cd Author: Randy Ramsey

ISSUED TO:	COPY NO.

GFX9 SPI Specification

Rev 1.0 – Last Edit: 3-Nov-16

THIS DOCUMENT CONTAINS INFORMATION THAT COULD BE SUBSTANTIALLY DETRIMENTAL TO THE INTEREST OF AMD THROUGH UNLICENSED USE OR UNAUTHORIZED DISCLOSURE.

1. **Preserve this document's integrity:**
   - ⇒ **Do not reproduce any portions of it.**
   - ⇒ **Do not separate any pages from this cover.**
2. **This document is issued to you alone. Do not transfer it to or share it with another person, even within your organization.**
3. **Store this document in a locked cabinet accessible only by authorized users. Do not leave it unattended.**
4. **When you no longer need this document, return it to AMD. Please do not discard it.**

"Copyright 2012, Advanced Micro Devices, Inc. ("AMD"). All rights reserved. This work contains confidential, proprietary to the reader information and trade secrets of AMD. No part of this document may be used, reproduced, or transmitted in any form or by any means without the prior written permission of AMD."

AMD, the AMD Arrow Logo and combinations thereof are trademarks of Advanced Micro Devices, Inc. PCIe is a registered trademark of PCI-SIG. HDMI is a trademark of HDMI Licensing, LLC.

AMD (NYSE: AMD) is a semiconductor design innovator leading the next era of vivid digital experiences with its ground-breaking AMD Fusion Accelerated Processing Units (APU). AMD's graphics and computing technologies power a variety of devices including PCs, game consoles and the powerful computers that drive the Internet and businesses. For more information, visit http://www.amd.com.

Revision History

Date	Revision	Description

Table of Contents

1 INTRODUCTION 6

1.1 OPEN ISSUES 6
1.2 DEFINITIONS 6
1.2.1 Acronyms 6
1.2.2 Terminology 7
1.3 TOP LEVEL DESCRIPTION 7
1.3.1 SPI Chip Level Data Flow Diagram 7
1.3.2 Chip Level Diagram 9
1.3.3 SPI Block Diagram 10

2 FEATURES / FUNCTIONALITY 11

2.1 STAGE AND ORGANIZE DATA FOR SHADER LAUNCH 11
2.2 COMPUTE SHADER (CS) 11
2.2.1 Resource Probing 13
2.2.2 Threadgroup Ordering 13
2.2.3 Threadgroup Halting and Discarding 14
2.2.4 Queue Status 15
2.2.5 Unordered Dispatches 15
2.2.6 State Forwarding to SQG 15
2.2.7 First Wave of Dispatch 15
2.2.8 Compute Shader Index Terms 15
2.3 VGT-SPI "VERT" SHADERS 16
2.3.1 ES, GS, VS Processing 17
2.3.2 On-chip GS 18
2.3.3 Tessellation 18
2.3.4 Distributed Tessellation 19
2.3.4.1 Work Creation Description 20
2.3.4.2 Offchip LDS ID Changes 21
2.3.4.3 Offchip LDS Deallocation Changes 21
2.4 PIXEL SHADER (PS) 23
2.4.1 Pixel Data Flow 24
2.4.1.1 Calculate Per-Pixel IJ Barycentric Coordinates 25
2.4.1.2 Pull Model 26
2.4.2 Scale Resolution Based on Screen Location (9.125) 27
2.4.2.1 Visualizing the Scaling 27
2.4.2.2 Impacts to BCI Equations 29

3 END OF SPEC UPDATES, BEYOND THIS POINT INFO MAY BE OUT OF DATE 31

3.1.1 Support for 16 pixels per SH 31
3.1.2 Unique Sample Positions per Pixel 34
3.2 LDS PARAMETER DATA LOADING FOR PIXELS 34
3.2.1 Organization of Data in the Parameter Cache 35
3.2.2 VS-PS Remapping 36
3.2.3 Flat Shading 36
3.2.4 Point Sprite Override 37
3.2.5 PARAM_GEN 37
3.2.6 Support Deeper Parameter Cache and Avoid Duplicate Data 37
3.2.6.1 Performance 37
3.3 PIXEL SHADER VGPR INITIALIZATION 38
3.4 VERTEX/PIXEL SYNCHRONIZATION 40

3.5 COMBINED DATA FLOW 41
3.6 RESOURCE ALLOCATION 42
3.6.1 CU and SIMD Assignment 42
3.6.1.1 SIMD Assignment for Work Distribution and Input Bandwidth 42
3.6.2 GPR Management 43
3.6.3 LDS Management 43
3.6.4 Wave Buffer 43
3.6.5 Scratch 43
3.6.6 Barrier 45
3.6.7 Bulky CS Threadgroups 45
3.6.8 Position Buffer and Parameter Cache 46
3.6.8.1 Late VS Allocation 46
3.6.9 Allocation Priority 47
3.6.10 Virtualization of Compute Unit Masks 50
3.6.11 Resource Reservations 50
3.6.12 Multiplier for Resource Limits 51
3.7 EXPORT ARBITRATION 51
3.7.1 Maintaining GDS order 53
3.7.2 Export Granting 53
3.8 PERSISTENT STATE 54
3.9 PARTIAL FLUSH EVENTS 54
3.10 WAVE/EVENT ORDERING 55
3.11 EVENT COLLECTION 55
3.12 H/V (HORIZONTAL/VERTICAL) PIXEL PICKER (FOR DEBUG AND PERFORMANCE ANALYSIS) 55
3.13 WAVEFRONT LIFETIME STATUS COUNTERS 56

4 PERFORMANCE 58

4.1 BARYCENTRIC CALCULATION 59
4.2 PARAMETER CACHE READ 59
4.3 GPR LOADING 59
4.4 PIXEL 59
4.5 GRAPHICS BALANCED THROUGHPUT CASES 59
4.6 PERFORMANCE COUNTERS 61
4.6.1 Performance Counter Binning 61

5 CLOCK GATING 61

Table of Figures

Figure 1 – SPI Chip Level Data Flow Diagram 8
Figure 2 – Chip Level Diagram 9
Figure 3 – Top Level Connectivity Block Diagram 10
Figure 4 – Block Diagram 11
Figure 5 – CS Data Flow 12
Figure 6 – Async Compute Block Diagram 13
Figure 7 – CS Threadgroup Ordering 14
Figure 8 – CS Thread Count Increment Example 16
Figure 9 – "Vertex" Data Flow VGT-SPI 16
Figure 10 – VGT ES, GS, VS Vertex Input 17
Figure 11 – LS,HS,ES,GS,VS Vertex Input 19
Figure 12 – Pixel Input Data 24
Figure 13 – Color Export Bus Arbitration, 1RB 32
Figure 14 – Color Export Bus Arbitration, 2RB 33
Figure 15 – Color Export Bus Arbitration, 4RB 33
Figure 16 – Color Export Bus Arbitration, 2RB+ 34
Figure 17 – LDS Logical Layout 35
Figure 18 – Parameter Cache Data Organization 36
Figure 19 – Combined Data Flow 41
Figure 20 – Persistent State Update FIFOs 54
Figure 21 – Persistent State Update FIFOs 56
Figure 22 - Performance, Balanced Throughput Case, VS-PS 60
Figure 23 – Performance, Balanced Throughput Case, ES-GS-VS-PS 60
Figure 24 – Performance, Balanced Throughput Case, LS-HS-ES-GS-VS-PS 61

1 Introduction

This document describes the requirements, functionality, and target performance of the Shader Processor Input (SPI) block.

