Radeon R6xx/R7xx Acceleration

Trademarks

AMD, the AMD Arrow logo, Athlon, and combinations thereof, ATI, ATI logo, Radeon, and Crossfire are trademarks of Advanced Micro Devices, Inc.

Microsoft and Windows are registered trademarks of Microsoft Corporation.

Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

Disclaimer

The contents of this document are provided in connection with Advanced Micro Devices, Inc. ("AMD") products. AMD makes no representations or warranties with respect to the accuracy or completeness of the contents of this publication and reserves the right to make changes to specifications and product descriptions at any time without notice. No license, whether express, implied, arising by estoppel, or otherwise, to any intellectual property rights are granted by this publication. Except as set forth in AMD's Standard Terms and Conditions of Sale, AMD assumes no liability whatsoever, and disclaims any express or implied warranty, relating to its products including, but not limited to, the implied warranty of merchantability, fitness for a particular purpose, or infringement of any intellectual property right. AMD's products are not designed, intended, authorized or warranted for use as components in systems intended for surgical implant into the body, or in other applications intended to support or sustain life, or in any other application in which the failure of AMD's product could create a situation where personal injury, death, or severe property or environmental damage may occur. AMD reserves the right to discontinue or make changes to its products at any time without notice.

1. Introduction

1.1 Overview

The R6xx family features a 2nd generation Superscaler Unified Shader Architecture with hundreds of individual stream processors, a dedicated command processor, ultra-threaded dispatch processors and dozens of innovative visual quality enhancements. The R7xx family builds on the R6xx foundation to provide the fastest and most advanced 2D, 3D, and multimedia graphics performance for desktop and mobile PCs.

1.2 R6xx/R7xx Feature Highlights

- Dynamic Game Computing Technology
- Unified Stream Processors
- Dynamic Geometry Acceleration
- High-speed 128-bit High Dynamic Range (HDR) rendering
- Physics processing support
- Plug-and-Play CrossFire support
- ATI Avivo™ HD Video and Display Technology
- Built-in HDMI with Multi-channel 5.1 surround audio
- Dual dual-link DVI
- HD video playback
- Dual 30-bpp display processors

1.3 R6xx 3D Features in Detail

- Fully DirectX 10 compliant, including full speed 32-bit (or higher) floating point per component operations.
- Shader Model 4.0, geometry and pixel support in a unified shader architecture.
- Full speed 32-bit floating point processing per component.
- High dynamic range rendering with floating point blending, texture filtering and anti-aliasing support.
- High performance dynamic branching and flow control.
- Nearly unlimited shader instruction storage, using an advance caching system.
- Advanced shader design, with ultra-threading sequencer for high efficiency operations.
- Advanced, high performance branching support, including static and dynamic branching.
- 32-bit floating point components for high dynamic range computations
- Full anti-aliasing on render surfaces up to and including 128-bit floating point formats.
- Support for OpenGL® 2.0.
- Anti-Aliasing Filtering:
    - 2x/4x/8x modes.
    - Sparse multi-sample algorithm with gamma correction, programmable sample patterns, and centroid sampling.
    - Temporal anti-aliasing.
    - Adaptive anti-aliasing mode.
- Up to 128-tap texture filtering.
- Improved quality mode due to improved subpixel precision, higher precision LOD computations, and rotationally invariant LOD computations.
- Advanced Texture Compression (3Dc+™):
    - High quality 4:1 compression for normal maps and luminance maps.
    - Works with any single-channel or two-channel data format.
- HW support to overcome "Small batch" issues in CPU limited applications.

- 3D resources virtualized to a 32-bit addressing space, for support of large numbers of render targets and textures.
- New vertex cache and vertex fetch design, to increase vertex throughput from previous generations.
- Full support of 64-bit and 128-bit textures and surfaces, which can be 4x to 8x faster than previous generations of hardware.
- Up to 8 K x 8 K textures, including 128-bits/pixel texture are supported.
- New multi-level texture cache to give optimal performance, 8x greater than the previous designs.
- High efficiency ring bus memory controller:
    - Programmable arbitration logic maximizes memory efficiency.
    - Fully associative texture, color, and Z cache design.
- New hierarchical Z, and Stencil buffers with Early Z Test.
- New lossless Z-buffer compression for both Z and Stencil.
- Fast Z-Buffer Clear.
- Z cache optimized for real-time shadow rendering.
- Z and color compression resources virtualized to a 32-bit addressing space, for support of multiple render targets and textures simultaneously.

1.4 Changes

1.4.1 R600 to R670 changes

- Add memory export (scatter) read/write.
- Added double-precision floating point support.
- Add MOVA (AR) instruction that can be used to index GPRs.

1.4.2 Changes from R670 to R7xx

- Improved methods for passing data between threads.
- MOVA updates.
- New shader instructions.
- Added support for fetch scheduling.
- Expand maximum texture or vertex fetch clause to 16 instructions (*was 8 in R6xx*)
- Removed fixed-function fog
- Added a bit to allow VS & GS to access each other's constants (*replaces "use_waterfall" bit)*
- ALU shifts (ashr, lshr, lshl) now work on all slots (xyzwt), not just scalar (t).

2. R6xx/R7xx 3D Pipeline

2.1 Overview

2.1.1 Command processing

The CP processes both ring buffer and linear buffer commands streams. For 3D, the commands are processed and this generates a stream of register activity. The basic index buffer addresses are sent to the vertex grouper and tesselator (VGT) with the trigger commands and the VGT DMAs the indices from the index buffers. It generates a series of vertex indices which are sent to the shader pipe interpolator (SPI), with triggers. VGT also sends primitive connectivity to primitive assembly (PA).

2.1.2 Vertex processing

All shader processing is performed in the Unified Shader block (aka shader core), which includes the Sequencer (SQ) and Shader Pipe (SP) blocks. Each shader program has access to a number of General Purpose Registers (GPRs) which are dynamically allocated before running the program. The SPIs will load the GPRs with the appropriate parameters including vertex base addresses. The SPI will then launch the execution of the vertex shader for the sequencer unit (SQ). The first thing that that shader will do is fetch the vertex data required, and then process it per the application vertex shader. The outputs of this shader go into the shader export (SX) buffers, depending on the type of data. When a geometry shader (GS) is present, the vertex shader (VS) is referred to as an export shader (ES).

2.1.3 Pixel processing

After vertex processing, the geometry (position) data passes back to the PA for assembly into primitives and setup computations. The output of the PA passes into the scan convertor (SC) for scan conversion. The SC checks with the depth buffers (DB) on the validity of the pixels. Early Z (i.e. early Z R/W test), Re-Z (check for Z pass/fail, but do not update Z buffer) and HiZ (Hierarchical Z and stencil tests) are all available as forms of early culling of unnecessary pixels. The generated pixels (geometry and barycentrics), are then sent to the SPI, and then to the shader core for pixel processing. The pixel shader application program is executed, which might include texture fetches, ALU functions or Scratch RAM Read/writes. Finally, the shader memory exchange (SMX) acts as the Read/Write cache for scratch memory accesses (to and from the shader core). Once completed, the geometry and color of the pixels are sent via the SX to the DBs and then color buffers (CBs) for final processing.

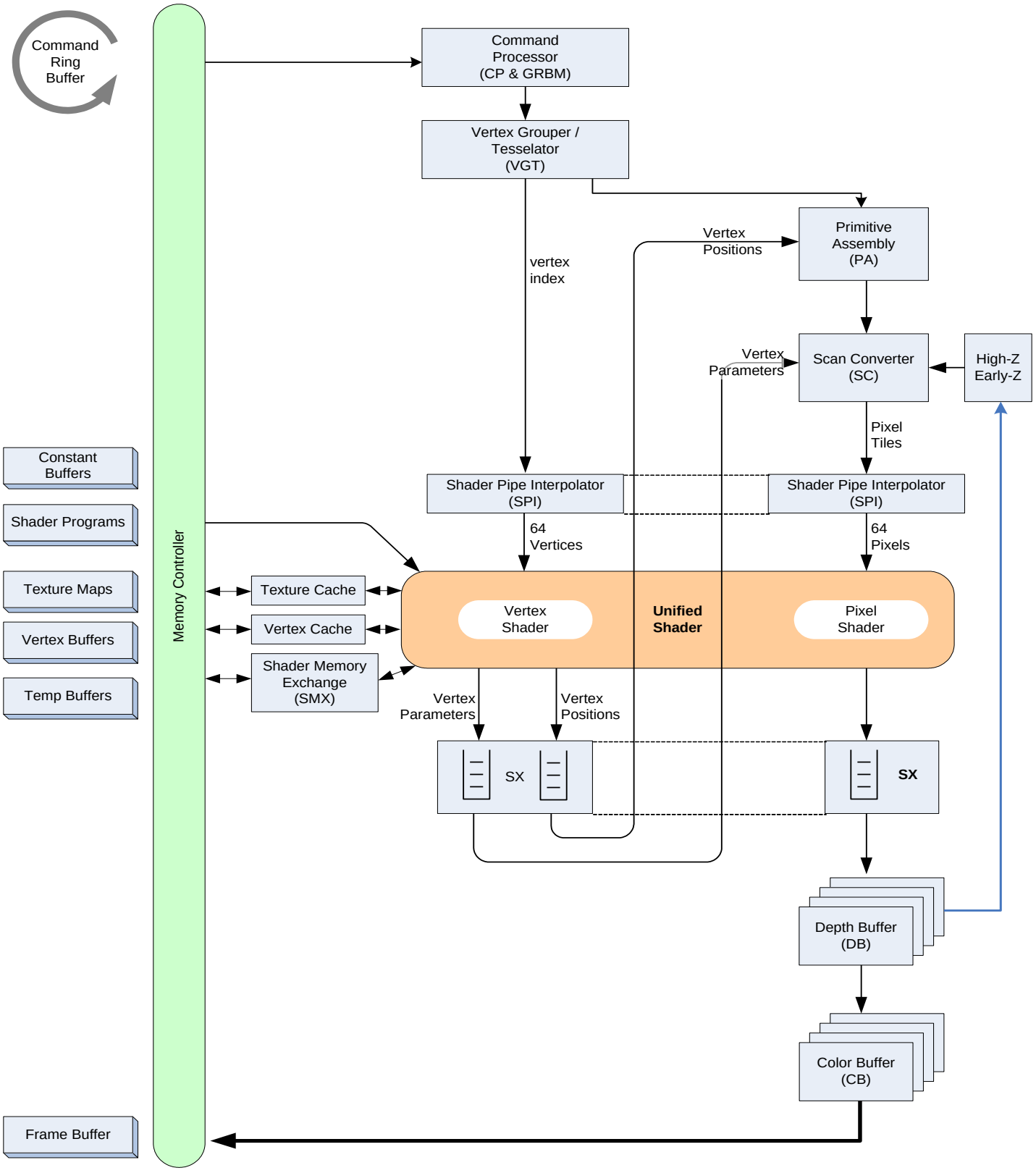
2.1.4 Final rendering

The final pixels are sent down to the DBs and CBs for final rendering. This includes alpha tests, depth testing and final blending/resolves. Multiple surface types will exist for depth (16b, 24b, 32b, 1-z (24), separate stencil) and blending. Z's and Colors are all cached and the DB supports compression for all of them.

