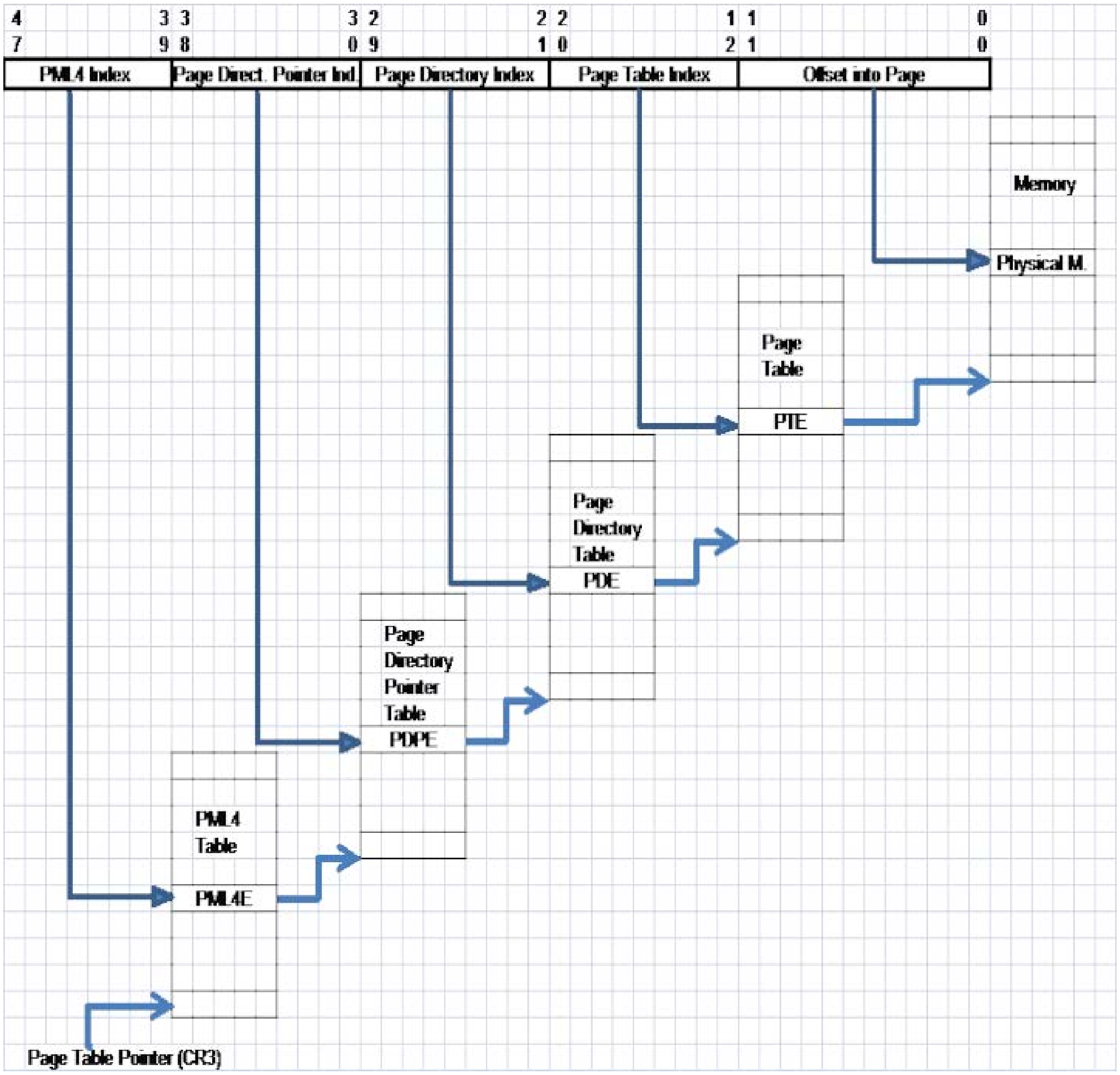
The same page walk is possible with 1GB page support as well.



Programming Note	
Context:	Walking with 1 GB Page
PPGTT32 is not going to 1GB pages.	

PPGTT for Standard Context (64b VA)

For advanced virtual addressing with legacy context, the full page walk mechanism needs to be exercised based on 48bit canonical addressing.



64bit (48b canonical) address requires 4-levels of page table format where the context carries a pointer to highest level page table (PML4 pointer or CR3). The rest of the walk is normal page walk thru various levels.

To repurpose the caches the following mechanism is used:

- 3D: 4KB to store PML4, 4KB as PDP cache, 2x4PD cache.
- Media: 4KB to store PML4, 4KB as PDP cache, 2x4PD cache.

Programming Note	
Context:	PPGTT for standard context (64b VA).
Design can section the 512 entries within 4KB to separate areas for PML4, PDP, and PD.	

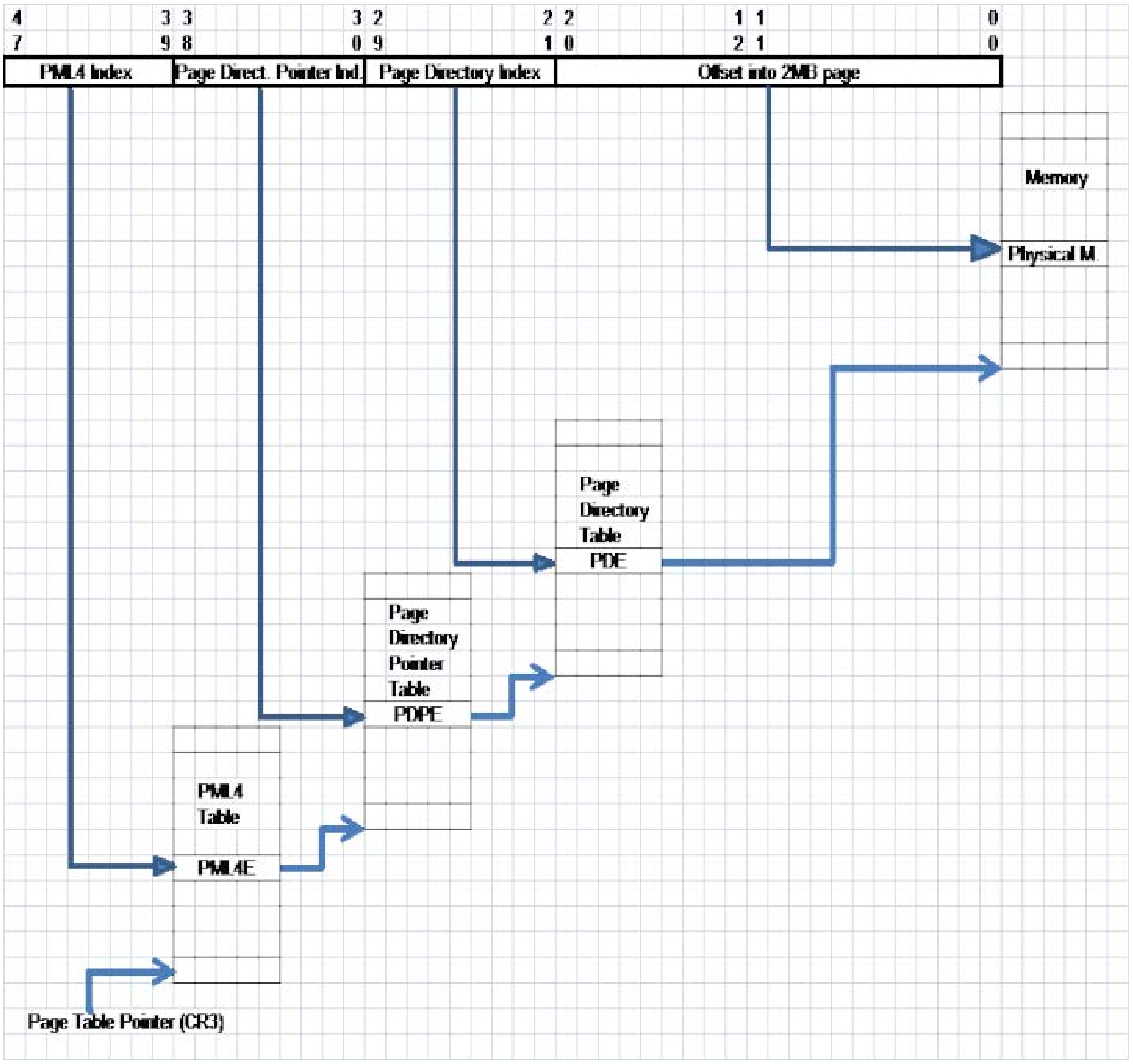
Walk with 64 KB Page

64 KB Page size has a slightly different usage for how PTEs are selected for the corresponding 64KB page. In page table every 16th entry (PTE#0, PTE#16, PTE#32, ... PTE#496) should be used to index. This is calculated using address[21:16] & "0000". Note that hardware should not make any assumptions for any other PTEs.

Programming Note	
Context:	Walk with 64 KB Page
BDW does not support 64KB page size.	

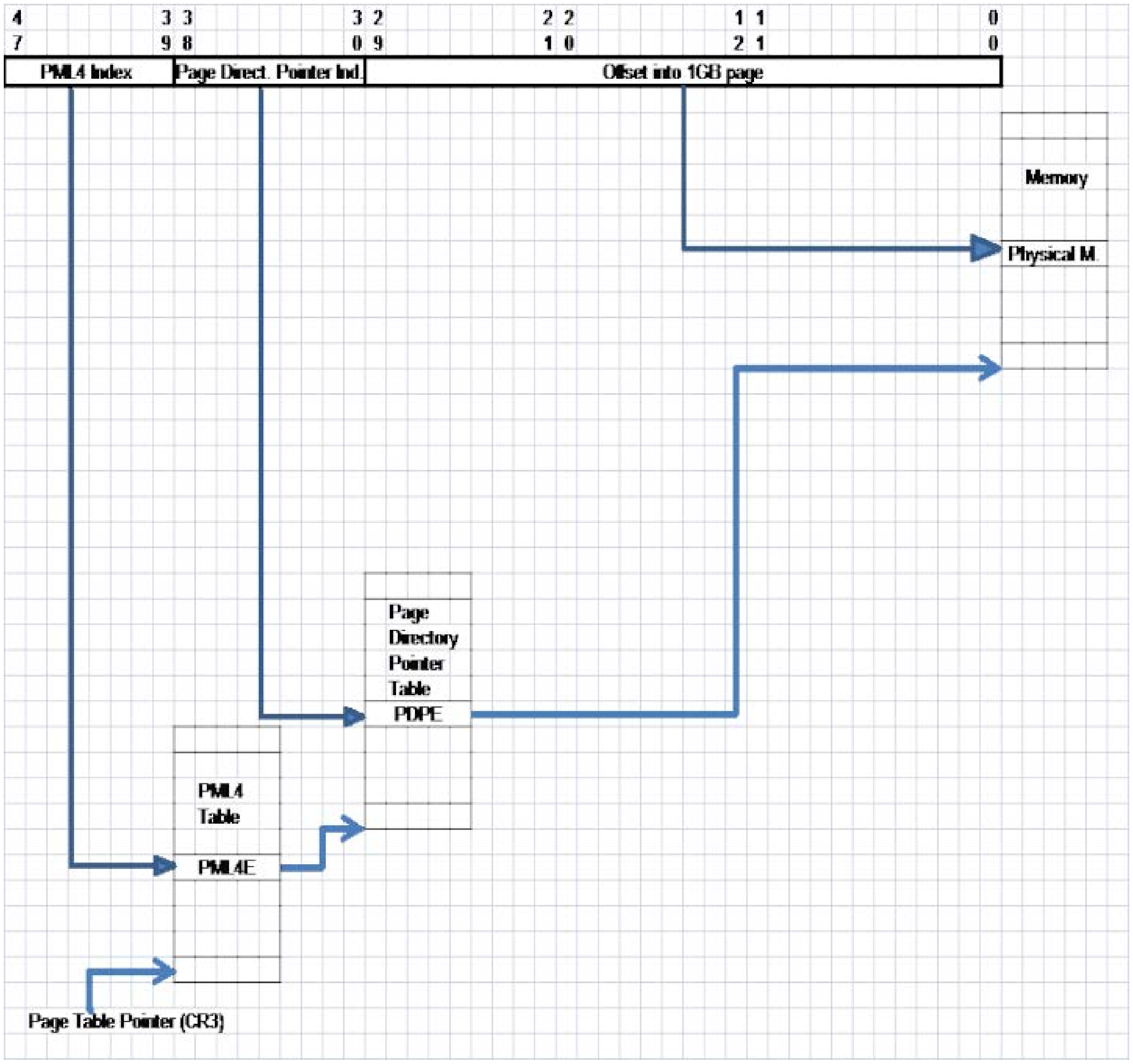
Walk with 2MB Page

Similar concept as the 32b VA walk, there is support for larger pages where one of the sizes supported is 2MB.



Walk with 1GB Page

For the support for 1GB page size, the following mechanism is needed.



Gen8 Global GTT

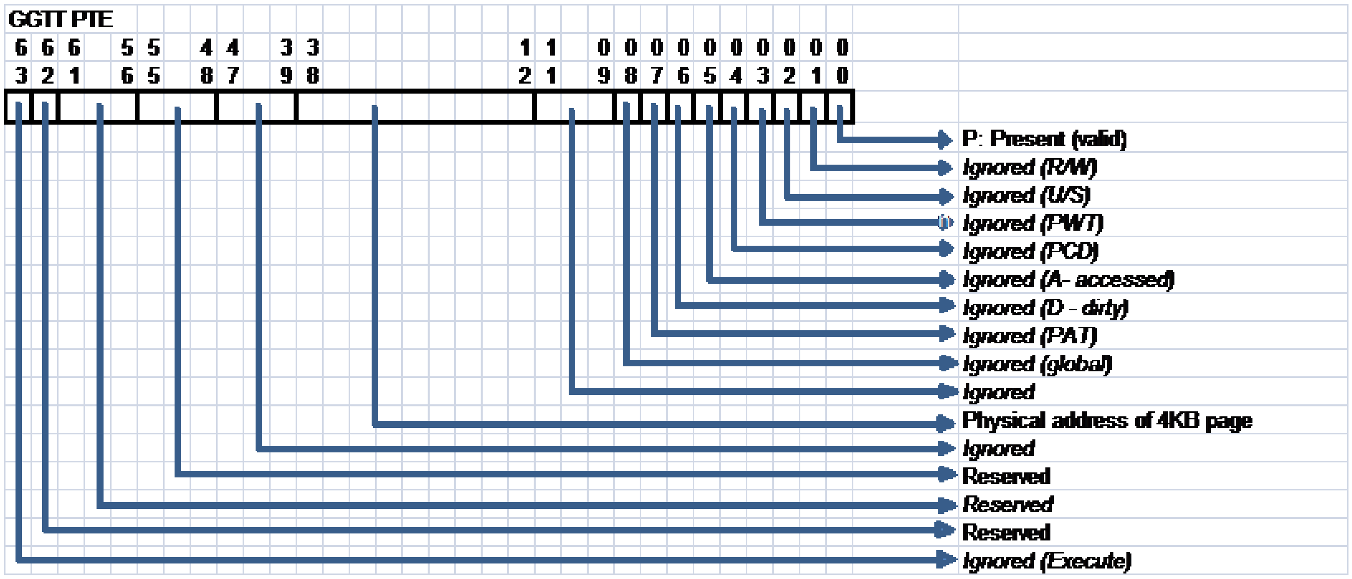
The Global GTT mechanism in gen8 looks very similar to pre-gen8 with the distinction of page table entry. Aperture and display will still use the global GTT even if GT core is mapped via per-process GTT.

The PTE format for Gen8 is updated to match per process GTT definitions and GSM is now expanded in size (2MB=>8MB) to cover for the entire 4GB (32b virtual addressing) space. Each entry corresponding to a 4KB page with 2^20 entries in GSM (each with 8B content)

For "*MI_update_GTT*", the page address provided 31:12 need to be shifted down to 22:3 for the correct QW position within the GGTT.

Page Table Entry

The following page table entry will be used for Global GTT:

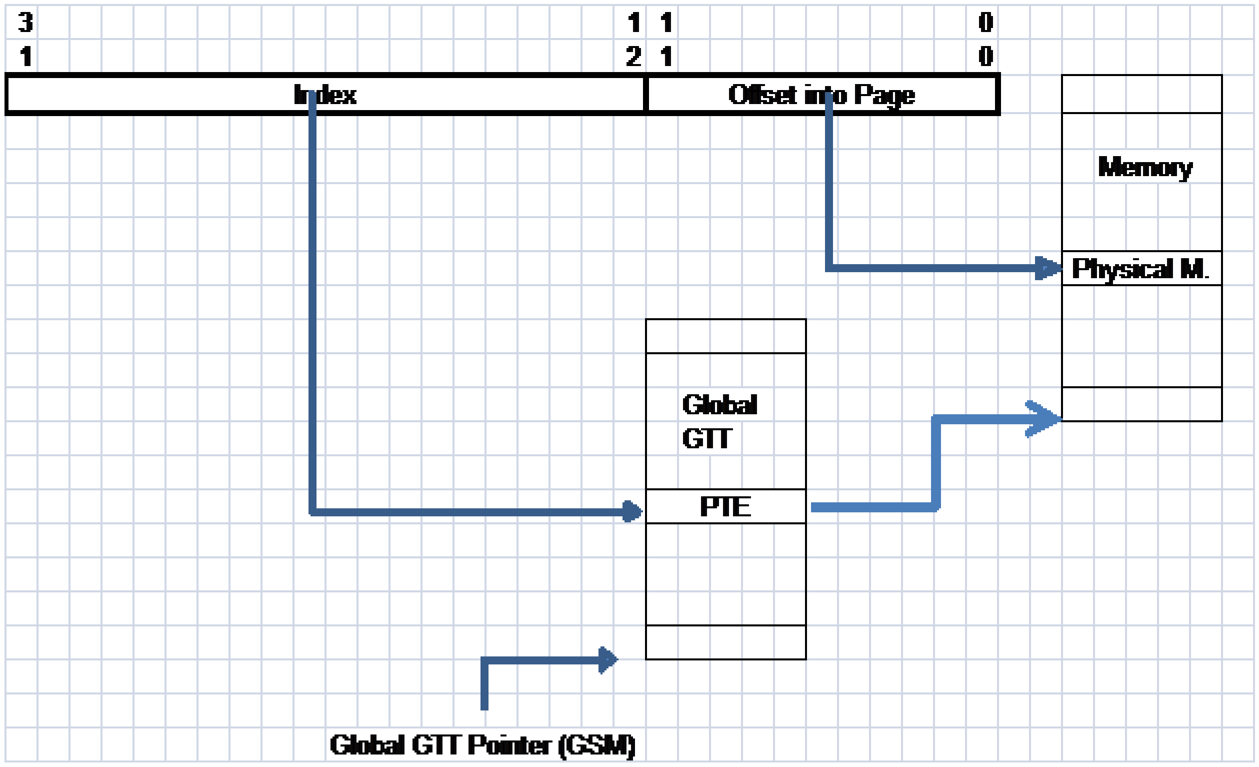


- Present (Valid): The pointed PTE is valid.
- *Ignored* - R/W (Read/Write): Are writes allowed to the region defined by this 4KB page. For GFX, in order 4KB memory to be usable it has to be both present and should also be write-able.
- *Ignored* - U/S (User/Supervisor access rights) : iGFX does not use these fields
- PWT/PCD/PAT bits are used as indexes into a PAT register which defines the cache attributes for the entire context. PAT field is used to do the look up in private PAT for memory typing
- *Ignored* - A (Accessed): It needs to be managed as the page table being accessed. Hardware needs to write this bit for the first access to the 4KB region defined with this PT entry.
- *Ignored* - D (Dirty): Hardware needs to set the dirty bit in page table if accessing this particular 4KB region in memory with the intention to modify it.
- *Ignored* - Global: this is not used by iGFX hardware, the field is used to identify global context where invalidation may not be required.
- Physical address of 4KB page

<u>For the treatment of the page bit0 AND bit1 defines the validity of the page, the rest of the information is not relevant for Aperture and Display usage.</u>

Page Walk

The global GTT page walk is identical to what it was before gen8. The only difference would be that each entry is 8B (instead of 4B) hence the entry selection needs to be updated once the corresponding Page Table miss read is returned.



GFX Page Walker (GAM)

The GPU supports various engines behind the same page walker. These streams/contexts are identified by Client level IDs which are carried via the arbitration pipeline. Page walker using look-up tables does the correct selection for the page tables in case of concurrent contexts running at the same time.

There are two types of page tables. Global graphics translation table (GGTT) is a single common translation table used for all processes. There can also be many Per-process graphics translation tables (PPGTTs). These tables require an additional lookup for translation.

Virtual Memory Structure	Memory Location
Global (GGTT)	GSM Only
Per-Process (PPGTT) – private	2 to 4-level, Page Tables anywhere
Per-Process (IA32e) – shared	4 levels, Page Tables anywhere

Programming Note	
Context:	Gfx Page Walker (GAM)
IA32e compatible PPGTT is added to Gen8 to enable SVM (shared virtual memory) functions.	

Context Definition for GFX Page Walker

Page Walker blocks need details about the context to decide on what type of page tables are used, what would be the error handling cases would be and many other details to operate. The information is passed to Page Walker (GAM) by the respective command streamer/DMA.

GAM supports the following engines:

- Render
- Media (VDBox) x2
- Blit
- WDBox

The following fields are sent to GAM:

- **Context Type** (4 bits) –
    - **Legacy vs Advanced Context.** Defines the context type and qualifies the rest of the fields. Same field may mean something else between the *Legacy* vs *Advanced* context. There is no restriction for what type of context can run in either combination.
        - *Requests without address-space-identifier (Legacy Context)*. These are the normal memory requests from endpoint devices. These requests typically specify the type of access (read/write/atomics), targeted DMA address/size, and identity of the device originating the request.
        - *Requests with address-space-identifier (Advanced Context)*. These are memory requests with additional information identifying the targeted process address space from endpoint devices supporting virtual memory capabilities. Beyond attributes in

normal requests, these requests specify the targeted process address space identifier (PASID), and extended attributes such as Execute-Requested (ER) flag (to indicate reads that are instruction fetches), and Privileged-mode-Requested (PR) flag (to distinguish user versus supervisor access). For details, refer to the Process Address Space ID (PASID) Capability in the PCI-Express specifications.

- **A/D Support Enable.** Access and Dirty bits are used when OS managing the page tables and have been added to IA32e compatible page walk. Context defines whether A/D bits need to be managed via GPU (only applicable in Advanced Context).
- **Privileged Context Support.** Enables GPU to run a privileged context which translates into page table accesses regardless of user vs. supervisor privileges (only applicable in Advanced Context).
    - *BDW (Gen8) GPU does not support privileged Context.*
- **32b vs 48b VA Support.** Enables 48b VA in page tables for the page walks. The rest of the HW is seamless to 32b vs 48b VA address walks, however GAM does the check and properly aligns the page walk to address bits.
    - *Only applicable in Legacy Context, Advanced context is always 48b.*
- **Page Fault Support Model:**
    - *Fault and Hang:* The only supported fault handling mode for legacy context and it does not apply to advanced mode. Optionally hang can be skipped for HW to make progress (same as Gen7.5).
    - *Fault and Stream:* Context can survive thru a number of page faults and could be switched out by the scheduler if a certain threshold is reached.

- **PASID** – Process Address Space IDentifier. Use to identify the context that is submitted to HW. We use the PASID in many places where during the page walk (i.e. PASID table look up) or while communicating with SW on page faults. Each engine could be running an independent context with different PASID. The page walker should have a mechanism to be able to cache at least some number of PASID table entries (matching to the engine count) for faster walk.
- **Context ID** (Queue ID, Bell ID) – Context ID is used to further qualify the running context beyond the PASID. PASID is given per process, and same process may allocate multiple queues to communicate with HW. The only way to further identify the process is to use an additional ID. For GFX HW Context ID could be same as the bell number assigned to it. GAM HW uses the context ID to populate the queue ID field while communicating page faults to SW.
- **Page Table Pointers** – The field could be up to 256 bits (i.e. 4x64bits) to identify the page table pointers associated with the context. For legacy 32b context, the entire 256b is valid representing the 4 PDPTR table entries. For 48b legacy context only the lower 64b is relevant pointing to base of PML4. In case of advanced context, PASID is given in the context definition.

Context Definition Delivery

Element Descriptor Register

General Description	Element Information: The register is populated by command streamer and consumed by GAM.
Register Offset	See per engine list below.

Bits	Access	Default	Field
63:32	RO	Xh	**Context ID.** Context identification number assigned to separate this context from others. Context IDs need to be recycled in such a way that there cannot be two active contexts with the same ID. This is a unique identification number by which a context is identified and referenced.
31:12	RO	Xh	**LRCA.** Command Streamer Only.
11:9	RO	Xh	**Function Number.** GFX device is considered to be on Bus0 with device number of 2. Function number is normally assigned as "0". Not used in Gen8.
8	RO	Xh	**Privileged Context/GGTT vs PPGTT mode.** Differs in legacy vs advanced context modes: **In Legacy Context:** Defines the page tables used. This is how page walker comes to know PPGTT vs GGTT selection for the entire context. 0: Use Global GTT. 1: Use Per-Process GTT. **In Advanced Context:** Defines the privilege level for the context. 0: User mode context. 1: Supervisor mode context.
7:6	RO	Xh	**Fault Model.** 00b: Fault & Hang. Same mode as Gen7.5. 01b: reserved. 10b: Fault & Stream & Switch. 11b: reserved.
5	RO	Xh	**Deeper IA coherency Support.** **In Advanced Context.** Defines the level of IA coherency. 0: IA coherency is provided at LLC level for all streams of GPU (i.e. Gen7.5 like mode). 1: IA coherency is provided at L3 level for EU data accesses of GPU.
4	RO	Xh	**A&D Support/ 32&64b Address Support.** Differs in legacy vs advanced context modes. **In Legacy Context:** Defines 32b vs 64b (48b canonical) addressing format. 0: 32b addressing format. 1: 64b (48b canonical) addressing format. **In Advanced Context:** Defines A&D bit support. 0: A&D bit management in page tables is NOT supported. 1: A&D bit management in page tables is supported.