1.1 Open Issues

1.2 Definitions

1.2.1 Acronyms

SPI - Shader Processor Input
SC - Scan Converter
SQ - Sequencer
SQC – Sequencer Cache
SQG – SQ Global Block, instanced in SPI
SX - Shader Export
SP - Shader Processor
CP – Command Processor
CPG – Command Processor, Graphics
CPC – Command Processor, Compute
SE – Shader Engine
SH – Shader Array
CU – Compute Unit
SIMD - Single Instruction Multiple Data unit in the shader processor (SP).
UL - Upper Left
UR - Upper Right
LL - Lower Left
LR - Lower Right
VGPR – Vector General Purpose Register in the SP
SGPR – Scalar General Purpose Register in the SQ
CS – Compute shader
LS – API Vertex shader stage when doing tessellation, writes to LDS
HS – Hull shader stage of tessellation
VS- Vertex shader, could be normal vertices, final pass of a Geometry Shader, or domain shader.
GS - Geometry Shader, processes primitives.
ES - Export Shader, first vertex pass of a Geometry Shader that processes vertices.
PS – Pixel Shader, processes pixels.
VSR –Vertex Input Staging Register, holds input data for vertex threads.
PSR –Pixel Input Staging Register, holds input data for pixel threads.
LDS – Local Data Store
se_id – Shader Engine Identification Number
sh_id – Shader Array Identification Number
MSAA – Multi-sample Anti-Aliasing
EQAA – Enhance Quality Anti-Aliasing

1.2.2 Terminology

Event – an event is a special token sent through the graphics pipeline which can be used to enforce synchronization, flush caches, and report status back to the CP. All blocks pipeline these tokens and keep them ordered with other graphics data.
Thread: one instance of a shader program being executed on a wavefront. Each thread has its own data which is unique from any other thread.
Wavefront: This is the basic unit of work. There are 64 threads per wavefront. It is a group of threads that can be executed simultaneously on a SIMD.
Threadgroup, Subgroup: Group of threads that may span several wavefronts. All threads are guaranteed to run on the same CU. This allows for shared CU resources such as the Local Data Store (LDS) and synchronization resources across all threads.
Tessellation Engine: A VGT module that implements DX11 tessellation functionality.
Pixel Quad: A 2x2 pixel region.
Pixel Center: Current pixel's screen coordinates, given as PIX_X.5, PIX_Y.5.
Pixel Centroid: Current pixel's centroid in screen coordinates, defined as the covered sample location closest to pixel center. If all samples of a pixel are hit, center will be used for centroid even if center is not one of the current sample locations.
Pixel Sample: Location of the sample ID of the current iteration when running at sample frequency.
Facedness: The PA determined face flag indicating front or back facing.
Param_gen: Automatically generated ST texture coordinates, typically used with points.
SIMD: Single Instruction Multiple Data unit in the shader processor (SP).
Shader Array: A combination of blocks separate and unique for shader processing, including a shader core consisting of Compute Units.
new_vector aka fpos, first_prim_of_slot: Parameter cache sync token received from the SC for pixels and used to make sure the SPI waits for VS to finish exporting parameter data before pixels start trying to read it.
Helper pixel: Any non-hit pixel being processed as a part of a quad with other hit pixels.

1.3 Top Level Description

The main purpose of the SPI is to manage shader resources and provide shader input data to the GPRs and wavefronts to the SQ. It accumulates "vertex" type shader input data from the VGT (VS, GS, ES, HS, LS) into wavefronts. It receives compute shader (CS) data and state from the CPG and CPC on csdata interfaces. Resources required to process wavefronts and CU/SIMD assignment in the shader array (SH) are managed by the SPI in terms of allocation and de-allocation. SPI passes data through for the VGT verts and prims. For HS and GS, SPI unrolls threadgroups and subgroups into wavefronts. For CS, SPI unrolls threadgroups into wavefronts and generates an index per thread based on the threadgroup size. Pixel quad data delivered from the SC is accumulated into wavefronts. The SPI processes this data, per pixel, to interpolate and produce barycentric gradient data (IJ) or screen X, Y, and/or primitive faceness data. The SPI loads IJ data into VGPRs and coordinates moving primitive attribute data from the parameter caches into a CU Local Data Store (LDS) for the pixel shader to use for attribute interpolation. SPI synchronizes the vertex shader attribute exports with the pixel shader reading those attributes, guaranteeing that attribute data has been written to the parameter cache before allowing PS to read.

1.3.1 SPI Chip Level Data Flow Diagram

Figure 1 shows the blocks and major data paths directly and functionally associated with the SPI.

Inputs from the VGT: subgroups, waves, events, and vertex input data for the data types VS, GS, ES, HS, LS.
Inputs from the SC: pixel data including coverage, primitive information and events.
From the CPG: compute state, events, threadgroups for GFX.
From the CPC: compute state, events, threadgroups for async compute.
Shader input data into the S/VGPRs and wavefront input to the SQ.
VS position and parameter cache data writes to the SX and PC.
Parameter cache read and LDS write controls.

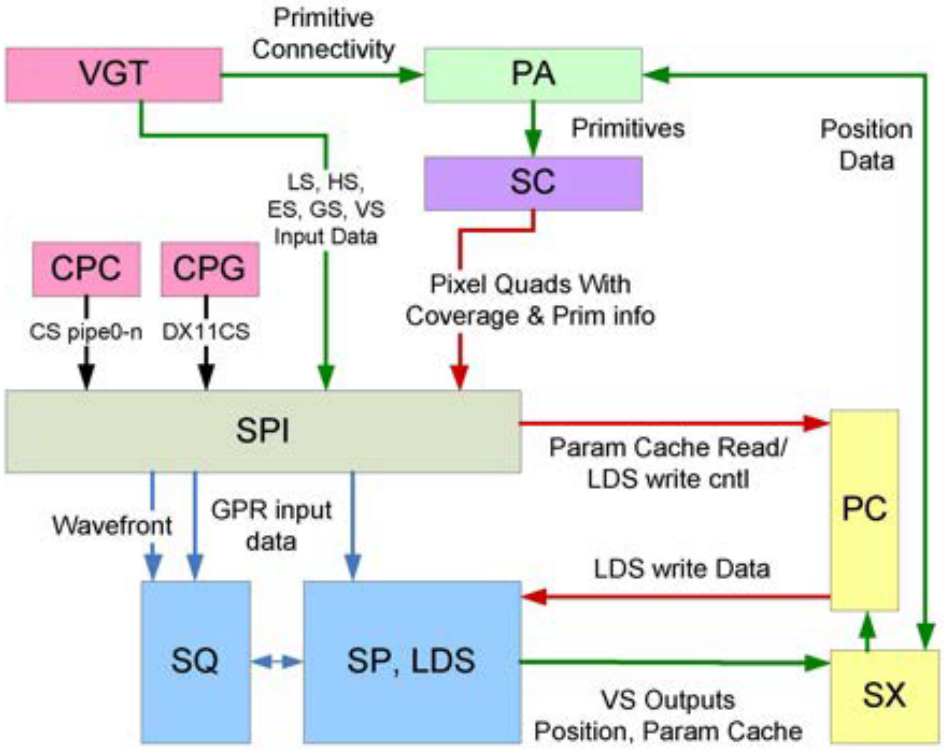


Figure 1 – SPI Chip Level Data Flow Diagram

Referencing Figure 1, for doing just vertex and pixel shading, vertex and primitive type processing are associated with the green colored lines. The VGT initially starts off sending vertex indices in the form of vsverts to the SPI and at the same time sending the primitive connectivity to the PA identifying how those vertices will get built back into primitives. The SPI buffers up the vsverts into a wavefront and once it has received a full wavefront of data, the wave transfer from the VGT will trigger the SPI to release the data to the SQ and feed associated data into the GPRs. When the vertex shader starts processing position data, typically it will send out position early to the position buffers in the SX which then allows the PA to read that position data and start building the primitives and producing those primitives which go through the Scan Converter (SC) to produce pixels. Once the SC has primitives, it will start producing pixels which are fed to the SPI. Once the SPI has a full wavefront of pixels, it will try and send associated data into the GPRs with the wavefront to the SQ. Reads are made to copy parameter data out of the parameter cache and write it into an SPI determined range of LDS in a particular CU.

1.3.2 Chip Level Diagram

Figure 2 shows the SPI block and its associated relationship to chip level inter-connections. Here, the physical partitioning of baryc logic is shown by the BCI blocks. For the purpose of this document, the BCI logic will be considered as part of the logical SPI design.