2.2 R6xx/R7xx 3D Pipeline

See next page.

R600 Graphics Pipeline



2.3 Basic Vertex and Pixel Shader Data Flow

In the non-GS mode, a vertex shader program is run on each vertex and a pixel shader program is run on each pixel. In this mode, the data flows as:

2.3.1 Vertex Program Flow

1. The VGT receives a pointer to a buffer of vertex indices. It retrieves those indices and sends one index per vertex to SPI.
2. The SPI groups these into vectors called wavefronts of up to 64 vertices (size is asic dependant) in its input buffers.
3. When the wavefront is ready to be processed:
    a. SPI allocates GPR and thread space for the Vertex Shader program to run based on driver-provided sizes
    b. The indices are transferred into the GPRs in the SP
    c. The shader core is then notified that a new wavefront is ready for execution
4. The shader core runs the vertex program on the wavefront
    a. The vertex program either executes vertex fetch instructions directly or calls the fetch shader which uses the index to fetch the full vertex data
    (Vertex Resource Constants in the SQ define the location of the vertex buffers in memory)
    b. Vertex data is fetched into GPRs and if a fetch shader was used, it returns control to the vertex shader program
    c. Next the transform and lighting (and whatever else) part of the program runs
    d. The program allocates space in SX's position buffer and then outputs (exports) position (XYZW)
    e. Just before exiting, the program allocates Parameter Cache space in the SX's PC buffer and exports parameters for this vertex. The program then exits.
5. The SPI is informed when a wavefront has completed so that it can de-allocate GPR space

2.3.2 Pixel Program Flow

1. The PA reads out position data from the SX's position buffer and together with connectivity data from the VGT assembles primitives
2. The primitives are then sent to SC for coarse scan conversion.
3. The tiles are then sent to SPI for final pixel interpolation:
    a. The SPI allocates a pixel thread and GPRs
    b. The SC and SPI read per-vertex parameter data from the SX
    c. It then interpolates that data along with barycentric data (I,J) from the SC to arrive at the per-pixel value of each parameter.
    d. These values are then loaded into GPRs
4. The shader core is then notified that a pixel wavefront and shader program are ready for execution
5. The shader program runs on all pixels in the wavefront and at the end exports data to the SX's export buffer
6. The SPI is informed when a wavefront has completed so that it can de-allocate GPR space

2.4 Hardware Support for Vertex Fetching

The hardware supports vertex fetches of the form:

fetch_addr = (index + index_offset) * stride + base + offset
GPR StateRegister Const Const Instr.

index = GPR value, typically vertex index
index_offset = either BaseVertexLoc or StartInstanceLoc, or zero: selected by an instruction bit.
stride = byte-stride from the fetch constant (AlignedByteStructureStride), or zero.

base = (VBResourceBase + AlignedVertexBufferByteOffset) *added together by the driver*
offset = AlignedByteOffset (which element within a vertex structure to read)

The SQ has a unique shader code base address for the fetch shader (FS) which allows the driver to load the FS and the VS (vertex shader) independently. The hardware supports separate shader base addresses for the FS and VS or ES (export shader; VS when GS is present) shaders, but will run them in a single pass with the fetch shader being called as a subroutine from the vertex shader (subroutine address in a register, not compiled into shader). Fetch data will be passed from FS to VS or ES through GPRs. Hardware will allocate at thread create time enough GPRs to support both, although the FS will often not require any dedicated GPRs. Vertex fetching can be done via a FS or directly from the VS. The fetch instruction includes the definition of the stream and vertex declaration. The vertex resource constant holds the vertex buffer definition. The FS will require its own GPR storage which is contiguous with VS GPR storage. The vertex-fetch-constant also contains the array-size for clamping. However the size for clamping is with respect to VBResourceBase, so when setting the size, the driver must subtract AlignedVertexBufferByteOffset since hardware will simply compare *base* to *size.*

2.5 Shader Linkage

2.5.1 Fetch Shader to Vertex Shader

The fetch shader (FS) fetches elements based on semantic name (position, normal, …). These are linked to the VS through a semantic table which correlates semantic name with VS input register number. The use of semantic names for vertex fetch is not available if the vertex fetch is done directly from the VS rather than via the FS.

2.5.2 Vertex Shader to Pixel Shader

The VS to PS linkage is handled via semantic mapping as well. VS exports are loaded into GPRs for the PS based on the semantic names set up in the SPI.

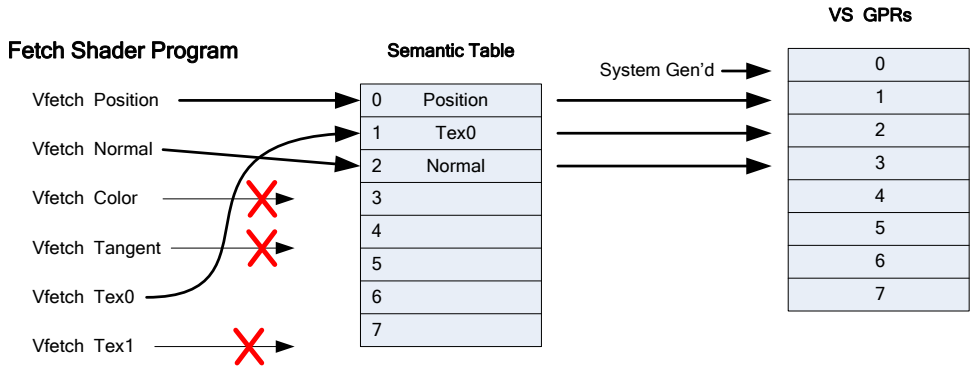
2.5.3 VS INPUT SEMANTIC TABLE

1. This table is used to provide semantic separation (independence) between the FS and the VS. FSs are optional as vertex fetches can be done by the VS directly, but without the semantic remapping.
2. This table contains an 8-bit semanticID field. The location in the table specifies the desired VS input GPR location (+1)
3. As semantic vertex fetches are made by the FS, the hardware will match the semanticID in the fetch instructions destination address with its counterpart in the semantic table. If the matching semanticID is found, the destination address for the vertex fetch is set to VS_GPR_BASE+1+table_location, where
    1. VS_GPR_BASE is typically 0
    2. The +1 is to leave room for the system generated input vector which contains the vertexID and instanceID
    3. The table_location is the relative entry of the semantic table in which the semanticID match is found.

    If the semanticID of the fetch instruction is not found in the semantic table this suggests that the VS does not use this element and the fetch would be terminated.

    The following diagram shows the independence between the FS and the VS. Among other things, this allows the FS to fetch vertexes with arbitrary formats and process them all with a standard VS.

    The driver would define a set of enums for each semanticID (POSITION, FRONT_COLOR, BACK_COLOR, TEXCOORD0, etc.). Then the same semanticID enum would be used in both the vertex fetch instruction and the semantic table.



The VS to PS transition provides a similar semantic mapping. See the *SPI (Shader Processor Interpolator)* setup section in below.

3. 3D Engine Programming

The following section describes the basic state and shader setup required to render a primitive using the 3D engine.

3.1 CP (Command Processor) setup

The CP microcode must be loaded and the ring buffer and read and write pointers must be initialized similar to previous asics before using the CP. The R6xx/R7xx asics are different from previous generations in that microcode must be loaded for both the CP microengine (ME) and the CP prefetch parser (PFP).

3.1.1 Surface Synchronization Packet Sequence

Through a single interface, the PM4 command stream and the host CPU can determine if rendering is complete to a surface and flush and invalidate caches. This mechanism is based on using four registers (CP_COHER_STATUS, CP_COHER_CNTL, CP_COHER_SIZE, CP_COHER_BASE), state management logic, and the SSU logic to determine if/when all queued rendering is complete to a surface.

At texture setup time the driver will synchronize the source caches and at draw primitive time the driver will synchronize the destination caches.

Source caches reside in the texture cache (TC), vertex cache (VC), and sequencer instruction cache (SH). For each of these fetch caches, each cache line that intersects the surface extents will be invalidated. Invalidating the source caches work different than invalidating the destination caches. The driver writes the CP_COHER_CNTL register with the source cache(s) to invalidate the specified surface, but it will not set any of the base registers. It will then set the CP_COHER_BASE.

For ***Destination Caches*** the surface coherency mechanism can optionally compare the CP_COHER_BASE address against one or more hardware destination addresses. The destination address registers in the CP are:

1. CB_COLOR_BASE[7:0]
2. DB_DEPTH_BASE
3. VGT_STRMOUT_BUFFER_BASE[3:0]
4. COHER_DEST_BASE[1:0]

The compare process occurs only once each time the CP_COHER_BASE register is written.

3.2 VGT (Vertex Grouper Tessellator) setup

- VGT_VTX_VECT_EJECT_REG: This register defines the number of primitives that are allowed to pass during the assembly of a single vertex vector. After this number of primitives has passed, the vertex vector is submitted to the shaders for processing even if it is not full.
- VGT_DMA_NUM_INSTANCES: VGT DMA Number of Instances (set to 1 generally).
- VGT_OUTPUT_PATH_CNTL: (Ignored in Major Mode 0) This register selects which backend path will be used by the VGT block. (Set to VGT_OUTPATH_VTX_REUSE).
- Enable Primitive ID in VGT: VGT_PRIMITIVEID_EN.
- VGT_MULTI_PRIM_IB_RESET_EN: This register enabling resetting of prim based on reset index (multi-prim reset set to OFF).
- VGT_MULTI_PRIM_IB_RESET_INDX: This register defines the index which resets primitive sets when MULTI_PRIM_IB is enabled.
- VGT_VTX_CNT_EN: Auto-index generation.
- VGT_GS_MODE: VGT GS Enable Mode (Geometry Shader is turned ON/OFF).