Bits	Access	Default	Field
3	RO	Xh	**Context Type: Legacy vs Advanced.** Defines the context type. 0: Advanced Context: Defines the rest of the advanced capabilities (i.e. OS page table support, fault models, ...). Note that advanced context is not bounded to GPGPU. 1: Legacy Context: Defines the context as legacy mode which is similar to prior generations of Gen8. **Note:** Bits [8:4] differ in functions when legacy vs advanced context modes are selected.
2	RO	Xh	**FR.** Command streamer specific.
1	RO	Xh	**Scheduling Mode.** 1: Execution List mode of scheduling. 0: Ring Buffer mode of scheduling.
0	RO	Xh	**Valid.** Indicates that element descriptor is valid. If GAM is programmed with an invalid descriptor, it continues but flags an error.

PDP0/PML4/PASID Descriptor Register

General Description	PDP0/PML4/PASID: The register is populated by command streamer and consumed by GAM. It contains one of the 3 values which is determined by looking at the element descriptor.
Register Offset	See per engine list below

Bits	Access	Default	Field
63:0	RO	Xh	**PDP0/PML4/PASID:** This register can contain three values which depend on the element descriptor definition. **PASID[19:0]:** Populated in the first 20bits of the register and selected when Advanced Context flag is set. **PML4[38:12]:** Pointer to base address of PML4 and selected when Legacy Context flag is set and 64b address support is selected **PDP0[38:12]:** Pointer to one of the four page directory pointer (lowest) and defines the first 0-1GB of memory mapping *Note: This is a guest physical address* (unused bits need to be populated as 0's)

PDP1 Descriptor Register

General Description	PDP1: The register is populated by command streamer and consumed by GAM. It contains one of the pointers to PD.
Register Offset	See per engine list below

Bits	Access	Default	Field
63:12	RO	Xh	**PDP1:** Pointer to one of the four page directory pointer (lowest+1) and defines the first 1-2GB of memory mapping *Note: This is a guest physical address* (unused bits need to be populated as 0's)

PDP2 Descriptor Register

General Description	PDP2: The register is populated by command streamer and consumed by GAM. It contains one of the pointers to PD.
Register Offset	See per engine list below

Bits	Access	Default	Field
63:12	RO	Xh	**PDP2:** Pointer to one of the four page directory pointer (lowest+2) and defines the first 2-3GB of memory mapping *Note: This is a guest physical address* (unused bits need to be populated as 0's)

PDP3 Descriptor Register

General Description	PDP3: The register is populated by command streamer and consumed by GAM. It contains one of the pointers to PD.
Register Offset	See per engine list below

Bits	Access	Default	Field
63:12	RO	Xh	**PDP3:** Pointer to one of the four page directory pointer (lowest+3) and defines the first 3-4GB of memory mapping *Note: This is a guest physical address* (unused bits need to be populated as 0's)

List of Registers and Command Streamers

The following registers are message registers and not written directly by SW.

Engine	Offset	Description
Render	x4400h	Element Descriptor Register
	x4408h	PDP0/PML4/PASID Descriptor Register
	x4410h	PDP1 Descriptor Register
	x4418h	PDP2 Descriptor Register
	x4420h	PDP3 Descriptor Register
Media0 (VDBOX0)	x4440h	Element Descriptor Register
	x4448h	PDP0/PML4/PASID Descriptor Register
	x4450h	PDP1 Descriptor Register
	x4458h	PDP2 Descriptor Register
	x4460h	PDP3 Descriptor Register
Blitter	x4500h	Element Descriptor Register
	x4508h	PDP0/PML4/PASID Descriptor Register
	x4510h	PDP1 Descriptor Register
	x4518h	PDP2 Descriptor Register
	x4520h	PDP3 Descriptor Register

GTT Cache

Processor graphics page walker implements a GTT cache which holds the remaining entries that are read as a cacheline but not used for the immediate page walk. This is only applicable in case of leaf walks and not including the 2MB/1GB page sizes. When s/w enables the use of 2MB/1GB page sizes, it will have to disable the GTT cache in Gen8.

Legacy Context

Legacy context could use either Global GTT or Per Process GTT which is given to page walker as part of the context descriptor. Even under PPGTT, there could be accesses from Command Streamers that would require to use Global GTT which requires to treat the walk requirement per transaction.

For Legacy context indicator command streamer is going to pass the “context type” information along with other parameters that define how certain behavior for paging needs to be.

Advanced Context

In the new context type, the Global GTT walk is same as legacy case however Per Process GTT walk starts earlier in the tables. This is where both the 1st level and 2nd level walks share the some portions of the page walk scheme.

Full Walk

In the full walk case (i.e. advanced context), the root of the 1st and 2nd level page tables share a common source. Both the root table and context table are walked with the assumption of GFX device is always on Bus#0 and it is always Device#2.

Function number however is part of the context GAM receives the function number (for Context table look-up) as part of the context. Both Root entry and Context Entry should be fetched along with PASID Table entry prior to running the context accesses.

Programming Note	
Context:	Full Walk Case in Complete Page Walk; Virtual Memory
Gen8 (Broadwell) GFX does not support different function numbers for GFX device.	

Root Table Entry

The root-entry functions as the top level structure to map devices on a specific PCI bus to their respective domains. Each root-entry structure contains the following fields:

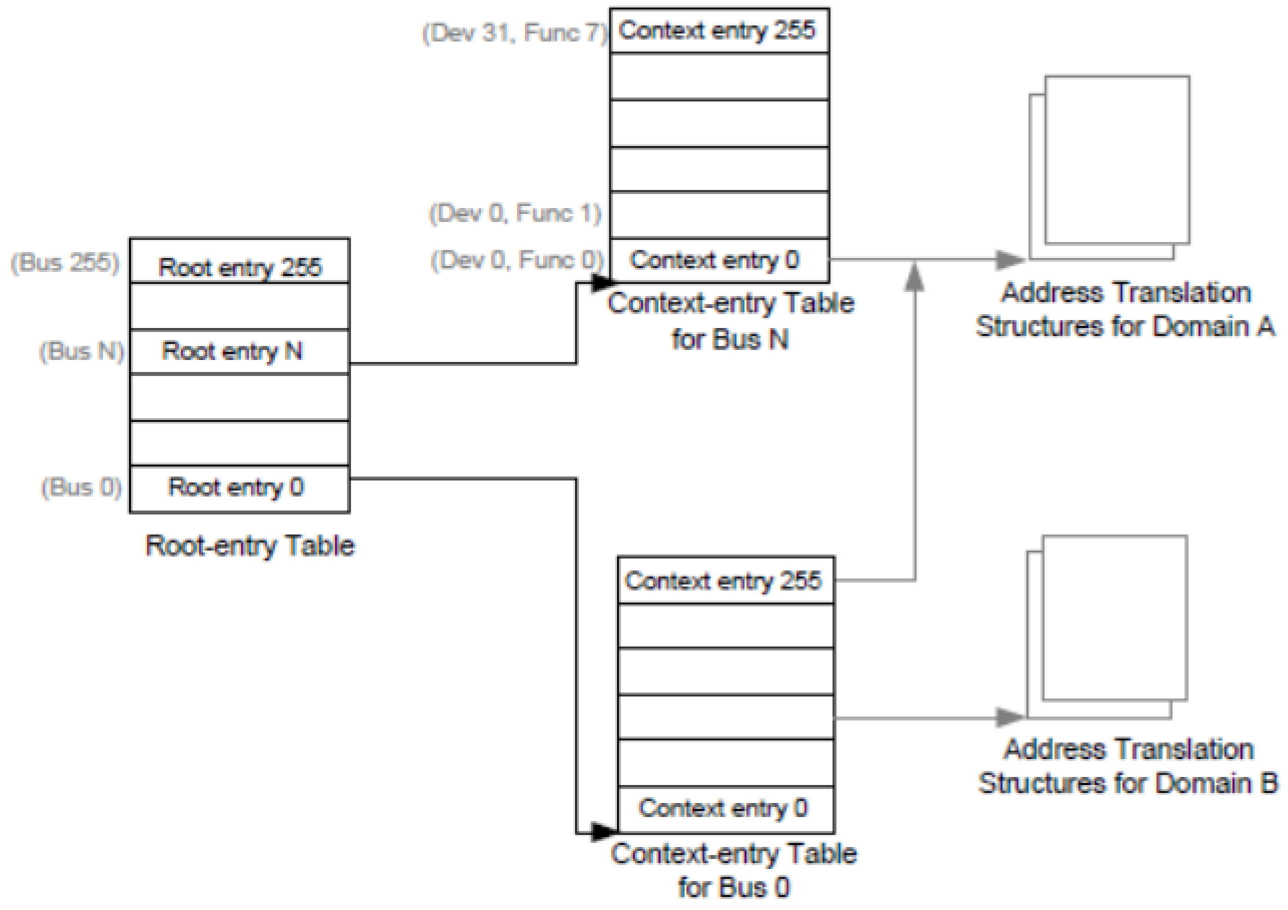
- **Present flag(s).** The present field is used by software to indicate to hardware whether the root entry is present and initialized. Software may clear the present field for root entries corresponding to bus numbers that are either not present in the platform, or don't have any downstream devices attached. If the present field of a root-entry used to process a DMA request is clear, the DMA request is blocked, resulting in a translation fault.
- **Context-table pointer.** The context-table pointer references the context-table for devices on the bus identified by the root-entry.

For implementations supporting Extended-Context-Support (ECS=1 in Extended Capability Register), the Root Table Address Register points to an extended-root-table when Root-Table-Type field in the Register is set (RTT=1). GT does support extended context as a capability.

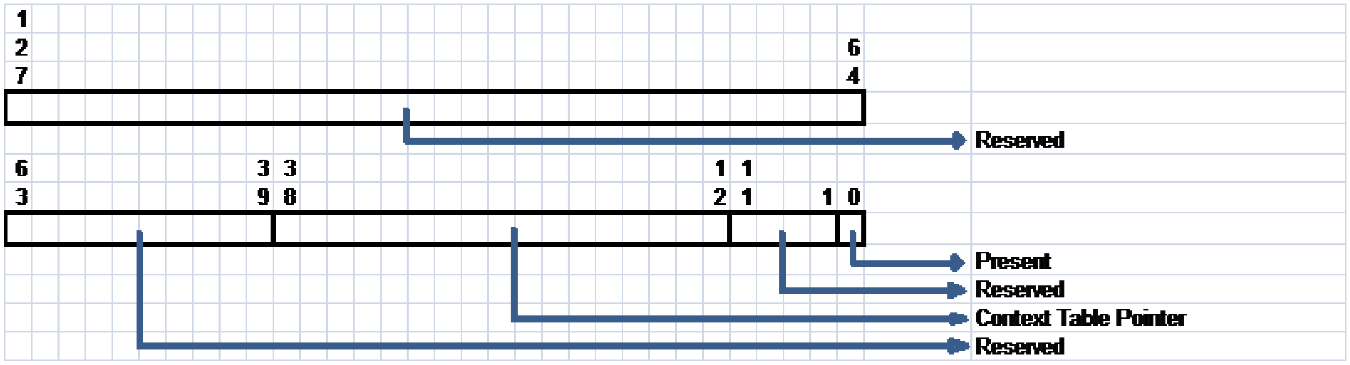
The extended-root-table is similar to the root-table (4KB in size and containing 256 extended-root-entries to cover the 0-255 PCI bus number space), but has an extended format to reference extended-context-tables.

- **Lower Present flag.** The lower-present field indicates the lower 64-bits of the extended-root-entry is present and the lower-context-table pointer (LCTP) field is initialized. Software may clear the lower-present field for extended-root-entries corresponding to bus numbers that are either not present in the platform, or don't have downstream devices with device numbers 0-15 attached. DMA requests processed through the lower part of an extended-root-entry with the lower-present field clear result in translation-fault.
- **(Lower) Context-entry table pointer.** The context-entry table pointer references the context-entry table for devices on the bus identified by the root-entry. For Legacy Context-Entry, this pointer maintains a context entry table with 256 entries and for extended context-entry, this pointer maintains a context entry table with 128 entries which correspond to lower part of the 256 context entries.
- **Upper Present flag.** The upper-present field indicates the upper 64-bits of the extended-root-entry is present and the upper-context-table pointer (UCTP) field is initialized. Software may clear the upper-present field for extended-root-entries corresponding to bus numbers that are either not present in the platform, or don't have downstream devices with device numbers 16-31 attached. DMA requests processed through the upper part of an extended-root-entry with the upper-present field clear result in translation-fault.
- **(Upper) Context-entry table pointer.** Only applicable for extended context entry support and maintains the context entry table with 128 entries which correspond to higher part of the 256 context entries.

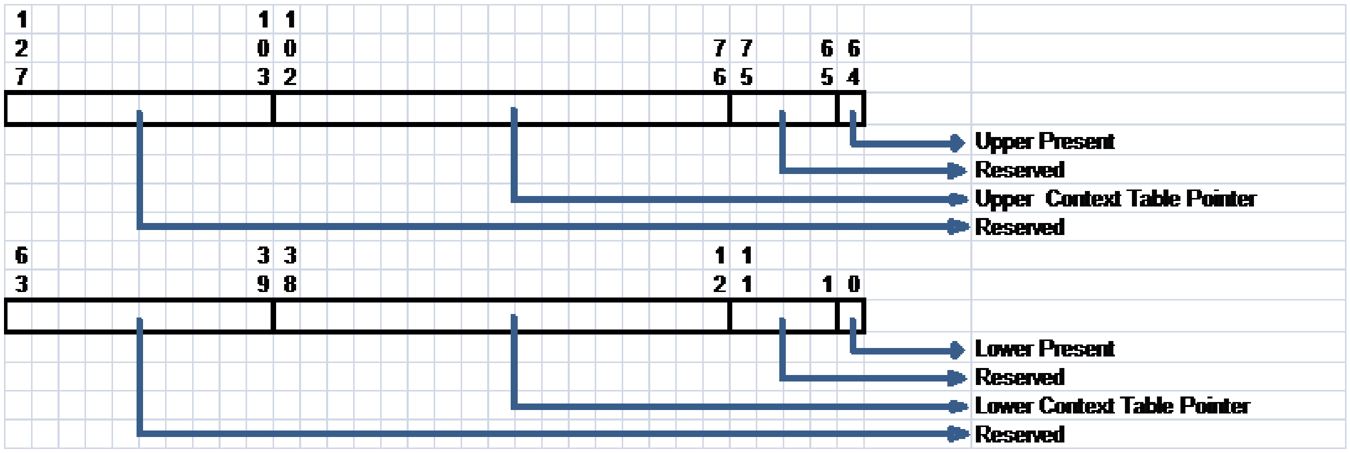
The root entries are programmed through the root-entry table. The location of the root-entry table in system memory is programmed through the Root-entry Table Address register. The root-entry table is 4KB in size and accommodates 256 root entries to cover the PCI bus number space (0-255). In the case of a PCI device, the bus number (upper 8-bits) encoded in a DMA transaction's source-id field is used to index into the root-entry structure.



Legacy Root Table Entry



Extended Root Table Entry



Note that extended root table entry has two “present” flags, one for lower context table pointer and one for upper context table pointer. This exception is added to ease the SW burden where system does not need to support devices 16-31 where most system do not have it.

PASID State Table

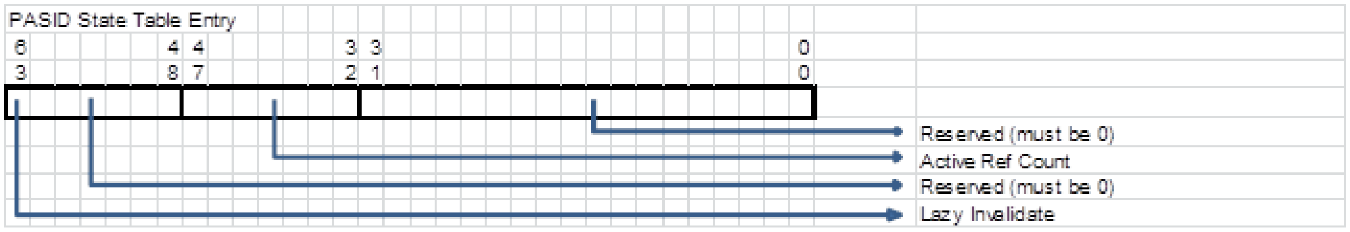
PASID state table is defined to indicate the usage of PASID in the hardware context. This contains information for PASIDs that are actively used as well as cached even if the PASID is not active. The later clause is not applicable to gen GFX given we only cache active PASID's page table entries and invalidate the page tables caches (TLBs) during context switch.

In the faultable (non-pinned) page allocation model, OS is required to dynamically move pages between applications, hence it is required to support TLB shootdowns to ensure a page is de-allocated from an application space that may be actively used in hardware. The process will required to involve IOMMU and messaging to hardware which has a large overhead.

Hence OS tends to prefers lazy-TLB shootdowns for performance. That is achieved by OS looking at the hardware status and

- If target PASID is not active in TLBs, skip the shootdown since the page table entries are not in the hardware anyways.
- If target PASID is not active but cached in TLBs, mark it as for deferred invalidation before the next time same PASID is activated. This is not applicable for GFX.

In order to enable Lazy-TLB shootdowns, we need a way in GFX to inform whether a PASID is actively used in the hardware, this is where the PASID state table is used.



- **Lazy-Invalidate:** AREFCNT is the 16 bit values (47:32) that defines the number when that particular PASID is used within the GFX domain. As long as AREFCNT is non-zero, s/w (OS) is required to send TLB shootdowns to the h/w. If the AREFCNT is zero, than s/w is allowed to deferred invalidation where no explicit TLB shootdowns are sent to h/w. However h/w may be caching GTT entries from a passive PASID, hence s/w is required to show the intention of deferred invalidation via setting the DINV field within the same PASID entry. Hardware detecting a non-zero DINV field while incrementing the AREFCNT should invalidate its TLBs.
  - GFX hardware does not cache GTT entries from non-active PASID (*i.e. all TLBs gets invalidated when an active PASID exits hardware execution*) hence DINV bit has no impact in the hardware, however hardware shall clear the DINV field with the increment of AREFCNT to follow the OS intention of supporting lazy invalidation.
- **Active Ref Count:** This is the main field for GFX to manage. The maintenance has to be done via atomic INC/DEC events from GAM to GTI on a 16bit level (for details of atomic events refer next section). The active Ref Count has to be updated once the initial context descriptor is delivered from the command streamer to the GAM and as part of the set up process (before bringing anything into TLBs or intermediate page table caches). GAM also needs to get the feedback (it has to be a read) as the atomic process complete before any further processing is done with

that PASID. Subsequent context descriptors should not update the active field in the PASID State Table. Hence command streamer and GAM needs to distinguish initial context descriptors from mid-stage context descriptor updates.

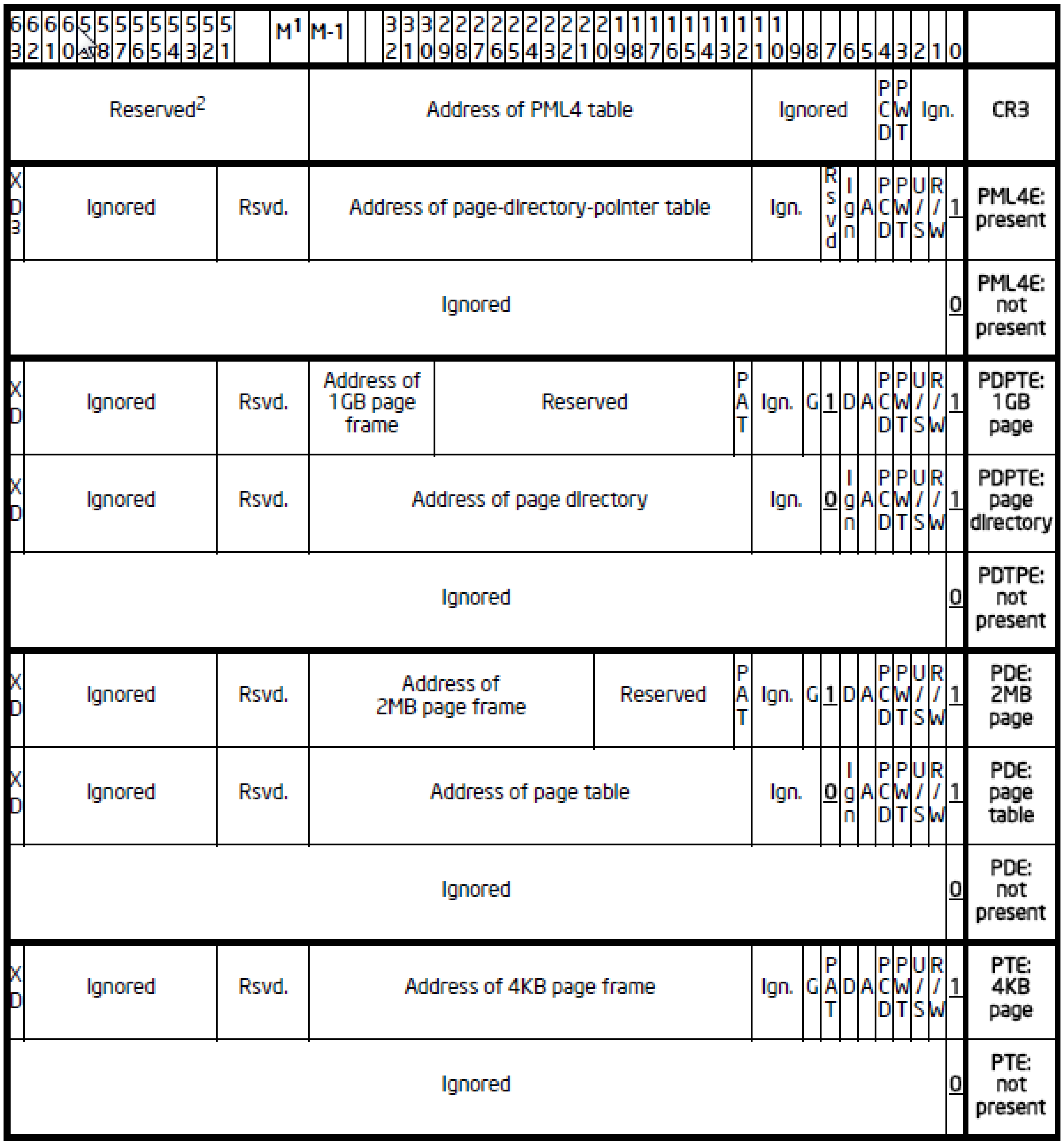
At the end of the context (when PASID is about to be retired), GAM needs to be notified hence the active ref count can be decremented.

Same decrement event has to be applied to device resets as well as FLR (device reset). For engine resets if the corresponding engine is running advanced context (PASID), the corresponding PASID state counter needs to be decremented. If device reset (FLR) is used, for all engines that are running advanced context, the PASID state counter(s) need to be decremented which could be many.

The reason an atomic increment/decrement is used, multiple engines within GT could be using the same PASID independently.

TLB Caching and Management

As compared to previous generation of TLB entry, IA32e page translation entry is quite different. At every stage of the page different bits need to be taken into account and proper treatment is required. Regardless of PPGTT vs GGTT usage, the paging entry has the same format. Linear addresses are translated using a hierarchy of in-memory paging structures located using the contents of CR3. IA-32e paging translates 48-bit linear addresses to 52-bit physical addresses. Although 52 bits corresponds to 4 PBytes, linear addresses are limited to 48 bits; at most 256 TBytes of linear-address space may be accessed at any given time. IA-32e paging may map linear addresses to 4-KByte pages, 2-MByte pages, or 1-GByte pages.



63	62–59	58–52	51 ... M[1]	M-1 ... 32–13	12	11–9	8	7	6	5	4	3	2	1	0	
Reserved[2]				Address of PML4 table		Ignored					PCD	PWT	Ign.			CR3
XD[3]	Ignored	Rsvd.		Address of page-directory-pointer table		Ign.		Rsvd	Ign	A	PCD	PWT	U/S	R/W	1	PML4E: present
Ignored															0	PML4E: not present
XD	Ignored	Rsvd.	Address of 1GB page frame	Reserved	PAT	Ign.	G	1	D	A	PCD	PWT	U/S	R/W	1	PDPTE: 1GB page
XD	Ignored	Rsvd.		Address of page directory		Ign.		0	Ign	A	PCD	PWT	U/S	R/W	1	PDPTE: page directory
Ignored															0	PDTPE: not present
XD	Ignored	Rsvd.	Address of 2MB page frame	Reserved	PAT	Ign.	G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 2MB page
XD	Ignored	Rsvd.		Address of page table		Ign.		0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
Ignored															0	PDE: not present
XD	Ignored	Rsvd.		Address of 4KB page frame		Ign.	G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored															0	PTE: not present

The following rules apply:

1. M is an abbreviation for MAXPHYSICAL ADDRESS.
2. Reserved fields must be "0".
3. Ignored fields must be ignored (there could be private information).
4. All ignore options are part of the context entry and coming from IOMMU definition.

TLB Caches

For Gen8 the caching structures are separated as following with the architectural view, this is also applicable to SW view of these caches when it comes to invalidations.

Context Cache - CC

This is the storage for context table entry which is achieved as part of root/context table walk.

Context cache can also be invalidated with directed invalidations, where HW needs to invalidate the content of the context cache along with all low level caches.

PASID Cache - PC

This is where the HW copy of the PASID table entry is kept and it is per context. This makes it unique for every HW engine that could be running an independent context (per GAM):

- Render/GPGPU
- MFX (VDBOX) – 1
- MFX (VDBOX) – 2
- Blitter

The cache content is updated if the corresponding engine is running an advanced context where its page table pointers are accessible via PASID table. In case of legacy context running engine, corresponding PASID Cache entry is not valid. Recommendation is to keep ONE physical storage per engine which is filled/invalidated during the context switch time.

PASID Cache can also be invalidated with the directed invalidations along with low level caches and needs to be re-filled prior to context resuming.

Intermediate Page Walk Caches (PML4, PDP, PD) – PWC

These are the stages where intermediate page walk entries are cached to speed-up/shorten the page walk when final TLB is missed. Each level can be cached separately or along with different levels, the cacheability structures will have programmability to move the boundary of different levels to accommodate more/less on each page walk level. However as a concept, for legacy 32b addressing mode, requirement is to cache 4PDPs along with 4x4KB PDs for certain engines, at least for render and media. The others will use cache concept.

TLB Entry Content

When a page walk entry is cached (or loaded prior to context start), certain bits need to be cached as well along with the physical address bits. The treatment on these bits would be considered when a HIT vs MISS decision needs to be made during a look up.