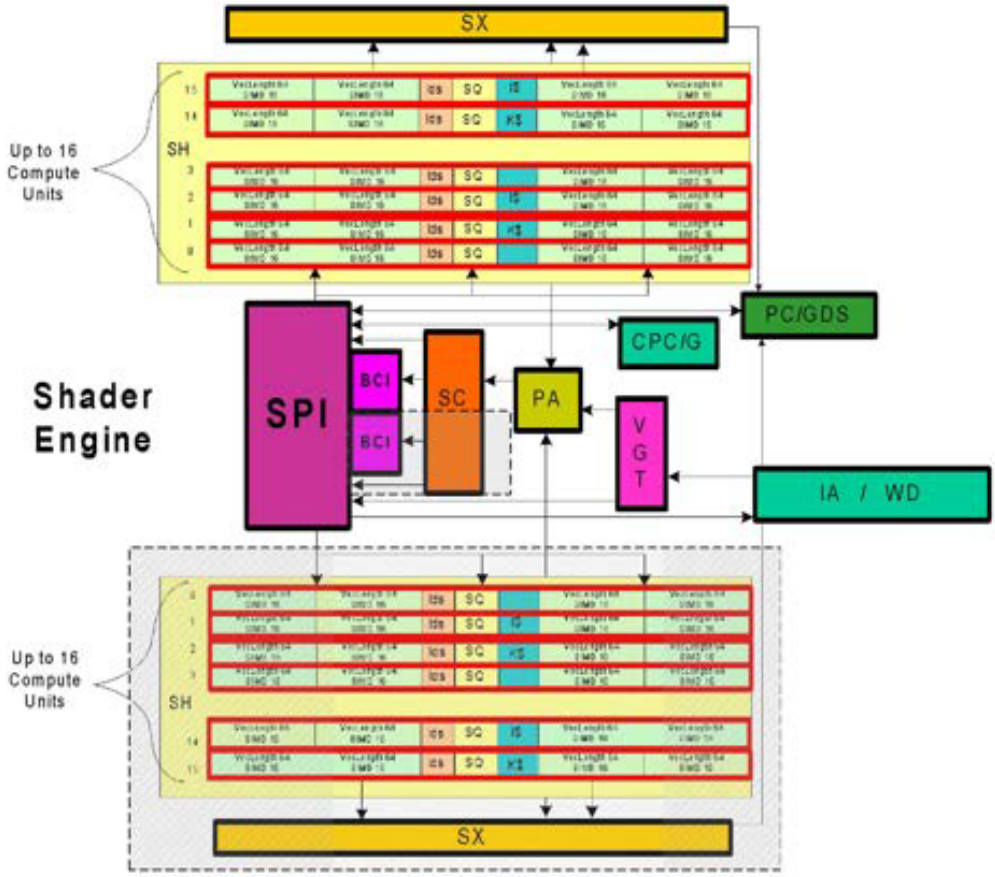


Figure 2 – Chip Level Diagram

1.3.3 SPI Block Diagram

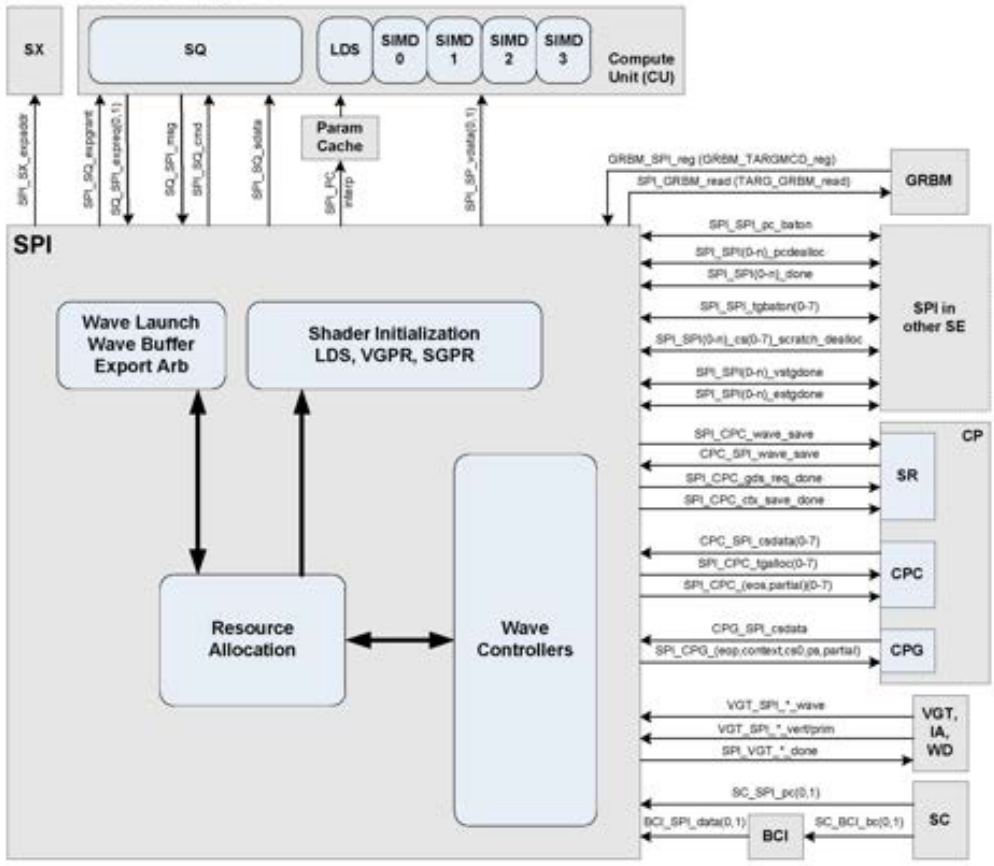


Figure 3 – Top Level Connectivity Block Diagram

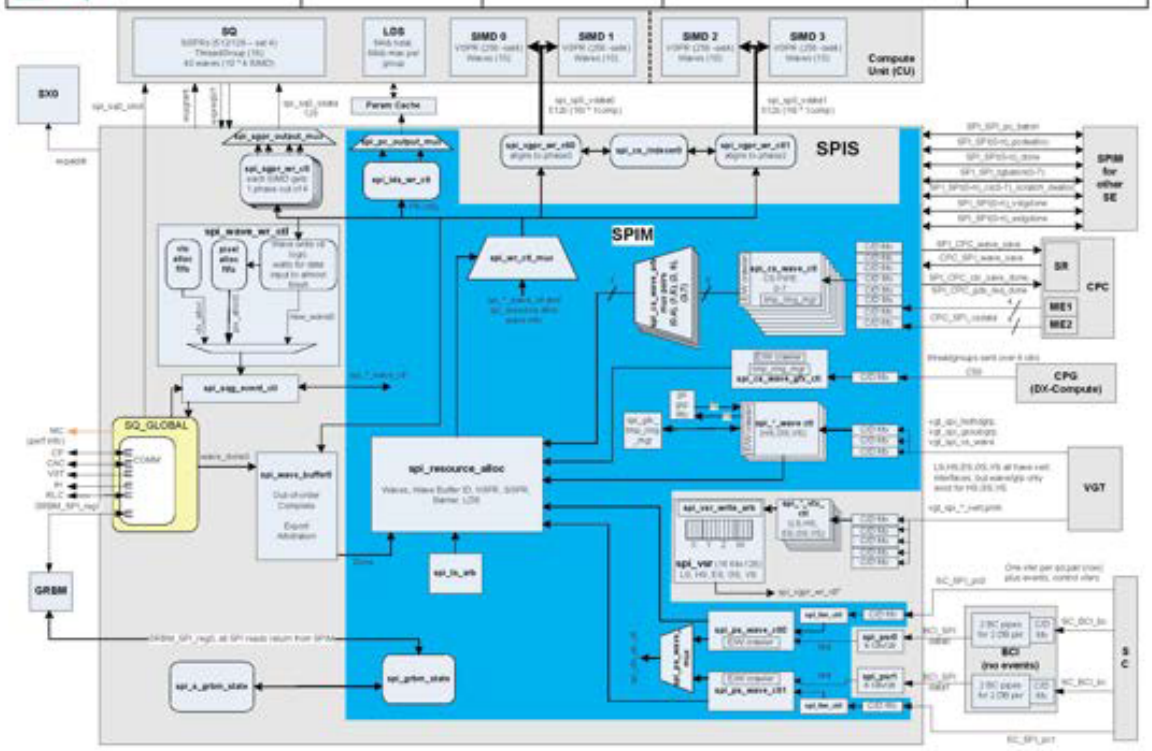


Figure 4 – Block Diagram

Diagram copied from //gfxip/gfx9/doc/design/blocks/spi/gfx9_SPI_Block_Diagram.vsd

2 Features / Functionality

2.1 Stage And Organize Data for Shader Launch

The SPI logical block stages and organizes efficient loading of shader input data to the Vector/Scalar General Purpose Registers (VGPR/SGPR) and Local Data Store (LDS) in the Shader Array and manages resources required to run those shader programs. The VGT will have several types of inputs to the SPI: normal vertices that will create positions and parameters for rasterization and pixel processing (VS, which could be normal vertices or the final pass of a Geometry Shader), Geometry Shader (GS) primitives, vertices that only export to memory (ES, which is the first vertex pass of a Geometry Shader), vertices acting as the first stage of tessellation processing (LS), and patch data associated with the Hull Shader (HS). The VS, GS, ES, HS, and LS are often generalized into the category of "verts" when discussing data moving through the SPI. The Scan Converter (SC) delivers pixel quads to the SPI for pixel shading. The CPG block delivers DX11 Compute threadgroups to the SPI for launching compute shaders. The CPC delivers Async Compute threadgroups to the SPI for launching compute shaders.