3.3 PA (Primitive Assembler) setup

Setup Scissors, Windows Offset, Clip Rectangles

- **Screen scissor rectangle specification**

    - PA_SC_SCREEN_SCISSOR_BR
    - PA_SC_SCREEN_SCISSOR_TL
- **Window offset** - Offset from screen coords to window coords. Vertices will be offset by these values if PA_SU_SC_MODE_CNTL.VTX_WINDOW_OFFSET_ENABLE is set. The WINDOW_SCISSOR will be offset by these values if the WINDOW_SCISSOR_TL.WINDOW_OFFSET_DISABLE is clear. If this value allows the window to extend beyond the Front Buffer (Surface) dimensions, it is expected that the SCREEN_SCISSOR is used to limit to FB surface.
    - PA_SC_WINDOW_OFFSET:
- **Window Scissor rectangle specification**
    - PA_SC_WINDOW_SCISSOR_BR : Scissor is conditionally (See WINDOW_OFFSET_ENABLE) offset by WINDOW_OFFSET.
    - PA_SC_WINDOW_SCISSOR_TL
- **4 Clip rectangles**
    - PA_SC_CLIPRECT_x_TL
    - PA_SC_CLIPRECT_RULE
- **Generic scissor rectangle specification**
    - PA_SC_GENERIC_SCISSOR_BR
    - PA_SC_GENERIC_SCISSOR_TL
- **WGF ViewportID Scissor rectangle specification**
    - PA_SC_VPORT_SCISSOR_0_BR
    - PA_SC_VPORT_SCISSOR_0_TL
- **Viewport Transform Z Min/Max Clamp**
    - PA_SC_VPORT_ZMIN_0: 0
    - PA_SC_VPORT_ZMAX_0: 1.0f
- **Setup PA CL (Clipper): Disable UCP (User Clipping Planes)**
    - PA_CL_CLIP_CNTL
- **Setup PA_SU_SC_MODE_CNTL.** Use this register for face culling, polymode, and to select what primitive the flat shading color comes from.
    - POLY_OFFSET_FRONT_ENABLE
    - POLY_OFFSET_BACK_ENABLE
    - POLY_OFFSET_PARA_ENABLE
    - VTX_WINDOW_OFFSET_ENABLE
    - PROVOKING_VTX_LAST
    - PERSP_CORR_DIS
    - MULTI_PRIM_IB_ENA
    - CENTER_GUARDBAND_ENABLE
- **Setup PA SU (Setup Unit):** Disable Viewport, Point Sprite, Guardband
    - PA_SU_POINT_SIZE
    - PA_SU_POINT_MINMAX
    - PA_SC_AA_CONFIG
    - PA_SC_AA_MASK
    - PA_CL_GB_HORZ_CLIP_ADJ
    - PA_CL_GB_HORZ_DISC_ADJ
    - PA_CL_GB_VERT_CLIP_ADJ
    - PA_CL_GB_VERT_DISC_ADJ
    - PA_SU_VTX_CNTL
        - *PIX_CENTER = 0;*
        - *ROUND_MODE = 2;*
        - *QUANT_MODE = 0;*
    - PA_CL_POINT_X_RAD: 0 (Point Sprite X Radius Expansion)
    - PA_CL_POINT_Y_RAD: 0 (Point Sprite Y Radius Expansion)
    - PA_CL_POINT_SIZE: 0 (Point Sprite Constant Size)
    - PA_CL_POINT_CULL_RAD: 0 (Point Sprite Culling Radius Expansion)

3.4 SQ (Sequencer) setup

- Controls the Shader Pipe to execute ALUs and export instructions.
- Controls Texture Pipes and Vertex Cache to execute texture and vertex fetch operations.
- Supports 4 thread types:
  - ES (Export Shader) – typically a vertex shader before a geometry shader (it only outputs to memory).
  - GS (Geometry shader) – optional.
  - VS (Vertex Shader) – null when GS is active, a normal vertex shader when GS is off.
  - PS (Pixel Shader).
- Allocates resources for shaders (#GPRS, threads, stack entries, vertex cache, etc.):
  - SQ_CONFIG:
    - *VC_ENABLE: 1 (if Vertex Cache is present)*
    - *DX9_CONSTS: 1 (Enable register-based constant file as opposed to memory-based constant file)*
    - *ALU_INST_PREFER_VECTOR: 1 (ALU clause instruction assignment. When a group of 4 or less instructions, there may be ambiguity whether to assign the last instruction to the vector pipe (according to the instruction`s dest-chan), or to the scalar pipe (trans). This bit controls that decision: 0 = send the last instruction word to the scalar (trans) pipe if possible, 1 = prefer to send it to the vector pipe).*
  - SQ_GPR_RESOURCE_MGMT_1
    - *NUM_PS_GPRS: Based on ASIC*
    - *NUM_VS_GPRS: Based on ASIC*
  - SQ_THREAD_RESOURCE_MGMT
    - *NUM_PS_THREADS: Based on ASIC*
    - *NUM_VS_THREADS: Based on ASIC*
  - SQ_STACK_RESOURCE_MGMT_1
    - *NUM_PS_STACK_ENTRIES: Based on ASIC*
    - *NUM_VS_STACK_ENTRIES: Based on ASIC*
  - SQ_VTX_BASE_VTX_LOC
    - *OFFSET: 0 (Vertex Base location for vertex fetching)*
  - SQ_VTX_START_INST_LOC
    - *OFFSET: 0 (Instance start location for vertex fetching)*

3.5 Vertex Setup

- **Vertex resource setup**
  - SQ_VTX_CONSTANT_WORD0_0
    - BASE_ADDRESS: dwDevAddr
  - SQ_VTX_CONSTANT_WORD0_1
    - SIZE: (dwNumEntries << 2) - 1
  - SQ_VTX_CONSTANT_WORD0_2
    - STRIDE: dwVtxSize << 2
    - CLAMP_X: 1
  - SQ_VTX_CONSTANT_WORD0_3
  - SQ_VTX_CONSTANT_WORD0_6
    - TYPE: 3

3.6 Texture Setup

For every used texture we are going to setup a set of SQ_TEX_RESOURCE_WORDx_y registers up to 160 resources are available per SQ thread and one of the 18 samplers per SQ thread using the SQ_TEX_SAMPLER_WORDx_y registers. Texture resources can be set up for PS or VS use. The details of these registers are as follows:

- **Texture resource setup**

  - SQ_TEX_RESOURCE_WORD0_0
    - *DIM: SQ_TEX_DIM_2D*
    - *TILE_MODE: pSurf->tileMode*
    - *TILE_TYPE: ARRAY_COLOR_TILE*
    - *PITCH*
    - *TEX_WIDTH: pSurf->dwWidth - 1*
  - SQ_TEX_RESOURCE_WORD1_0
    - *TEX_HEIGHT: pSurf->dwHeight - 1*
    - *TEX_DEPTH: 0*
    - *DATA_FORMAT: pSurf->dwTexFmt*
  - SQ_TEX_RESOURCE_WORD2_0
    - *BASE_ADDRESS*
  - SQ_TEX_RESOURCE_WORD3_0
    - *MIP_ADDRESS*
  - SQ_TEX_RESOURCE_WORD4_0: pSurf->dwTexFmtEx
    - *FORMAT_COMP_X*
    - *FORMAT_COMP_Y*
    - *FORMAT_COMP_Z*
    - *FORMAT_COMP_W*
    - *NUM_FORMAT_ALL*
    - *SRF_MODE_ALL*
    - *FORCE_DEGAMMA*
    - *ENDIAN_SWAP*
    - *REQUEST_SIZE*
    - *DST_SEL_X*
    - *DST_SEL_Y*
    - *DST_SEL_Z*
    - *DST_SEL_W*
    - *BASE_LEVEL*
  - SQ_TEX_RESOURCE_WORD5_0
    - *LAST_LEVEL*
    - *BASE_ARRAY*
    - *LAST_ARRAY*
  - SQ_TEX_RESOURCE_WORD6_0
    - *MPEG_CLAMP*
    - *PERF_MODULATION: 1*
    - *INTERLACED*
    - *TYPE: TVX_Type_ValidTextureResource*
- **Samplers setup**
  - SQ_TEX_SAMPLER_WORD0_0
    - *CLAMP_X*
    - *CLAMP_Y*
    - *CLAMP_Z*
    - *XY_MAG_FILTER*
    - *XY_MIN_FILTER*
    - *Z_FILTER*
    - *MIP_FILTER*
    - *BORDER_COLOR_TYPE*
    - *POINT_SAMPLING_CLAMP*
    - *TEX_ARRAY_OVERRIDE*
    - *DEPTH_COMPARE_FUNCTION*
    - *CHROMA_KEY*
    - *LOD_USES_MINOR_AXIS*

- SQ_TEX_SAMPLER_WORD1_0
  - *MIN_LOD*
  - *MAX_LOD*
  - *LOD_BIAS*
- SQ_TEX_SAMPLER_WORD2_0
  - *LOD_BIAS_SEC*
  - *MC_COORD_TRUNCATE*
  - *FORCE_DEGAMMA*
  - *HIGH_PRECISION_FILTER*
  - *PERF_MIP*
  - *PERF_Z*
  - *FETCH_4*
  - *SAMPLE_IS_PCF*
  - *TYPE*

3.7 Shader Setup

All shaders need to be in GPU accessible memory. *Texture and vertex fetches* are initiated by a shader fetch instruction. A texture or vertex fetch instruction defines the **source GPR** which contains the address, the **destination GPR** where the fetched data goes and a pointer to a **surface descriptor – the resource** (texture/vertex fetch constants). The texture or vertex data for the wavefront is fetched from cache or memory, filtered, and loaded into the destination GPRs.

- *Fetch Shader* info setup (optional):
  - SQ_PGM_START_FS: Base video memory address for the fetch shader
  - SQ_PGM_RESOURCES_FS: Info about the resources used by this shader
- *Vertex Shader* info setup:
  - SQ_PGM_START_VS: Base video memory address for the vertex shader
  - SQ_PGM_RESOURCES_VS: Info about the resources used by this shader
- Clear the semantic table (SQ_VTX_SEMANTIC_CLEAR)
- Update the Vertex Fetch Semantic table that will provide the inputs for the Vertex Shader (using the SQ_VTX_SEMANTIC_x set of registers).
- Update the output semantics for the Vertex Shader using the SPI_VS_OUT_ID_x set of registers and the SPI_VS_OUT_CONFIG reg.
  - SPI_VS_OUT_CONFIG:
    - *VS_PER_COMPONENT: When set, each entry in SPI_VS_OUT_ID_0-9 represents one component of a vector. Otherwise each entry represents an entire vector. This should be set to 0 generally.*
    - *VS_EXPORT_COUNT: Number of vectors exported by the VS (value minus 1).*
    - *VS_EXPORTS_FOG: Set when VS exports fog.*
    - *VS_OUT_FOG_VEC_ADDR: Vector address where VS exported fog. Fog factor will always be in the X channel.*
  - SPI_VS_OUT_ID_x
    - *SEMANTIC_0*
    - *SEMANTIC_1*
    - *SEMANTIC_2*
    - *SEMANTIC_3*
- *Pixel Shader info* setup (similar SQ setup with FS and VS).
- *PS constants* setup using the SQ_ALU_CONSTANT* registers.
- Setup color component mask fields for the colors output by the pixel shader using the CB_SHADER_MASK and Early_Z or Late_Z using DB_SHADER_CONTROL register.
  - CB_SHADER_MASK:
    - *OUTPUT0_ENABLE: If zero, this field disables writes to render target 0, else it specifies which components are enabled in the shader. The low order bit corresponds to the red*

channel. A one bit passes the shader output component value to the color block. A zero bit replaces the component with the default value: 0.0 for RGB or 1.0 for alpha.
        - *OUTPUT1_ENABLE*
        - *…*
        - *OUTPUT7_ENABLE*
- *SPI (Shader Processor Interpolator)* setup to organize the loading of the shader input data to GPRs (General Purpose Registers) and SPs (Shader Pipes).
    - SPI_PS_IN_CONTROL_0 and SPI_PS_IN_CONTROL_1.
        - *NUM_INTERP: Number of parameters to interpolate (no minus 1). Does not include fog, param_gen, or gen_indx, but should include position and frontface.*
        - *POSITION_ENA: Load per-pixel position into the PS.*
        - *POSITION_CENTROID: Calculate per-pixel position at pixel centroid.*
        - *POSITION_ADDR: Relative GPR address where position is loaded.*
        - *PARAM_GEN: Generate up to 4 sets of ST coordinates. Bit 0=persp/center, 1=persp/centroid, 2=linear/center, 3=linear/centroid.*
        - *PARAM_GEN_ADDR: First relative GPR address where param_gen values are loaded.*
        - *BARYC_SAMPLE_CNTL*
        - *PERSP_GRADIENT_ENA: Enable perspective gradients (if linear is set to 0, persp is always enabled).*
        - *LINEAR_GRADIENT_ENA: Enable linear gradients.*
    - Set of SPI_PS_INPUT_CNTL_x registers. Each of these registers maps to a GPR. SPI_PS_INPUT_CNTL_0 maps to GPR 0. These define how data is loaded into the PS from the VS or elsewhere.
        - *SEMANTIC: PS input semantic mapping.*
        - *DEFAULT_VAL: Selects value to force into GPR if no semantic match found.*
        - *FLAT_SHADE: Flat shade select.*
        - *SEL_CENTROID: Use IJ data sampled at pixel centroid.*
        - *SEL_LINEAR: Use IJ data from linear gradients.*
        - *CYL_WRAP: 4-bit cylindrical wrap control (1 bit per component).*
        - *PT_SPRITE_TEX: Override this parameter with texture coordinates if global enable set and prim is a point.*