The purpose of caching is to accelerate the paging process by caching individual translations in *translation look-aside buffers (TLBs)*. Each entry in a TLB is an individual translation. Each translation is referenced by a page number. It contains the following information from the paging-structure entries used to translate linear addresses with the page number:

- The physical address corresponding to the page number (the page frame).
- The access rights from the paging-structure entries used to translate linear addresses with the page number:
    - The logical-AND of the R/W flags.
    - The logical-AND of the U/S flags.
    - The logical-OR of the XD flags.
- Attributes from a paging-structure entry that identifies the final page frame for the page number (either a PTE or a paging-structure entry in which the PS flag is 1):
    - The dirty flag.
    - The memory type.

PRESENT. This is the same VALID bit description as in previous page table designs. The lack of present bit (i.e. bit[0]=0) points that rest of the information in the page table entry is being invalid. For some fault models, even NOT PRESENT entries are cached to filter further page faults (see *fault models on caching page faulting entries*). If such an entry is cached, there are ways that it can be removed from the page tables:

1. LRA selection where the entry becomes a victim for replacement.
2. Global or Selective invalidation.
3. Page fault response stating the faulting page is now fixed.

R/W Privilege. Certain pages can be allocated as read-only and write operations are not allowed. To make this check work, TLB has to keep the R/W bit. This bit has no affect on read operations; however for write operations privilege needs to be checked. If there is a mismatch, the result of the TLB lookup should be a MISS. This does not mean a page fault immediately; the walk has to be redone as for any

TLB MISS result. There are cases where the OS may change page table privileges without invalidating pages in TLB. (**Note:** All downgrades result in invalidation of the TLB, however upgrades can be done silently hence re-walk is required.) In case where the TLB Miss is due to privilege mis-match, the existing entry from TLB has to be invalidated and page walk brings in the most up-to-date copy from memory.

The R/W privilege on final frame is generated as a logical-AND process of all upper page walks pointing to this location.

User vs Supervisor Privilege. The GPU typically operates in user mode when it comes to page tables. So the GTT walk can be treated as faulted when GPU encounters a page with supervisor privileges and the context is marked as user mode. The faulted entry can be cached back into TLB with "P" bit off indicating a faulted entry. However the page fault report should carry the correct reason why HW detected the fault in the first place which was the user vs supervisor privilege. There is an option in context header to define the context as supervisor, than it legal to access supervisor pages.

- This is not stored in TLB.

The U/S privilege on final frame is generated as a logical-AND process of all upper page walks pointing to this location.

Accessed Bit. This where a stage of the page walk cannot be used if the accessed bit is not set for that level in the page walk. This is true for both storage into TLB as well as to make progress on the page walk. To achieve the process of Accessed bit, every stage of the ppGTT read is done via a new semantics between the GAM and GTI such that GTI can atomically process A-bit without running into access violations. The details of the semantics are defined as part of the following sections. The "A" bit does not need to be stored as part of the TLB, just the fact that a valid page table entry is present in the TLB does mean that HW took care of the "A" bit at the time the page was brought up to TLB. Note that TLB prefetching is disabled when A-bit management is enabled.

IA32e mode page tables cannot coexist with TLB pre-fetching due to lack of A-bit management for all entries of the line.

- This is not stored in TLB.

Dirty Bit. Similar to accessed bit, dirty bit needs to be managed. It only applies for "write" accesses. Given that there are cases where a TLB entry was acquired as part of a read operation, the presence of D-bit should be maintained with the TLB. This gives us the capability to declare a TLB miss for a write access when the D-bit is not set even though TLB has a valid translation. In such case, the TLB entry needs to invalidated and the final stage of the walk needs to be redone to ensure the most up-to-date copy of the GTT entry is brought into HW. The operation of Dirty bit update is also atomic similar to A-bit management.

Execute (XD) Bit. XD bit is also present on every stage of the walk and applicable to executable code that GT would be fetching. In the first pass, instruction cache accesses are not allowed to proceed if the corresponding page does not have the execute credentials set properly. Similar treatment of the TLB entry as privilege bits is expected. A page entry that was already cached in TLB and later accessed for instruction space has to check the XD bit which is also stored in TLB. If mis-match, the end result is a TLB miss and walk has to be redone replacing the different stages of the walk.

The XD privilege on final frame is generated as a logical-OR process of all upper page walks pointing to this location.

Faulted Bit. There are usage models where the faulted entries are cached in TLB. This is to filter further faults to the same page as an opportunistic way to prevent fault storms. When faulted bit is set the address is included in the TLB lookup but final treatment is fault filtering. The rest of the bits are used to define what would be the reason for the fault. If the look-up conflicts with the original faulted reason, a re-walk is required. As a basic case, take a read access bringing up a PTE with W-flag cleared. A subsequent write access has a conflict on privilege, and it performs a re-walk. If the result of the re-walk is W-flag set, then TLB is upgraded and write makes progress. However if the result is still W-flag cleared, the write access faults and TLB entry is tagged as a faulted entry with only read-allowed. Subsequent write accesses are filtered as fault but read accesses should cause a re-walk of the page and if successful, the TLB can be updated with PTE as valid with read-only attribute.

TLB Accessed and Dirty Flags

For any paging-structure entry that is used during linear-address translation, bit 5 is the *accessed* flag. For paging-structure entries that map a page (as opposed to referencing another paging structure), bit 6 is the *dirty* flag. These flags are provided for use by memory-management software to manage the transfer of pages and paging structures into and out of physical memory.

Whenever the processor and/or GPU uses a paging-structure entry as part of linear-address translation, it sets the accessed flag in that entry (if it is not already set).

Whenever there is a write to a linear address, the processor and/or GPU sets the dirty flag (if it is not already set) in the paging-structure entry that identifies the final physical address for the linear address (either a PTE or a paging-structure entry in which the PS flag is 1).

Memory-management software may clear these flags when a page or a paging structure is initially loaded into physical memory. These flags are “sticky,” meaning that, once set, the processor and/or GPU does not clear them; only software can clear them.

A processor and/or GPU may cache information from the paging-structure entries in TLBs and paging-structure caches (see Section 4.10). This fact implies that, if software changes an accessed flag or a dirty flag from 1 to 0, the GPU might not set the corresponding bit in memory on a subsequent access using an affected linear address.

The accessed bit applies to every stage of the page walk; however the dirty bit only applies to the final stage of the walk.

The rule states that a particular access cannot be committed until the Accessed and/or Dirty bits are not visible to page management SW. For the GPU to follow the rule, GTT accesses (when A/D bits are supported) are done via a special cycle definition between GAM and GTI.

Replacement

TLB replacements during runtime are based on LRA algorithm. In addition invalidations and page responses have to invalidate the TLB entries.

Invalidations of TLB

There are various ways to invalidate TLBs:

1. **Traditional invalidation from command streamer:** Could be part of any fence accesses including newly added atomics
2. **SVM based invalidations:** Listed as part of the new SVM related invalidations, various stages of TLBs including intermediate stages can be invalidated selectively and/or as a whole.
3. **Context Switch:** A context switch has to invalidate caches to make sure we have no residual value of the TLBs across multiple PASIDs. GAM will treat the context reload message from CS as a form of TLB invalidation.
4. **A page response:** should invalidate faulted recordings. It should be done via address matching to kick the faulted entries within the matching PASID.

Invalidation response "Invalidation Wait Descriptor" should also be a fence for both READs and WRITEs that used the previous TLB entries. Gam can only respond to "invalidation wait descriptor" after getting a GTI EMPTY indication.

Optional Invalidations

The following cases are listed as page table updates which software may choose not to invalidate the TLBs.

Note: GAM only puts faulted entries to its TLBs if there has been page request for it. This means only faultable surfaces can be stored in GAM TLBs as a faulted entry.

- If a paging-structure is modified to change the Present (Valid) flag from 0 to 1, SW may choose not to invalidate TLBs. This affects only the case where GPU keeps the faulted page in its TLB to filter out future faults. Regardless of whether SW does invalidation or not, for the cases where HW cares, there is a page response from SW which is used to shootdown the faulted record from the TLB.
- If a paging-structure entry is modified to change the accessed flag from 0 to 1,no invalidation is necessary (assuming that an invalidation was performed the last time the accessed flag was changed from 1 to 0). This is because no TLB entry or paging-structure cache entry is created with information from a paging structure entry in which the accessed flag is 0.
- If a paging-structure entry is modified to change the R/W or U/S or XD flag from 0 to 1, failure to perform an invalidation may result in a "spurious" page-fault exception (e.g., in response to an attempted write access) but no other adverse behavior. Such an exception occurs at most once for each affected linear address.

Memory Types and Cache Interface

This section has additional information on the types of memory which are accessible via the various GT mechanisms. It includes discussion on how the various paging models are used and accessed. See the Graphics Translation Tables for more detailed discussions on paging models.

This section also includes descriptions of how different surface types (MOCS) can be cached in the L3 and the different behaviors which can be enabled.

Memory Object Control State (MOCS)

The memory object control state defines the behavior of memory accesses beyond the graphics core, including graphics data types that allow selective flushing of data from outer caches, and controlling cacheability in the outer caches.

This control uses several mechanisms. Control state for all memory accesses can be defined page by page in the GTT entries. Memory objects that are defined by state per surface generally have additional memory object control state in the state structure that defines the other surface attributes. Memory objects without state defining them have memory object state control defined per class in the STATE_BASE_ADDRESS command, with class divisions the same as the base addresses. Finally, some memory objects only have the GTT entry mechanism for defining this control. The table below enumerates the memory objects and the location of the control state for each:

Memory Object	Location of Control State
surfaces defined by SURFACE_STATE: sampling engine surfaces, render targets, media surfaces, pull constant buffers, streamed vertex buffers	SURFACE_STATE
depth, stencil, and hierarchical depth buffers	corresponding state command that defined the buffer attributes
stateless buffers accessed by data port	STATE_BASE_ADDRESS
indirect state objects	STATE_BASE_ADDRESS
kernel instructions	STATE_BASE_ADDRESS
push constant buffers	3DSTATE_CONSTANT_(VS | GS | PS)
index buffers	3DSTATE_INDEX_BUFFER
vertex buffers	3DSTATE_VERTEX_BUFFERS
indirect media object	STATE_BASE_ADDRESS
generic state prefetch	GTT control only
ring/batch buffers	GTT control only
context save buffers	GTT control only
store DWord	GTT control only

MOCS Registers

These registers provide the detailed format of the MOCS table entries that need to be programmed to define each surface state.

MEMORY_OBJECT_CONTROL_STATE

Page Walker Access and Memory Types

Most of these notes are further explained in the document however summarized as part of the page table behavior:

Page Walker Memory Types

1. Legacy Contexts
    a. GT access to root/extended-root table and context/extended-context table
    b. GTT access to private paging (PPGTT) entries
    c. GT access to GPA-to-HPA paging entries
    d. GT access to the translated page
2. Advanced context (without nesting)
    a. GT access to extended-root table and extended-context table
    b. GT access to PASID-entry & PASID-state entry
    c. GT access to IA-32e paging entries
    d. GT access to the translated page
3. Advanced context (with nesting)
    a. GT access to extended-root table and extended-context table
    b. GT access to PASID-entry & PASID-state entry
    c. GT access to IA-32e paging entries
    d. GT access to the translated page
    e. GT access to GPA-to-HPA paging entries to translate address of PASID-entry and PASID-state entry
    f. GT access to GPA-to-HPA paging entries to translate address of IA-32e paging entries
    g. GT access to GPA-to-HPA paging entries to translate address of page

Gen8 Memory Typing for Paging

The following information is duplicated in the Page Walker Memory Types topic:

1. Legacy Contexts
   a. GT access to root/extended-root table and context/extended-context table
   b. GTT access to private paging (PPGTT) entries
   c. GT access to GPA-to-HPA paging entries
   d. GT access to the translated page
2. Advanced context (without nesting)
   a. GT access to extended-root table and extended-context table
   b. GT access to PASID-entry & PASID-state entry
   c. GT access to IA-32e paging entries
   d. GT access to the translated page
3. Advanced context (with nesting)
   a. GT access to extended-root table and extended-context table
   b. GT access to PASID-entry & PASID-state entry
   c. GT access to IA-32e paging entries
   d. GT access to the translated page
   e. GT access to GPA-to-HPA paging entries to translate address of PASID-entry and PASID-state entry
   f. GT access to GPA-to-HPA paging entries to translate address of IA-32e paging entries
   g. GT access to GPA-to-HPA paging entries to translate address of page

This information is new in this topic and references the cases and subcases enumerated above:

For case [1]:

- [1.a] is always covered as a non-cacheable access
- [1.b] & [1.c] is covered with MMIO register where PPGTT entries can be forced to be cached in LLC (default option is cached).
- [1.d] is defined via private PAT (MMIO based) and surface state.

For case [2]:

- [2.a] is always covered as a non-cacheable access
- [2.b] is always cached & PASID state table entry is always accessed “atomically”
- [2.c] is accessed as cached
- [2.d] use memory-type as evaluated through MTRR, CD, and PCD/PWT/PAT bits in leaf IA-32e paging entry

For case [3]:

- [3.a] is always covered as a non-cacheable access
- [3.b] is always cached & PASID state table entry is always accessed "atomically"
- [3.c] is accessed as cached
- [3.d] use memory-type as follows (this section is further described in detail in memory typing section)
    - If CD=1, memory-type is UC
    - If CD=0:
        - If EMTE=0 in extended-context-entry, it is handled same as [2.d]
        - If EMTE=1 in extended-context-entry:
            - If IGMT=1 in leaf GPA-to-HPA entry, memory type used is the EMT field in this GPA-to-HPA entry.
            - If IGMT=0 in leaf GPA-to-HPA entry, memory type from [2.d] is combined with EMT field in this GPA-to-HPA entry.
- [3.e] is always cached & PASID state table entry is always accessed "atomically"
- [3.f] &[3.g] is accessed as cached

Error Cases

- A/D bit update attempt for paging entry in non-WB memory, causes page-walk to be aborted; Error reported to device in Translation Response; For Gen, gets reported to driver as GPGPU context in error – catastrophic error case.
- Locked/Atomic operations to pages in non-WB memory aborted; For Gen, gets reported to driver as GPGPU context in error (catastrophic error).
- CD=1 treated same as non-WB memory, for above lock behavior.

Common Surface Formats

This section documents surfaces and how they are stored in memory, including 3D and video surfaces, including the details of compressed texture formats. Also covered are the surface layouts based on tiling mode and surface type.

Non-Video Surface Formats

This section describes the lowest-level organization of a surfaces containing discrete “pixel” oriented data (e.g., discrete pixel (RGB,YUV) colors, subsampled video data, 3D depth/stencil buffer pixel formats, bump map values etc. Many of these pixel formats are common to the various pixel-oriented memory object types.

Surface Format Naming

Unless indicated otherwise, all pixels are **stored** in "**little endian**" byte order. i.e., pixel bits 7:0 are stored in byte *n*, pixel bits 15:8 are stored in byte *n*+1, and so on. The format labels include color components in little endian order (e.g., R8G8B8A8 format is physically stored as R, G, B, A).

The name of most of the surface formats specifies its format. Channels are listed in little endian order (LSB channel on the left, MSB channel on the right), with the channel format specified following the channels with that format. For example, R5G5_SNORM_B6_UNORM contains, from LSB to MSB, 5 bits of red in SNORM format, 5 bits of green in SNORM format, and 6 bits of blue in UNORM format.

Intensity Formats

All surface formats containing "I" include an intensity value. When used as a source surface for the sampling engine, the intensity value is replicated to all four channels (R,G,B,A) before being filtered. Intensity surfaces are not supported as destinations.

Luminance Formats

All surface formats containing “L” include a luminance value. When used as a source surface for the sampling engine, the luminance value is replicated to the three color channels (R,G,B) before being filtered. The alpha channel is provided either from another field or receives a default value. Luminance surfaces are not supported as destinations.

R1_UNORM (same as R1_UINT) and MONO8

When used as a texel format, the R1_UNORM format contains 8 1-bit Intensity (I) values that are replicated to all color channels. Note that T0 of byte 0 of a R1_UNORM-formatted texture corresponds to Texel[0,0]. This is different from the format used for monochrome sources in the BLT engine.

7	6	5	4	3	2	1	0
T7	T6	T5	T4	T3	T2	T1	T0

Bit	Description
T0	**Texel 0** On texture reads, this (unsigned) 1-bit value is replicated to all color channels. Format: U1
...	**...**
T7	**Texel 7** On texture reads, this (unsigned) 1-bit value is replicated to all color channels. Format: U1

MONO8 format is identical to R1_UNORM but has different semantics for filtering. MONO8 is the only supported format for the MAPFILTER_MONO filter. See the *Sampling Engine* chapter.

Palette Formats

Palette formats are supported by the sampling engine. These formats include an index into the palette (Px) that selects the actual channel values from the palette, which is loaded via the 3DSTATE_SAMPLER_PALETTE_LOAD0 command.

P4A4_UNORM

This surface format contains a 4-bit Alpha value (in the high nibble) and a 4-bit Palette Index value (in the low nibble).

7			4	3			0
Alpha				Palette Index			

Bit	Description
7:4	**Alpha** Alpha value which will be replicated to both the high and low nibble of an 8-bit value, and then divided by 255 to yield a [0.0,1.0] Alpha value. Format: U4
3:0	**Palette Index** A 4-bit index which is used to lookup a 24-bit (RGB) value in the texture palette (loaded via 3DSTATE_SAMPLER_PALETTE_LOADx) Format: U4

A4P4_UNORM

This surface format contains a 4-bit Alpha value (in the low nibble) and a 4-bit Color Index value (in the high nibble).

7			4	3			0
Palette Index				Alpha			

Bit	Description
7:4	**Palette Index** A 4-bit color index which is used to lookup a 24-bit RGB value in the texture palette. Format: U4
3:0	**Alpha** Alpha value which will be replicated to both the high and low nibble of an 8-bit value, and then divided by 255 to yield a [0.0,1.0] alpha value. Format: U4

P8A8_UNORM

This surface format contains an 8-bit Alpha value (in the high byte) and an 8-bit Palette Index value (in the low byte).

15			8	7			0
Alpha				Palette Index			

Bit	Description
15:8	**Alpha** Alpha value which will be divided by 255 to yield a [0.0,1.0] Alpha value. Format: U8
7:0	**Palette Index** An 8-bit index which is used to lookup a 24-bit (RGB) value in the texture palette (loaded via 3DSTATE_SAMPLER_PALETTE_LOADx) Format: U8

A8P8_UNORM

This surface format contains an 8-bit Alpha value (in the low byte) and an 8-bit Color Index value (in the high byte).

15			8	7			0
Palette Index				Alpha			

Bit	Description
15:8	**Palette Index** An 8-bit color index which is used to lookup a 24-bit RGB value in the texture palette. Format: U8
7:0	**Alpha** Alpha value which will be divided by 255 to yield a [0.0,1.0] alpha value. Format: U8

P8_UNORM

This surface format contains only an 8-bit Color Index value.

Bit	Description
7:0	**Palette Index** An 8-bit color index which is used to lookup a 32-bit ARGB value in the texture palette. Format: U8

P2_UNORM

This surface format contains only a 2-bit Color Index value.

Bit	Description
1:0	**Palette Index** A 2-bit color index which is used to lookup a 32-bit ARGB value in the texture palette. Format: U2

Compressed Surface Formats

This section contains information on the internal organization of compressed surface formats.

ETC1_RGB8

This format compresses UNORM RGB data using an 8-byte compression block representing a 4x4 block of texels. The texels are labeled as texel[row][column] where both row and column range from 0 to 3. Texel[0][0] is the upper left texel.

The 8-byte compression block is laid out as follows.

High 24 bits if "diff" is zero (individual mode):

Bits	Description
7:4	R0[3:0]
3:0	R1[3:0]
15:12	G0[3:0]
11:8	G1[3:0]
23:20	B0[3:0]
19:16	B1[3:0]

High 24 bits if "diff" is one (differential mode):

Bits	Description
7:3	R0[4:0]
2:0	dR1[2:0]
15:11	G0[4:0]
10:8	dG1[2:0]
23:19	B0[4:0]
18:16	dB1[2:0]

Low 40 bits:

Bits	Description
31:29	lum table index for sub-block 0
28:26	lum table index for sub-block 1
25	diff
24	flip
39	texel[3][3] index MSB
38	texel[2][3] index MSB
37	texel[1][3] index MSB
36	texel[0][3] index MSB
35	texel[3][2] index MSB

Bits	Description
34	texel[2][2] index MSB
33	texel[1][2] index MSB
32	texel[0][2] index MSB
47	texel[3][1] index MSB
46	texel[2][1] index MSB
45	texel[1][1] index MSB
44	texel[0][1] index MSB
43	texel[3][0] index MSB
42	texel[2][0] index MSB
41	texel[1][0] index MSB
40	texel[0][0] index MSB
55	texel[3][3] index LSB
54	texel[2][3] index LSB
53	texel[1][3] index LSB
52	texel[0][3] index LSB
51	texel[3][2] index LSB
50	texel[2][2] index LSB
49	texel[1][2] index LSB
48	texel[0][2] index LSB
63	texel[3][1] index LSB
62	texel[2][1] index LSB
61	texel[1][1] index LSB
60	texel[0][1] index LSB
59	texel[3][0] index LSB
58	texel[2][0] index LSB
57	texel[1][0] index LSB
56	texel[0][0] index LSB

The 4x4 is divided into two 8-pixel sub-blocks, either two 2x4 sub-blocks or two 4x2 sub-blocks controlled by the "flip" bit. If flip=0, sub-block 0 is the 2x4 on the left and sub-block 1 is the 2x4 on the right. If flip=1, sub-block 0 is the 4x2 on the top and sub-block 1 is the 4x2 on the bottom.

The "diff" bit controls whether the red/green/blue values (R0/G0/B0/R1/G1/B1) are stored as one 444 value per sub-block ("individual" mode with diff = 0), or a single 555 value for the first sub-block (R0/G0/B0) and a 333 delta value (dR1/dG1/dB1) for the second sub-block ("differential" mode with diff = 1). The delta values are 3-bit two's-complement values that hold values in the range [-4,3]. These values are added to the 5-bit values for sub-block 0 to obtain the 5-bit values for sub-block 1 (if the value is outside of the range [0,31], the result of the decompression is undefined). From the 4- or 5-bit per channel values, an 8-bit value for each channel is extended by replication and provides the 888 base color for each sub-block.

For each sub-block one of 8 different luminance columns is selected based on the 3-bit lum table index. Then each texel selects one of the 4 rows of the selected column with a 2-bit per-texel index. The chosen value in the table is added to the 8-bit base color for the sub-block (obtained in the previous step) to obtain the texel's color. Values in the table are given in decimal, representing an 8-bit UNORM as an 8-bit signed integer.

Luminance Table

	0	1	2	3	4	5	6	7
0	2	5	9	13	18	24	33	47
1	8	17	29	42	60	80	106	183
2	-2	-5	-9	-13	-18	-24	-33	-47
3	-8	-17	-29	-42	-60	-80	-106	-183

ETC2_RGB8 and ETC2_SRGB8

The ETC2_RGB8 format builds on top of ETC1_RGB8, using a set of invalid bit sequences to enable three new modes. The two modes of ETC1_RGB8 are also supported with ETC2_RGB8, and will not be documented in this section as they are covered in the ETC1_RGB8 section.

The detection of the three new modes is based on RGB and diff bits in locations as defined for ETC1 differential mode. The mode is determined as follows (x indicates don't care):

diff	Rt	Gt	Bt	mode
0	x	x	x	individual
1	0	x	x	T
1	1	0	x	H
1	1	1	0	planar
1	1	1	1	differential

The inputs in the above table are defined as follows:

```
 Rt = (R0 + dR1) in [0,31]
Gt = (G0 + dG1) in [0,31]
Bt = (G0 + dB1) in [0,31]
```

8-byte compression block for mode determination

Bits	Description
7:3	R0[4:0]
2:0	dR1[2:0]
15:11	G0[4:0]
10:8	dG1[2:0]
23:19	B0[4:0]
18:16	dB1[2:0]
31:26	ignored
25	diff
24	ignored
63:32	ignored

The fields in the table above are used *only* for mode determination. Some of the bits in this table are overloaded with other values within each mode. The algorithm is defined such that there is no ambiguity in modes when this is done.

T mode

The “T” mode has the following bit definition:

8-byte compression block for “T” mode

Bits	Description
7:5	ignored
4:3	R0[3:2]
2	ignored
1:0	R0[1:0]
15:12	G0[3:0]
11:8	B0[3:0]
23:20	R1[3:0]
19:16	G1[3:0]
31:28	B1[3:0]
27:26	di[2:1]
25	diff = 1
24	di[0]
39	texel[3][3] index MSB
38	texel[2][3] index MSB
37	texel[1][3] index MSB
36	texel[0][3] index MSB
35	texel[3][2] index MSB
34	texel[2][2] index MSB
33	texel[1][2] index MSB
32	texel[0][2] index MSB
47	texel[3][1] index MSB
46	texel[2][1] index MSB
45	texel[1][1] index MSB
44	texel[0][1] index MSB
43	texel[3][0] index MSB
42	texel[2][0] index MSB
41	texel[1][0] index MSB
40	texel[0][0] index MSB
55	texel[0][0] index LSB
54	texel[2][3] index LSB
53	texel[1][3] index LSB
52	texel[0][3] index LSB

Bits	Description
51	texel[3][2] index LSB
50	texel[2][2] index LSB
49	texel[1][2] index LSB
48	texel[0][2] index LSB
63	texel[3][1] index LSB
62	texel[2][1] index LSB
61	texel[1][1] index LSB
60	texel[0][1] index LSB
59	texel[3][0] index LSB
58	texel[2][0] index LSB
57	texel[1][0] index LSB
56	texel[0][0] index LSB

The "T" mode has two base colors stored as 4 bits per channel, R0/G0/B0 and R1/G1/B1, as in the individual mode, however the bit positions for these are different. For each channel, the 4 bits are extended to 8 bits by bit replication.

A 3-bit distance index "di" is also defined in the compression block. This value is used to look up the distance in the following table:

distance index "di"	distance "d"
0	3
1	6
2	11
3	16
4	23
5	32
6	41
7	64

Four colors are possible on each texel. These colors are defined as the following:

```
 P0 = (R0, G0, B0)
P1 = (R1, G1, B1) + (d, d, d)
P2 = (R1, G1, B1)
P3 = (R1, G1, B1) – (d, d, d)
```

All resulting channels are clamped to the range [0,255]. One of the four colors is then assigned to each texel in the block based on the 2-bit texel index.