2.2 Compute Shader (CS)

As shown in Figure 1, Compute Shader input data can come from either the CPG (GFX-CS) or the CPC (Async CS). CS waves go through the same resource arbitration and allocation as all other supported SPI wavefront types.

DX11 requires support for compute shaders, and the SPI plays a role in getting compute shaders into the shader array. Both the CPG and CPC deliver threadgroups to the SPI along with persistent state data that tells the SPI how to process those threadgroups. The SPI is responsible for unrolling each threadgroup into the number of wavefronts required to process all of the threads for the threadgroup.

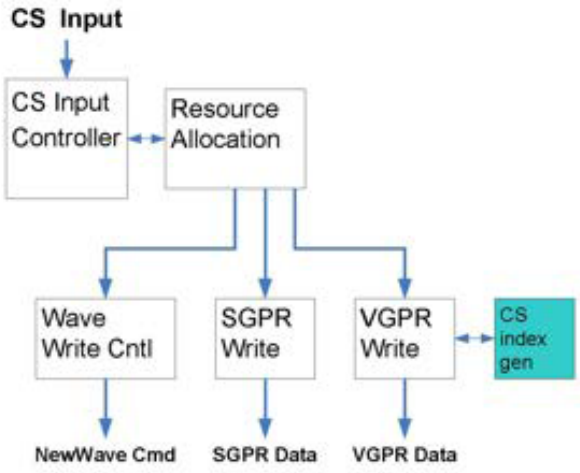


Figure 5 – CS Data Flow

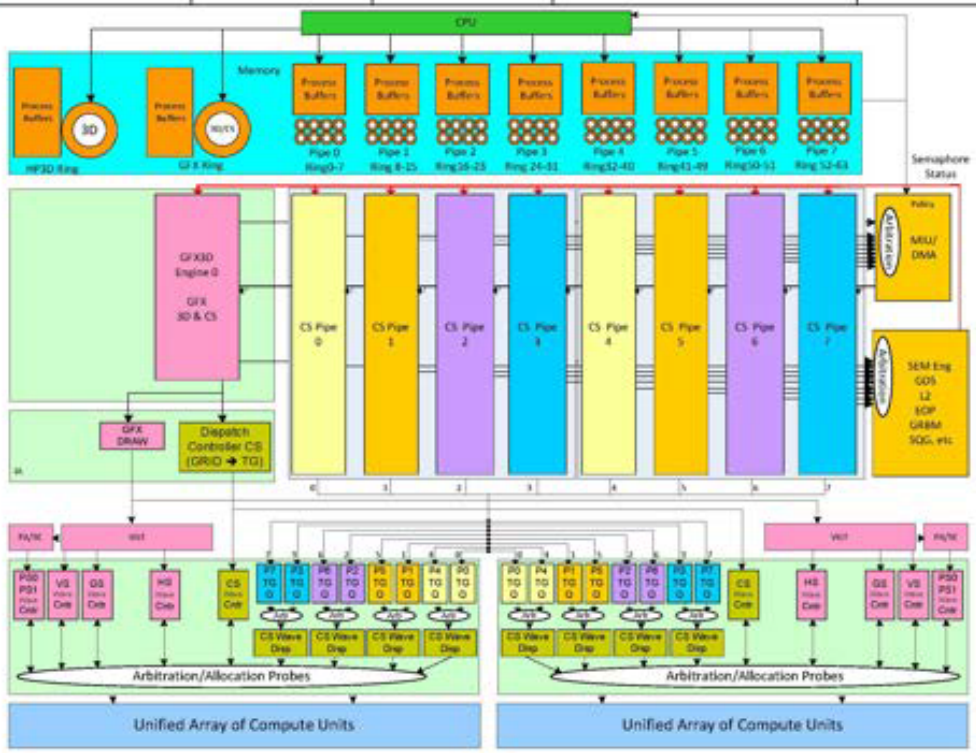


Figure 6 – Async Compute Block Diagram

2.2.1 Resource Probing

If there are more than 4 Async Compute Pipes present in a configuration (more than 1 compute ME) then pairs of compute wave controllers will share a single probe to Resource Alloc (RA) for allocating resources. Each of the pair takes alternating turns using the probe to request resources. This probing opportunity will alternate between the two pipelines once every four clocks until a probing pipeline has a work group that fits and is selected by RA. Once a pipeline is selected, it will allocate resources for all waves in its threadgroup before releasing the probe. If only one pipe of a pair has a threadgroup ready to allocate, it will have exclusive use of the probe for requesting resources and can continue requesting on every four clock cycle.

Each CS controller should check its tg_per_cu limit, wave_per_sh limit, scratch limit, and crawler space before requesting resources so it doesn't take cycles away from the other cs_ctl sharing a common probe.

2.2.2 Threadgroup Ordering

Ordering of threadgroups for a given async compute pipe needs to be maintained across all SE. The Dispatch Controller (DC) assigns threadgroups round-robin to all SE in the chip, and the SPIs from each of those SE must cooperate to ensure that a threadgroup from a given SE is not allowed to probe until the threadgroup before it has won allocation. The SPI needs to wait until the first wave of the previous group allocates, but does not need to wait for all waves of the previous threadgroup.

The Dispatch Controller will send two signals with each threadgroup, first_group and last_group to tell the SPI when each dispatch starts and finishes. If a group is marked as first_group, the CS controller can start requesting immediately without waiting on any previous group. If a task is pre-empted and restarted, the first threadgroup of

the restart should be marked as first_group even if it is not the first of the dispatch. Once that first_group allocates, the allocating controller sends a tg_alloc pulse to the next SPI in the dispatch sequence so that it can start requesting for its group. For allocating groups marked as last_group, no tg_alloc pulse is sent. This scheme avoids any problems that can arise from an implicit ordering scheme where the DC and the SPI both independently manage threadgroup ordering. First_groups can be sent to any SPI, regardless of where the previous group was sent, and last_groups won't create any left-over status in the SPI. Power gating and soft_reset issues are also avoided since no duplicate status needs to be kept in sync between DC and SPI, which are physically in separate tiles.

SPI also supports a mode where a DISPATCH_INITIATOR write clears the baton for that async compute pipe such that the last_tg from the dispatch controller is not necessary. This is the default behavior for SPI, but it can be disabled by setting SPI_CONFIG_CNTL_1.BATON_RESET_DISABLE to 1.