3.8 SPI (Shader Processor Interpolator) Fixed Setup

- Turn on/off fog, inputZ, flat shading and point sprites.
    - SPI_CONFIG_CNTL
    - SPI_INTERP_CONTROL_0
    - SPI_FOG_CNTL
    - SPI_INPUT_Z
    - SPI_FOG_FUNC_SCALE
    - SPI_FOG_FUNC_BIAS

3.9 DB (Depth Block) Setup

- Responsible for doing visibility testing with the z and stencil buffer. Turn on/off features like Hierarchical Z (HiZ), Hierarchical Stencil (HiS), Early-Z/S, Re-Z
    - DB_DEPTH_CONTROL
    - DB_DEPTH_INFO
    - DB_RENDER_OVERRIDE

3.10 CB (Color Block) setup

- Responsible converting pixels into their final numerical form and writing them to the frame buffer via the Memory Controller (MC).
    - CB_COLOR0_BASE

- CB_COLOR0_SIZE
- CB_COLOR0_VIEW
- CB_COLOR0_INFO
- CB_COLOR0_TILE
- CB_COLOR0_MASK
- CB_COLOR_CONTROL
- CB_CLRCMP_CONTROL
- CB_TARGET_MASK

3.11 Draw Setup

In order to start the rendering we need to add the draw command to the PM4 buffer. This draw command will consist from the following parts:

- *Primitive* setup (set VGT_PRIMITIVE_TYPE register)
- *Index type* setup (use INDEX_TYPE packet)
- *Num instances* setup (use NUM_INSTANCES packet)
- *Draw packets* setup (use Type 3 draw packets)
    - These will set the VGT_DRAW_INITIATOR register

4. Synchronization and Cache Flushing

4.1 Overview

The GPU and CPU can render based on data residing in frame buffer or system memory. The data in these memories is generated by previous rendering actions which are performed by GPU/CPU itself or another part (CPU/GPU). To have correct rendering, one has to maintain data coherency which implies each previous rendering action must have completed and the result data of the rendering has arrived in corresponding memory (frame buffer or system memory).

Even though rendering action has completed, the resulting data may have not have arrived in memory because when rendering to the frame buffer, the data is first put into cache before going to the frame buffer, and when rendering to system memory, the data is also first put into cache then goes through the bus controller and finally reaches out the physical system memory. It is necessary to flush the data out of ASIC cache and system bus controller.

For some cases, GPU/CPU operation itself can guarantee the data coherency, while for other cases, drivers need to take action. This depends on the sequence of read/write combinations that GPU/CPU performed. The situation also varies with ASICs. For pre-R600 ASICs, entire ASIC cache can be automatically flushed when GUI becomes idle, using PM4_IDLE where needed. R6xx/R7xx does not have this auto flush mechanism.

On previous asics, the WAIT_UNTIL register was used by the driver to do explicit synchronization between software and hardware or between different hardware blocks. For R6xx/R7xx, there are two methods to do explicit synchronization in driver: using the WAIT_UNTIL register and using WAIT_REG_MEM type 3 PM4 packet.

The main difference between those two methods is that WAIT_UNTIL waits at the end of CP to GRBM fifo so it does not block the CP from processing the following packets (until the FIFO fills up), same as on previous asics. WAIT_REG_MEM waits in CP.

4.2 WAIT_UNTIL Synchronization

When comparing with R5xx WAIT_UNTIL register, only the bits waiting for status of GRBM are left in the R6xx/R7xx WAIT_UNTIL register. Bits defined on both R5xx and R6xx/R7xx:

WAIT_CMDFIFO
WAIT_2D_IDLE
WAIT_3D_IDLE
WAIT_2D_IDLECLEAN
WAIT_3D_IDLECLEAN
WAIT_EXTERN_SIG
CMDFIFO_ENTRIES

The following table lists the bits that are defined in the R5xx WAIT_UNTIL register and the registers and masks that R6xx/R7xx need to poll (since some drivers may be using R3xx bit field names for R5xx programming, R3xx bits are listed below as well). The WAIT_REG_MEM packet can be used to poll the R6xx/R7xx registers below.

WAIT_UNTIL bits on R5xx	R6xx/R7xx registers and bits to poll
WAIT_GRPH_FLIP_DONE Or	D1GRPH_UPDATE.D1GRPH_SURFACE_UPDATE_PENDING or

(R3xx) WAIT_CRTC_PFLIP (Use ENG_DISPLAY_SELECT to select display engine)	D2GRPH_UPDATE.D2GRPH_SURFACE_UPDATE_PENDING
WAIT_RE_VBLANK Or (R3xx) WAIT_FE_CRTC_VLINE (Use ENG_DISPLAY_SELECT to select display engine)	This could be replaced by setting up mmD1MODE_VLINE_START_END/ mmD2MODE_VLINE_START_END registers as follow: VLINE_START = first VLINE of active display VLINE_END = last VLINE of active display + 1 VLINE_INV = 1 (vline status is outside of vline start and end) And poll on D1MODE_VLINE_STATUS.D1MODE_VLINE_STAT or D2MODE_VLINE_STATUS.D2MODE_VLINE_STAT
WAIT_OVL_FLIP_DONE (Use ENG_DISPLAY_SELECT to select display engine)	D1OVL_UPDATE.D1OVL_UPDATE_PENDING or D2OVL_UPDATE.D2OVL_UPDATE_PENDING
WAIT_VLINE_ACTIVE Or (R3xx) WAIT_CRTC_VLINE (Use ENG_DISPLAY_SELECT to select display engine)	D1MODE_VLINE_STATUS.D1MODE_VLINE_STAT or D2MODE_VLINE_STATUS.D2MODE_VLINE_STAT
WAIT_BOTH_CRTC_PFLIP	Poll both D1GRPH_UPDATE.D1GRPH_SURFACE_UPDATE_PENDING and D2GRPH_UPDATE.D2GRPH_SURFACE_UPDATE_PENDING back to back.

4.3 WAIT_REG_MEM Synchronization

The other method is to use WAIT_REG_MEM type 3 packet. See the PM4 section for a packet description. Since the CP is stalled when processing WAIT_REG_MEM, it is recommended that POLL_INTERVAL be set to a small interval. This can ensure quick response time on the wait so that CP can be freed up sooner.

4.4 Mechanism of Data Coherency and Cache Flush

There are several data coherency mechanisms that let driver know that PM4 packets have been fetched into CP and consumed, or request cache flush and be notified that cache flush has completed. Followings are part of data coherency mechanisms. Some of them are new features supported only by R6xx/R7xx while some already exist in pre-R600 ASICs.

4.4.1 CP Command Buffer Fetch Acknowledging/Consuming Timestamp

Driver can know that submitted PM4 packets have been fetched by CP. One of the ways is (fence/time stamp write back to scratch register) which is often used to confirm some packets have been processed by CP.

4.4.2 Data Retirement Acknowledgement

Driver can be aware that data associated with a packet has been consumed. There is a type 3 PM4 packet EVENT_WRITE_EOP, which can generate events to indicate that all data has been consumed.

4.4.3 Wait Until the Completion of a Rendering

There are two ways that can be used to explicitly wait for completion of a rendering. One is to submit a packet to set a bit in the WAIT_UNTIL register. The other is to use PM4 packet WAIT_REG_MEM which can request CP to poll either a memory location or a register until it becomes a value specified in the packet, and then continue next PM4 packet parsing.

4.4.4 Surface Synchronization

Surface synchronization is a new feature of data coherency for R6xx/R7xx CP block. It can determine whether renderings to a surface have completed, all data has made into corresponding memory and all caches are flushed and invalidated. This mechanism is based on operations to register CP_COHER_STATUS, CP_COHER_CNTL, CP_COHER_SIZE and CP_COHER_BASE and wait for PM4 packet consumed logic.

Driver can submit following packets to synchronize single surface:

- Submit PM4 packets to write following registers:
  CP_COHER_CNTL = base register to be compared and block receiving start signal (destination / source surface).
  CP_COHER_SIZE = surface size.
  CP_COHER_BASE = surface base MC address
- Submit PM4 packet WAIT_REG_MEM to let ASIC poll register CP_COHER_STATUS. When bit STATUS in the register becomes 0, then it means surface has been synchronized.
- Submit a time stamp write back packet.
- Driver polls the write back scratch register. When write back value presents in the register, driver knows synchronization process has completed.

To synchronize all surfaces (flush all caches) , enable all corresponding bits in CP_COHER_CNTL and set CP_COHER_SIZE = 0xFFFFFFFF, and CP_COHER_BASE = 0x00000000.

4.4.5 Flush Cache by Setting Event in Register

Writing CACHE_FLUSH to EVENT_TYPE field in the register can request to flush the caches of ASIC functional blocks. This can be done with packet EVENT_WRITE(_EOP).

4.4.6 Flush Data to System Memory:

To flush data from bus controller to physical system memory, driver just needs do a read back from system memory.

4.5 Recommendations for R6xx/R7xx Cache Flush

R6xx/R7xx supports some new data coherency mechanisms so that individual surfaces can be synchronized. Automatic cache flush is not available when GUI becomes idle like on previous asics. The following sequence

combinations of GPU/CPU read/write to frame buffer/system memory require driver handling to ensure data coherency:

- After the last HW(GPU) rendering, the driver needs to flush ASIC cache since R6xx/R7xx does not have an auto flush mechanism when GUI is idle.
- When switching from HW(GPU) rendering to SW(CPU) rendering on frame buffer, the driver needs to make sure that the cache flush for last HW rendering has completed before SW rendering begins.
- When switching from HW rendering to SW rendering on system memory, 2D driver also needs to make sure that all data generated by HW has been in system memory before SW rendering begins.
- When flipping display, the driver needs to make sure all renderings to the display surface are completed and the surface data has been in frame buffer before display engine can be switched to that surface.
- When there is a mode switch/change (including normal mode change, power saving, PCI-E lane switch), the driver needs to make sure all renderings are completed, the engines are idle, and rendering data has been correspondingly in frame buffer or system memory.