H mode

The “H” mode has the following bit definition:

8-byte compression block for “H” mode

Bits	Description
7	ignored
6:3	R0[3:0]
2:0	G0[3:1]
15:13	ignored
12	G0[0]
11	B0[3]
10	ignored
9:8	B0[2:1]
23	B0[0]
22:19	R1[3:0]
18:16	G1[3:1]
31	G1[0]
30:27	B1[3:0]
26	di[2]
25	diff = 1
24	di[1]
39	texel[3][3] index MSB
38	texel[2][3] index MSB
37	texel[1][3] index MSB
36	texel[0][3] index MSB
35	texel[3][2] index MSB
34	texel[2][2] index MSB
33	texel[1][2] index MSB
32	texel[0][2] index MSB
47	texel[3][1] index MSB
46	texel[2][1] index MSB
45	texel[1][1] index MSB
44	texel[0][1] index MSB
43	texel[3][0] index MSB
42	texel[2][0] index MSB
41	texel[1][0] index MSB
40	texel[0][0] index MSB

Bits	Description
55	texel[3][3] index LSB
54	texel[2][3] index LSB
53	texel[1][3] index LSB
52	texel[0][3] index LSB
51	texel[3][2] index LSB
50	texel[2][2] index LSB
49	texel[1][2] index LSB
48	texel[0][2] index LSB
63	texel[3][1] index LSB
62	texel[2][1] index LSB
61	texel[1][1] index LSB
60	texel[0][1] index LSB
59	texel[3][0] index LSB
58	texel[2][0] index LSB
57	texel[1][0] index LSB
56	texel[0][0] index LSB

The “H” mode has two base colors stored as 4 bits per channel, R0/G0/B0 and R1/G1/B1, as in the individual and T modes, however the bit positions for these are different. For each channel, the 4 bits are extended to 8 bits by bit replication.

A 3-bit distance index “di” is defined by 2 MSBs in the compression block and the LSB computed by the following equation, where R/G/B values are the 8-bit values from the first step:

```
 di[0] = ((R0 « 16) | (G0 « 8) | B0) >= ((R1 « 16) | (G1 « 8) | B1)
```

The distance “d” is then looked up in the same table used for T mode. The four colors for H mode are computed as follows:

```
 P0 = (R0, G0, B0) + (d, d, d)
P1 = (R0, G0, B0) - (d, d, d)
P2 = (R1, G1, B1) + (d, d, d)
P3 = (R1, G1, B1) – (d, d, d)
```

All resulting channels are clamped to the range [0,255]. One of the four colors is then assigned to each texel in the block based on the 2-bit texel index as in T mode.

Planar mode

The “planar” mode has the following bit definition:

8-byte compression block for “planar” mode

Bits	Description
7	ignored
6:1	R0[5:0]
0	G0[6]
15	ignored
14:9	G0[5:0]
8	B[5]
23:21	ignored
20:19	B[4:3]
18	ignored
17:16	B0[2:1]
31	B0[0]
30:26	RH[5:1]
25	diff = 1
24	RH[0]
39:33	GH[6:0]
32	BH[5]
47:43	BH[4:0]
42:40	RV[5:3]
55:53	RV[2:0]
52:48	GV[6:2]
63:62	GV[1:0]
61:56	BV[5:0]

The “planar” mode has three base colors stored as RGB 676, with red & blue having 6 bits and green having 7 bits. These three base colors are each extended to RGB 888 with bit replication.

The color of each texel is then computed using the following equations, with x and y representing the texel position within the compression block:

```
 texel[y][x].R = x(RH-R0)/4 + y(RV-R0)/4 + R0
texel[y][x].G = x(GH-G0)/4 + y(GV-G0)/4 + G0
texel[y][x].B = x(BH-B0)/4 + y(BV-B0)/4 + B0
```

All resulting channels are clamped to the range [0,255].

The ETC2_SRGB8 format is decompressed as if it is ETC2_RGB8, then a conversion from the resulting RGB values to SRGB space is performed.

EAC_R11 and EAC_SIGNED_R11

These formats compress UNORM/SNORM single-channel data using an 8-byte compression block representing a 4x4 block of texels. The texels are labeled as texel[row][column] where both row and column range from 0 to 3. Texel[0][0] is the upper left texel.

The 8-byte compression block is laid out as follows.

EAC_R11 compression block layout

Bits	Description
7:0	R0[7:0]
15:12	m[3:0]
11:8	ti[3:0]
23:21	texel[0][0] index
20:18	texel[1][0] index
17:16,31	texel[2][0] index
30:28	texel[3][0] index
27:25	texel[0][1] index
24,39:38	texel[1][1] index
37:35	texel[2][1] index
34:32	texel[3][1] index
47:45	texel[0][2] index
44:42	texel[1][2] index
41:40,55	texel[2][2] index
54:52	texel[3][2] index
51:49	texel[0][3] index
48,63:62	texel[1][3] index
61:59	texel[2][3] index
58:56	texel[3][3] index

The “ti” (table index) value from the compression block is used to select one of the columns in the table below.

Intensity modifier (im) table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	-3	-3	-2	-2	-3	-3	-4	-3	-2	-2	-2	-2	-3	-1	-4	-3
1	-6	-7	-5	-4	-6	-7	-7	-5	-6	-5	-4	-5	-4	-2	-6	-5
2	-9	-10	-8	-6	-8	-9	-8	-8	-8	-8	-8	-7	-7	-3	-8	-7
3	-15	-13	-13	-13	-12	-11	-11	-11	-10	-10	-10	-10	-10	-10	-9	-9
4	2	2	1	1	2	2	3	2	1	1	1	1	2	0	3	2
5	5	6	4	3	5	6	6	4	5	4	3	4	3	1	5	4
6	8	9	7	5	7	8	7	7	7	7	7	6	6	2	7	6
7	14	12	12	12	11	10	10	10	9	9	9	9	9	9	8	8

The eight possible color values R_i are then computed from the 8 values in the column labeled im_i, where i ranges from 0 to 7:

For EAC_R11:

`if (m == 0)` R_i `= R0*8 + 4 +` im_i `else` R_i `= R0*8 + 4 + (`im_i `* m * 8)`

Each value is clamped to the range [0,2047].

For EAC_SIGNED_R11:

`if (m == 0)` R_i `= R0*8 +` im_i `else` R_i `= R0*8 + (`im_i `* m * 8)`

Each value is clamped to the range [-1023,1023].

Note that in the signed case, the R0 value is a signed, 2’s complement value in the range [-127, 127]. Before being used in the above equations, an R0 value of -128 must be clamped to -127.

Finally, each texel red value is selected from the 8 possible values R_i using the 3-bit index for that texel. The green, blue, and alpha values are set to their default values.

The final value represents an 11-bit UNORM or SNORM as an unsigned/signed integer.

ETC2_RGB8_PTA and ETC2_SRGB8_PTA

The ETC2_RGB8_PTA format is similar to ETC2_RGB8 but eliminates the “individual” mode in favor of allowing a punch-through alpha. The “diff” bit from ETC2_RGB8 is renamed to “opaque” in this format, and the mode selection behaves as if the “diff” bit is always 1, making the “individual” mode inaccessible for these formats.

An alpha value of either 0 or 255 (representing 0.0 or 1.0) is possible with this format. If alpha is determined to be zero, the three other channels are also forced to zero, regardless of what value the normal decompression algorithm would have produced.

Differential Mode

In differential mode, if the opaque bit is set, the luminance table for ETC2_RGB8 is used. If the opaque bit is not set, the following luminance table is used (note that rows 0 and 2 have been zeroed out, otherwise the table is the same):

Luminance Table for opaque bit not set

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	8	17	29	42	60	80	106	183
2	0	0	0	0	0	0	0	0
3	-8	-17	-29	-42	-60	-80	-106	-183

For each texel, if the opaque bit is zero and the corresponding texel index is equal to 2, the alpha value is set to zero (and therefore RGB for that texel will also end up at zero). Otherwise alpha is set to 255 and RGB is the result of the normal decompression calculations.

T and H Modes

In both of these modes, if the opaque bit is zero and the texel index is equal to 2, the alpha value is set to zero (and therefore RGB will also end up at zero). Otherwise alpha is set to 255.

Planar Mode

In planar mode, the opaque bit is ignored and alpha is set to 255.

The ETC2_SRGB8_PTA format is decompressed as if it is ETC2_RGB8_PTA, then a conversion from the resulting RGB values to SRGB space is performed, with alpha remaining unchanged.

ETC2_EAC_RGBA8 and ETC2_EAC_SRGB8_A8

The ETC2_EAC_RGBA8 format is a combination of ETC2_RGB8 and EAC_R8. A 16-byte compression block represents each 4x4. The low-order 8 bytes are used to compute alpha (instead of red) using the EAC_R8 algorithm. The high-order 8 bytes are used to compute RGB using the ETC2_RGB8 algorithm. The EAC_R8 format differs from EAC_R11 as described below.

The ETC2_EAC_SRGB8_A8 format is decompressed as if it is ETC2_EAC_RGBA8, then a conversion from the resulting RGB values to SRGB space is performed, with alpha remaining unchanged.

EAC_R8 Format:

The EAC_R8 format used within these surface formats is identical to EAC_R11 described in an earlier section, except the procedure for computing the eight possible color values Ri is performed as follows:

Ri = R0 + (imi * m)

Each value is clamped to the range [0,255].

EAC_RG11 and EAC_SIGNED_RG11

These formats compress UNORM/SNORM double-channel data using a 16-byte compression block representing a 4x4 block of texels. The texels are labeled as texel[row][column] where both row and column range from 0 to 3. Texel[0][0] is the upper left texel.

The 16-byte compression block is laid out as follows.

EAC_RG11 compression block layout

Bits	Description
63:56	G0[7:0]
55:52	Gm[3:0]
51:48	Gti[3:0]
47:45	texel[0][0] G index
44:42	texel[1][0] G index
41:39	texel[2][0] G index
38:36	texel[3][0] G index
35:33	texel[0][1] G index
32:30	texel[1][1] G index
29:27	texel[2][1] G index
26:24	texel[3][1] G index
23:21	texel[0][2] G index
20:18	texel[1][2] G index
17:15	texel[2][2] G index
14:12	texel[3][2] G index
11:9	texel[0][3] G index
8:6	texel[1][3] G index
5:3	texel[2][3] G index
66:64	texel[3][3] G index
63:56	R0[7:0]
55:52	Rm[3:0]
51:48	Rti[3:0]
47:45	texel[0][0] R index
44:42	texel[1][0] R index
41:39	texel[2][0] R index
38:36	texel[3][0] R index
35:33	texel[0][1] R index
32:30	texel[1][1] R index
29:27	texel[2][1] R index
26:24	texel[3][1] R index

Bits	Description
23:21	texel[0][2] R index
20:18	texel[1][2] R index
17:15	texel[2][2] R index
14:12	texel[3][2] R index
11:9	texel[0][3] R index
8:6	texel[1][3] R index
5:3	texel[2][3] R index
2:0	texel[3][3] R index

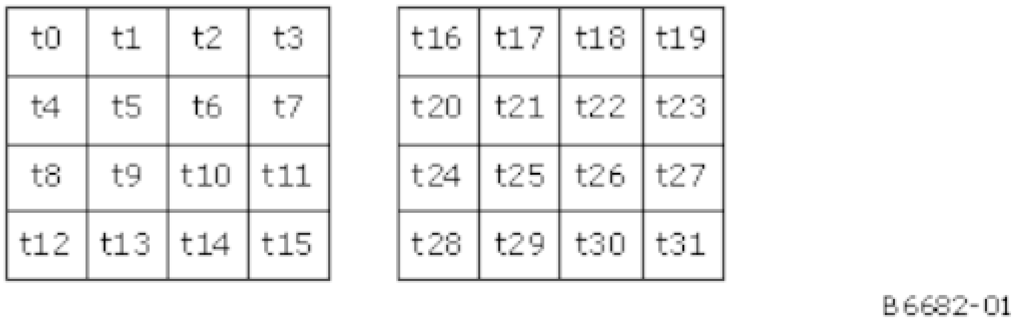
These compression formats are identical to the EAC_R11 and EAC_SIGNED_R11 formats, except that they supply two channels of output data, both red and green, from two independent 8-byte portions of the compression block. The low half of the compression block contains the red information, and the high half contains the green information. Blue and alpha channels are set to their default values.

Refer to the EAC_R11 and EAC_SIGNED_R11 specification for details on how the red and green channels are generated using the data in the compression block.

FXT Texture Formats

There are four different FXT1 compressed texture formats. Each of the formats compress two 4x4 texel blocks into 128 bits. In each compression format, the 32 texels in the two 4x4 blocks are arranged according to the following diagram:

FXT1 Encoded Blocks



t0	t1	t2	t3
t4	t5	t6	t7
t8	t9	t10	t11
t12	t13	t14	t15

t16	t17	t18	t19
t20	t21	t22	t23
t24	t25	t26	t27
t28	t29	t30	t31

Overview of FXT1 Formats

During the compression phase, the encoder selects one of the four formats for each block based on which encoding scheme results in best overall visual quality. The following table lists the four different modes and their encodings:

FXT1 Format Summary

Bit 127	Bit 126	Bit 125	Block Compression Mode	Summary Description
0	0	X	**CC_HI**	2 R5G5B5 colors supplied. Single LUT with 7 interpolated color values and transparent black
0	1	0	**CC_CHROMA**	4 R5G5B5 colors used directly as 4-entry LUT.
0	1	1	**CC_ALPHA**	3 A5R5G5B5 colors supplied. LERP bit selects between 1 LUT with 3 discrete colors + transparent black and 2 LUTs using interpolated values of Color 0,1 (t0-15) and Color 1,2 (t16-31).
1	x	x	**CC_MIXED**	4 R5G5B5 colors supplied, where Color0,1 LUT is used for t0-t15, and Color2,3 LUT used for t16-31. Alpha bit selects between LUTs with 4 interpolated colors or 3 interpolated colors + transparent black.

FXT1 CC_HI Format

In the CC_HI encoding format, two base 15-bit R5G5B5 colors (Color 0, Color 1) are included in the encoded block. These base colors are then expanded (using high-order bit replication) to 24-bit RGB colors, and used to define an 8-entry lookup table of interpolated color values (the 8th entry is transparent black). The encoded block contains a 3-bit index value per texel that is used to lookup a color from the table.

CC_HI Block Encoding

The following table describes the encoding of the 128-bit (DQWord) CC_HI block format:

FXT CC_HI Block Encoding

Bit	Description
127:126	Mode = '00'b (CC_HI)
125:121	Color 1 Red
120:116	Color 1 Green
115:111	Color 1 Blue
110:106	Color 0 Red
105:101	Color 0 Green
100:96	Color 0 Blue
95:93	Texel 31 Select
...	...
50:48	Texel 16 Select
47:45	Texel 15 Select
...	...
2:0	Texel 0 Select

CC_HI Block Decoding

The two base colors, Color 0 and Color 1 are converted from R5G5B5 to R8G8B8 by replicating the 3 MSBs into the 3 LSBs, as shown in the following table:

FXT CC_HI Decoded Colors

Expanded Color Bit	Expanded Channel Bit	Encoded Block Source Bit
Color 1 [23:19]	Color 1 Red [7:3]	[125:121]
Color 1 [18:16]	Color 1 Red [2:0]	[125:123]
Color 1 [15:11]	Color 1 Green [7:3]	[120:116]
Color 1 [10:08]	Color 1 Green [2:0]	[120:118]
Color 1 [07:03]	Color 1 Blue [7:3]	[115:111]
Color 1 [02:00]	Color 1 Blue [2:0]	[115:113]
Color 0 [23:19]	Color 0 Red [7:3]	[110:106]
Color 0 [18:16]	Color 0 Red [2:0]	[110:108]
Color 0 [15:11]	Color 0 Green [7:3]	[105:101]
Color 0 [10:08]	Color 0 Green [2:0]	[105:103]
Color 0 [07:03]	Color 0 Blue [7:3]	[100:96]
Color 0 [02:00]	Color 0 Blue [2:0]	[100:98]

These two 24-bit colors (Color 0, Color 1) are then used to create a table of seven interpolated colors (with Alpha = 0FFh), along with an eight entry equal to RGBA = 0,0,0,0, as shown in the following table:

FXT CC_HI Interpolated Color Table

Interpolated Color	Color RGB	Alpha
0	Color0.RGB	0FFh
1	(5 * Color0.RGB + 1 * Color1.RGB + 3) / 6	0FFh
2	(4 * Color0.RGB + 2 * Color1.RGB + 3) / 6	0FFh
3	(3 * Color0.RGB + 3 * Color1.RGB + 3) / 6	0FFh
4	(2 * Color0.RGB + 4 * Color1.RGB + 3) / 6	0FFh
5	(1 * Color0.RGB + 5 * Color1.RGB + 3) / 6	0FFh
6	Color1.RGB	0FFh
7	RGB = 0,0,0	0

This table is then used as an 8-entry Lookup Table, where each 3-bit Texel n Select field of the encoded CC_HI block is used to index into a 32-bit A8R8G8B8 color from the table completing the decode of the CC_HI block.

FXT1 CC_CHROMA Format

In the CC_CHROMA encoding format, four 15-bit R5B5G5 colors are included in the encoded block. These colors are then expanded (using high-order bit replication) to form a 4-entry table of 24-bit RGB colors. The encoded block contains a 2-bit index value per texel that is used to lookup a 24-bit RGB color from the table. The Alpha component defaults to fully opaque (0FFh).

CC_CHROMA Block Encoding

The following table describes the encoding of the 128-bit (DQWord) CC_CHROMA block format:

FXT CC_CHROMA Block Encoding

Bit	Description
127:125	Mode = '010'b (CC_CHROMA)
124	Unused
123:119	Color 3 Red
118:114	Color 3 Green
113:109	Color 3 Blue
108:104	Color 2 Red
103:99	Color 2 Green
98:94	Color 2 Blue
93:89	Color 1 Red
88:84	Color 1 Green
83:79	Color 1 Blue
78:74	Color 0 Red
73:69	Color 0 Green
68:64	Color 0 Blue
63:62	Texel 31 Select
...	
33:32	Texel 16 Select
31:30	Texel 15 Select
...	
1:0	Texel 0 Select

CC_CHROMA Block Decoding

The four colors (Color 0-3) are converted from R5G5B5 to R8G8B8 by replicating the 3 MSBs into the 3 LSBs, as shown in the following tables:

FXT CC_CHROMA Decoded Colors

Expanded Color Bit	Expanded Channel Bit	Encoded Block Source Bit
Color 3 [23:17]	Color 3 Red [7:3]	[123:119]
Color 3 [18:16]	Color 3 Red [2:0]	[123:121]
Color 3 [15:11]	Color 3 Green [7:3]	[118:114]
Color 3 [10:08]	Color 3 Green [2:0]	[118:116]
Color 3 [07:03]	Color 3 Blue [7:3]	[113:109]
Color 3 [02:00]	Color 3 Blue [2:0]	[113:111]
Color 2 [23:17]	Color 2 Red [7:3]	[108:104]
Color 2 [18:16]	Color 2 Red [2:0]	[108:106]
Color 2 [15:11]	Color 2 Green [7:3]	[103:99]
Color 2 [10:08]	Color 2 Green [2:0]	[103:101]
Color 2 [07:03]	Color 2 Blue [7:3]	[98:94]
Color 2 [02:00]	Color 2 Blue [2:0]	[98:96]
Color 1 [23:17]	Color 1 Red [7:3]	[93:89]
Color 1 [18:16]	Color 1 Red [2:0]	[93:91]
Color 1 [15:11]	Color 1 Green [7:3]	[88:84]
Color 1 [10:08]	Color 1 Green [2:0]	[88:86]
Color 1 [07:03]	Color 1 Blue [7:3]	[83:79]
Color 1 [02:00]	Color 1 Blue [2:0]	[83:81]
Color 0 [23:17]	Color 0 Red [7:3]	[78:74]
Color 0 [18:16]	Color 0 Red [2:0]	[78:76]
Color 0 [15:11]	Color 0 Green [7:3]	[73:69]
Color 0 [10:08]	Color 0 Green [2:0]	[73:71]
Color 0 [07:03]	Color 0 Blue [7:3]	[68:64]
Color 0 [02:00]	Color 0 Blue [2:0]	[68:66]

This table is then used as a 4-entry Lookup Table, where each 2-bit Texel n Select field of the encoded CC_CHROMA block is used to index into a 32-bit A8R8G8B8 color from the table (Alpha defaults to 0FFh) completing the decode of the CC_CHROMA block.

FXT CC_CHROMA Interpolated Color Table

Texel Select	Color ARGB
0	Color0.ARGB
1	Color1.ARGB
2	Color2.ARGB
3	Color3.ARGB

FXT1 CC_MIXED Format

In the CC_MIXED encoding format, four 15-bit R5G5B5 colors are included in the encoded block: Color 0 and Color 1 are used for Texels 0-15, and Color 2 and Color 3 are used for Texels 16-31.

Each pair of colors are then expanded (using high-order bit replication) to form 4-entry tables of 24-bit RGB colors. The encoded block contains a 2-bit index value per texel that is used to lookup a 24-bit RGB color from the table. The Alpha component defaults to fully opaque (0FFh).

CC_MIXED Block Encoding

The following table describes the encoding of the 128-bit (DQWord) CC_MIXED block format:

FXT CC_MIXED Block Encoding

Bit	Description
127	Mode = '1'b (CC_MIXED)
126	Color 3 Green [0]
125	Color 1 Green [0]
124	Alpha [0]
123:119	Color 3 Red
118:114	Color 3 Green
113:109	Color 3 Blue
108:104	Color 2 Red
103:99	Color 2 Green
98:94	Color 2 Blue
93:89	Color 1 Red
88:84	Color 1 Green
83:79	Color 1 Blue
78:74	Color 0 Red
73:69	Color 0 Green
68:64	Color 0 Blue
63:62	Texel 31 Select
...	...
33:32	Texel 16 Select
31:30	Texel 15 Select
...	...
1:0	Texel 0 Select

CC_MIXED Block Decoding

The decode of the CC_MIXED block is modified by Bit 124 (Alpha [0]) of the encoded block.

Alpha[0] = 0 Decoding

When Alpha[0] = 0 the four colors are encoded as 16-bit R5G6B5 values, with the Green LSB defined as per the following table:

FXT CC_MIXED (Alpha[0]=0) Decoded Colors

Encoded Color Bit	Definition
Color 3 Green [0]	Encoded Bit [126]
Color 2 Green [0]	Encoded Bit [33] XOR Encoded Bit [126]
Color 1 Green [0]	Encoded Bit [125]
Color 0 Green [0]	Encoded Bit [1] XOR Encoded Bit [125]

The four colors (Color 0-3) are then converted from R5G5B6 to R8G8B8 by replicating the 3 MSBs into the 3 LSBs, as shown in the following table:

FXT CC_MIXED Decoded Colors (Alpha[0] = 0)

Expanded Color Bit	Expanded Channel Bit	Encoded Block Source Bit
Color 3 [23:17]	Color 3 Red [7:3]	[123:119]
Color 3 [18:16]	Color 3 Red [2:0]	[123:121]
Color 3 [15:11]	Color 3 Green [7:3]	[118:114]
Color 3 [10]	Color 3 Green [2]	[126]
Color 3 [09:08]	Color 3 Green [1:0]	[118:117]
Color 3 [07:03]	Color 3 Blue [7:3]	[113:109]
Color 3 [02:00]	Color 3 Blue [2:0]	[113:111]
Color 2 [23:17]	Color 2 Red [7:3]	[108:104]
Color 2 [18:16]	Color 2 Red [2:0]	[108:106]
Color 2 [15:11]	Color 2 Green [7:3]	[103:99]
Color 2 [10]	Color 2 Green [2]	[33] XOR [126]]
Color 2 [09:08]	Color 2 Green [1:0]	[103:100]
Color 2 [07:03]	Color 2 Blue [7:3]	[98:94]
Color 2 [02:00]	Color 2 Blue [2:0]	[98:96]
Color 1 [23:17]	Color 1 Red [7:3]	[93:89]
Color 1 [18:16]	Color 1 Red [2:0]	[93:91]
Color 1 [15:11]	Color 1 Green [7:3]	[88:84]
Color 1 [10]	Color 1 Green [2]	[125]
Color 1 [09:08]	Color 1 Green [1:0]	[88:86]
Color 1 [07:03]	Color 1 Blue [7:3]	[83:79]

Expanded Color Bit	Expanded Channel Bit	Encoded Block Source Bit
Color 1 [02:00]	Color 1 Blue [2:0]	[83:81]
Color 0 [23:17]	Color 0 Red [7:3]	[78:74]
Color 0 [18:16]	Color 0 Red [2:0]	[78:76]
Color 0 [15:11]	Color 0 Green [7:3]	[73:69]
Color 0 [10]	Color 0 Green [2]	[1] XOR [125]
Color 0 [09:08]	Color 0 Green [1:0]	[73:71]
Color 0 [07:03]	Color 0 Blue [7:3]	[68:64]
Color 0 [02:00]	Color 0 Blue [2:0]	[68:66]

The two sets of 24-bit colors (Color 0,1 and Color 2,3) are then used to create two tables of four interpolated colors (with Alpha = 0FFh). The Color0,1 table is used as a lookup table for texel 0-15 indices, and the Color2,3 table used for texels 16-31 indices, as shown in the following figures:

FXT CC_MIXED Interpolated Color Table (Alpha[0]=0, Texels 0-15)

Texel 0-15 Select	Color RGB	Alpha
0	Color0.RGB	0FFh
1	(2*Color0.RGB + Color1.RGB + 1) /3	0FFh
2	(Color0.RGB + 2*Color1.RGB + 1) /3	0FFh
3	Color1.RGB	0FFh

FXT CC_MIXED Interpolated Color Table (Alpha[0]=0, Texels 16-31)

Texel 16-31 Select	Color RGB	Alpha
0	Color2.RGB	0FFh
1	(2/3) * Color2.RGB + (1/3) * Color3.RGB	0FFh
2	(1/3) * Color2.RGB + (2/3) * Color3.RGB	0FFh
3	Color3.RGB	0FFh

Alpha[0] = 1 Decoding

When Alpha[0] = 1, Color0 and Color2 are encoded as 15-bit R5G5B5 values. Color1 and Color3 are encoded as RGB565 colors, with the Green LSB obtained as shown in the following table:

FXT CC_MIXED (Alpha[0]=0) Decoded Colors

Encoded Color Bit	Definition
Color 3 Green [0]	Encoded Bit [126]
Color 1 Green [0]	Encoded Bit [125]

All four colors are then expanded to 24-bit R8G8B8 colors by bit replication.