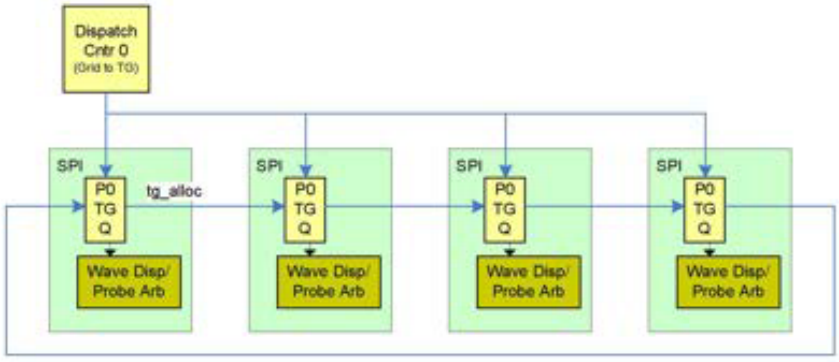


Figure 7 – CS Threadgroup Ordering

The compute controllers also support disabling of entire SH for a given pipe using the COMPUTE_STATIC_THREAD_MGMT register. This feature is also known as "steering", and allows a dispatch to be sent only to a subset of the possible SH in a given config. The DC will shadow CU_EN settings and only send threadgroups to SPI with at least one CU enabled for the dispatch. When passing/receiving tg_alloc, each SPI needs to check its own CU_EN settings. If the receiving SPI has a CU_EN of 0 then it should pass the token along to the next SPI. This passing of the token through disabled SPI adds extra time between threadgroups starting. The ADC will optimize for the case where only a single SH is enabled for a dispatch by marking every threadgroup sent to that single SH as both first and last of group. This way no ordering tokens are passed by the SPI and the single enabled SH is allowed to launch threadgroups as fast as possible.

2.2.3 Threadgroup Halting and Discarding

The CS controller will also respond to halt signaling to accomplish precise launch pre-emption. Upon being commanded to halt by the CPC, the controller will finish out any wavefronts from partially started work groups and then stall any subsequent traffic from that pipe.

Name:	Bits:	Description:
CLIENT_TARGET_halt_req	1	If asserted the receiving block must halt the production of compute work at a well-defined pipeline location. After halting, the receiver must return a halt_ack.

If a discard is then requested, any other entries in the input fifo will be popped and discarded before signaling back to the grid dispatcher the SPI has prepared to switch. A discard_req will always happen within a halt_req/halt_ack window. The SPI must be halted before it can be told to discard.

Name:	Bits:	Description:
CLIENT_TARGET_discard_req	1	If asserted the receiving block discard any pending compute work that has not yet been allocated shader resources. A client should only assert this when both CLIENT_TARGET_halt_req and TARGET_CLIENT_halt_ack are assserted.

The CS controller will drive a tg_allocated signal to the CP notifying the DC when a threadgroup allocates. This is needed so the DC can track the exact number of groups that launch versus those that are discarded after a halt.

2.2.4 Queue Status

Each CS controller maintains a count of active waves for all 8 queues that can drive that pipe. SPI provides that status through GRBM reads using several register decodes. One register, SPI_CSQ_WF_ACTIVE_STATUS, contains a single ACTIVE bit for each queue of each pipe of a given ME. SPI_CSQ_WF_ACTIVE_STATUS is indexed by GRBM.ME_ID. SPI_CSQ_WF_ACTIVE_COUNT_{0-7}.COUNT provides the actual number of wavefronts that are in flight for a specific queue. SPI_CSQ_WF_ACTIVE_COUNT_{0-7}.EVENTS provides the actual number of events that are in flight for a specific queue. SPI_CSQ_WF_ACTIVE_COUNT is indexed by GRBM.ME_ID and GRBM.PIPE_ID.

2.2.5 Unordered Dispatches

DC and SPI also support an Unordered Dispatch mode using the ORDER_MODE field of the DISPATCH_INITIATOR. When launching an Unordered dispatch, the Dispatch Controller will send every threadgroup marked with both first/last_group. This allows the SPI in each SH to launch threadgroups independently without passing or expecting the order baton.
Unordered mode also changes the way SPI responds to halt requests. In the ordered mode, SPI can halt on any threadgroup boundary and return halt_ack with threadgroups still pending in its input fifo. In the unordered mode, SPI will allocate all threadgroups that have been sent from the DC before returning halt_ack.

2.2.6 State Forwarding to SQG

All state traffic to each compute pipe needs to be passed to the SQG for logging. State writes are sent from the outputs of credit/debit fifos with arbitration and backpressure to ensure that only 1 controller sends per clock.

2.2.7 First Wave of Dispatch

SPI supports SQ/SQC volatile cache deallocation control by marking the first threadgroup of a dispatch that is sent to each CU and SQC (group of CU). The scoreboard logic used to track when threadgroups are sent to CU/SQC needs to be reset at the start of each dispatch, so each CS wave controller needs to provide this information. The CS wave controller will signal "first wave of dispatch" to RA for the first_wave request of the first threadgroup after each DISPATCH_INITIATOR.
SPI is aware of SQ to SQC mapping, both for this invalidate volatile feature as well as CU busy signaling for clk-gate control. The SPI is ifdef'ed to handle both different numbers of CU (GPU__GC__NUM_CU_PER_SH) and different numbers of CU-per-SQC (GPU__GC__MAX_3_CU_PER_SQC).

2.2.8 Compute Shader Index Terms

For CS, the SPI can load up to 3 index terms as input into the VGPR. This is a 1 to 3-Dimensional incrementing index that represents the relative ID of the thread within its threadgroup known as ThreadIDInGroup. COMPUTE_PGM_RSRC2.TIDIG_COMP_CNT is used to control the number of components written by the SPI. Here is a simple example of how the SPI would generate the ThreadIDInGroup across the wavefronts with incrementing 3d indices.

For a threadgroup with dimension X=3, Y=16, Z=2, the SPI would create 2 wavefronts to process the 96 valid threads (3*16*2). Sequentially, the thread input values would look like this where the X increments first and wraps back to zero. At each wrap point, the Y term would increment all the way up to the Z term incrementing.

Thread0 (X,Y,Z) = 0,0,0
Thread1 = 1,0,0
Thread2 = 2,0,0
Thread3 = 0,1,0
Thread95 = 2,15,1

16 threads wide and 4 clocks deep counts demonstrated in Figure 8.

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

Figure 8 – CS Thread Count Increment Example

2.3 VGT-SPI "Vert" Shaders

The SPI can receive one thread per clock from the VGT for each of LS, HS, ES, GS, and VS. The LS, ES, and VS interfaces are all 128 bits wide, GS is 87 bits wide, and HS is 45 bits. The SPI takes a serial stream of up to 64 threads from the VGT (one wavefront) and accumulates it into four parallel lines in the Vertex Staging Register (VSR), matching the VGPR write format and allowing the SPI to minimize the VGPR input cycles for vertex data. The interface between the SPI and the VGPRs is 16 verts * 1 component wide and the SPI is always trying to write 16 threads per cycle into the GPRs. The SPI arbitrates on 4 clock cycles so every time a particular type gets to write into the VGPRs it really wants to write 64 threads, 16 at a time, over 4 cycles. If the SPI tried to write immediately to the VGPRs every time the VGT came in with 1 serial thread, the other 63 threads of the 4 clock cycle would be wasted.

Figure 9 shows the serial stream from the VGT being packed into this 16 wide over 4 clock wavefront.

Figure 9 – "Vertex" Data Flow VGT-SPI

2.3.1 ES, GS, VS Processing

In GFX9 the change was made to combine ES and GS processing into a single shader stage so there is no need to synchronize ES-done to GS-start. There is also no need for SPI to pass parent CU information from ES to GS groups like was necessary for onchip-GS processing in previous families. The synchronization of GS to VS processing is handled outside of the SPI (VGT waits on gscount_done from GS shader before generating VS). If GS is passing data to VS using onchip LDS (onchip-GS) then SPI must pass subgroup information from the producing GS to the consuming VS subgroup.

Each vertex controller runs independently, with the only interaction being the arbitration for writes to a particular VSR, until wavefronts request for resource allocation. There is only one copy of VSR memory composed of multiple banks which hold the different components. There is a simple priority arbitration here to make sure there are no data collisions when multiple controllers need to write to the same memory banks. The priority order is a fixed lowest-to-highest of LS, HS, ES, GS, VS. Space for multiple wavefronts exists for each type in the SPI which allows the SPI to start copying one wavefront while the VGT starts sending the next wavefront.