4.5.1 Cases Where Maintaining Data Coherency Is Necessary for R600

There are several cases that the driver has to handle to ensure data coherency.

4.5.1.1 After the Last HW Rendering on Frame Buffer

CP can ensure data coherency between two HW renderings actions, but if the rendering action is the last one, data coherency cannot be guaranteed as R6xx/R7xx has no auto flush mechanism. Theoretically the driver just needs to do cache flush for the last HW rendering. In practice, the driver is unable to know which rendering action is the last one. The solution is to do a cache flush after each HW rendering.

4.5.1.2 Switching from HW rendering to SW rendering on Frame Buffer

There are many occasions that one needs to switch from HW rendering to SW rendering on frame buffer. Before SW rendering begins, driver has to make sure cache flush has completed. For older ASICs, the driver just needs to use PM4_IDLE / PM4_IDLE_FLUSH to wait for GUI idle. For R6xx/R7xx, since driver requests cache flush after each HW rendering, when switching to SW rendering driver needs to know the last cache flush has finished. Following methods can be used as PM4_IDLE for R6xx/R7xx for this purpose:

4.5.1.2.1 Method 1

- Submit PM4 packet EVENT_WRITE_EOP with event “Cache Flush and Invalidate TS”.
- Wait for timestamp in “Cache Flush and Invalidate TS” to be written back.
- Set a flag for cache flush when switching back to HW rendering.
- Flush HDP read cache.

4.5.1.2.2 Method 2

- Wait for engines become idle.
- Submit type 3 PM4 packets to set CP_COHER_CNTL= specify surfaces, CP_COHER_SIZE=0xFFFFFFFF, CP_COHER_BASE=0x00000000.
- Submit a type 3 PM4 packet WAIT_REG_MEM to wait till STATUS in CP_COHER_STATUS becomes 0.
- Submit timestamp write back packet and poll the write back.
- Set a flag for cache flush when switching back to HW rendering.
- Flush HDP read cache.

4.5.1.2.3 Method 3

For synchronizing individual surface, the driver can submit following PM4 packets and wait for the synchronization process to complete:

- Submit type 0 PM4 packets to write following registers as:
  CP_COHER_CNTL = specify surface (source / destination).
  CP_COHER_SIZE = surface size.
  CP_COHER_BASE = surface base MC address
- Submit type 3 PM4 packet WAIT_REG_MEM to let ASIC poll register CP_COHER_STATUS. When bit STATUS in the register becomes 0, then surface has been synchronized.
- Submit a time stamp write back packet.
- Driver polls the write back scratch register. When write back value presenting in the register, driver knows all packets in PM4 command buffer have been consumed and synchronization has completed.
- Set a flag for cache flush when switching back to HW rendering.
- Flush HDP read cache.

4.5.2 Flipping Display

When flipping display, driver needs to ensure the renderings to the display surface have completed and the data of the surface has arrived in corresponding memory before switching display engine to the surface.

For pre-R600 ASICs, when flipping display driver does following:

- Flush 3D cache.
- Wait for 2D_IDELCLEAN and 3D_IDELCLEAN.
- Submit a type 0 packet to write CRTC base address register with the address we want to switch to.
- Submit a type 0 packet to poll WAIT_UNTIL register and wait for WAIT_CRTC_PFLIP / WAIT_BOTH_CRTC_PFLIP becomes zero to confirm display flipping completed.

For R6xx/R7xx, driver can use surface synchronization mechanism:

- Submit type 0 PM4 packets to write following registers as
  CP_COHER_CNTL = specify the display surface.
  CP_COHER_SIZE = surface size.
  CP_COHER_BASE = display base MC address
- Submit type 3 PM4 packet WAIT_REG_MEM to let ASIC poll register CP_COHER_STATUS. When bit STATUS in the register becomes 0, then surface has been synchronized.
- Submit a time stamp write back packet.
- Driver polls the write back scratch register. When write back value presenting in the register, driver knows synchronization has completed.
- Submit type 0 PM4 packet to write display base address into register DxGRPH_PRIMARY_SURFACE_ADDRESS.
- Submit a type 3 PM4 packet WAIT_REG_MEM to let ASIC poll(read) register DxGRPH_PRIMARY_SURFACE_ADDRESS and confirm the read value to be what we wrote in, so as to make sure display base address is in effect.
- Submit a type 0 packet to poll WAIT_UNTIL register and wait for DxGRPH_SURFACE_UPDATE_PENDING becomes zero to confirm display flipping completed.

4.5.3 Mode Change

When doing mode change (including normal mode changing, power saving, or PCI-E lane switching), it is necessary to make sure all engines are idle and all caches are flushed.

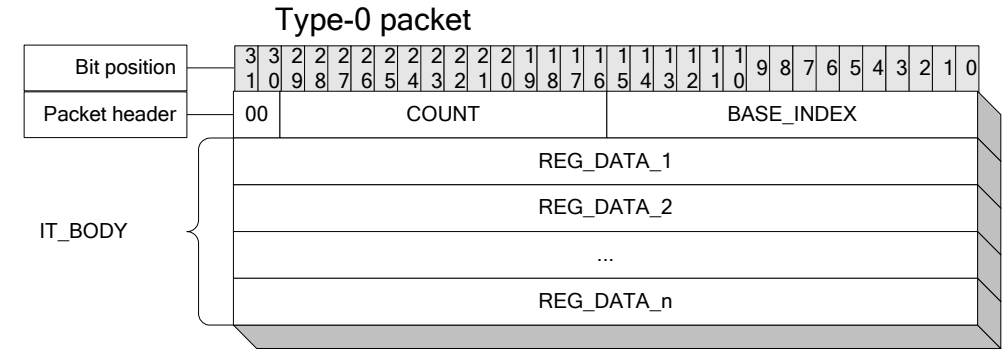
5. PM4

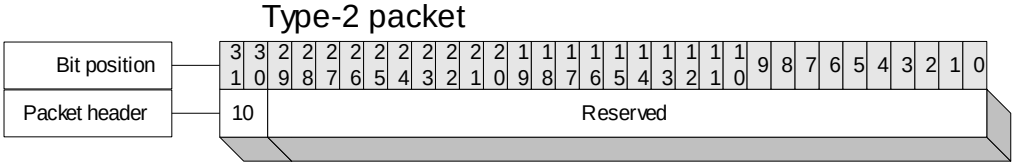
5.1 Overview

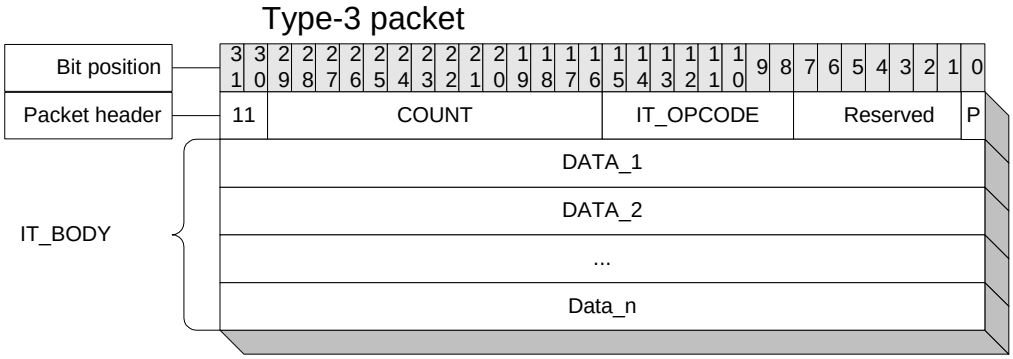
When programming in the PM4 mode, the driver does not write directly to the GPU registers to carry out drawing operations on the screen. Instead, it prepares data in the format of PM4 *Command Packets* in either system or video (a.k.a. local) memory, and lets the Micro Engine to do the rest of the job.

5.2 Packet Types

Three types of PM4 command packets are currently defined. They are types 0, 2 and 3 as shown in the following figure. A PM4 command packet consists of a *packet header*, identified by field HEADER, and an *information body*, identified by IT_BODY, that follows the header. The packet header defines the operations to be carried out by the PM4 micro-engine, and the information body contains the data to be used by the engine in carrying out the operation. In the following, we use brackets [.] to denote a 32-bit field (referred to as DWORD) in a packet, and braces {.} to denote a size-varying field that may consist of a number of DWORDs. If a DWORD consists of more than one field, the fields are separated by '|'. The field that appears on the far left takes the most significant bits, and the field that appears on the far right takes the least significant bits. For example, DWORD [HI_WORD | LO_WORD] denotes that HI_WORD is defined on bits 16-31, and LO_WORD on bits 0-15. A C-style notation of referencing an element of a structure is used to refer to a sub-field of a main field. For example, MAIN_FIELD.SUBFIELD refers to the sub-field SUBFIELD of MAIN_FIELD.







5.2.1 Type-0 Packet

Functionality

Write *N* DWORDs in the information body to the *N* consecutive registers, or to the register, pointed to by the BASE_INDEX field of the packet header.

Format

Ordinal	Field Name
1	[HEADER]
2	[REG_DATA_1]
3	[REG_DATA_2]
…	...
N+1	[REG_DATA_N]

Header Fields

Bit(s)	Field Name	Description
15:0	BASE_INDEX	The BASE_INDEX[15:0] correspond to byte address bits [17:2]. The BASE_INDEX is the DWORD Memory-mapped address. This field width supports a memory map up to 64K DWORDs (256K Bytes).
29:16	COUNT	Count of DWORDs in the information body. Its value should be N-1 if there are N DWORDs in the information body.
31:30	TYPE	Packet identifier. It should be zero.

Information Body

Bit(s)	Field Name	Description
31:0	REG_DATA _x	The bits correspond to those defined for the relevant register. Note the suffix x of REG_DATA_x stands for an integer ranging from 1 to N.

5.2.2 Type-2 Packet

Functionality

This is a filler packet. It has only the header, and its content is not important except for bits 30 and 31. It is used to fill up the trailing space left when the allocated buffer for a packet, or packets, is not fully filled. This allows the CP to skip the trailing space and to fetch the next packet.

Format

Ordinal	Field Name
1	[HEADER]

Header fields

Bit(s)	Field Name	Description
29:0	Reserved	Reserved
31:30	TYPE	Packet identifier. It should be 2.

5.2.3 Type-3 Packet

Functionality

Carry out the operation indicated by field IT_OPCODE.