FXT CC_MIXED Decoded Colors (Alpha[0] = 1)

Expanded Color Bit	Expanded Channel Bit	Encoded Block Source Bit
Color 3 [23:17]	Color 3 Red [7:3]	[123:119]
Color 3 [18:16]	Color 3 Red [2:0]	[123:121]
Color 3 [15:11]	Color 3 Green [7:3]	[118:114]
Color 3 [10]	Color 3 Green [2]	[126]
Color 3 [09:08]	Color 3 Green [1:0]	[118:117]
Color 3 [07:03]	Color 3 Blue [7:3]	[113:109]
Color 3 [02:00]	Color 3 Blue [2:0]	[113:111]
Color 2 [23:19]	Color 2 Red [7:3]	[108:104]
Color 2 [18:16]	Color 2 Red [2:0]	[108:106]
Color 2 [15:11]	Color 2 Green [7:3]	[103:99]
Color 2 [10:08]	Color 2 Green [2:0]	[103:101]
Color 2 [07:03]	Color 2 Blue [7:3]	[98:94]
Color 2 [02:00]	Color 2 Blue [2:0]	[98:96]
Color 1 [23:17]	Color 1 Red [7:3]	[93:89]
Color 1 [18:16]	Color 1 Red [2:0]	[93:91]
Color 1 [15:11]	Color 1 Green [7:3]	[88:84]
Color 1 [10]	Color 1 Green [2]	[125]
Color 1 [09:08]	Color 1 Green [1:0]	[88:87]
Color 1 [07:03]	Color 1 Blue [7:3]	[83:79]
Color 1 [02:00]	Color 1 Blue [2:0]	[83:81]
Color 0 [23:19]	Color 0 Red [7:3]	[78:74]
Color 0 [18:16]	Color 0 Red [2:0]	[78:76]
Color 0 [15:11]	Color 0 Green [7:3]	[73:69]
Color 0 [10:08]	Color 0 Green [2:0]	[73:71]
Color 0 [07:03]	Color 0 Blue [7:3]	[68:64]
Color 0 [02:00]	Color 0 Blue [2:0]	[68:66]

The two sets of 24-bit colors (Color 0,1 and Color 2,3) are then used to create two tables of four colors. The Color0,1 table is used as a lookup table for texel 0-15 indices, and the Color2,3 table used for texels 16-31 indices. The color at index 1 is the linear interpolation of the base colors, while the color at index 3 is defined as Black (0,0,0) with Alpha = 0:

FXT CC_MIXED Interpolated Color Table (Alpha[0]=1, Texels 0-15)

Texel 0-15 Select	Color RGB	Alpha
0	Color0.RGB	0FFh
1	(Color0.RGB + Color1.RGB) /2	0FFh
2	Color1.RGB	0FFh
3	Black (0,0,0)	0

FXT CC_MIXED Interpolated Color Table (Alpha[0]=1, Texels 16-31)

Texel 16-31 Select	Color RGB	Alpha
0	Color2.RGB	0FFh
1	(Color2.RGB + Color3.RGB) /2	0FFh
2	Color3.RGB	0FFh
3	Black (0,0,0)	0

These tables are then used as a 4-entry Lookup Table, where each 2-bit Texel n Select field of the encoded CC_MIXED block is used to index into the appropriate 32-bit A8R8G8B8 color from the table, completing the decode of the CC_CMIXED block.

FXT1 CC_ALPHA Format

In the CC_ALPHA encoding format, three A5R5G5B5 colors are provided in the encoded block. A control bit (LERP) is used to define the lookup table (or tables) used to dereference the 2-bit Texel Selects.

CC_ALPHA Block Encoding

The following table describes the encoding of the 128-bit (DQWord) CC_ALPHA block format:

FXT CC_ALPHA Block Encoding

Bit	Description
127:125	Mode = '011'b (CC_ALPHA)
124	LERP
123:119	Color 2 Alpha
118:114	Color 1 Alpha
113:109	Color 0 Alpha
108:104	Color 2 Red
103:99	Color 2 Green
98:94	Color 2 Blue
93:89	Color 1 Red
88:84	Color 1 Green
83:79	Color 1 Blue
78:74	Color 0 Red
73:69	Color 0 Green
68:64	Color 0 Blue
63:62	Texel 31 Select
...	...
33:32	Texel 16 Select
31:30	Texel 15 Select
...	...
1:0	Texel 0 Select

CC_ALPHA Block Decoding

Each of the three colors (Color 0-2) are converted from A5R5G5B5 to A8R8G8B8 by replicating the 3 MSBs into the 3 LSBs, as shown in the following tables:

FXT CC_ALPHA Decoded Colors

Expanded Color Bit	Expanded Channel Bit	Encoded Block Source Bit
Color 2 [31:27]	Color 2 Alpha [7:3]	[123:119]
Color 2 [26:24]	Color 2 Alpha [2:0]	[123:121]
Color 2 [23:17]	Color 2 Red [7:3]	[108:104]
Color 2 [18:16]	Color 2 Red [2:0]	[108:106]
Color 2 [15:11]	Color 2 Green [7:3]	[103:99]
Color 2 [10:08]	Color 2 Green [2:0]	[103:101]
Color 2 [07:03]	Color 2 Blue [7:3]	[98:94]
Color 2 [02:00]	Color 2 Blue [2:0]	[98:96]
Color 1 [31:27]	Color 1 Alpha [7:3]	[118:114]
Color 1 [26:24]	Color 1 Alpha [2:0]	[118:116]
Color 1 [23:17]	Color 1 Red [7:3]	[93:89]
Color 1 [18:16]	Color 1 Red [2:0]	[93:91]
Color 1 [15:11]	Color 1 Green [7:3]	[88:84]
Color 1 [10:08]	Color 1 Green [2:0]	[88:86]
Color 1 [07:03]	Color 1 Blue [7:3]	[83:79]
Color 1 [02:00]	Color 1 Blue [2:0]	[83:81]
Color 0 [31:27]	Color 0 Alpha [7:3]	[113:109]
Color 0 [26:24]	Color 0 Alpha [2:0]	[113:111]
Color 0 [23:17]	Color 0 Red [7:3]	[78:74]
Color 0 [18:16]	Color 0 Red [2:0]	[78:76]
Color 0 [15:11]	Color 0 Green [7:3]	[73:69]
Color 0 [10:08]	Color 0 Green [2:0]	[73:71]
Color 0 [07:03]	Color 0 Blue [7:3]	[68:64]
Color 0 [02:00]	Color 0 Blue [2:0]	[68:66]

LERP = 0 Decoding

When LERP = 0, a single 4-entry lookup table is formed using the three expanded colors, with the 4th entry defined as transparent black (ARGB=0,0,0,0). Each 2-bit Texel n Select field of the encoded CC_ALPHA block is used to index into a 32-bit A8R8G8B8 color from the table completing the decode of the CC_ALPHA block.

FXT CC_ALPHA Interpolated Color Table (LERP=0)

Texel Select	Color	Alpha
0	Color0.RGB	Color0.Alpha
1	Color1.RGB	Color1.Alpha
2	Color2.RGB	Color2.Alpha
3	Black (RGB=0,0,0)	0

LERP = 1 Decoding

When LERP = 1, the three expanded colors are used to create two tables of four interpolated colors. The Color0,1 table is used as a lookup table for texel 0-15 indices, and the Color1,2 table used for texels 16-31 indices, as shown in the following figures:

FXT CC_ALPHA Interpolated Color Table (LERP=1, Texels 0-15)

Texel 0-15 Select	Color ARGB
0	Color0.ARGB
1	(2*Color0.ARGB + Color1.ARGB + 1) /3
2	(Color0.ARGB + 2*Color1.ARGB + 1) /3
3	Color1.ARGB

FXT CC_ALPHA Interpolated Color Table (LERP=1, Texels 16-31)

Texel 16-31 Select	Color ARGB
0	Color2.ARGB
1	(2*Color2.ARGB + Color1.ARGB + 1) /3
2	(Color2.ARGB + 2*Color1.ARGB + 1) /3
3	Color1.ARGB

DXT Texture Formats

Note that non-power-of-2 dimensioned maps may require the surface to be padded out to the next multiple of four texels – here the pad texels are not referenced by the device.

An 8-byte (QWord) block encoding can be used if the source texture contains no transparency (is opaque) or if the transparency can be specified by a one-bit alpha. A 16-byte (DQWord) block encoding can be used to support source textures that require more than one-bit alpha: here the 1st QWord is used to encode the texel alpha values, and the 2nd QWord is used to encode the texel color values.

These three types of format are discussed in the following sections:

- Opaque and One-bit Alpha Textures (DXT1)
- Opaque Textures (DXT1_RGB)
- Textures with Alpha Channels (DXT2-5)

Programming Note	
Context:	DXT Texture Formats
• Any single texture must specify that its data is stored as 64 or 128 bits per group of 16 texels. If 64-bit blocks—that is, format DXT1—are used for the texture, it is possible to mix the opaque and one-bit alpha formats on a per-block basis within the same texture. In other words, the comparison of the unsigned integer magnitude of color_0 and color_1 is performed uniquely for each block of 16 texels. • When 128-bit blocks are used, then the alpha channel must be specified in either explicit (format DXT2 or DXT3) or interpolated mode (format DXT4 or DXT5) for the entire texture. Note that as with color, once interpolated mode is selected then either 8 interpolated alphas or 6 interpolated alphas mode can be used on a block-by-block basis. Again the magnitude comparison of alpha_0 and alpha_1 is done uniquely on a block-by-block basis.	

Opaque and One-bit Alpha Textures (DXT1/BC1)

Texture format DXT1 is for textures that are opaque or have a single transparent color. For each opaque or one-bit alpha block, two 16-bit R5G6B5 values and a 4x4 bitmap with 2-bits-per-pixel are stored. This totals 64 bits (1 QWord) for 16 texels, or 4-bits-per-texel.

In the block bitmap, there are two bits per texel to select between the four colors, two of which are stored in the encoded data. The other two colors are derived from these stored colors by linear interpolation.

The one-bit alpha format is distinguished from the opaque format by comparing the two 16-bit color values stored in the block. They are treated as unsigned integers. If the first color is greater than the second, it implies that only opaque texels are defined. This means four colors will be used to represent the texels. In four-color encoding, there are two derived colors and all four colors are equally distributed in RGB color space. This format is analogous to R5G6B5 format. Otherwise, for one-bit alpha transparency, three colors are used and the fourth is reserved to represent transparent texels. Note that the color blocks in DXT2-5 formats strictly use four colors, as the alpha values are obtained from the alpha block .

In three-color encoding, there is one derived color and the fourth two-bit code is reserved to indicate a transparent texel (alpha information). This format is analogous to A1R5G5B5, where the final bit is used for encoding the alpha mask.

The following piece of pseudo-code illustrates the algorithm for deciding whether three- or four-color encoding is selected:

```
 if (color_0 > color_1)
{
   // Four-color block: derive the other two colors.  // 00 = color_0, 01 = color_1, 10 =
color_2, 11 = color_3
   // These two bit codes correspond to the 2-bit fields
   // stored in the 64-bit block.  color_2 = (2 * color_0 + color_1) / 3;
    color_3 = (color 0 + 2 * color_1) / 3;
}
else
{
   // Three-color block: derive the other color.  // 00 = color_0, 01 = color_1, 10 =
color_2,
   // 11 = transparent.  // These two bit codes correspond to the 2-bit fields
   // stored in the 64-bit block.  color_2 = (color_0 + color_1) / 2;
    color_3 = transparent;
}
```

The following tables show the memory layout for the 8-byte block. It is assumed that the first index corresponds to the y-coordinate and the second corresponds to the x-coordinate. For example, Texel[1][2] refers to the texture map pixel at (x,y) = (2,1).

Here is the memory layout for the 8-byte (64-bit) block:

Word Address	16-bit Word
0	Color_0
1	Color_1
2	Bitmap Word_0
3	Bitmap Word_1

Color_0 and Color_1 (colors at the two extremes) are laid out as follows:

Bits	Color
15:11	Red color component
10:5	Green color component
4:0	Blue color component

Bits	Texel
1:0 (LSB)	Texel[0][0]
3:2	Texel[0][1]
5:4	Texel[0][2]
7:6	Texel[0][3]
9:8	Texel[1][0]
11:10	Texel[1][1]
13:12	Texel[1][2]
15:14	Texel[1][3]

Bitmap Word_1 is laid out as follows:

Bits	Texel
1:0 (LSB)	Texel[2][0]
3:2	Texel[2][1]
5:4	Texel[2][2]
7:6	Texel[2][3]
9:8	Texel[3][0]
11:10	Texel[3][1]
13:12	Texel[3][2]
15:14 (MSB)	Texel[3][3]

Example of Opaque Color Encoding

As an example of opaque encoding, we will assume that the colors red and black are at the extremes. We will call red color_0 and black color_1. There will be four interpolated colors that form the uniformly distributed gradient between them. To determine the values for the 4x4 bitmap, the following calculations are used:

```
 00 ? color_0
01 ? color_1
10 ? 2/3 color_0 + 1/3 color_1
11 ? 1/3 color_0 + 2/3 color_1
```

Example of One-bit Alpha Encoding

This format is selected when the unsigned 16-bit integer, color_0, is less than the unsigned 16-bit integer, color_1. An example of where this format could be used is leaves on a tree to be shown against a blue sky. Some texels could be marked as transparent while three shades of green are still available for the leaves. Two of these colors fix the extremes, and the third color is an interpolated color.

The bitmap encoding for the colors and the transparency is determined using the following calculations:

```
 00 ? color_0
01 ? color_1
10 ? 1/2 color_0 + 1/2 color_1
11 ? Transparent
```

Opaque Textures (DXT1_RGB)

Texture format DXT1_RGB is identical to DXT1, with the exception that the One-bit Alpha encoding is removed. Color 0 and Color 1 are not compared, and the resulting texel color is derived strictly from the Opaque Color Encoding. The alpha channel defaults to 1.0.

Programming Note	
Context:	Opaque Textures (DXT1_RGB)
The behavior of this format is not compliant with the OGL spec. As a workaround, the **Surface Format** should be set to BC1 and the **Shader Channel Select A** should be set to SCS_ONE. This workaround is only available for BDW due to lack of Shader Channel Select support on earlier products.	

Compressed Textures with Alpha Channels (DXT2-5 / BC2-3)

There are two ways to encode texture maps that exhibit more complex transparency. In each case, a block that describes the transparency precedes the 64-bit block already described. The transparency is either represented as a 4x4 bitmap with four bits per pixel (explicit encoding), or with fewer bits and linear interpolation analogous to what is used for color encoding.

The transparency block and the color block are laid out as follows:

Word Address	64-bit Block
3:0	Transparency block
7:4	Previously described 64-bit block

Explicit Texture Encoding

For explicit texture encoding (DXT2 and DXT3 formats), the alpha components of the texels that describe transparency are encoded in a 4x4 bitmap with 4 bits per texel. These 4 bits can be achieved through a variety of means such as dithering or by simply using the 4 most significant bits of the alpha data. However they are produced, they are used just as they are, without any form of interpolation.

Note: DirectDraw's compression method uses the 4 most significant bits.

The following tables illustrate how the alpha information is laid out in memory, for each 16-bit word.

This is the layout for Word 0:

Bits	Alpha
3:0 (LSB)	[0][0]
7:4	[0][1]
11:8	[0][2]
15:12 (MSB)	[0][3]

This is the layout for Word 1:

Bits	Alpha
3:0 (LSB)	[1][0]
7:4	[1][1]
11:8	[1][2]
15:12 (MSB)	[1][3]

This is the layout for Word 2:

Bits	Alpha
3:0 (LSB)	[2][0]
7:4	[2][1]
11:8	[2][2]
15:12 (MSB)	[2][3]

This is the layout for Word 3:

Bits	Alpha
3:0 (LSB)	[3][0]
7:4	[3][1]
11:8	[3][2]
15:12 (MSB)	[3][3]

Three-Bit Linear Alpha Interpolation

The encoding of transparency for the DXT4 and DXT5 formats is based on a concept similar to the linear encoding used for color. Two 8-bit alpha values and a 4x4 bitmap with three bits per pixel are stored in the first eight bytes of the block. The representative alpha values are used to interpolate intermediate alpha values. Additional information is available in the way the two alpha values are stored. If alpha_0 is greater than alpha_1, then six intermediate alpha values are created by the interpolation. Otherwise, four intermediate alpha values are interpolated between the specified alpha extremes. The two additional implicit alpha values are 0 (fully transparent) and 255 (fully opaque).

The following pseudo-code illustrates this algorithm:

```
 // 8-alpha or 6-alpha block?
if (alpha_0 > alpha_1) {
    // 8-alpha block: derive the other 6 alphas.
    // 000 = alpha_0, 001 = alpha_1, others are interpolated
   alpha_2 = (6 * alpha_0 + alpha_1) / 7;     // Bit code 010
   alpha_3 = (5 * alpha_0 + 2 * alpha_1) / 7; // Bit code 011
   alpha_4 = (4 * alpha_0 + 3 * alpha_1) / 7; // Bit code 100
   alpha_5 = (3 * alpha_0 + 4 * alpha_1) / 7; // Bit code 101
   alpha_6 = (2 * alpha_0 + 5 * alpha_1) / 7; // Bit code 110
   alpha_7 = (alpha_0 + 6 * alpha_1) / 7;     // Bit code 111
 }
else {
    // 6-alpha block: derive the other alphas.
    // 000 = alpha_0, 001 = alpha_1, others are interpolated
   alpha_2 = (4 * alpha_0 + alpha_1) / 5;     // Bit code 010
   alpha_3 = (3 * alpha_0 + 2 * alpha_1) / 5; // Bit code 011
   alpha_4 = (2 * alpha_0 + 3 * alpha_1) / 5; // Bit code 100
   alpha_5 = (alpha_0 + 4 * alpha_1) / 5;     // Bit code 101
   alpha_6 = 0;                               // Bit code 110
   alpha_7 = 255;                             // Bit code 111
}
```

The memory layout of the alpha block is as follows:

Byte	Alpha
0	Alpha_0
1	Alpha_1
2	[0][2] (2 LSBs), [0][1], [0][0]
3	[1][1] (1 LSB), [1][0], [0][3], [0][2] (1 MSB)
4	[1][3], [1][2], [1][1] (2 MSBs)
5	[2][2] (2 LSBs), [2][1], [2][0]
6	[3][1] (1 LSB), [3][0], [2][3], [2][2] (1 MSB)
7	[3][3], [3][2], [3][1] (2 MSBs)

BC4

These formats (BC4_UNORM and BC4_SNORM) compresses single-component UNORM or SNORM data. An 8-byte compression block represents a 4x4 block of texels. The texels are labeled as texel[row][column] where both row and column range from 0 to 3. Texel[0][0] is the upper left texel.

The 8-byte compression block is laid out as follows:

Bit	Description
7:0	red_0
15:8	red_1
18:16	texel[0][0] bit code
21:19	texel[0][1] bit code
24:22	texel[0][2] bit code
27:25	texel[0][3] bit code
30:28	texel[1][0] bit code
33:31	texel[1][1] bit code
36:34	texel[1][2] bit code
39:37	texel[1][3] bit code
42:40	texel[2][0] bit code
45:43	texel[2][1] bit code
48:46	texel[2][2] bit code
51:49	texel[2][3] bit code
54:52	texel[3][0] bit code
57:55	texel[3][1] bit code
60:58	texel[3][2] bit code
63:61	texel[3][3] bit code

There are two interpolation modes, chosen based on which reference color is larger. The first mode has the two reference colors plus six equal-spaced interpolated colors between the reference colors, chosen based on the three-bit code for that texel. The second mode has the two reference colors plus four interpolated colors, chosen by six of the three-bit codes. The remaining two codes select min and max values for the colors. The values of red_0 through red_7 are computed as follows:

```
 red_0 = red_0;                          // bit code 000
red_1 = red_1;                          // bit code 001
if (red_0 > red_1) {
     red_2 = (6 * red_0 + 1 * red_1) / 7; // bit code 010
     red_3 = (5 * red_0 + 2 * red_1) / 7; // bit code 011
     red_4 = (4 * red_0 + 3 * red_1) / 7; // bit code 100
     red_5 = (3 * red_0 + 4 * red_1) / 7; // bit code 101
     red_6 = (2 * red_0 + 5 * red_1) / 7; // bit code 110
     red_7 = (1 * red_0 + 6 * red_1) / 7; // bit code 111
}
else {
     red_2 = (4 * red_0 + 1 * red_1) / 5; // bit code 010
     red_3 = (3 * red_0 + 2 * red_1) / 5; // bit code 011
     red_4 = (2 * red_0 + 3 * red_1) / 5; // bit code 100
     red_5 = (1 * red_0 + 4 * red_1) / 5; // bit code 101
     red_6 = UNORM ? 0.0 : -1.0;          // bit code 110 (0 for UNORM, -1 for SNORM)
     red_7 = 1.0;                         // bit code 111
}
```

BC5

These formats (BC5_UNORM and BC5_SNORM) compresses dual-component UNORM or SNORM data. A 16-byte compression block represents a 4x4 block of texels. The texels are labeled as texel[row][column] where both row and column range from 0 to 3. Texel[0][0] is the upper left texel.

The 16-byte compression block is laid out as follows:

Bit	Description
7:0	red_0
15:8	red_1
18:16	texel[0][0] red bit code
21:19	texel[0][1] red bit code
24:22	texel[0][2] red bit code
27:25	texel[0][3] red bit code
30:28	texel[1][0] red bit code
33:31	texel[1][1] red bit code
36:34	texel[1][2] red bit code
39:37	texel[1][3] red bit code
42:40	texel[2][0] red bit code
45:43	texel[2][1] red bit code
48:46	texel[2][2] red bit code
51:49	texel[2][3] red bit code
54:52	texel[3][0] red bit code
57:55	texel[3][1] red bit code
60:58	texel[3][2] red bit code
63:61	texel[3][3] red bit code
71:64	green_0
79:72	green_1
82:80	texel[0][0] green bit code
85:83	texel[0][1] green bit code
88:86	texel[0][2] green bit code
91:89	texel[0][3] green bit code
94:92	texel[1][0] green bit code
97:95	texel[1][1] green bit code
100:98	texel[1][2] green bit code
103:101	texel[1][3] green bit code
106:104	texel[2][0] green bit code
109:107	texel[2][1] green bit code
112:110	texel[2][2] green bit code

Bit	Description
115:113	texel[2][3] green bit code
118:116	texel[3][0] green bit code
121:119	texel[3][1] green bit code
124:122	texel[3][2] green bit code
127:125	texel[3][3] green bit code

There are two interpolation modes, chosen based on which reference color is larger. The first mode has the two reference colors plus six equal-spaced interpolated colors between the reference colors, chosen based on the three-bit code for that texel. The second mode has the two reference colors plus four interpolated colors, chosen by six of the three-bit codes. The remaining two codes select min and max values for the colors. The values of red_0 through red_7 are computed as follows:

```
 red_0 = red_0;                          // bit code 000
red_1 = red_1;                          // bit code 001
if (red_0 > red_1) {
     red_2 = (6 * red_0 + 1 * red_1) / 7; // bit code 010
     red_3 = (5 * red_0 + 2 * red_1) / 7; // bit code 011
     red_4 = (4 * red_0 + 3 * red_1) / 7; // bit code 100
     red_5 = (3 * red_0 + 4 * red_1) / 7; // bit code 101
     red_6 = (2 * red_0 + 5 * red_1) / 7; // bit code 110
     red_7 = (1 * red_0 + 6 * red_1) / 7; // bit code 111
}
else {
     red_2 = (4 * red_0 + 1 * red_1) / 5; // bit code 010
     red_3 = (3 * red_0 + 2 * red_1) / 5; // bit code 011
     red_4 = (2 * red_0 + 3 * red_1) / 5; // bit code 100
     red_5 = (1 * red_0 + 4 * red_1) / 5; // bit code 101
     red_6 = UNORM ? 0.0 : -1.0;          // bit code 110 (0 for UNORM, -1 for SNORM)
     red_7 = 1.0;                         // bit code 111
}
```

The same calculations are done for green, using the corresponding reference colors and bit codes.

BC6H

For BDW, these formats (BC6H_UF16 and BC6H_SF16) compresses 3-channel images with high dynamic range (> 8 bits per channel). BC6H supports floating point denorms but there is no support for INF and NaN, other than with BC6H_SF16 –INF is supported. The alpha channel is not included, thus alpha is returned at its default value.

The BC6H block is 16 bytes and represents a 4x4 block of texels. The texels are labeled as texel[row][column] where both row and column range from 0 to 3. Texel[0][0] is the upper left texel. BC6H has 14 different modes, the mode that the block is in is contained in the least significant bits (either 2 or 5 bits).

The basic scheme consists of interpolating colors along either one or two lines, with per-texel indices indicating which color along the line is chosen for each texel. If a two-line mode is selected, one of 32 partition sets is indicated which selects which of the two lines each texel is assigned to.

Field Definition

There are 14 possible modes for a BC6H block, the format of each is indicated in the 14 tables below. The mode is selected by the unique mode bits specified in each table. The first 10 modes use two lines ("TWO"), and the last 4 use one line ("ONE"). The difference between the various two-line and one-line modes is with the precision of the first endpoint and the number of bits used to store delta values for the remaining endpoints. Two modes (9 and 10) specify each endpoint as an original value rather than using the deltas (these are indicated as having no delta values).

The endpoints values and deltas are indicated in the tables using a two-letter name. The first letter is "r", "g", or "b" indicating the color channel. The second letter is "w", "x", "y", or "z" indicating which of the four endpoints. The first line has endpoints "w" and "x", with "w" being the endpoint that is fully specified (i.e. not as a delta). The second line has endpoints "y" and "z". Modes using ONE mode do not have endpoints "y" and "z" as they have only one line.

In addition to the mode and endpoint data, TWO blocks contain a 5-bit "partition" which selects one of the partition sets, and a 46-bit set of indices. ONE blocks contain a 63-bit set of indices. These are described in more detail in the following tables.