Once a full wavefront of vertex indices are written into the VSR and the associated wave transfer from the VGT has occurred to let the SPI know it is ok to issue that wavefront, the Vertex Wave Controllers will try to allocate the resources that the shader needs to execute in the shader complex. In the case of LS/HS and ES/GS groups, SPI waits until all transfers of all waves of the group (LS-vert/HS-vert or ES-vert/GS-prim) have been received before trying to launch the group. This means the VSR must be able to hold an entire group's worth of data, up to a max of 4 wavefronts, for each of these group types. If the wave/group wins resources allocation, the wave control information (resource bases/sizes, state_id, pipe_id, etc) is sent to the shader input write controllers to load the wavefront to the Shader Array.

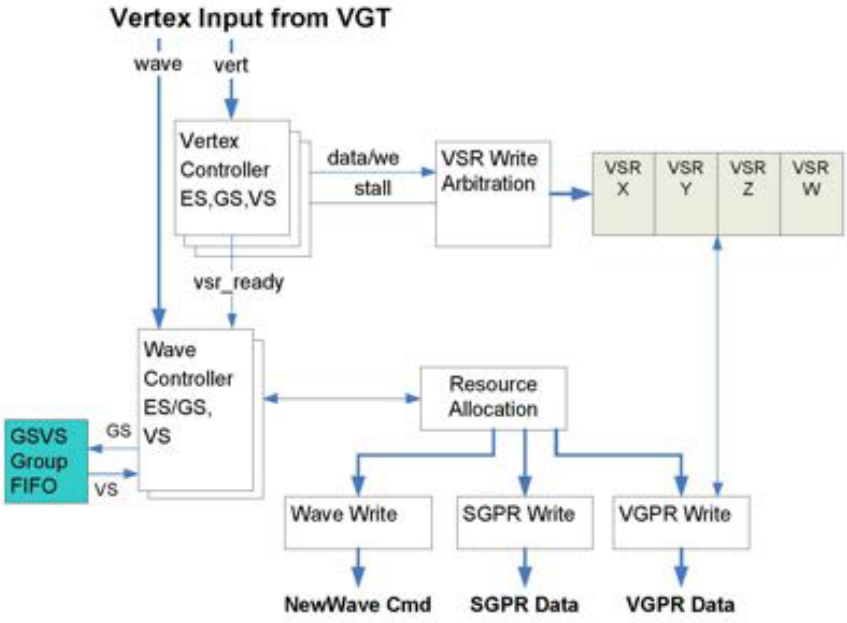


Figure 10 – VGT ES, GS, VS Vertex Input

2.3.2 On-chip GS

Onchip GS mode allows the use of onchip LDS space to store the ESGS and GSVS ring buffers, eliminating the traffic to offchip memory that is necessary for offchip modes. Typically scenarios with small amplification will benefit the most from this approach. Prior offchip ES/GS modes used two offchip ring buffers that wrapped individually, and were of a fixed size and base for the entire application. In the new combined ES/GS shader the passing of data from ES to GS is always done using onchip LDS space. Onchip versus offchip GS now refers to how data is being passed from the GS stage to the VS stage.

The VGT partitions the ES/GS work into smaller chunks called subgroups. Each ES/GS subgroup gets allocated to a Compute Unit (CU) by the SPI and for onchip GS that subgroup stays on the same CU for the duration of its lifetime (which is ES/GS/VS). Onchip GS mode requires the SPI to maintain a task grouping for ES/GS/VS processing. ES/GS waves in a subgroup (waves between first/last of subgroup) must all go to the same CU and need to share a common LDS base and size. The subsequent VS subgroup must also go to the same CU and launch with the same LDS base and size and this information is communicated through the GS-to-VS group fifo.

SPI offers a means of controlling "subgroups between GS output and VS consumption". SPI_SHADER_PGM_RSRC4_GS.GROUP_FIFO_DEPTH will set a limit on the number of groups between GS launch and VS launch. This limit will also support scaling through the SPI_WCL_PIPE_PERCENT_GFX.GS_GRP_VALUE register field.

The SPI also has to know when all waves of a VS subgroup have completed so that the onchip-GS LDS can be freed.

2.3.3 Tessellation

DX11 Tessellation requires two vertex types through the SPI: LS and HS. LS is the API vertex shader which writes to the LDS, and HS is the hardware stage that creates tessellation factors for the tessellation engine and output data for the Domain Shader. The hardware shader stage flow is LS/HS->DS(ES/GS,VS), depending on whether geometry shading is enabled (ES/GS) or not (VS). The passing of computed results from LS to HS is always done using the onchip LDS of the executing CU. LDS is a per-CU resource and only waves sent to that CU can access it, so all waves of a tessellation threadgroup must be sent to the same CU. The onchip LDS space associated with an LS/HS threadgroup is freed when the final HS of the group completes

HS-to-DS data is always passed using offchip buffering and SPI is responsible for managing this offchip LDS space. SPI allocates a buffer with each LS/HS threadgroup and then frees the buffer once all DS that source the data are complete.

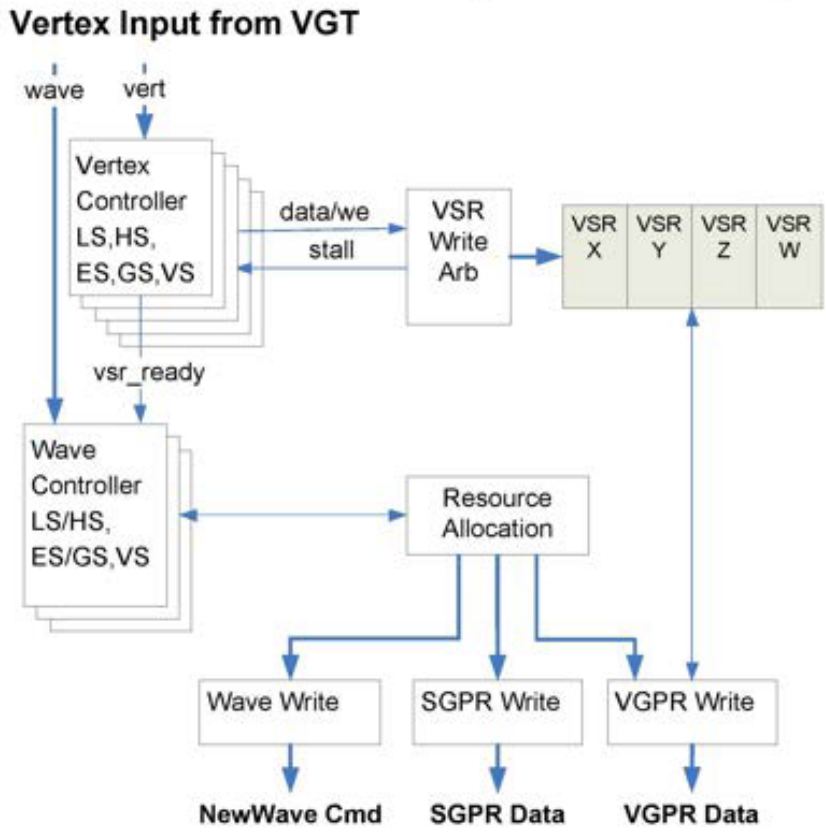
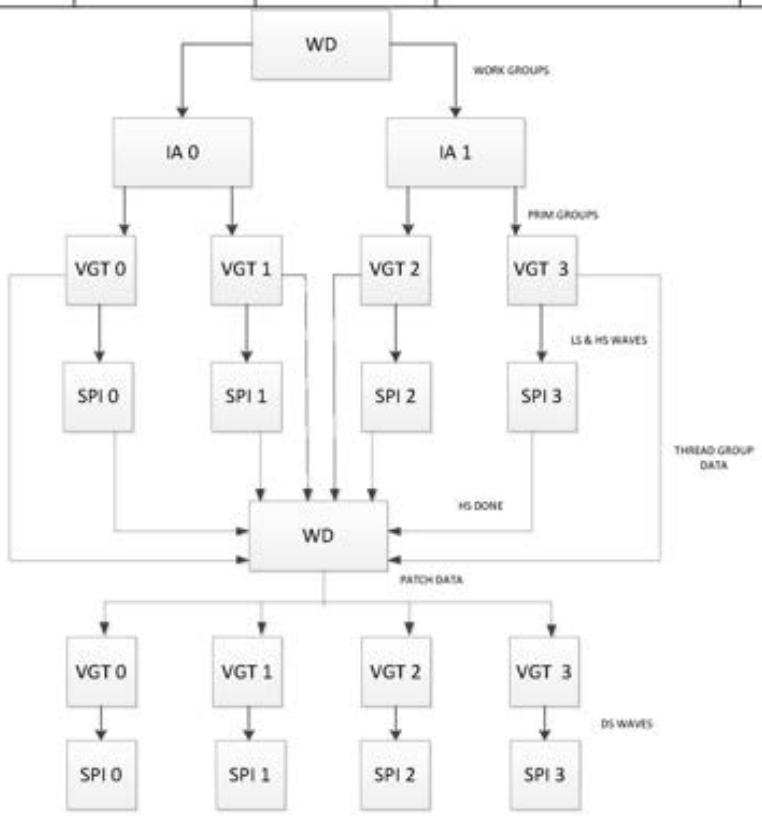


Figure 11 – LS,HS,ES,GS,VS Vertex Input

2.3.4 Distributed Tessellation

The intention of distributed tessellation is to rebalance the DS work after the HS stage in order to generate new primgroups with post tessellation primitives. This enables the workload to be distributed more uniformly amongst the available VGT and SH units for an overall higher performance.