Format

Ordinal	Field Name
1	[HEADER]
2	{IT_BODY}

Header fields

Bit(s)	Field Name	Description
0	PREDICATE	Predicated version of packet when bit 0 is set.
7:1	Reserved	This field is undefined, and is set to zero by default.
15:8	IT_OPCODE	Operation to be carried out. Bit 15 is the GMC bit. See section 4 for details.
29:16	COUNT	Number of DWORDs -1 in the information body. It is N-1 if the information body contains N DWORDs.
31:30	TYPE	Packet identifier. It should be 3.

Information Body

The information body IT_BODY will be described extensively in the following sections.

5.3 Definition of Type-3 Packets

Type-3 packets have a common format for their headers. However, the size of their information body may vary depending on the value of field IT_OPCODE. The size of the information body is indicated by field COUNT. If the size of the information is *N* DWORDs, the value of COUNT is *N-1*. In the following packet definitions, we will describe the field IT_BODY for each packet with respect to a given IT_OPCODE, and omit the header. The MSB of the IT_OPCODE identifies whether this packet requires the GUI_CONTROL field (described later). A "1" in the MSB of the IT_OPCODE indicates that GUI control is required. A "0" in the MSB of the IT_OPCODE indicates that the GUI_CONTROL should be omitted.

5.3.1 Type-3 Packet Summary

Packet Name	IT_OPCODE	Description
Draw Packets		
DRAW_INDEX_IMMD_BE	0x29	Draw packet used with big endian (BE) immediate (embedded) 16 bit index data in the packet
INDEX_TYPE	0x2A	Sends the current index type to the VGT
DRAW_INDEX	0x2B	Draw packet used when fetching indices from memory is required
DRAW_INDEX_AUTO	0x2D	Draw packet used when auto-index generation is required
DRAW_INDEX_IMMD	0x2E	Draw packet used with immediate (embedded) index data in the packet
NUM_INSTANCES	0x2F	Sends the number of instances to the VGT
MPEG_INDEX	0x3A	MPEG Packed Register Writes and Index Generation
State Management Packets		
SET_CONFIG_REG	0x68	Write Register Data (single context) to a Location on Chip.
SET_CONTEXT_REG	0x69	Write Render State Data (multi context) to a Location on Chip.
SET_ALU_CONST	0x6A	Write ALU Constants to a Location on Chip.
SET_BOOL_CONST	0x6B	Write Boolean Constants to a Location on Chip.
SET_LOOP_CONST	0x6C	Write Loop Constants to a Location on Chip.
SET_RESOURCE	0x6D	Write Resource Constants to a Location on Chip.
SET_SAMPLER	0x6E	Write Sampler Constants to a Location on Chip.
SET_CTL_CONST	0x6F	Write Control Constants to a Location on Chip.
SURFACE_BASE_UPDATE	0x73	Inform the CP which base register has been updated. (work around for CP on RV6xx).
Wait/Synchronization Packets		
MEM_SEMAPHORE	0x39	Sends Signal & Wait semaphores to the Semaphore Block.
WAIT_REG_MEM	0x3C	Wait Until a Register or Memory Location is a Specific Value.
MEM_WRITE	0x3D	Write DWORD to Memory For Synchronization
CP_INTERRUPT	0x40	Generate Interrupt from the Command Stream
SURFACE_SYNC	0x43	Synchronize Surface or Cache
COND_WRITE	0x45	Conditional Write to Memory or to a Register
EVENT_WRITE	0x46	Generate an Event write to the VGT Event Initiator

Packet Name	IT_OPCODE	Description
EVENT_WRITE_EOP	0x47	Generate an Event that Creates a Write to Memory when Completed
Command Buffer Packets		
INDIRECT_BUFFER	0x32	Indirect Buffer Dispatch

5.3.2 Draw Packets

This section documents the formats of the 3D draw packets

5.3.2.1 DRAW_INDEX_IMMD_BE

Functionality

Draws a set of primitives using indices in the packet where 16in32 bit swapping is required. Therefore, this packet should only be used when the indices are 16 bits and the 8in32 bit swapping was done when the CP fetched the command buffer (programmed in the CP_RB_CNTL register). The SOURCE_SELECT field in the DRAW_INITIATOR indicates that the VGT will use immediate data for the indices. This packet is generally used for draw packets with a "small number" of indices. It is faster for the driver to just put the indices into the command buffer instead of copying them to a separate index buffer.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[INDEX_COUNT]	INDEX_COUNT [31:0] – Count of indices in the Index Buffer. *Written to the VGT_NUM_INDICES register.*
3	[DRAW_INITIATOR]	Primitive type and other control. *Written to the VGT_DRAW_INITIATOR register.*
4 to End	[indx16 #1 | indx16 #0]	Index Data. *Written to the VGT_IMMED_DATA register.* See the INDEX_TYPE packet for details on how to specify the 16-bit indices.

5.3.2.2 INDEX_TYPE

Functionality

This packet sets the index type and is considered part of the draw packet sequence. This packet is required to be sent before the first draw command, however, is recommended to set it before each draw command.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[INDEX_TYPE | SWAP_MODE]	[0] Index Type – 0 = 16-bits; 1 = 32-bits. [3:2] Swap Mode[1:0] – Byte swapping control.

5.3.2.3 DRAW_INDEX

Functionality

Draws a set of primitives using fetched indices. The SOURCE_SELECT field in the DRAW_INITIATOR indicates

that the VGT will DMA the indices.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[INDEX_BASE_LO]	Base Address [31:1] of Index Buffer (Word-Aligned). *Written to the VGT_DMA_BASE register.*
3	[INDEX_BASE_HI]	Bits [7:0] Base Address Hi [39:32] of Index Buffer. *Written to the VGT_DMA_BASE_HI register.*
4	[INDEX_COUNT]	INDEX_COUNT [31:0] – Number of indices in the Index Buffer. *Written to the VGT_DMA_SIZE register.* *Written to the VGT_NUM_INDICES register for the assigned context.*
5	[DRAW_INITIATOR]	Draw Initiator Register in the R6xx. *Written to the VGT_DRAW_INITIATOR register for the assigned context.*

5.3.2.4 DRAW_INDEX_AUTO

Functionality

Draws a set of primitives using indices auto-generated by the VGT. The SOURCE_SELECT field in the DRAW_INITIATOR indicates that the VGT need to auto-generate the indices.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[INDEX_COUNT]	INDEX_COUNT [31:0] – Number of indices to generate. *Written to the VGT_NUM_INDICES register.*
3	[DRAW_INITIATOR]	Primitive type and other control. *Written to the VGT_DRAW_INITIATOR register.*

5.3.2.5 DRAW_INDEX_IMMD

Functionality

Draws a set of primitives using indices in the packet. The SOURCE_SELECT field in the DRAW_INITIATOR indicates that the VGT will use immediate data for the indices. This packet is generally used for draw packets with a "small number" of indices. It is faster for the driver to just put the indices into the command buffer instead of copying them to a separate index buffer.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[INDEX_COUNT]	INDEX_COUNT [31:0] – Number of indices that will be written to the VGT_IMMED_DATA register.

		Written to the VGT_NUM_INDICES register.
3	[DRAW_INITIATOR]	Primitive type and other control. *Written to the VGT_DRAW_INITIATOR register.*
4 to End	[indx16 #1 | indx16 #0] or [indx32 #0]	Index Data. *Written to the VGT_IMMED_DATA register.* See the DRAW_INDEX_TYPE packet for details on how to specify the 16 or 32-bit indices.

5.3.2.6 NUM_INSTANCES

Functionality

This packet sets the number of instances for the VGT and is considered part of the draw packet sequence. This packet is required to be sent before the first draw command, however, is recommended to set it before each draw command.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[NUM_INSTANCES]	[31:0] Number of Instances. Minimum value is one; if zero is programmed, it will be treated as one. This allows for a max of $2^{\wedge 32} - 1$ instances.

5.3.2.7 MPEG_INDEX

Functionality

Packed register writes for MPEG and Generation of Indices. The VGT_PRIMITIVE_TYPE:PRIM_TYPE should be a DI_PT_RECTLIST.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header field of the packet.
2	[NUM_INDICES]	Number of Indices the VGT will actually fetch = 3 * number of base indices given at end of this packet. Valid values are 0x0003 to 0x3FFF.
3	[DRAW_INITIATOR]	Written Unconditional to VGT_DRAW_INITIATOR register
4 to 4 + ((NUM_INDICES/3)-1)	[32-Bit INDEX]	First Index of Rect. (0x00000000 to 0xFFFFFFFD) For each "First Index", CP will generate the other 2 indices and output: FIRST_INDEX FIRST_INDEX+1 FIRST_INDEX+2

		All indices are written to the VGT_IMMED_DATA register.

5.3.3 State Management Packets

Previous asics used type 0 packets to setup engine state. While it is still possible to use type 0 packets on R6xx/R7xx chips, the use of these type 3 packets is the recommended method for programming registers.

5.3.3.1 SET_CONFIG_REG

Functionality

This packet loads the register data, which is embedded in the packet, into the chip.

The REG_OFFSET field is a DWord-offset from the starting address. All the register data in the packet is written to consecutive register addresses beginning at the starting address. The starting address for register data is computed as follows:

Reg_Start_Address[17:2] = 0x2000 + REG_OFFSET

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[REG_OFFSET]	[15:0] – Offset in DWords from the register base address (0x2000 in DWs).
3 to N	[REG_DATA]	DWORD Data for Constants.

5.3.3.2 SET_CONTEXT_REG

Functionality

This packet loads the register data, which is embedded in the packet, into the chip.

The REG_OFFSET field is a DWord-offset from the starting address. All the render state data in the packet is written to consecutive register addresses beginning at the starting address. The starting address for register data is computed as follows:

Reg_Start_Address[17:2] = 0xA000 + REG_OFFSET

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[REG_OFFSET]	[15:0] – Offset in DWords from the register base address (0xA000 in DWs).
3 to N	[REG_DATA]	DWORD Data for Constants.

5.3.3.3 SET_ALU_CONST

Functionality

This packet loads the ALU constant data, which is embedded in the packet, into the chip.

The CONST_OFFSET field is a DWord-offset from the starting address. All the constant data in the packet is written to consecutive register addresses beginning at the starting address. The starting address for ALU constants is computed as follows:

Reg_Start_Address[17:2] = SQ_ALU_CONSTANT0_0[17:2] + CONST_OFFSET

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[CONST_OFFSET]	[15:0] – Offset in DWords from the ALU const base address (DW address of SQ_ALU_CONSTANT0_0).
3 to N	[CONST_DATA]	DWORD Data for Constants.

5.3.3.4 SET_BOOL_CONST

Functionality

This packet loads the constant Bool data, which is embedded in the packet, into the chip.

The CONST_OFFSET field is a DWord-offset from the starting address. All the constant data in the packet is written to consecutive register addresses beginning at the starting address. The starting address for Boolean constants is computed as follows:

Reg_Start_Address[17:2] = SQ_BOOL_CONST_0[17:2] + CONST_OFFSET

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[CONST_OFFSET]	[15:0] – Offset in DWords from the BOOL const base address (DW address of SQ_BOOL_CONSTANT0_0).
3 to N	[CONST_DATA]	DWORD Data for Constants.