Mode 0: (TWO) Red, Green, Blue: 10-bit endpoint, 5-bit deltas

Bit	Description
1:0	mode = 00
2	gy[4]
3	by[4]
4	bz[4]
14:5	rw[9:0]
24:15	gw[9:0]
34:25	bw[9:0]
39:35	rx[4:0]
40	gz[4]
44:41	gy[3:0]
49:45	gx[4:0]
50	bz[0]
54:51	gz[3:0]
59:55	bx[4:0]
60	bz[1]
64:61	by[3:0]
69:65	ry[4:0]
70	bz[2]
75:71	rz[4:0]
76	bz[3]
81:77	partition
127:82	indices

Mode 1: (TWO) Red, Green, Blue: 7-bit endpoint, 6-bit deltas

Bit	Description
1:0	mode = 01
2	gy[5]
3	gz[4]
4	gz[5]
11:5	rw[6:0]
12	bz[0]
13	bz[1]
14	by[4]
21:15	gw[6:0]
22	by[5]
23	bz[2]
24	gy[4]
31:25	bw[6:0]
32	bz[3]
33	bz[5]
34	bz[4]
40:35	rx[5:0]
44:41	gy[3:0]
50:45	gx[5:0]
54:51	gz[3:0]
60:55	bx[5:0]
64:61	by[3:0]
70:65	ry[5:0]
76:71	rz[5:0]
81:77	partition
127:82	indices

Mode 2: (TWO) Red: 11-bit endpoint, 5-bit deltas

Green, Blue: 11-bit endpoint, 4-bit deltas

Bit	Description
4:0	mode = 00010
14:5	rw[9:0]
24:15	gw[9:0]
34:25	bw[9:0]
39:35	rx[4:0]
40	rw[10]
44:41	gy[3:0]
48:45	gx[3:0]
49	gw[10]
50	bz[0]
54:51	gz[3:0]
58:55	bx[3:0]
59	bw[10]
60	bz[1]
64:61	by[3:0]
69:65	ry[4:0]
70	bz[2]
75:71	rz[4:0]
76	bz[3]
81:77	partition
127:82	indices

Mode 3: (TWO) Red, Blue: 11-bit endpoint, 4-bit deltas

Green: 11-bit endpoint, 5-bit deltas

Bit	Description
4:0	mode = 00110
14:5	rw[9:0]
24:15	gw[9:0]
34:25	bw[9:0]
38:35	rx[3:0]
39	rw[10]
40	gz[4]
44:41	gy[3:0]
49:45	gx[4:0]
50	gw[10]
54:51	gz[3:0]
58:55	bx[3:0]
59	bw[10]
60	bz[1]
64:61	by[3:0]
68:65	ry[3:0]
69	bz[0]
70	bz[2]
74:71	rz[3:0]
75	gy[4]
76	bz[3]
81:77	partition
127:82	indices

Mode 4: (TWO) Red, Green: 11-bit endpoint, 4-bit deltas

Blue: 11-bit endpoint, 5-bit deltas

Bit	Description
4:0	mode = 01010
14:5	rw[9:0]
24:15	gw[9:0]
34:25	bw[9:0]
38:35	rx[3:0]
39	rw[10]
40	by[4]
44:41	gy[3:0]
48:45	gx[3:0]
49	gw[10]
50	bz[0]
54:51	gz[3:0]
59:55	bx[4:0]
60	bw[10]
64:61	by[3:0]
68:65	ry[3:0]
69	bz[1]
70	bz[2]
74:71	rz[3:0]
75	bz[4]
76	bz[3]
81:77	partition
127:82	indices

Mode 5: (TWO) Red, Green, Blue: 9-bit endpoint, 5-bit deltas

Bit	Description
4:0	mode = 01110
13:5	rw[8:0]
14	by[4]
23:15	gw[8:0]
24	gy[4]
33:25	bw[8:0]
34	bz[4]
39:35	rx[4:0]
40	gz[4]
44:41	gy[3:0]
49:45	gx[3:0]
50	bz[0]
54:51	gz[3:0]
59:55	bx[4:0]
60	bz[1]
64:61	by[3:0]
69:65	ry[4:0]
70	bz[2]
75:71	rz[4:0]
76	bz[3]
81:77	partition
127:82	indices

Mode 6: (TWO) Red: 8-bit endpoint, 6-bit deltas

Green, Blue: 8-bit endpoint, 5-bit deltas

Bit	Description
4:0	mode = 10010
12:5	rw[7:0]
13	gz[4]
14	by[4]
22:15	gw[7:0]
23	bz[2]
24	gy[4]
32:25	bw[7:0]
33	bz[3]
34	bz[4]
40:35	rx[5:0]
44:41	gy[3:0]
49:45	gx[4:0]
50	bz[0]
54:51	gz[3:0]
59:55	bx[4:0]
60	gz[1]
64:61	by[3:0]
70:65	ry[5:0]
76:71	rz[5:0]
81:77	partition
127:82	indices

Mode 7: (TWO) Red, Blue: 8-bit endpoint, 5-bit deltas

Green: 8-bit endpoint, 6-bit deltas

Bit	Description
4:0	mode = 10110
12:5	rw[7:0]
13	bz[0]
14	by[4]
22:15	gw[7:0]
23	gy[5]
24	gy[4]
32:25	bw[7:0]
33	gz[5]
34	bz[4]
39:35	rx[4:0]
40	gz[4]
44:41	gy[3:0]
50:45	gx[5:0]
54:51	gz[3:0]
59:55	bx[4:0]
60	bz[1]
64:61	by[3:0]
69:65	ry[4:0]
70	bz[2]
75:71	rz[4:0]
76	bz[3]
81:77	partition
127:82	indices

Mode 8: (TWO) Red, Green: 8-bit endpoint, 5-bit deltas

Blue: 8-bit endpoint, 6-bit deltas

Bit	Description
4:0	mode = 11010
12:5	rw[7:0]
13	bz[1]
14	by[4]
22:15	gw[7:0]
23	by[5]
24	gy[4]
32:25	bw[7:0]
33	bz[5]
34	bz[4]
39:35	rx[4:0]
40	gz[4]
44:41	gy[3:0]
49:45	gx[4:0]
50	bz[0]
54:51	gz[3:0]
60:55	bx[5:0]
64:61	by[3:0]
69:65	ry[4:0]
70	bz[2]
75:71	rz[4:0]
76	bz[3]
81:77	partition
127:82	indices

Mode 9: (TWO) Red, Green, Blue: 6-bit endpoints for all four, no deltas

Bit	Description
4:0	mode = 11110
10:5	rw[5:0]
11	gz[4]
12	bz[0]
13	bz[1]
14	by[4]
20:15	gw[5:0]
21	gy[5]
22	by[5]
23	bz[2]
24	gy[4]
30:25	bw[5:0]
31	gz[5]
32	bz[3]
33	bz[5]
34	bz[4]
40:35	rx[5:0]
44:41	gy[3:0]
50:45	gx[5:0]
54:51	gz[3:0]
60:55	bx[5:0]
64:61	by[3:0]
70:65	ry[5:0]
76:71	rz[5:0]
81:77	partition
127:82	indices

Mode 10: (ONE) Red, Green, Blue: 10-bit endpoints for both, no deltas

Bit	Description
4:0	mode = 00011
14:5	rw[9:0]
24:15	gw[9:0]
34:25	bw[9:0]
44:35	rx[9:0]
54:45	gx[9:0]
64:55	bx[9:0]
127:65	indices

Mode 11: (ONE) Red, Green, Blue: 11-bit endpoints, 9-bit deltas

Bit	Description
4:0	mode = 00111
14:5	rw[9:0]
24:15	gw[9:0]
34:25	bw[9:0]
43:35	rx[8:0]
44	rw[10]
53:45	gx[8:0]
54	gw[10]
63:55	bx[8:0]
64	bw[10]
127:65	indices

Mode 12: (ONE) Red, Green, Blue: 12-bit endpoints, 8-bit deltas

Bit	Description
4:0	mode = 01011
14:5	rw[9:0]
24:15	gw[9:0]
34:25	bw[9:0]
42:35	rx[7:0]
43	rw[11]
44	rw[10]
52:45	gx[7:0]
53	gw[11]
54	gw[10]
62:55	bx[7:0]
63	bw[11]
64	bw[10]
127:65	indices

Mode 13: (ONE) Red, Green, Blue: 16-bit endpoints, 4-bit deltas

Bit	Description
4:0	mode = 01111
14:5	rw[9:0]
24:15	gw[9:0]
34:25	bw[9:0]
38:35	rx[3:0]
39	rw[15]
40	rw[14]
41	rw[13]
42	rw[12]
43	rw[11]
44	rw[10]
48:45	gx[3:0]
49	gw[15]
50	gw[14]
51	gw[13]
52	gw[12]
53	gw[11]
54	gw[10]
58:55	bx[3:0]
59	bw[15]
60	bw[14]
61	bw[13]
62	bw[12]
63	bw[11]
64	bw[10]
127:65	indices

Undefined mode values (10011, 10111, 11011, and 11111) return zero in the RGB channels.

The “indices” fields are defined as follows:

TWO mode *indices* field with fix-up index [1] at texel[3][3]

Bit	Description
83:82	texel[0][0] index
86:84	texel[0][1] index
89:87	texel[0][2] index
92:90	texel[0][3] index
95:93	texel[1][0] index
98:96	texel[1][1] index
101:99	texel[1][2] index
104:102	texel[1][3] index
107:105	texel[2][0] index
110:108	texel[2][1] index
113:111	texel[2][2] index
116:114	texel[2][3] index
119:117	texel[3][0] index
122:120	texel[3][1] index
125:123	texel[3][2] index
127:126	texel[3][3] index

TWO mode *indices* field with fix-up index [1] at texel[0][2]

Bit	Description
83:82	texel[0][0] index
86:84	texel[0][1] index
88:87	texel[0][2] index
91:89	texel[0][3] index
94:92	texel[1][0] index
97:95	texel[1][1] index
100:98	texel[1][2] index
103:101	texel[1][3] index
106:104	texel[2][0] index
109:107	texel[2][1] index
112:110	texel[2][2] index
115:113	texel[2][3] index
118:116	texel[3][0] index
121:119	texel[3][1] index
124:122	texel[3][2] index
127:125	texel[3][3] index

TWO mode *indices* field with fix-up index [1] at texel[2][0]

Bit	Description
83:82	texel[0][0] index
86:84	texel[0][1] index
89:87	texel[0][2] index
92:90	texel[0][3] index
95:93	texel[1][0] index
98:96	texel[1][1] index
101:99	texel[1][2] index
104:102	texel[1][3] index
106:105	texel[2][0] index
109:107	texel[2][1] index
112:110	texel[2][2] index
115:113	texel[2][3] index
118:116	texel[3][0] index
121:119	texel[3][1] index
124:122	texel[3][2] index
127:125	texel[3][3] index

ONE mode *indices* field

Bit	Description
67:65	texel[0][0] index
71:68	texel[0][1] index
75:72	texel[0][2] index
79:76	texel[0][3] index
83:80	texel[1][0] index
87:84	texel[1][1] index
91:88	texel[1][2] index
95:92	texel[1][3] index
99:96	texel[2][0] index
103:100	texel[2][1] index
107:104	texel[2][2] index
111:108	texel[2][3] index
115:112	texel[3][0] index
119:116	texel[3][1] index
123:120	texel[3][2] index
127:124	texel[3][3] index

Endpoint Computation

The endpoints can be defined in many different ways, as shown above. This section describes how the endpoints are computed from the bits in the compression block. The method used depends on whether the BC6H format is signed (BC6H_SF16) or unsigned (BC6H_UF16).

First, each channel (RGB) of each endpoint is extended to 16 bits. Each is handled identically and independently, however in some modes different channels have different incoming precision which must be accounted for. The following rules are employed:

- If the format is BC6H_SF16 or the endpoint is a delta value, the value is sign-extended to 16 bits
- For all other cases, the value is zero-extended to 16 bits

If there are no endpoints that are delta values, endpoint computation is complete. For endpoints that are delta values, the next step involves computing the absolute endpoint. The “w” endpoint is always absolute and acts as a base value for the other three endpoints. Each channel is handled identically and independently.

```
 x = w + x
y = w + y
z = w + z
```

The above is performed using 16-bit integer arithmetic. Overflows beyond 16 bits are ignored (any resulting high bits are dropped).

Palette Color Computation

The next step involves computing the color palette values that provide the available values for each texel's color. The color palette for each line consists of the two endpoint colors plus 6 (TWO mode) or 14 (ONE mode) interpolated colors. Again each channel is processed independently.

First the endpoints are unquantized, with each channel of each endpoint being processed independently. The number of bits in the original base *w* value represents the precision of the endpoints. The input endpoint is called *e*, and the resulting endpoints are represented as 17-bit signed integers and called e' below.

For the BC6H_UF16 format:

- if the precision is already 16 bits, e' = e
- if e = 0, e' = 0
- if e is the maximum representible in the precision, e' = 0xFFFF
- otherwise, e' = ((e « 16) + 0x8000) » precision

For the BC6H_SF16 format, the value is treated as sign magnitude. The sign is not changed, e' and e refer only to the magnitude portion:

- if the precision is already 16 bits, e' = e
- if e = 0, e' = 0
- if e is the maximum representible in the precision, e' = 0x7FFF
- otherwise, e' = ((e « 15) + 0x4000) » (precision - 1)

Next, the palette values are generated using predefined weights, using the tables below:

```
palette[i] = (w' * (64 - weight[i]) + x' * weight[i] + 32) » 6
```

TWO mode weights:

palette index	0	1	2	3	4	5	6	7
weight	0	9	18	27	37	46	55	64

ONE mode weights:

palette index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
weight	0	4	9	13	17	21	26	30	34	38	43	47	51	55	60	64

The two end palette indices are equal to the two endpoints given that the weights are 0 and 64. In the above equation w' and x' represent the endpoints e' computed in the previous step corresponding to w and x, respectively. For the second line in TWO mode, w and x are replaced with y and z.

The final step in computing the palette colors is to rescale the final results. For BC6H_UF16 format, the values are multiplied by 31/64. For BC6H_SF16, the values are multiplied by 31/32, treating them as sign magnitude. These final 16-bit results are ultimately treated as 16-bit floats.

Texel Selection

The final step is to select the appropriate palette index for each texel. This index then selects the 16-bit per channel palette value, which is re-interpreted as a 16-bit floating point result for input into the filter. This procedure differs depending on whether the mode is TWO or ONE.

ONE Mode

In ONE mode, there is only one set of palette colors, but the “indices” field is 63 bits. This field consists of a 4-bit palette index for each of the 16 texels, with the exception of the texel at [0][0] which has only 3 bits, the missing high bit being set to zero.

TWO Mode

32 partitions are defined for TWO, which are defined below. Each of the 32 cases shows the 4x4 block of texels, and is indexed by adding its hexadecimal row number (00-1C) to its column number (0-3). Each texel in the 4x4 block indicates whether that texel is part of line 0 (endpoints *w* and *x*) or line 1 (endpoints *y* and *z*). Each case has one texel each of "[0]" and "[1]", the index that this is at is termed the "fix-up index". These texels have one less bit in the index.

	0				1				2				3			
00	[0]	0	1	1	[0]	0	0	1	[0]	1	1	1	[0]	0	0	1
	0	0	1	1	0	0	0	1	0	1	1	1	0	0	1	1
	0	0	1	1	0	0	0	1	0	1	1	1	0	0	1	1
	0	0	1	[1]	0	0	0	[1]	0	1	1	[1]	0	1	1	[1]
04	[0]	0	0	0	[0]	0	1	1	[0]	0	0	1	[0]	0	0	0
	0	0	0	1	0	1	1	1	0	0	1	1	0	0	0	1
	0	0	0	1	0	1	1	1	0	1	1	1	0	0	1	1
	0	0	1	[1]	1	1	1	[1]	1	1	1	[1]	0	1	1	[1]
08	[0]	0	0	0	[0]	0	1	1	[0]	0	0	0	[0]	0	0	0
	0	0	0	0	0	1	1	1	0	0	0	1	0	0	0	0
	0	0	0	1	1	1	1	1	0	1	1	1	0	0	0	1
	0	0	1	[1]	1	1	1	[1]	1	1	1	[1]	0	1	1	[1]
0C	[0]	0	0	1	[0]	0	0	0	[0]	0	0	0	[0]	0	0	0
	0	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0
	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
	1	1	1	[1]	1	1	1	[1]	1	1	1	[1]	1	1	1	[1]
10	[0]	0	0	0	[0]	1	[1]	1	[0]	0	0	0	[0]	1	[1]	1
	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1
	1	1	1	0	0	0	0	0	[1]	0	0	0	0	0	0	1
	1	1	1	[1]	0	0	0	0	1	1	1	0	0	0	0	0
14	[0]	0	[1]	1	[0]	0	0	0	[0]	0	0	0	[0]	1	1	1
	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	1
	0	0	0	0	[1]	1	0	0	[1]	0	0	0	0	0	1	1
	0	0	0	0	1	1	1	0	1	1	0	0	0	0	0	[1]
18	[0]	0	[1]	1	[0]	0	0	0	[0]	1	[1]	0	[0]	0	[1]	1
	0	0	0	1	1	0	0	0	0	1	1	0	0	1	1	0
	0	0	0	1	[1]	0	0	0	0	1	1	0	0	1	1	0
	0	0	0	0	1	1	0	0	0	1	1	0	1	1	0	0
1C	[0]	0	0	1	[0]	0	0	0	[0]	1	[1]	1	[0]	0	[1]	1
	0	1	1	1	1	1	1	1	0	0	0	1	1	0	0	1
	[1]	1	1	0	[1]	1	1	1	1	0	0	0	1	0	0	1
	1	0	0	0	0	0	0	0	1	1	1	0	1	1	0	0

The 46-bit "indices" field consists of a 3-bit palette index for each of the 16 texels, with the exception of the bracketed texels that have only two bits each. The high bit of these texels is set to zero.

BC7

These formats (BC7_UNORM and BC7_UNORM_SRGB) compresses 3-channel and 4-channel fixed point images.

The BC7 block is 16 bytes and represents a 4x4 block of texels. The texels are labeled as texel[row][column] where both row and column range from 0 to 3. Texel[0][0] is the upper left texel. BC7 has 8 different modes, the mode that the block is in is contained in the least significant bits (1-8 bits depending on mode).

The basic scheme consists of interpolating colors and alpha in some modes along either one, two, or three lines, with per-texel indices indicating which color/alpha along the line is chosen for each texel. If a two- or three-line mode is selected, one of 64 partition sets is indicated which selects which of the two lines each texel is assigned to, although some modes are limited to the first 16 partition sets. In the color-only modes, alpha is always returned at its default value of 1.0.

Some modes contain the following fields:

- **P-bits.** These represent shared LSB for all components of the endpoint, which increases the endpoint precision by one bit. In some cases both endpoints of a line share a P-bit.
- **Rotation bits.** For blocks with separate color and alpha, this 2-bit field allows selection of which of the four components has its own indexes (scalar) vs. the other three components (vector).
- **Index selector.** This 1-bit field selects whether the scalar or vector components uses the 3-bit index vs. the 2-bit index.

Field Definition

There are 8 possible modes for a BC7 block, the format of each is indicated in the 8 tables below. The mode is selected by the unique mode bits specified in each table. Each mode has particular characteristics described at the top of the table.

Mode 0: Color only, 3 lines (THREE), 4-bit endpoints with one P-bit per endpoint, 3-bit indices, 16 partitions

Bit	Description
0	mode = 0
4:1	partition
8:5	R0
12:9	R1
16:13	R2
20:17	R3
24:21	R4
28:25	R5
32:29	G0
36:33	G1
40:37	G2
44:41	G3
48:45	G4
52:49	G5
56:53	B0
60:57	B1
64:61	B2
68:65	B3
72:69	B4
76:73	B5
77	P0
78	P1
79	P2
80	P3
81	P4
82	P5
127:83	indices

Mode 1: Color only, 2 lines (TWO), 6-bit endpoints with one shared P-bit per line, 3-bit indices, 64 partitions

Bit	Description
1:0	mode = 10
7:2	partition
13:8	R0
19:14	R1
25:20	R2
31:26	R3
37:32	G0
43:38	G1
49:44	G2
55:50	G3
61:56	B0
67:62	B1
73:68	B2
79:74	B3
80	P0
81	P1
127:82	indices

Mode 2: Color only, 3 lines (THREE), 5-bit endpoints, 2-bit indices, 64 partitions

Bit	Description
2:0	mode = 100
8:3	partition
13:9	R0
18:14	R1
23:19	R2
28:24	R3
33:29	R4
38:34	R5
43:39	G0
48:44	G1
53:49	G2
58:54	G3
63:59	G4
68:64	G5
73:69	B0
78:74	B1
83:79	B2
88:84	B3
93:89	B4
98:94	B5
127:99	indices

Mode 3: Color only, 2 lines (TWO), 7-bit endpoints with one P-bit per endpoint, 2-bit indices, 64 partitions

Bit	Description
3:0	mode = 1000
9:4	partition
16:10	R0
23:17	R1
30:24	R2
37:31	R3
44:38	G0
51:45	G1
58:52	G2
65:59	G3
72:66	B0
79:73	B1
86:80	B2
93:87	B3
94	P0
95	P1
96	P2
97	P3
127:98	indices

Mode 4: Color and alpha, 1 line (ONE), 5-bit color endpoints, 6-bit alpha endpoints, 16 2-bit indices, 16 3-bit indices, 2-bit component rotation, 1-bit index selector

Bit	Description
4:0	mode = 10000
6:5	rotation
7	index selector
12:8	R0
17:13	R1
22:18	G0
27:23	G1
32:28	B0
37:33	B1
43:38	A0
49:44	A1
80:50	2-bit indices
127:81	3-bit indices

Mode 5: Color and alpha, 1 line (ONE), 7-bit color endpoints, 8-bit alpha endpoints, 2-bit color indices, 2-bit alpha indices, 2-bit component rotation

Bit	Description
5:0	mode = 100000
7:6	rotation
14:8	R0
21:15	R1
28:22	G0
35:29	G1
42:36	B0
49:43	B1
57:50	A0
65:58	A1
96:66	color indices
127:97	alpha indices

Mode 6: Combined color and alpha, 1 line (ONE), 7-bit endpoints with one P-bit per endpoint, 4-bit indices

Bit	Description
6:0	mode = 1000000
13:7	R0
20:14	R1
27:21	G0
34:28	G1
41:35	B0
48:42	B1
55:49	A0
62:56	A1
63	P0
64	P1
127:65	indices

Mode 7: Combined color and alpha, 2 lines (TWO), 5-bit endpoints with one P-bit per endpoint, 2-bit indices, 64 partitions

Bit	Description
7:0	mode = 10000000
13:8	partition
18:14	R0
23:19	R1
28:24	R2
33:29	R3
38:34	G0
43:39	G1
48:44	G2
53:49	G3
58:54	B0
63:59	B1
68:64	B2
73:69	B3
78:74	A0
83:79	A1
88:84	A2
93:89	A3
94	P0
95	P1
96	P2
97	P3
127:98	indices

Undefined mode values (bits 7:0 = 00000000) return zero in the RGB channels.

The indices fields are variable in length and due to the different locations of the fix-up indices depending on partition set there are a very large number of possible configurations. Each mode above indicates how many bits each index has, and the fix-up indices (one in ONE mode, two in TWO mode, and three in THREE mode) each have one less bit than indicated. However, the indices are always packed into the index fields according to the table below, with the specific bit assignments of each texel following the rules just given.

Bit	Description
LSBs	texel[0][0] index
	texel[0][1] index
	texel[0][2] index
	texel[0][3] index
	texel[1][0] index
	texel[1][1] index
	texel[1][2] index
	texel[1][3] index
	texel[2][0] index
	texel[2][1] index
	texel[2][2] index
	texel[2][3] index
	texel[3][0] index
	texel[3][1] index
	texel[3][2] index
MSBs	texel[3][3] index

Endpoint Computation

The endpoints can be defined with different precision depending on mode, as shown above. This section describes how the endpoints are computed from the bits in the compression block. Each component of each endpoint follows the same steps.

If a P-bit is defined for the endpoint, it is first added as an additional LSB at the bottom of the endpoint value. The endpoint is then bit-replicated to create an 8-bit fixed point endpoint value with a range from 0x00 to 0xFF.

Palette Color Computation

The next step involves computing the color palette values that provide the available values for each texel's color. The color palette for each line consists of the two endpoint colors plus 2, 6, or 14 interpolated colors, depending on the number of bits in the indices. Again each channel is processed independently.

The equation to compute each palette color with index i, given two endpoints is as follows, using the tables below to determine the weight for each palette index:

```
palette[i] = (E0 * (64 - weight[i]) + E1 * weight[i] + 32) » 6
```

2-bit index weights:

palette index	0	1	2	3
weight	0	21	43	64

3-bit index weights:

palette index	0	1	2	3	4	5	6	7
weight	0	9	18	27	37	46	55	64

4-bit index weights:

palette index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
weight	0	4	9	13	17	21	26	30	34	38	43	47	51	55	60	64

The two end palette indices are equal to the two endpoints given that the weights are 0 and 64. In the above equation E0 and E1 represent the even-numbered and odd-numbered endpoints computed in the previous step for the component and line currently being computed.

Texel Selection

The final step is to select the appropriate palette index for each texel. This index then selects the 8-bit per channel palette value, which is interpreted as an 8-bit UNORM value for input into the filter (In BC7_UNORM_SRGB to UNORM values first go through inverse gamma conversion). This procedure differs depending on whether the mode is ONE, TWO, or THREE.

ONE Mode

In ONE mode, there is only one set of palette colors, thus there is only a single “partition set” defined, with all texels selecting line 0 and texel [0][0] being the “fix-up index” with one less bit in the index.

TWO Mode

64 partitions are defined for TWO, which are defined below. Each of the 64 cases shows the 4x4 block of texels, and is indexed by adding its hexadecimal row number (00-3C) to its column number (0-3). Each texel in the 4x4 block indicates whether that texel is part of line 0 (endpoints *0* and *1*) or line 1 (endpoints *2* and *3*). Each case has one texel each of “[0]” and “[1]”, the index that this is at is termed the “fix-up index”. These texels have one less bit in the index.