The diagram above shows the data flow.
Note that commonly named blocks such as VGT 0, WD etc are the same physical block (not duplicated) just re-drawn for clarity.
Note that all variants have this architecture, the legacy patch distribution method of all DS sent to parent SH is still supported but the infrastructure is changed to the above.

2.3.4.1 Work Creation Description

The unit of work that the VGT creates is a threadgroup. There could be multiple threadgroups present in the primgroup each VGT receives. LS/HS verts are created for the entire threadgroup and then the HS threadgroup transfer is sent to the SPI. A FLUSH_HS_OUTPUT event is inserted after all each threadgroup transfer. The SPI allocates onchip LDS space and an offchip LDS buffer for each LS/HS threadgroup. The SPI sends the VGT an HS_done signal per threadgroup when the entire threadgroup completes.

The WD needs to process threadgroups in the order they were issued originally by the WD/IA/VGT when producing DS threadgroups. There is a HS threadgroup done counter per VGT which is incremented when the respective HS_done is received from the SPI. At the beginning of the packet, the VGT fifo with a threadgroup tagged first_primgroup will be processed first. This will ensure the launch order of the LS/HS is maintained. After this threadgroup is popped off, its next_fe_id field will be used to determine which fifo to read from next. All the

tessellation factors for the patches in an entire threadgroup are fetched from memory and the patches are then sent to the distribution logic.

Each VGT will receive patches that will potentially be tagged with start/stop points. Each VGT will only tessellate the portion of the patch that is active and will issue the respective DS waves to the SPI. The WD will create and insert the OFFCHIP_HS_DEALLOC event at the end of each DS threadgroup and broadcast it to all VGT, attaching the appropriate VGT_ID to the event so the SPI knows which frontend allocated the offchip space for the threadgroup. The VGT_ID of the original threadgroup will be sent on the new parent_se[1:0] field of the VS_wave or GS_subgrp interfaces. The SPI will launch the DS waves on any available CU and the HS output data will be fetched from offchip memory. Once all DS of the threadgroup complete the SPI will handle the OFFCHIP_HS_DEALLOC event by incrementing either gs_offchip_done_count or vs_offchip_done_count in the appropriate SPI as identified by parent_se. If an SPI sees an event with a parent_se that does not match its own SE_ID, a done signal will be forwarded to the appropriate SPI. Offchip LDS will always be deallocated by the OFFCHIP_HS_DEALLOC event, occurring when the event pops off the SPI's event/wave crawler.

2.3.4.2 Offchip LDS ID Changes

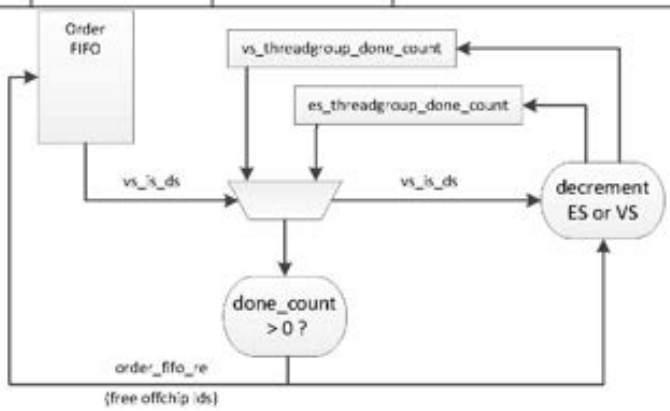
With distributed tessellation the DS from a given LS/HS can be sent to other SE and those other SE do not have access to the parent SPI's group information. Because of that, the distributed DS need another way to get the offchip_lds_id allocated to their parent LS/HS. The WD will add a per-SE offchip_lds_id counter that increments for each tessellation threadgroup that is issued to that SE. This offchip_lds_id will be stored through the WD and VGT for each outstanding threadgroup and patch and then sent to the SPI with every DS wave on the GS or VS wave interface. The event_id field on these interfaces is increased to 7 bits and is used to send offchip_lds_id for wave or subgroup transfers.
SPI still allocates offchip space with LS/HS and makes sure that the next buffer is available, but the ID is assigned by and delivered from the WD. VGT_HS_OFFCHIP_PARAM.OFFCHIP_BUFFERING specifies the current number of offchip buffers and offchip_lds_id should reset to 0 for each SE whenever that register value changes (SPI does not reset if the register is rewritten with the same value). OFFCHIP_BUFFERING is divided between the number of SE in the config, regardless of front-end harvesting, and each SE's ID should count from 0 to ((BUFFERING/NUM_SE) – 1).

The OFFCHIP_BUFFERING field will range 0-511 representing 1-512 buffers. A setting of 0 is not useful for this register and being able to represent 512 allocated buffers allows support for 128 buffers * 4 SE in larger configurations.

The offchip_lds_id sent to SPI will be a total of nine bits, two bits of parent_se and the seven bit event_id carrying offchip_lds_id count from the parent LS/HS.

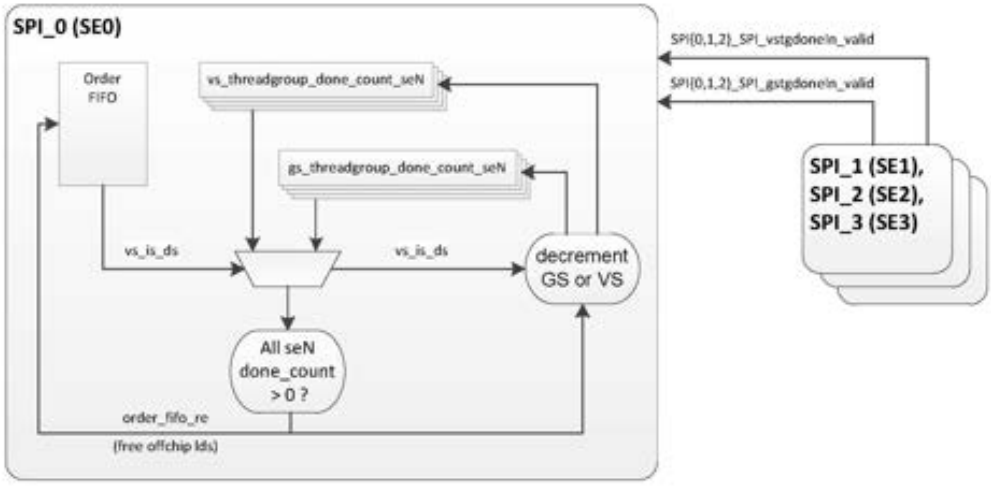
2.3.4.3 Offchip LDS Deallocation Changes

In previous projects the work associated with a given SE's offchip LDS allocation was only ever sent to that same SE. This meant that all LS/HS/DS for a threadgroup went to the same SE and a given SE's offchip space could be deallocated based solely on work completing in that SE. ES and VS executing as DS could also potentially complete out of order with respect to each other, and offchip LDS manager dealt with that by keeping a FIFO history of allocation order (ES/VS is DS) along with group done counts for each of GS and DS.