5.3.3.5 SET_LOOP_CONST

Functionality

This packet loads the constant Loop data, which is embedded in the packet, into the chip.

The CONST_OFFSET field is a DWord-offset from the starting address. All the constant data in the packet is written to consecutive register addresses beginning at the starting address. The starting address for Loop constants is computed as follows:

Reg_Start_Address[17:2] = SQ_LOOP_COUNT_CONST_0[17:2] + CONST_OFFSET

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet

2	[CONST_OFFSET]	[15:0] – Offset in DWords from the LOOP const base address (DW address of SQ_LOOP_COUNT_CONSTANT0_0).
3 to N	[CONST_DATA]	DWORD Data for Constants.

5.3.3.6 SET_RESOURCE

Functionality

This packet loads the Resource data, which is embedded in the packet, into the chip.

The CONST_OFFSET field is a DWord-offset from the starting address. All the Resource data in the packet is written to consecutive register addresses beginning at the starting address. The starting address for Resources is computed as follows:

Reg_Start_Address[17:2] = SQ_TEX_RESOURCE_WORD0_0[17:2] + CONST_OFFSET

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[CONST_OFFSET]	[15:0] – Offset in DWords from the RESOURCE const base address (DW address of SQ_ TEX_RESOURCE_WORD0_0).
3 to N	[CONST_DATA]	DWORD Data for Constants.

5.3.3.7 SET_SAMPLER

Functionality

This packet loads the Sampler data, which is embedded in the packet, into the chip.

The CONST_OFFSET field is a DWord-offset from the starting address. All the Sampler data in the packet is written to consecutive register addresses beginning at the starting address. The starting address for Samplers is computed as follows:

Reg_Start_Address[17:2] = SQ_TEX_SAMPLER_WORD0_0[17:2] + CONST_OFFSET

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[CONST_OFFSET]	[15:0] – Offset in DWords from the SAMPLER const base address (DW address of SQ_ TEX_SAMPLER_WORD0_0).
3 to N	[CONST_DATA]	DWORD Data for Constants.

5.3.3.8 SET_CTL_CONST

Functionality

This packet loads the constant Control data, which is embedded in the packet, into the chip.

The CONST_OFFSET field is a DWord-offset from the starting address. All the constant data in the packet is written to consecutive register addresses beginning at the starting address. The starting address for Control constants is computed as follows:

Reg_Start_Address[17:2] = mmSQ_VTX_BASE_VTX_LOC[17:2] + CONST_OFFSET

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[CONST_OFFSET]	[15:0] – Offset in DWords from the CTL const base address (DW address of mmSQ_VTX_BASE_VTX_LOC).
3 to N	[CONST_DATA]	DWORD Data for Constants.

5.3.3.9 SURFACE_BASE_UPDATE

Functionality

This packet is inserted by the driver each time a surface base register is written. It should be inserted with the corresponding bit set to indicate which surface base register was just written. This is needed on RV6xx chips for the CP surface synchronization logic. R600 and R7xx do not require the use of this packet.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[COHER_BASE1	Bits [14] COHER_DEST_BASE_1 has been updated.
	COHER_BASE0	Bits [13] COHER_DEST_BASE_0 has been updated.
	STRMOUT_BASE3	Bits [12] VGT_STRMOUT_BUFFER_BASE_3 has been updated.
	STRMOUT_BASE2	Bits [11] VGT_STRMOUT_BUFFER_BASE_2 has been updated.
	STRMOUT_BASE1	Bits [10] VGT_STRMOUT_BUFFER_BASE_1 has been updated.
	STRMOUT_BASE0	Bits [9] VGT_STRMOUT_BUFFER_BASE_0 has been updated.
	COLOR7_BASE	Bits [8] CB_COLOR7_BASE has been updated.
	COLOR6_BASE	Bits [7] CB_COLOR6_BASE has been updated.
	COLOR5_BASE	Bits [6] CB_COLOR5_BASE has been updated.
	COLOR4_BASE	Bits [5] CB_COLOR4_BASE has been updated.
	COLOR3_BASE	Bits [4] CB_COLOR3_BASE has been updated.
	COLOR2_BASE	Bits [3] CB_COLOR2_BASE has been updated.
	COLOR1_BASE	Bits [2] CB_COLOR1_BASE has been updated.
	COLOR0_BASE	Bits [1] CB_COLOR0_BASE has been updated.
	DEPTH_BASE]	Bits [0] DB_DEPTH_BASE has been updated.

5.3.4 WAIT-ON / Synchronization Packets

5.3.4.1 MEM_SEMAPHORE

Functionality

This packet supports Signal and Wait Semaphores. The Signal is an end-of-pipe event with cache flush, where as the Wait is a top of pipe event.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[ADDRESS_LO]	[31:3] Lower bits of QWORD-Aligned Address.
3	[SEM_SEL | ADDRESS_HI]	[31:29] – Select either Wait or Signal. This is a multi-bit field to be DW compatible with EVENT_WRITE_EOP. 110 – Signal Semaphore. 111 – Wait Semaphore. [7:0] – Upper bits (39:32) of Address

5.3.4.2 WAIT_REG_MEM

Functionality

The packet can be processed by either the CP PFP or the CP ME, as indicated by the ENGINE field. The CP PFP is limited to polling a memory location, where the ME can be programmed to poll either a memory location or a register (indicated by MEM_SPACE). The polled value is then tested against the reference value given in the command packet. The test is qualified by both the specified function and mask. If the test passes, the parsing continues. If it fails, the CP waits for the Wait_Interval * 16 Clocks, then tests the Poll Address again.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[ENGINE | MEM_SPACE | FUNCTION]	[31:9] - Reserved [8] - ENGINE: 0=>ME , 1=>PFP [7:5] - Reserved [4] - MEM_SPACE: 0=>Register, 1=>Memory. If ENGINE = PFP, only Memory is valid. [3] - Reserved [2:0] - FUNCTION 000 – Always (Compare Passes). Still does read operation and waits for read results to come back. 001 – Less Than (<) the Reference Value. 010 – Less Than or Equal (<=) to the Reference Value. 011 – Equal (=) to the Reference Value. 100 – Not Equal (!=) to the Reference Value. 101 – Greater Than or Equal (>=) to the Reference Value. 110 – Greater Than (>) the Reference Value. 111 – Reserved. If ENGINE = PFP, only "101/Greater Than or Equal " is valid
3	[POLL_ADDRESS_LO]	Lower portion of Address to poll If the address is a memory location then bits [31:4] specify the lower bits of the address and Bits [1:0] specify SWAP used for memory read. Bits[3:2] must be programmed to '0'. ***Note: Address must be 16 byte aligned.*** If the address is a memory-mapped register, then bits [15:0] is the DWORD memory-mapped register address that the CP will read.
4	[POLL_ADDRESS_HI]	Higher portion Address to poll If the address is a memory location then bits [7:0] specify bits 39:32 of the address.

		If the address is a memory-mapped register, then this ordinal should be 0.
5	[REFERENCE]	[31:0] - Reference Value.
6	[MASK]	[31:0] - Mask for Comparison.
7	[POLL_INTERVAL]	[15:0] - Poll_Interval: Interval to wait between the time an unsuccessful polling result is returned and a new poll is issued. Time between these is 16*Poll_Interval clocks.

5.3.4.3 MEM_WRITE

Functionality

This command packet provides the ability to either write two DWORDs to a QWORD-aligned memory location or to write one DWORD to a DWORD-aligned memory location at the top of the graphics pipe.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[ADDRESS_LO | SWAP]	[31:3] Lower bits of QWORD-Aligned Address. OR [31:2] Lower bits of DWORD-Aligned Address (when DATA32 bit is set in ordinal 3). Bits [1:0] – Swap function used for data write.
3	[DATA32 | WR_CONFIRM | CNTR_SEL | ADDRESS_HI]	[18] – Data32 writes only the lower 32-bits to memory when set to '1'. [17] – Write Confirm is requested for this write when set. [16] – Selects sending embedded packet Data (0) or CP's copy of the 64 –bit GPU counter (1) value. [7:0] – Upper bits (39:32) of Address
	[DATA_LO]	[31:0] Data
5	[DATA_HI]	[31:0] Data – Discarded when DATA32 is set.

5.3.4.4 CP_INTERRUPT

Functionality

Sets the interrupt status corresponding to the stream where the CP_INTERRUPT packet was parsed as indicated by the INT_ID field in ordinal 2.

This provides the ability for the Driver to identify the stream that generates the interrupt. Because the Driver puts this packet in the command stream, it can also use the interrupt as an "almost-finished-parsing" flag from the CP if it is placed near the end of a command stream.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header field of the packet.
2	[INT_ID]	The Driver must set the appropriate interrupt bit. Bit 31 - Interrupt for Ring Buffer Bit 30 - Interrupt for Indirect Buffer #1 Bit 29 - Interrupt for Indirect Buffer #2 Bit 28:0 - Reserved

5.3.4.5 SURFACE_SYNC

Functionality

The purpose of this packet is to allow the driver to place the surface sync commands as one atomic packet and to allow the driver to send the same COHER_CNTL value regardless of the ASIC. The first feature simplifies the driver in that the end of command buffer calculations becomes easier. This calculation is important since the surface sync sequence should not cross command boundaries. The second feature simplifies the driver since it does not have to do if-then-else decision for the VC, as an example. Since the CP uCode is unique per ASIC it can just do the right action regarding the VC. If the VC does exist, then there is no change. If it does not exist, then it needs to 'OR' the VC_ACTION_ENA bit with the TC_ACTION_ENA bit and use that as the final TC_ACTION_ENA.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[COHER_CNTL]	See the CP_COHER_CNTL register for a definition
3	[COHER_SIZE]	Coherency Surface Size has a granularity of 256 Bytes.
4	[COHER_BASE]	CP_COHER_BASE[31:0] = virtual memory address [39:8]. This value times 256 is the byte address of the start of the surface to be synchronized (to create the high 32-bits of a 40-bit virtual device address).
5	[POLL_INTERVAL]	[15:0] - Poll_Interval: Interval to wait between the time an unsuccessful polling result is returned and a new poll is issued. Time between these is 16*Poll_Interval clocks.

5.3.4.6 COND_WRITE

Functionality

The CP reads either a memory or register location (indicated by POLL_SPACE) and tests the polled value with the reference value provided in the command packet. The test is qualified by both the specified function and mask.

If the test passes, the write occurs to either a register or memory depending on WRITE_SPACE.

If the test fails, the CP skips the write. In either case, the CP then continues parsing the command stream.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[WRITE_SPACE | POLL_SPACE | FUNCTION]	31:9: Reserved WRITE_SPACE 8: 0=>Register, 1=>Memory 7:5 : Reserved POLL_SPACE 4: 0=>Register, 1=>Memory FUNCTION [2:0] 000 – Always (Compare Passes). Still does read operation and waits for data to return. 001 – Less Than (<) the Reference Value. 010 – Less Than or Equal (<=) to the Reference Value. 011 – Equal (=) to the Reference Value. 100 – Not Equal (!=) to the Reference Value. 101 – Greater Than or Equal (>=) to the Reference Value. 110 – Greater Than (>) the Reference Value.