	0				1				2				3			
00	[0]	0	1	1	[0]	0	0	1	[0]	1	1	1	[0]	0	0	1
	0	0	1	1	0	0	0	1	0	1	1	1	0	0	1	1
	0	0	1	1	0	0	0	1	0	1	1	1	0	0	1	1
	0	0	1	[1]	0	0	0	[1]	0	1	1	[1]	0	1	1	[1]
04	[0]	0	0	0	[0]	0	1	1	[0]	0	0	1	[0]	0	0	0
	0	0	0	1	0	1	1	1	0	0	1	1	0	0	0	1
	0	0	0	1	0	1	1	1	0	1	1	1	0	0	1	1
	0	0	1	[1]	1	1	1	[1]	1	1	1	[1]	0	1	1	[1]
08	[0]	0	0	0	[0]	0	1	1	[0]	0	0	0	[0]	0	0	0
	0	0	0	0	0	1	1	1	0	0	0	1	0	0	0	0
	0	0	0	1	1	1	1	1	0	1	1	1	0	0	0	1
	0	0	1	[1]	1	1	1	[1]	1	1	1	[1]	0	1	1	[1]
0C	[0]	0	0	1	[0]	0	0	0	[0]	0	0	0	[0]	0	0	0
	0	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0
	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
	1	1	1	[1]	1	1	1	[1]	1	1	1	[1]	1	1	1	[1]
10	[0]	0	0	0	[0]	1	[1]	1	[0]	0	0	0	[0]	1	[1]	1
	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1
	1	1	1	0	0	0	0	0	[1]	0	0	0	0	0	0	1
	1	1	1	[1]	0	0	0	0	1	1	1	0	0	0	0	0
14	[0]	0	[1]	1	[0]	0	0	0	[0]	0	0	0	[0]	1	1	1
	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	1
	0	0	0	0	[1]	1	0	0	[1]	0	0	0	0	0	1	1
	0	0	0	0	1	1	1	0	1	1	0	0	0	0	0	[1]
18	[0]	0	[1]	1	[0]	0	0	0	[0]	1	[1]	0	[0]	0	[1]	1
	0	0	0	1	1	0	0	0	0	1	1	0	0	1	1	0
	0	0	0	1	[1]	0	0	0	0	1	1	0	0	1	1	0
	0	0	0	0	1	1	0	0	0	1	1	0	1	1	0	0
1C	[0]	0	0	1	[0]	0	0	0	[0]	1	[1]	1	[0]	0	[1]	1
	0	1	1	1	1	1	1	1	0	0	0	1	1	0	0	1
	[1]	1	1	0	[1]	1	1	1	1	0	0	0	1	0	0	1
	1	0	0	0	0	0	0	0	1	1	1	0	1	1	0	0

	0				1				2				3			
20	[0]	1	0	1	[0]	0	0	0	[0]	1	0	1	[0]	0	1	1
	0	1	0	1	1	1	1	1	1	0	[1]	0	0	0	1	1
	0	1	0	1	0	0	0	0	0	1	0	1	[1]	1	0	0
	0	1	0	[1]	1	1	1	[1]	1	0	1	0	1	1	0	0
24	[0]	0	[1]	1	[0]	1	0	1	[0]	1	1	0	[0]	1	0	1
	1	1	0	0	0	1	0	1	1	0	0	1	1	0	1	0
	0	0	1	1	[1]	0	1	0	0	1	1	0	1	0	1	0
	1	1	0	0	1	0	1	0	1	0	0	[1]	0	1	0	[1]
28	[0]	1	[1]	1	[0]	0	0	1	[0]	0	[1]	1	[0]	0	[1]	1
	0	0	1	1	0	0	1	1	0	0	1	0	1	0	1	1
	1	1	0	0	[1]	1	0	0	0	1	0	0	1	1	0	1
	1	1	1	0	1	0	0	0	1	1	0	0	1	1	0	0
2C	[0]	1	[1]	0	[0]	0	1	1	[0]	1	1	0	[0]	0	0	0
	1	0	0	1	1	1	0	0	0	1	1	0	0	1	[1]	0
	1	0	0	1	1	1	0	0	1	0	0	1	0	1	1	0
	0	1	1	0	0	0	1	[1]	1	0	0	[1]	0	0	0	0
30	[0]	1	0	0	[0]	0	[1]	0	[0]	0	0	0	[0]	0	0	0
	1	1	[1]	0	0	1	1	1	0	0	[1]	0	0	1	0	0
	0	1	0	0	0	0	1	0	0	1	1	1	[1]	1	1	0
	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0
34	[0]	1	1	0	[0]	0	1	1	[0]	1	[1]	0	[0]	0	[1]	1
	1	1	0	0	0	1	1	0	0	0	1	1	1	0	0	1
	1	0	0	1	1	1	0	0	1	0	0	1	1	1	0	0
	0	0	1	[1]	1	0	0	[1]	1	1	0	0	0	1	1	0
38	[0]	1	1	0	[0]	1	1	0	[0]	1	1	1	[0]	0	0	1
	1	1	0	0	0	0	1	1	1	1	1	0	1	0	0	0
	1	1	0	0	0	0	1	1	1	0	0	0	1	1	1	0
	1	0	0	[1]	1	0	0	[1]	0	0	0	[1]	0	1	1	[1]
3C	[0]	0	0	0	[0]	0	[1]	1	[0]	0	[1]	0	[0]	1	0	0
	1	1	1	1	0	0	1	1	0	0	1	0	0	1	0	0
	0	0	1	1	1	1	1	1	1	1	1	0	0	1	1	1
	0	0	1	[1]	0	0	0	0	1	1	1	0	0	1	1	[1]

THREE Mode

64 partitions are defined for THREE, which are defined below. Each of the 64 cases shows the 4x4 block of texels, and is indexed by adding its hexadecimal row number (00-3C) to its column number (0-3). Each texel in the 4x4 block indicates whether that texel is part of line 0 (endpoints *0* and *1*), line 1 (endpoints *2* and *3*), or line 2 (endpoints *4* and *5*). Each case has one texel each of "[0]", "[1]", and "[2]", the index that this is at is termed the "fix-up index". These texels have one less bit in the index.

	0				1				2				3			
00	[0]	0	1	[1]	[0]	0	0	[1]	[0]	0	0	0	[0]	2	2	[2]
	0	0	1	1	0	0	1	1	2	0	0	1	0	0	2	2
	0	2	2	1	[2]	2	1	1	[2]	2	1	1	0	0	1	1
	2	2	2	[2]	2	2	2	1	2	2	1	[1]	0	1	1	[1]
04	[0]	0	0	0	[0]	0	1	[1]	[0]	0	2	[2]	[0]	0	1	1
	0	0	0	0	0	0	1	1	0	0	2	2	0	0	1	1
	[1]	1	2	2	0	0	2	2	1	1	1	1	[2]	2	1	1
	1	1	2	[2]	0	0	2	[2]	1	1	1	[1]	2	2	1	[1]
08	[0]	0	0	0	[0]	0	0	0	[0]	0	0	0	[0]	0	1	2
	0	0	0	0	1	1	1	1	1	1	[1]	1	0	0	[1]	2
	[1]	1	1	1	[1]	1	1	1	2	2	2	2	0	0	1	2
	2	2	2	[2]	2	2	2	[2]	2	2	2	[2]	0	0	1	[2]
0C	[0]	1	1	2	[0]	1	2	2	[0]	0	1	[1]	[0]	0	1	[1]
	0	1	[1]	2	0	[1]	2	2	0	1	1	2	2	0	0	1
	0	1	1	2	0	1	2	2	1	1	2	2	[2]	2	0	0
	0	1	1	[2]	0	1	2	[2]	1	2	2	[2]	2	2	2	0
10	[0]	0	0	[1]	[0]	1	1	[1]	[0]	0	0	0	[0]	0	2	[2]
	0	0	1	1	0	0	1	1	1	1	2	2	0	0	2	2
	0	1	1	2	[2]	0	0	1	[1]	1	2	2	0	0	2	2
	1	1	2	[2]	2	2	0	0	1	1	2	[2]	1	1	1	[1]
14	[0]	1	1	[1]	[0]	0	0	[1]	[0]	0	0	0	[0]	0	0	0
	0	1	1	1	0	0	0	1	0	0	[1]	1	1	1	0	0
	0	2	2	2	[2]	2	2	1	0	1	2	2	[2]	2	[1]	0
	0	2	2	[2]	2	2	2	1	0	1	2	[2]	2	2	1	0
18	[0]	1	2	[2]	[0]	0	1	2	[0]	1	1	0	[0]	0	0	0
	0	[1]	2	2	0	0	1	2	1	2	[2]	1	0	1	[1]	0
	0	0	1	1	[1]	1	2	2	[1]	2	2	1	1	2	[2]	1
	0	0	0	0	2	2	2	[2]	0	1	1	0	1	2	2	1
1C	[0]	0	2	2	[0]	1	1	0	[0]	0	1	1	[0]	0	0	0
	1	1	0	2	0	[1]	1	0	0	1	2	2	2	0	0	0

	0				1				2				3			
	[1]	1	0	2	2	0	0	2	0	1	[2]	2	[2]	2	1	1
	0	0	2	[2]	2	2	2	[2]	0	0	1	[1]	2	2	2	[1]
20	[0]	0	0	0	[0]	2	2	[2]	[0]	0	1	[1]	[0]	1	2	0
	0	0	0	2	0	0	2	2	0	0	1	2	0	[1]	2	0
	[1]	1	2	2	0	0	1	2	0	0	2	2	0	1	[2]	0
	1	2	2	[2]	0	0	1	[1]	0	2	2	[2]	0	1	2	0
24	[0]	0	0	0	[0]	1	2	0	[0]	1	2	0	[0]	0	1	1
	1	1	[1]	1	1	2	0	1	2	0	1	2	2	2	0	0
	2	2	[2]	2	[2]	0	[1]	2	[1]	[2]	0	1	1	1	[2]	2
	0	0	0	0	0	1	2	0	0	1	2	0	0	0	1	[1]
28	[0]	0	1	1	[0]	1	0	[1]	[0]	0	0	0	[0]	0	2	2
	1	1	[2]	2	0	1	0	1	0	0	0	0	1	[1]	2	2
	2	2	0	0	2	2	2	2	[2]	1	2	1	0	0	2	2
	0	0	1	[1]	2	2	2	[2]	2	1	2	[1]	1	1	2	[2]
2C	[0]	0	2	[2]	[0]	2	2	0	[0]	1	0	1	[0]	0	0	0
	0	0	1	1	1	2	[2]	1	2	2	[2]	2	2	1	2	1
	0	0	2	2	0	2	2	0	2	2	2	2	[2]	1	2	1
	0	0	1	[1]	1	2	2	[1]	0	1	0	[1]	2	1	2	[1]
30	[0]	1	0	[1]	[0]	2	2	[2]	[0]	0	0	2	[0]	0	0	0
	0	1	0	1	0	1	1	1	1	[1]	1	2	2	[1]	1	2
	0	1	0	1	0	2	2	2	0	0	0	2	2	1	1	2
	2	2	2	[2]	0	1	1	[1]	1	1	1	[2]	2	1	1	[2]
34	[0]	2	2	2	[0]	0	0	2	[0]	1	1	0	[0]	0	0	0
	0	[1]	1	1	1	1	1	2	0	[1]	1	0	0	0	0	0
	0	1	1	1	[1]	1	1	2	0	1	1	0	2	1	[1]	2
	0	2	2	[2]	0	0	0	[2]	2	2	2	[2]	2	1	1	[2]
38	[0]	1	1	0	[0]	0	2	2	[0]	0	2	2	[0]	0	0	0
	0	[1]	1	0	0	0	1	1	1	1	2	2	0	0	0	0
	2	2	2	2	0	0	[1]	1	[1]	1	2	2	0	0	0	0
	2	2	2	[2]	0	0	2	[2]	0	0	2	[2]	2	[1]	1	[2]
3C	[0]	0	0	[2]	[0]	2	2	2	[0]	1	0	[1]	[0]	1	1	[1]
	0	0	0	1	1	2	2	2	2	2	2	2	2	0	1	1
	0	0	0	2	0	2	2	2	2	2	2	2	[2]	2	0	1
	0	0	0	[1]	[1]	2	2	[2]	2	2	2	[2]	2	2	2	0

Video Pixel/Texel Formats

This section describes the "video" pixel/texel formats with respect to memory layout. See the Overlay chapter for a description of how the Y, U, V components are sampled.

Packed Memory Organization

Color components are all 8 bits in size for YUV formats. For YUV 4:2:2 formats each DWord will contain two pixels and only the byte order affects the memory organization.

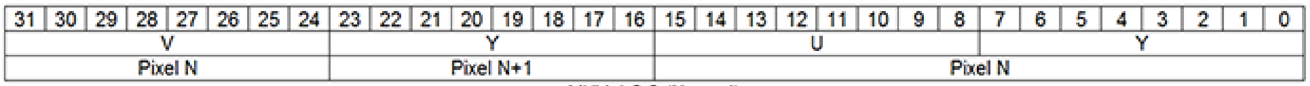
The following four YUV 4:2:2 surface formats are supported, listed with alternate names:

- YCRCB_NORMAL (YUYV/YUY2)
- YCRCB_SWAPUVY (VYUY) (R8G8_B8G8_UNORM)
- YCRCB_SWAPUV(YVYU) (G8R8_G8B8_UNORM)
- YCRCB_SWAPY (UYVY)

The channels are mapped as follows:

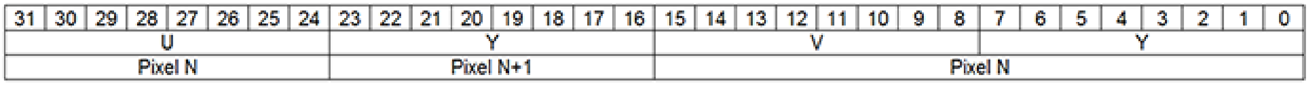
Cr (V)	Red
Y	Green
Cb (U)	Blue

Memory layout of packed YUV 4:2:2 formats



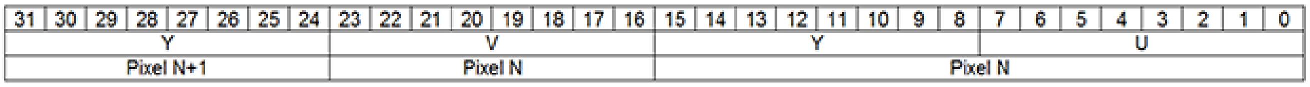
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
V								Y								U								Y							
Pixel N								Pixel N+1								Pixel N															

YUV 4:2:2 (Normal)



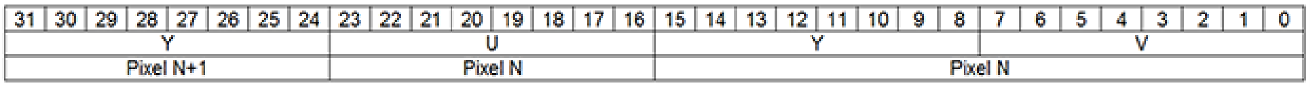
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
U								Y								V								Y							
Pixel N								Pixel N+1								Pixel N															

YUV 4:2:2 (UV Swap)



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Y								V								Y								U							
Pixel N+1								Pixel N								Pixel N															

YUV 4:2:2 (Y Swap)



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Y								U								Y								V							
Pixel N+1								Pixel N								Pixel N															

YUV 4:2:2 (UV/Y Swap)

Planar Memory Organization

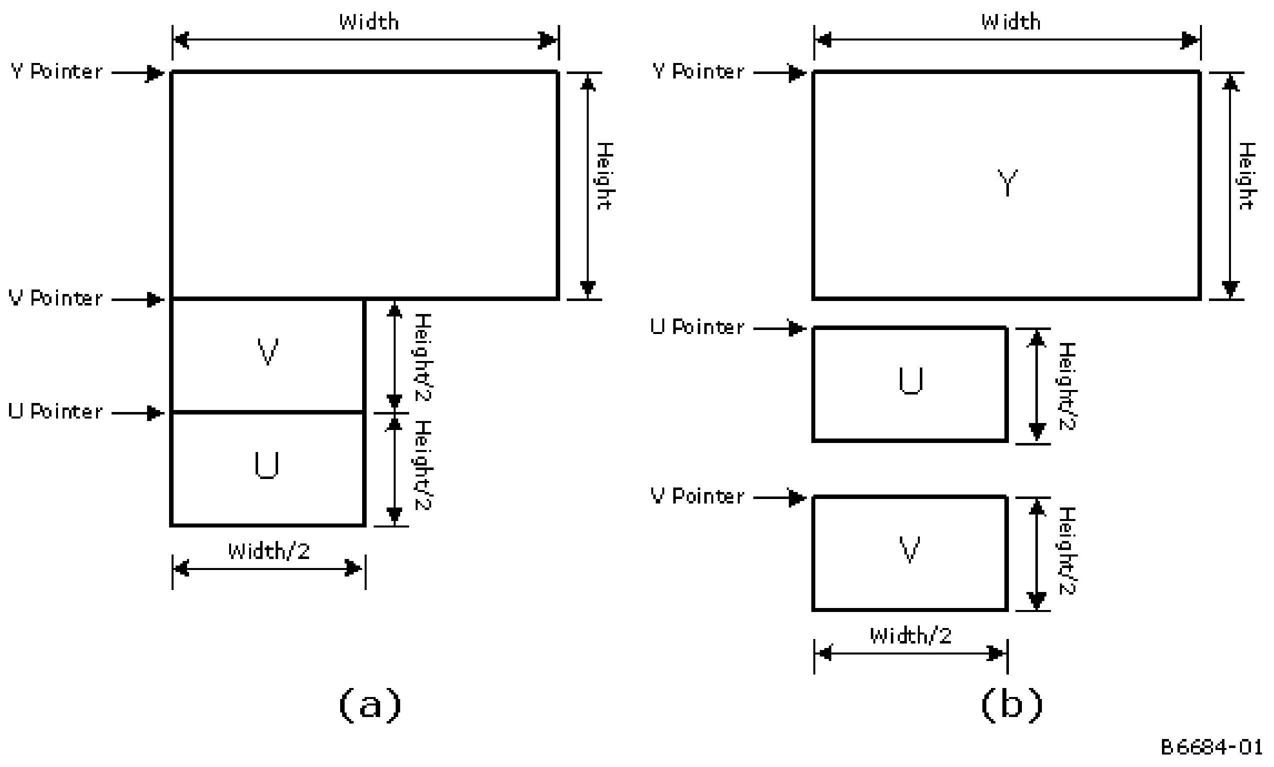
Planar formats use what could be thought of as separate buffers for the three color components. Because there is a separate stride for the Y and U/V data buffers, several memory footprints can be supported.

Note: There is no direct support for use of planar video surfaces as textures. The sampling engine can be used to operate on each of the 8bpp buffers separately (via a single-channel 8-bit format such as I8_UNORM). The U and V buffers can be written concurrently by using multiple render targets from the pixel shader. The Y buffer must be written in a separate pass due to its different size.

The following figure shows two types of memory organization for the YUV 4:2:0 planar video data:

1. The memory organization of the common YV12 data, where all three planes are contiguous and the strides of U and V components are half of that of the Y component.
2. An alternative memory structure that the addresses of the three planes are independent but satisfy certain alignment restrictions.

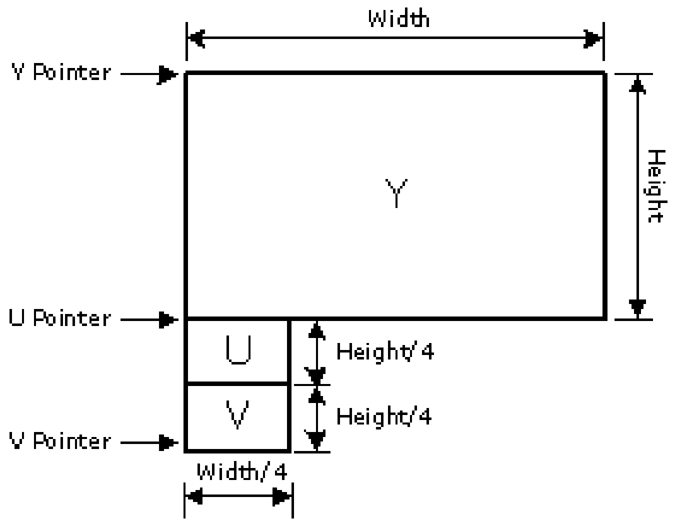
YUV 4:2:0 Format Memory Organization



The following figure shows memory organization of the planar YUV 4:1:0 format where the planes are contiguous.

Note: The chroma planes (U and V), when separate (case b above) are treated as half-pitch with respect to the Y plane. In general, YV12 is supported only in linear format because separate planes cannot be supported correctly with a tiled format.

YUV 4:1:0 Format Memory Organization



B6685-01

The table below shows how position within a Planar YUV surface chroma plane is calculated for various cases ot U and V pitch and position. It also shows restrictions on the alignment of the planes in memory.

Case	Interleave Chroma	Pitch	Vertical U/V Offset	Restrictions

YUV with Half Pitch Chroma	No	Half	When U is below Y
			Y_Uoffset = Y_Height * 2
			Y_Voffset = Y_Height * 2 + V_Height
			When V is below Y
			Y_Uoffset = Y_Height * 2 + V_Height
			Y_Voffset = Y_Height * 2
				(Y Height)%4 = 0
				(U Height)%2 = 0
				(V Height)%2 = 0
				Vertical for Y surface must be 0

YUV with Full Pitch Chroma	Yes	Full	When U is below Y
			Y_Uoffset = Y_Height
			Y_Voffset = Y_Height + V_Height
			When V is below Y
			Y_Uoffset = Y_Height + V_Height
			Y_Voffset = Y_Height
				(Y Height)%2 = 0
				(U Height)%2 = 0
				(V Height)%2 = 0

YUV for Media Sampling	Yes	Always Full	Same as 3D full pitch	Same as 3D full pitch

Raw Format

RAW format is supported only with the untyped surface read/write and atomic operation data port messages. It means that the surface has no inherent format. Surfaces of type RAW are addressed with byte-based offsets that must be DWord-aligned (multiple of 4). Data is returned in DWord quantities. The RAW surface format can be applied only to surface types of BUFFER and STRBUF.

Surface Memory Organizations

See *Memory Interface Functions* chapter for a discussion of tiled vs. linear surface formats.

Display, Overlay, Cursor Surfaces

These surfaces are memory image buffers (planes) used to refresh a display device in non-VGA mode. See the Display chapter for specifics on how these surfaces are defined/used.

2D Render Surfaces

These surfaces are used as general source and/or destination operands in 2D BLT operations.

Note that there is no coherency between 2D render surfaces and the texture cache. Software must explicitly invalidate the texture cache before using a texture that has been modified via the BLT engine.

See the 2D Instruction and 2D Rendering chapters for specifics on how these surfaces are used, restrictions on their size, placement, etc.

2D Monochrome Source

These 1 BPP (bit per pixel) surfaces are used as source operands to certain 2D BLT operations, where the BLT engine expands the 1 BPP source to the required color depth.

The texture cache stores any monochrome sources. There is no mechanism to maintain coherency between 2D render surfaces and texture-cached monochrome sources. Software must explicitly invalidate the texture cache before using a memory-based monochrome source that has been modified via the BLT engine. (Here the assumption is that SW enforces memory-based monochrome source surfaces as read-only surfaces.)

See the 2D Instruction and 2D Rendering chapters for specifics on how these surfaces are used, restrictions on their size, placement, coherency rules, etc.

2D Color Pattern

Color pattern surfaces are used as special pattern operands in 2D BLT operations.

The device uses the texture cache to store color patterns. There is no mechanism to maintain coherency between 2D render surfaces and (texture)-cached color patterns. Software is required to explicitly invalidate the texture cache before using a memory-based color pattern that has been modified via the BLT engine. (Here the assumption is that SW enforces memory-based color pattern surfaces as read-only surfaces.)

See the *2D Instruction* and *2D Rendering* chapters for specifics on how these surfaces are used, restrictions on their size, placement, etc.

3D Color Buffer (Destination) Surfaces

3D Color Buffer surfaces hold per-pixel color values for use in the 3D Pipeline. The 3D Pipeline always requires a Color Buffer to be defined.

See the Non-Video Pixel/Texel Formats section in this chapter for details on the Color Buffer pixel formats. See the 3D Instruction and 3D Rendering chapters for Color Buffer usage details.

The Color Buffer is defined as the BUFFERID_COLOR_BACK memory buffer via the 3DSTATE_BUFFER_INFO instruction. That buffer can be mapped to LM or SM (snooped or unsnooped), and can be linear or tiled. When both the Depth and Color Buffers are tiled, the respective Tile Walk directions must match.

When a linear Color Buffer and a linear Depth Buffer are used together:

- The buffers may have different pitches, though both pitches must be a multiple of 32 bytes.
- The buffers must be co-aligned with a 32-byte region.

3D Depth Buffer Surfaces

Depth Buffer surfaces hold per-pixel depth values and per-pixel stencil values for use in the 3D Pipeline. The 3D Pipeline does not require a Depth Buffer in general, though a Depth Buffer is required to perform non-trivial Depth Test and Stencil Test operations.

The Depth Buffer is specified via the 3DSTATE_DEPTH_BUFFER command. See the description of that instruction in *Windower* for restrictions.

See *Depth Buffer Formats* below for a summary of the possible depth buffer formats. See the Depth Buffer Formats section in this chapter for details on the pixel formats. See the *Windower* and *DataPort* chapters for details on the usage of the Depth Buffer.

Depth Buffer Formats

DepthBufferFormat / DepthComponent	BPP (Bits Per Pixel)	Description
D32_FLOAT_S8X24_UINT	64	32-bit floating point Z depth value in first DWord, 8-bit stencil in lower byte of second DWord
D32_FLOAT	32	32-bit floating point Z depth value
D24_UNORM_S8_UINT	32	24-bit fixed point Z depth value in lower 3 bytes, 8-bit stencil value in upper byte
D16_UNORM	16	16-bit fixed point Z depth value

3D Separate Stencil Buffer Surfaces

Separate Stencil Buffer surfaces hold per-pixel stencil values for use in the 3D Pipeline. Note that the 3D Pipeline does not require a Stencil Buffer to be allocated, though a Stencil Buffer is required to perform non-trivial Stencil Test operations.

UNRESOLVED CROSS REFERENCE, Depth Buffer Formats summarizes Stencil Buffer formats. Refer to the Stencil Buffer Formats section in this chapter for details on the pixel formats. Refer to the *Windower* chapters for Stencil Buffer usage details.

The Stencil buffer is specified via the 3DSTATE_STENCIL_BUFFER command. See that instruction description in *Windower* for restrictions.