		111 – Reserved
3	[POLL_ADDRESS_LO]	Lower portion of Address to poll If the address is a memory location then bits [31:2] specify the lower bits of the address and [1:0] is the swap code to be used. If the address is a memory-mapped register, then bits [15:0] is the DWORD memory-mapped register address that the CP will read.
4	[POLL_ADDRESS_HI]	Higher portion Address to poll If the address is a memory location then bits [7:0] specify bits 39:32 of the address. If the address is a memory-mapped register, then this ordinal is a don't care.
5	[REFERENCE]	Reference Value [31:0].
6	[MASK]	Mask for Comparison [31:0]
7	[WRITE_ADDRESS_LO]	If WRITE_SPACE = Register: WRITE_ADDRESS[15:0] – DWORD memory-mapped register address that the will be written. If WRITE_SPACE = Memory: WRITE_ADDRESS[31:2] – DWORD-Aligned Address of destination memory location. WRITE_ADDRESS[1:0] – SWAP Used for Memory Write.
8	[WRITE_ADDRESS_HI]	If WRITE_SPACE = Register: This DWORD does not matter If WRITE_SPACE = Memory: Bits 7:0 - WRITE_ADDRESS[39:32]
9	[WRITE_DATA]	Write Data[31:0] that will be conditionally written to the ADDRESS.

5.3.4.7 EVENT_WRITE

Functionality

This packet is used when the driver wants to create a non-TimeStamp/Fence event. See EVENT_WRITE_EOP to send timestamps and fences.

The EVENT_WRITE supports two categories of events. Those are:

1. 4 DW (ordinal) event where special handling is required (ZPASS).
2. 2 DW (ordinal) event where no special handling is required; CP just writes EVENT_INITIATOR (ordinal 2 from the packet) into VGT event initiator register and ordinals 3 and 4 do not exist, i.e., the packet is only 2 ordinals for these events. These include all other events.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[EVENT_INITIATOR]	EVENT_INITIATOR[5:0] The CP writes this value to the VGT_EVENT_INITIATOR register for the assigned context.
3	[ADDRESS_LO]	Bits [31:3] – Lower bits of QWORD-Aligned Address. Bits [2:0] – Reserved & must be programmed to zero. Driver should only supply this ordinal for ZPASS (Occlusion).
4	[ADDRESS_HI]	Bits [7:0] – Upper bits of Address [39:32] Driver should only supply this ordinal for ZPASS (Occlusion).

5.3.4.8 EVENT_WRITE_EOP

Functionality

This packet is used when the driver wants to create any end-of-pipe event. "TS" indicates either fence data or actual timestamp will be written back.

Supported Events are:

1. Cache Flush TS: provides the driver with a pipelined fence/timestamp indicating that the CBs, DBs, and SMX have completed flushing their caches.
2. Cache Flush And Inval TS: same as above but the CBs, DBs, and SMX, also invalidate their caches before sending the pulse back to the CP.
3. Bottom Of Pipe TS: provides the driver with a pipelined timestamp indicating that the CBs, DBs, and SMX have completed all work before the time stamp. This can be considered a read EOP event in that all reads have occurred but the CBs/DBs/SMX have not written out all the data in their caches.

Use the EVENT_WRITE packet for all others.

Supported actions when requested event has completed are:

1. timestamps – 64-bit global GPU clock counter value with optional interrupt
2. fences - 32 or 64 bit embedded data in the packet with optional interrupt.
3. traps - interrupt only.

The CP will generate the end-of-pipe event given in the packet by writing to the VGT_EVENT_INITIATOR register. The CP will also set up a write to memory and/or interrupt that will occur at the completion of the specified event. The Address & Data fields are therefore required.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[EVENT_INITIATOR]	[5:0] - EVENT_INITIATOR The CP writes this value to the VGT_EVENT_INITIATOR register for the assigned context.
3	[ADDRESS_LO]	[31:2] – Lower bits of DWORD-Aligned Address if DATA_SEL = "001", [31:3] – Lower bits of QWORD-Aligned Address if DATA_SEL = "010" or "011", else don't care. Bits [1:0] – Reserved & must be programmed to zero.
4	[DATA_SEL | INT_EN | ADDRESS_HI]	[31:29] - DATA_SEL – Selects Source of Data to be written for a End-of-Pipe Done event. 000 – None, i.e., Discard Data. Used when only an interrupt is needed. Program INT_SEL to "01". 001 - Send 32-bit Data Low (Discard Data High). 010 - Send 64-bit Data. 011 - Send current value of the 64 bit global GPU clock counter. *10x - Reserved.* *110 – Reserved.* *111 - Reserved.* [25:24] - INT_SEL Selects interrupt action for End-of-Pipe Done

		event. 00 - None (Do not send an interrupt). 01 - Send Interrupt Only. Program DATA_SEL "000'. 10 - Send Interrupt when Write Confirm is received from the MC. [7:0] - ADDR_HI, address bits[39:32]. External memory address written for a End-of-Pipe Done event. Read returns last value written to memory.
5	[DATA_LO]	Data [31:0] value that will be written to memory when event occurs. Driver should always supply this ordinal
6	[DATA_HI]	Data [63:32] value that will be written to memory when event occurs. Driver should always supply this ordinal

5.3.5 Command Buffer Packets

5.3.5.1 INDIRECT_BUFFER

Functionality

This packet initiates an indirect buffer fetch.

Format

Ordinal	Field Name	Description
1	[HEADER]	Header of the packet
2	[IB_BASE_LO]	[31:2] - Indirect Buffer Base Address[31:2] – DW-Aligned [1:0] – Swap function used for data write.
3	[IB_BASE_HI]	[7:0] – Upper bits of Address [39:32]
4	[IB_SIZE]	Indirect Buffer Size [19:2] – Size of the Indirect Buffer in DWORDs. ***Note: Size must be modulo 4. Use Type-2 "filler packets" to pad buffer if necessary.***

6. Driver Notes

6.1 Shaders

6.1.1 SQ Constant Registers (ALU, sampler, tex/vertex-buffer, loop, vtx-inst-start/base)

It is very important that SQ constant register writes be optimized. When there are too many constant updates, the CP/GRBM can become stalled. The following rules should be used to optimize these register writes:

- **Never write the same constant more than once between draw calls.** This would count as 2 updates.
- **Do not write more constants than absolutely needed.**
- **Always write a complete constant** (e.g. you cannot update just the Y channel of an alu constant).
- **Use the SET_ALU_CONST packet**

6.1.2 Number of shader threads in R600

R600 requires the # of threads of each type to be a multiple of 4. This affects the SQ_THREAD_RESOURCE_MGMT register. Note that the fields in this register are "true" values, not "value-1".

6.1.3 Shader memory export instructions

In R600, RV630 and RV610, all export instructions which go through the SMX (scratch, reductions, rings and stream-out) must have the barrier bit set.

In RV670, the channel mask for an indexed-memexport-read must be set to a non-zero value. The value may correspond to the read-channel mask for the data (if the SMX feature is enabled). This is not an issue for R600, RV630 and RV610.

6.1.4 Shader GPR Indexing may return incorrect result

This affects R600, RV630 and RV610, but not RV670 or RS780.

Given a GPR indexing instruction such as:

```
ADD   R0, R5,R6
MOV  R7, R[A0.x + 0]   // A0.x != 0
```

The hardware will incorrectly replace R[A0.x+3] with PV. The rule the hardware applies is: use PV if the source GPR is the same as the destination of the previous instruction. It does not account for gpr indexing (since that gets applied later), and so it thinks the source of the MOV is R0 (it is based solely on the +0 part).

The same problem can happen if the GPR indexing is in the destination, and the next instruction uses a GPR which will be confused as being the same as the destination:

```
MOV R[A0.x +2], R33
ADD  R20, R20, R2    // H/w thinks R2 is the same as the prev dest and will substitute PV
```

So the rules are: if the hardware will mistakenly apply PV, you must insert a NOP or other instructions. Also, you must still obey the rule that after a gpr-indexed write (destination), you must not use the result of the write in the next reason (for similar reasons – hardware will not know to use PV if the source really does match the last write from the gpr-index move).

6.1.5 Restrictions on number of ES/GS threads in the SQ

In RV670, when turning on the ES/GS modes, you need to set the SQ thread limit for ES and GS to a minimum of 16 entries regardless of CUT mode setting.

6.2 DB

6.2.1 Setting Driver Provided DB Hints

The DB needs hints from the driver to when to enable certain performance optimizations such as EarlyZ, Re-Z, HiZ and HiS. Most of these are in DB_SHADER_CONTROL, which has descriptions in the register spec. In general Late Z is best for very short shaders, Early Z for middle length shaders and ReZ is only good for very long shaders. The Z_ORDER field should be derived from the required shader setup.

6.2.2 ReZ

The driver must turn off ReZ mode if the ZFunc is set to Not Equal and Z writes are enabled. Instead of Re_then_early, earlyz_then_late should be used. Or instead of Re_then_late, late should be used.

6.2.3 R6xx Fog

- MRT0 is always fogged when fog is enabled
- With multiwrite_enable, we must fog all of the MRTs
- All other cases it doesn't matter what we do
- The fog factor only comes from pixel shader color output 0 and MRT1-7 are only fogged if multiwrite_enable is set
- The color in MRT1-7 will be that of MRT0 when multiwrite_enable is set
- A fog factor exported in MRT1-7 is essentially ignored and not used
- Fog is not supported for render targets using 32FP components

Additionally, the fog factor, source color, and fog color should not be set to NAN, INF, or denorm values.

6.3 CB

6.3.1 DB->CB Copies: RV6xx

In RV610, RV630, RV620, and RV635 but not R600, RV670, or RS780, the DB->CB copies needs the following programming sequence to ensure the DB will read the Z/Stencil data.

1. Z_ENABLE = true
2. STENCIL_ENABLE = true
3. Z_WRITE_ENABLE = false
4. STENCILWRITEMASK = 0xff
5. Z vertices = 0.0. (like a Z clear to 0.0)
6. ZFUNC= LEQUAL
7. STENCILFUNC = REF_ALWAYS
8. STENCILZPASS = STENCIL_KEEP
9. STENCILZFAIL = STENCIL_INCR_CLAMP
10. DB_RENDER_OVERRIDE.FORCE_HIZ_ENABLE = FORCE_DISABLE

6.3.2 8 MRT + 8-Sample Issue

On R600 and RV630, if 8 MRTs are in use, there are 8 samples per pixel, and MRT7 had a fast-clear performed on it, the driver must perform an expand_color on MRT7 if there are 8 samples per pixel to ensure proper operation. This is only required if all 8 MRTs are enabled, none of them are set to COLOR_INVALID, none of them have 0 as the target_mask, and none of them have 0 as the shader_mask.

6.3.3 4 MRT + 8-Sample Restriction

On RV610, there is a maximum of 4 enabled render targets for 8-sample surfaces.