Depth Buffer Formats

DepthBufferFormat / DepthComponent	BPP (bits per pixel)	Description
R8_ UNIT	8	8-bit stencil value in a byte

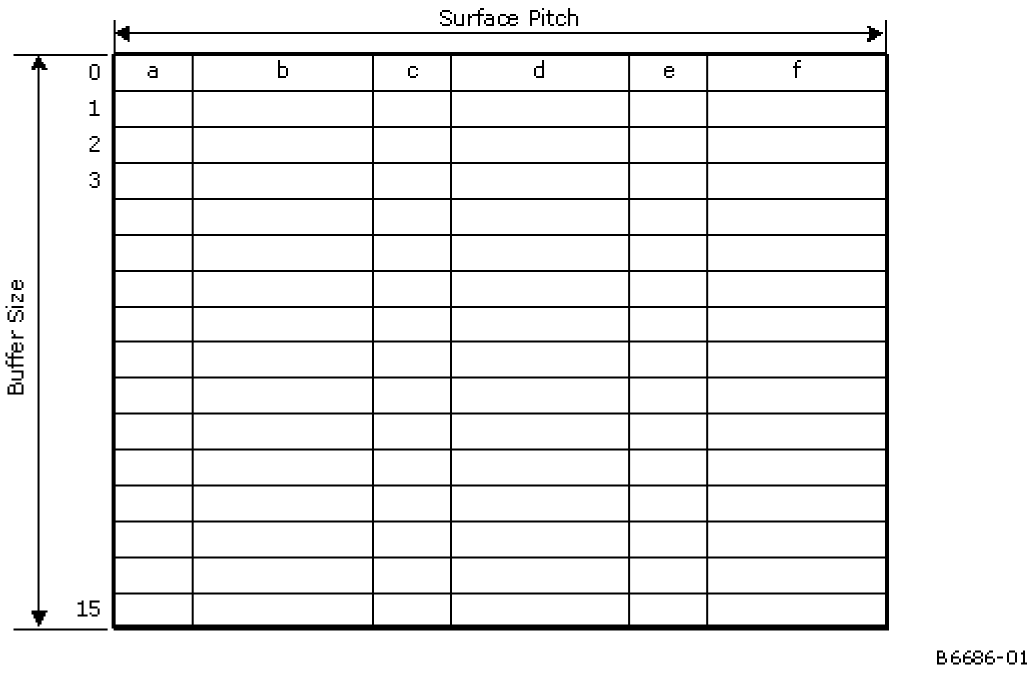
Surface Layout

In addition to restrictions on maximum height, width, and depth, surfaces are also restricted to a maximum size in bytes. This maximum is 2 GB for all products and all surface types.

Buffers

A buffer is an array of structures. Each structure contains up to 2048 bytes of elements. Each element is a single surface format using one of the supported surface formats depending on how the surface is being accessed. The surface pitch state for the surface specifies the size of each structure in bytes.

The buffer is stored in memory contiguously with each element in the structure packed together, and the first element in the next structure immediately following the last element of the previous structure. Buffers are supported only in linear memory.



Structured Buffers

A structured buffer is a surface type that is accessed by a 2-dimensional coordinate. It can be thought of as an array of structures, where each structure is a predefined number of DWords in size. The first coordinate (U) defines the array index, and the second coordinate (V) is a byte offset into the structure which must be a multiple of 4 (DWord-aligned). A structured buffer must be defined with **Surface Format** RAW.

The structured buffer has only one dimension programmed in SURFACE_STATE which indicates the array size. The byte offset dimension (V) is assumed to be bounded only by the **Surface Pitch**.

1D Surfaces

One-dimensional surfaces are identical to 2D surfaces with height of one. Arrays of 1D surfaces are also supported. Please refer to the 2D Surfaces section for details on how these surfaces are stored.

2D Surfaces

Surfaces that comprise texture mip-maps are stored in a fixed “monolithic” format and referenced by a single base address. The base map and associated mipmaps are located within a single rectangular area of memory identified by the base address of the upper left corner and a pitch. The base address references the upper left corner of the base map. The pitch must be specified at least as large as the widest mip-map. In some cases it must be wider; see the section on Minimum Pitch below.

These surfaces may be overlapped in memory and must adhere to the following memory organization rules:

- For non-compressed texture formats, each mipmap must start on an even row within the monolithic rectangular area. For 1-texel-high mipmaps, this may require a row of padding below the previous mipmap. This restriction does not apply to any compressed texture formats; each subsequent (lower-res) compressed mipmap is positioned directly below the previous mipmap.
- Vertical alignment restrictions vary with memory tiling type: 1 DWord for linear, 16-byte (DQWord) for tiled. (Note that tiled mipmaps are *not* required to start at the left edge of a tile row.)

Computing MIP Level Sizes

Map width and height specify the size of the largest MIP level (LOD 0). Less detailed LOD level (i+1) sizes are determined by dividing the width and height of the current (i) LOD level by 2 and truncating to an integer (floor). This is equivalent to shifting the width/height by 1 bit to the right and discarding the bit shifted off. The map height and width are clamped on the low side at 1.

In equations, the width and height of an LOD "*L*" can be expressed as:

$W_L = ((width \gg L) > 0 ?\ width \gg L : 1)$

$H_L = ((height \gg L) > 0 ?\ height \gg L : 1)$

If the surface is multisampled and it is a depth or stencil surface or **Multisampled Surface StorageFormat** in SURFACE_STATE is MSFMT_DEPTH_STENCIL, W_L and H_L must be adjusted as follows before proceeding:

Number of Multisamples	W_L =	H_L =
2	ceiling(W_L / 2) * 4	H_L [no adjustment]
4	ceiling(W_L / 2) * 4	ceiling(H_L / 2) * 4
8	ceiling(W_L / 2) * 8	ceiling(H_L / 2) * 4d
16	ceiling(W_L / 2) * 8	ceiling(H_L / 2) * 8

Base Address for LOD Calculation

It is conceptually easier to think of the space that the map uses in Cartesian space (x, y), where x and y are in units of texels, with the upper left corner of the base map at (0, 0). The final step is to convert from Cartesian coordinates to linear addresses as documented at the bottom of this section.

It is useful to think of the concept of "stepping" when considering where the next MIP level will be stored in rectangular memory space. We either step down or step right when moving to the next higher LOD.

- for MIPLAYOUT_RIGHT maps:
  - step right when moving from LOD 0 to LOD 1
  - step down for all of the other MIPs
- for MIPLAYOUT_BELOW maps:
  - step down when moving from LOD 0 to LOD 1
  - step right when moving from LOD 1 to LOD 2
  - step down for all of the other MIPs

To account for the cache line alignment required, we define *i* and *j* as the width and height, respectively, of an *alignment unit*. This alignment unit is defined below. We then define lower-case w_L and h_L as the padded width and height of LOD "*L*" as follows:

$$w_L = i * ceil\left(\frac{W_L}{i}\right)$$

$$h_L = j * ceil\left(\frac{H_L}{j}\right)$$

For separate stencil buffer, the width must be mutiplied by 2 and height divided by 2 as follows:

$$w_l = 2 * i * ceil\left(\frac{W_L}{i}\right)$$

$$h_L = 1/2 * j * ceil\left(\frac{H_L}{j}\right)$$

Equations to compute the upper left corner of each MIP level are then as follows:

for *MIPLAYOUT_RIGHT* maps:

$LOD_0 = (0,0)$

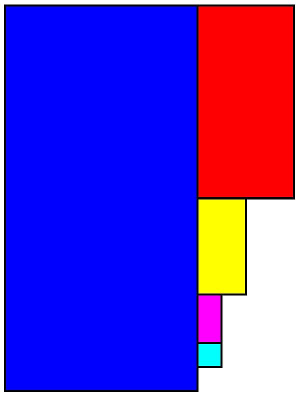
$LOD_1 = (w_0, 0)$

$LOD_2 = (w_0, h_1)$

$LOD_3 = (w_0, h_1 + h_2)$

$LOD_4 = (w_0, h_1 + h_2 + h_3)$

...



for *MIPLAYOUT_BELOW* maps:

$LOD_0 = (0,0)$

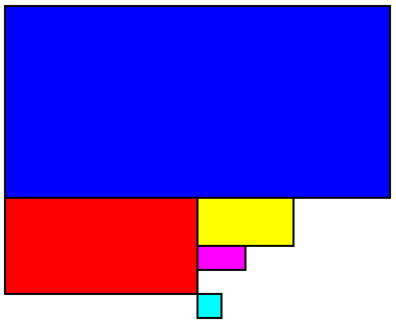
$LOD_1 = (0, h_0)$

$LOD_2 = (w_1, h_0)$

$LOD_3 = (w_1, h_0 + h_2)$

$LOD_4 = (w_1, h_0 + h_2 + h_3)$

...



Minimum Pitch for MIPLAYOUT_RIGHT and Other Maps

For MIPLAYOUT_RIGHT maps, the minimum pitch must be calculated before choosing a fence to place the map within. This is approximately equal to 1.5x the pitch required by the base map, with possible adjustments made for cache line alignment. For MIPLAYOUT_BELOW and MIPLAYOUT_LEGACY maps, the minimum pitch required is equal to that required by the base (LOD 0) map.

A safe but simple calculation of minimum pitch is equal to 2x the pitch required by the base map for MIPLAYOUT_RIGHT maps. This ensures that enough pitch is available, and since it is restricted to MIPLAYOUT_RIGHT maps, not much memory is wasted. It is up to the driver (hardware independent) whether to use this simple determination of pitch or a more complex one.

Cartesian to Linear Address Conversion

A set of variables are defined in addition to the i and j defined above.

- b = bytes per texel of the native map format (0.5 for DXT1, FXT1, and 4-bit surface format, 2.0 for YUV 4:2:2, others aligned to surface format)
- t = texel rows / memory row (4 for DXT1-5 and FXT1, 1 for all other formats)
- p = pitch in bytes (equal to pitch in dwords * 4)
- B = base address in bytes (address of texel 0,0 of the base map)
- x, y = cartesian coordinates from the above calculations in units of texels (assumed that x is always a multiple of i and y is a multiple of j)
- A = linear address in bytes

$$A = B + \frac{yp}{t} + xbt$$

This calculation gives the linear address in bytes for a given MIP level (taking into account L1 cache line alignment requirements).

Compressed Mipmap Layout

Mipmaps of textures using compressed (DXTn, FXT) texel formats are also stored in a monolithic format. The compressed mipmaps are stored in a similar fashion to uncompressed mipmaps, with each block of source (uncompressed) texels represented by a 1 or 2 QWord compressed block. The compressed blocks occupy the same logical positions as the texels they represent, where each row of compressed blocks represent a 4-high row of uncompressed texels. The format of the blocks is preserved, i.e., there is no “intermediate” format as required on some other devices.

The following exceptions apply to the layout of compressed (vs. uncompressed) mipmaps:

- Mipmaps are not required to start on even rows, therefore each successive mip level is located on the texel row immediately below the last row of the previous mip level. Pad rows are neither required nor allowed.
- The dimensions of the mip maps are first determined by applying the sizing algorithm presented in Non-Power-of-Two Mipmaps above. Then, if necessary, they are padded out to compression block boundaries.

Surface Arrays

Arrays of 1D and 2D surfaces can be treated as a single surface. This section covers the layout of these composite surfaces.

For All Surface Other Than Separate Stencil Buffer

Both 1D and 2D surfaces can be specified as an array. The only difference in the surface state is the presence of a depth value greater than one, indicating multiple array “slices”.

A value *QPitch* is defined which indicates the worst-case height for one slice in the texture array. This *QPitch* is multiplied by the array index to and added to the vertical component of the address to determine the vertical component of the address for that slice. Within the slice, the map is stored identically to a MIPLAYOUT_BELOW 2D surface. *MIPLAYOUT_BELOW is the only format supported by 1D non-arrays and both 2D and 1D arrays, the programming of the MIP Map Layout Mode state variable is ignored when using a TextureArray.*

The following equation is used for surface formats other than compressed textures:

$$QPitch = (h_0 + h_1 + 11j)$$

The input variables in this equation are defined in sections above.

The equation for compressed textures (BC* and FXT1 surface formats) follows:

$$QPitch = \frac{(h_0 + h_1 + 11j)}{4}$$

For All Surfaces

Both 1D and 2D surfaces can be specified as an array. An array surface is indicated by enabling the **Surface Array** field in SURFACE_STATE. 2D multisampled surfaces with Multisampled **Surface Storage Format** set to MSFMT_MSS also are stored like an array in memory.

A value QPitch is defined which indicates the worst-case height for one slice in the texture array. This QPitch is multiplied by the array index to and added to the vertical component of the address to determine the vertical component of the address for that slice. Within the slice, the map is stored identically to a 2D surface.

Since cube surfaces are stored identically to 2D arrays, QPitch is used to determine the spacing between faces of the cube.

For CMS/UMS multisampled surfaces, QPitch is used to determine the spacing between sample slices. For IMS multisampled surfaces, QPitch must account for the additional slice size due to sample storage.

For surfaces defined with SURFACE_STATE, The QPitch field in this state defines the value of QPitch. Software must ensure that QPitch is sufficiently large to avoid overlap between array slices in the memory layout. If an auxiliary surface is defined, a separate QPitch must be set for that surface.

For *depth* and stencil surfaces, QPitch is set in the corresponding state command.

Multisampled Surfaces

Multisampled render targets and sampling engine surfaces are supported. There are three types of multisampled surface layouts designated as follows:

- **IMS** Interleaved Multisampled Surface
- **CMS** Compressed Mulitsampled Surface
- **UMS** Uncompressed Multisampled Surface

These surface layouts are described in the following sections.

Compressed Multisampled Surfaces

Multisampled render targets can be compressed. If **MCS Enable** is enabled in SURFACE_STATE, hardware handles the compression using a software-invisible algorithm. However, performance optimizations in the multisample resolve kernel using the sampling engine are possible if the internal format of these surfaces is understood by software. This section documents the formats of the Multisample Control Surface (MCS) and Multisample Surface (MSS).

The MCS surface consists of one element per pixel, with the element size being an 8-bit unsigned integer value for 4x multisampled surfaces and a 32-bit unsigned integer value for 8x multisampled surfaces. Each field within the element indicates which sample slice (SS) the sample resides on.

For BDW, the 2x MCS is 8 bits per pixel. The 8 bits are encoded as follows:

2x MCS

7:2	1	0
reserved	sample 1 SS	sample 0 SS

Each 1-bit field indicates which sample slice (SS) the sample's color value is stored. An MCS value of 0x00 indicates that both samples are stored in sample slice 0 (thus have the same color). This is the fully compressed case. An MCS value of 0x03 indicates that all samples in the pixel are in the clear state and none of the sample slices are valid. The pixel's color must be replaced with the surface's clear value.

For BDW, the 4x MCS is 8 bits per pixel. The 8 bits are encoded as follows:

4x MCS

7:6	5:4	3:2	1:0
sample 3 SS	sample 2 SS	sample 1 SS	sample 0 SS

Each 2-bit field indicates which sample slice (SS) the sample's color value is stored. An MCS value of 0x00 indicates that all four samples are stored in sample slice 0 (thus all have the same color). This is the fully compressed case. An MCS value of 0xff indicates that all samples in the pixel are in the clear state, and none of the sample slices are valid. The pixel's color must be replaced with the surface's clear value.

For BDW, extending the mechanism used for the 4x MCS to 8x requires 3 bits per sample times 8 samples, or 24 bits per pixel. The 24-bit MCS value per pixel is placed in a 32-bit footprint, with the upper 8 bits unused as shown below.

8x MCS

31:24	23:21	20:18	17:15	14:12	11:9	8:6	5:3	2:0
reserved	sample 7 SS	sample 6 SS	sample 5 SS	sample 4 SS	sample 3 SS	sample 2 SS	sample 1 SS	sample 0 SS

Other than this, the 8x algorithm is the same as the 4x algorithm. The MCS value indicating clear state is 0x00ffffff.

Physical MSS Surface

The physical MSS surface is stored identically to a 2D array surface, with the height and width matching the *pixel* dimensions of the logical multisampled surface. The number of array slices in the physical surface is 2, 4, 8, or 16 times that of the logical surface (depending on the number of multisamples). Sample slices belonging to the same logical surface array slice are stored in adjacent physical slices. The sampling engine *ld2dss* message gives direct access to a specific sample slice.

Uncompressed Multisampled Surfaces

UMS surfaces similar to CMS, except that the MCS is disabled, and there is no MCS surface. UMS contains only an MSS surface, where each sample is stored on its sample slice (SS) of the same index.

Cube Surfaces

The 3D Pipeline supports *cubic environment maps*, conceptually arranged as a cube surrounding the origin of a 3D coordinate system aligned to the cube faces. These maps can be used to supply texel (color/alpha) data of the environment in any direction from the enclosed origin, where the direction is supplied as a 3D “vector” texture coordinate. These cube maps can also be mipmapped.

Each texture map level is represented as a group of six, square *cube face* texture surfaces. The faces are identified by their relationship to the 3D texture coordinate system. The subsections below describe the cube maps as described at the API as well as the memory layout dictated by the hardware.

Hardware Cube Map Layout

The cube face textures are stored in the same way as 2D array surfaces are stored (see section *2D Surfaces*for details). For cube surfaces, the depth (array instances) is equal to 6. The array index “q” corresponds to the face according to the following table:

“q” coordinate	face
0	+x
1	-x
2	+y
3	-y
4	+z
5	-z

Restrictions

- The cube map memory layout is the same whether or not the cube map is mip-mapped, and whether or not all six faces are “enabled”, though the memory backing disabled faces or non-supplied levels can be used by software for other purposes.
- The cube map faces all share the same **Surface Format**

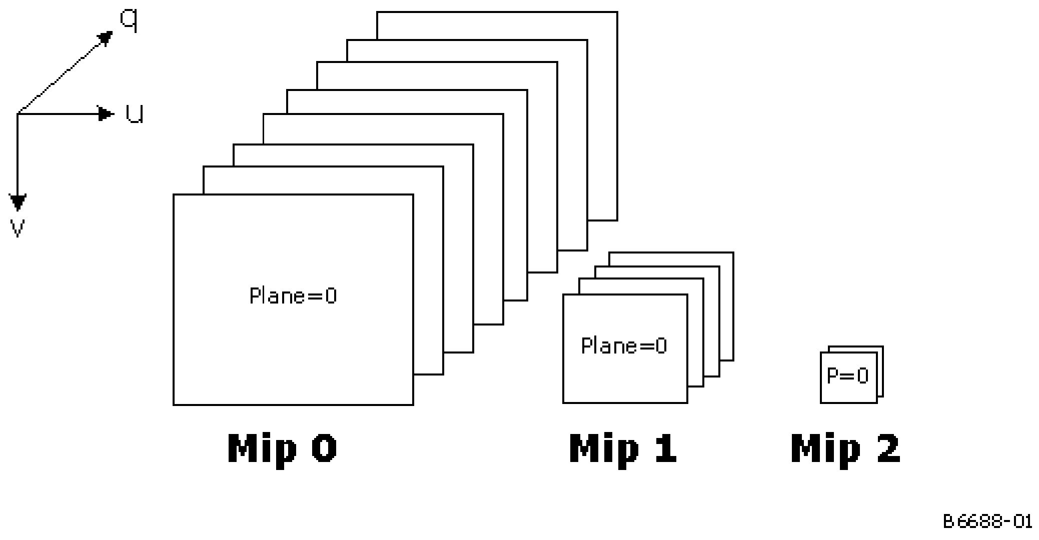
Cube Arrays

Cube arrays are stored identically to 2D surface arrays. A group of 6 consecutive array elements makes up a single cube map. A cube array with N array elements is stored identically to a 2D array with 6N array elements.

3D Surfaces

Multiple texture map surfaces (and their respective mipmap chains) can be arranged into a structure known as a Texture3D (volume) texture. A volume texture map consists of many *planes* of 2D texture maps. See *Sampler* for a description of how volume textures are used.

Volume Texture Map



The number of planes defined at each successive mip level is halved. Volumetric texture maps are stored as follows. All of the LOD=0 q-planes are stacked vertically, then below that, the LOD=1 q-planes are stacked two-wide, then the LOD=2 q-planes are stacked four-wide below that, and so on.

The width, height, and depth of LOD "L" are as follows:

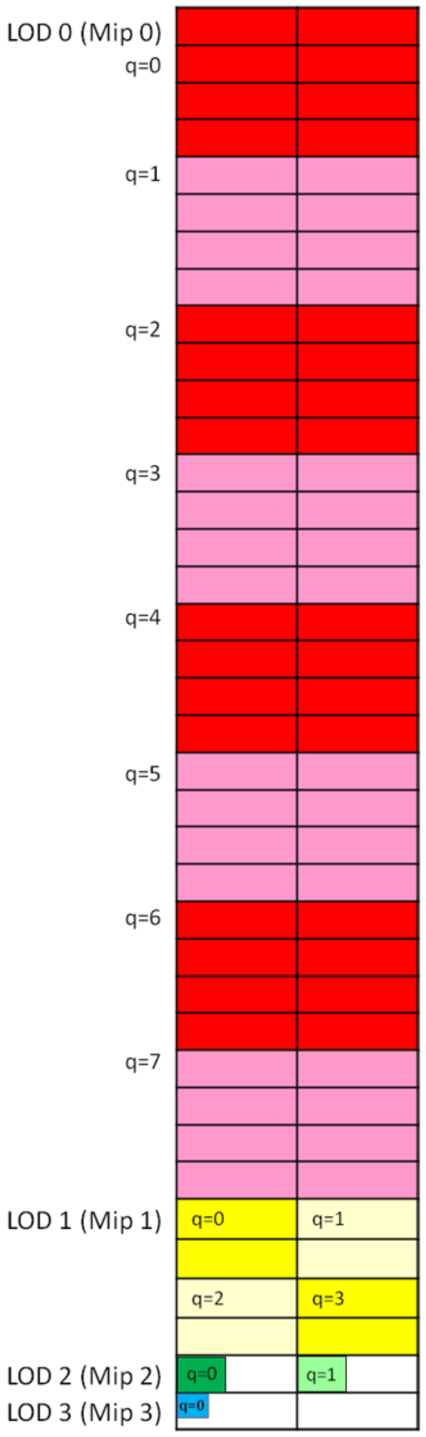
W_L = *((width » L) > 0 ? width » L:1)*

H_L = *((height » L) > 0 ? height » L:1)*

This is the same as for a regular texture. For volume textures we add:

D_L = *((depth » L) > 0 ? depth » L:1)*

Cache-line aligned width and height are as follows, with i and j being a function of the map format as shown in Alignment Unit Size.



$$w_L = i * ceil\left(\frac{W_L}{i}\right)$$

$$h_L = j * ceil\left(\frac{H_L}{j}\right)$$

It is not necessary to cache-line align in the "depth" dimension (i.e. lowercase "*d*").

The following equations for $LOD_{L,q}$ give the base address Cartesian coordinates for the map at LOD L and depth q.

$$LOD_{0,q} = (0, q * h_0)$$

$$LOD_{1,q} = ((q\%2) * w_1, D_0 * h_0 + (q >> 1) * h_1)$$

$$LOD_{2,q} = ((q\%4) * w_2, D_0 * h_0 + ceil\left(\frac{D_1}{2}\right) * h_1 + (q >> 2) * h_2)$$

$$LOD_{3,q} = ((q\%8) * w_3, D_0 * h_0 + ceil\left(\frac{D_1}{2}\right) * h_1 + ceil\left(\frac{D_2}{4}\right) * h_2 + (q >> 3) * h_3)$$

...

These values are then used as "base addresses" and the 2D MIP Map equations are used to compute the location within each LOD/q map.

Minimum Pitch

The minimum pitch required to store the 3D map may in some cases be greater than the minimum pitch required by the LOD=0 map. This is due to cache line alignment requirements that may impact some of the MIP levels requiring additional spacing in the horizontal direction.

Surface Padding Requirements

This section covers the requirements for padding around surfaces stored in memory, as there are cases where the device will overfetch beyond the bounds of the surface due to implementation of caches and other hardware structures.

Alignment Unit Size

This section documents the alignment parameters *i* and *j* that are used depending on the surface.

Alignment Parameters

Surface Defined By	Surface Format	Alignment Unit Width "*i*"	Alignment Unit Height "*j*"
3DSTATE_DEPTH_BUFFER	D16_UNORM	8	4
	not D16_UNORM	4	4
3DSTATE_STENCIL_BUFFER	N/A	8	8
SURFACE_STATE	BC*, ETC*, EAC*	4	4
	FXT1	8	4
	all others	set by **Surface Horizontal Alignment**	set by **Surface Vertical Alignment**

Sampling Engine Surfaces

The sampling engine accesses texels outside of the surface if they are contained in the same cache line as texels that are within the surface. These texels will not participate in any calculation performed by the sampling engine and will not affect the result of any sampling engine operation, however if these texels lie outside of defined pages in the GTT, a GTT error will result when the cache line is accessed. In order to avoid these GTT errors, “padding” at the bottom and right side of a sampling engine surface is sometimes necessary.

It is possible that a cache line will straddle a page boundary if the base address or pitch is not aligned. All pages included in the cache lines that are part of the surface must map to valid GTT entries to avoid errors. To determine the necessary padding on the bottom and right side of the surface, refer to the table in Alignment Unit Size section for the i and j parameters for the surface format in use. The surface must then be extended to the next multiple of the alignment unit size in each dimension, and all texels contained in this extended surface must have valid GTT entries.

For example, suppose the surface size is 15 texels by 10 texels and the alignment parameters are i=4 and j=2. In this case, the extended surface would be 16 by 10. Note that these calculations are done in texels, and must be converted to bytes based on the surface format being used to determine whether additional pages need to be defined.

Buffer Padding Requirements

For compressed textures (BC*, FXT1, ETC*, and EAC* surface formats), padding at the bottom of the surface is to an even compressed row. This is equivalent to a multiple of 2*q*, where *q* is the compression block height in texels. Thus, for padding purposes, these surfaces behave as if *j* = *2q* only for surface padding purposes. The value of *j* is still equal to *q* for mip level alignment and QPitch calculation.For cube surfaces, an additional two rows of padding are required at the bottom of the surface. This must be ensured regardless of whether the surface is stored tiled or linear. This is due to the potential rotation of cache line orientation from memory to cache.

For packed YUV, 96 bpt, 48 bpt, and 24 bpt surface formats, additional padding is required. These surfaces require an extra row plus 16 bytes of padding at the bottom in addition to the general padding requirements.

For linear surfaces, additional padding of 64 bytes is required at the bottom of the surface. This is in addition to the padding required above.

Programming Note	
Context:	Sampling Engine Surfaces.
For SURFTYPE_BUFFER, SURFTYPE_1D, and SURFTYPE_2D non-array, non-MSAA, non-mip-mapped surfaces in linear memory, the only padding requirement is to the next aligned 64-byte boundary beyond the end of the surface. The rest of the padding requirements documented above do not apply to these surfaces.	

Render Target and Media Surfaces

The data port accesses data (pixels) outside of the surface if they are contained in the same cache request as pixels that are within the surface. These pixels will not be returned by the requesting message, however if these pixels lie outside of defined pages in the GTT, a GTT error will result when the cache request is processed. In order to avoid these GTT errors, “padding” at the bottom of the surface is sometimes necessary.

If the surface contains an odd number of rows of data, a final row below the surface must be allocated. If the surface will be accessed in field mode (**Vertical Stride** = 1), enough additional rows below the surface must be allocated to make the extended surface height (including the padding) a multiple of 4.