SPI incremented the ES/VS group done count when an ES/VS lastwave that launched with offchip LDS enabled popped from a wave controller's crawler, maintaining order within a shader stage. Once the done count corresponding to the order FIFO output was greater than 0, SPI would deallocate the offchip space, pop the order FIFO, and decrement the appropriate done_count.

Distributed Tessellation means that tessellation work originating on a given SE (and therefore associated with that SE's managed offchip space) can also be sent to other SE. This means that a given SE has to ensure that all other SE are done using an offchip allocation before deallocating and reusing that space. In order to handle this, the SPI will use a scheme similar to the one used for parameter cache deallocation across multiple Shader Engines where the SPI signal to each other when they see a dealloc they do not own. For instance, if SPI2 pops an OFFCHIP_HS_DEALLOC event with a VGT_ID of 0 from a VS or GS crawler then it will send a signal to SPI0 rather than doing anything with its own offchip LDS mgmt.



Each SPI has to keep a done count for every SE and check that all done counts are greater than zero before freeing an offchip LDS buffer. When offchip LDS is freed, all SE done counts for the freeing shader stage (GS or VS)

decrement together. If a frontend is disabled (as specified by CC_GC/GC_USER_PRIM_CONFIG regs) then the SPI attached to that frontend will not receive an OFFCHIP_HS_DEALLOC event. The SPI's offchip LDS deallocation logic should not wait for pulses from other SPI that are connected to disabled frontends, and also should not decrement the done count associated with those SPI. All counts associated with a disabled frontend SPI should be held at 0 as long as the frontend is disabled so that the counts will be in a known good state if the frontend is enabled at a later time.

The move to only supporting offchip tessellation (HS-DS only through offchip LDS) means that SPI no longer needs an HS-DS group fifo to pass information to the DS stage. However, SPI does still offer a means of controlling "groups between HS output and DS consumption" similar to what the programmable depth group fifo offered previously. SPI_SHADER_PGM_RSRC3_HS.GROUP_FIFO_DEPTH will set a limit on the number of groups between LS/HS offchip allocation and DS consumption. This limit will also support scaling through the SPI_WCL_PIPE_PERCENT_GFX.HS_GRP_VALUE register field.

2.4 Pixel Shader (PS)

Figure 12 shows the flow of pixel data through the SPI. The SPI gets input data for pixels from the Scan Converter (SC). Wave control information tells the SPI which quads are hit, parameter cache sync and de-allocation tokens, and if the transfer is an event. The SC also delivers quad and per primitive barycentric information to the Baryc interpolation math. As quads are being received from the SC, the Pixel Input Controller will buffer the control information until a full wavefront is received. The quad information is also flowing down the barycentric math pipe where the SPI calculates the per pixel IJ and W terms and can also store off screen X/Y, primitive facedness, and other ancillary terms. That data gets delivered to the Pixel Staging Registers for storage until the wavefront is ready to launch. Once a full wavefront is accumulated, the SPI requests to allocate resources and, once granted, the wavefront is sent to the various Write Controllers which coordinate the loading of PS data to the SH.

PSR data is read and sent into VGPRs, attribute data is copied from the PC into the LDS of the appropriate CU, and other various terms are written into SGPRs. Once the full wavefront of pixel data is sent to VGPRs, SGPRs, and the LDS, the Wave Write Controller sends the wave to the SQ to initiate pixel shader execution.

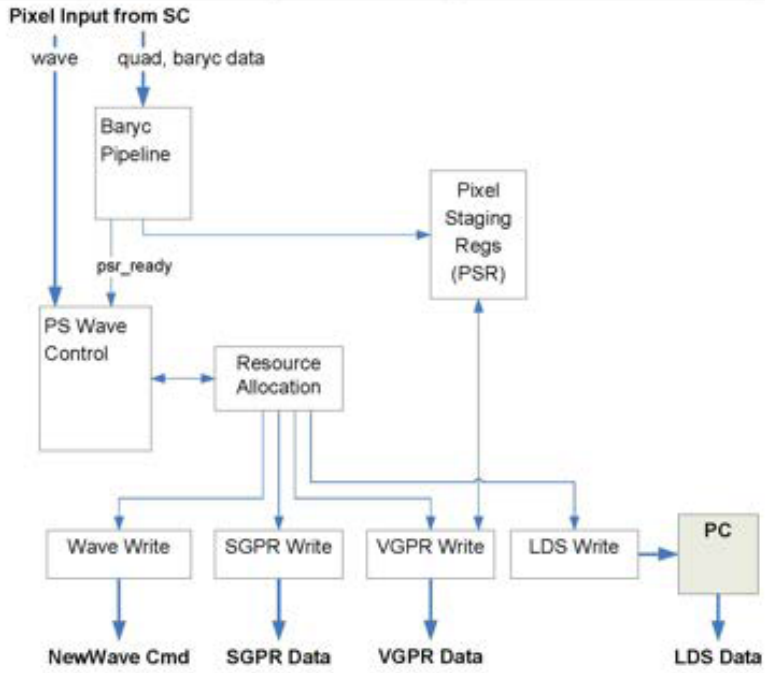


Figure 12 – Pixel Input Data

2.4.1 Pixel Data Flow

SPI accepts pixel quad rows from the SC at the peak rate of 2 quads (8 pixels) per clock per packer containing:

Primitive control data:

- parameter cache base pointers - where attributes of the vertices that created the primitives that created those pixels are located in the parameter cache
- first_prim_of_slot aka new_vector - this is how the SPI makes sure the attribute data for the primitives that created these pixels is actually in the parameter cache before a read is attempted
- dealloc tokens - lets the SPI know that this is the last prim from the last vertex of a VS wavefront so it is ok to free up all of the associated attribute data
- end_of_vector flag - Informs the SPI this is the last row of a pixel wavefront, and can happen early prior to getting all 16 quads
- first_quad_of_prim – Attached to the first quad created by each primitive
- prim_type - associated with each quad

Quad Data:

- screen (X,Y) - where the quad is located in screen space
- centermost sample id - The centroid of each pixel in each quad
- iterated sample number - which sample is running during sample frequency pixel shading
- per pixel coverage – mask identifying all hit pixel samples as determined by the SC

Primitive Data:

- perspective/linear barycentric gradients, depth (Z) information

2.4.1.1 Calculate Per-Pixel IJ Barycentric Coordinates

The SPI receives IJ gradient information from the SC on a per quad basis (all values are 32-bit IEEE).

InvW0 – value of 1/W at ref vtx (value of line_stipple_tex_coord at ref vtx with linear gradient)
InvWdx – 1/W rate of change in x (line_stipple_tex_coord rate of change with linear gradient)
InvWdy – 1/W rate of change in y (line_stipple_tex_coord rate of change with linear gradient)
IW0 – value of I/W at ref vtx
IWdx – I/W rate of change in x
IWdy – I/W rate of change in y
JW0 – value of J/W at ref vtx
JWdx – J/W rate of change in x
JWdy – J/W rate of change in y

The SPI uses the following equations to calculate per-pixel I,J,W. One barycentric triplet (I,J,W) is computed per cycle per quad, so all of the math below is instanced per quad in each SPI.

The SPI calculates the distance of the current quad's upper left pixel to the reference vertex and uses that distance to calculate the terms IWref, JWref, and InvWref which are the values at the upper left pixel of the quad and known as the "reference pixel".

Reference Pixel (1 per quad):
(delta_x and delta_y are 16.8 2's comp convert to flt pt distance of quad UL pixel from vertex 0)
IWref = IWo + (delta_x * Iwdx) + (delta_y * Iwdy)
JWref = JWo + (delta_x * Jwdx) + (delta_y * Jwdy)
InvWref = InvWo + (delta_x * InvWdx) + (delta_y * InvWdy)

I,J and per pixel W values are stored in 32-bit IEEE float.

Delta Pixels (3 per quad):
The SPI only does the full distance math to calculate the value for the upper left pixel. The other 3 pixels are calculated as delta distances from the reference vertex. This is a hardware savings in the subtraction math since the subtraction will be a small distance within the quad versus a potentially large distance from the reference vertex. The trade off is latency on the other three pixels since those calculations cannot complete until the reference pixel obtains its calculated result. The delta calculations are fixed point math, while all other calculations are in float.

delta_x, delta_y – distance from upper left reference pixel to this pixel (UR, LL, or LR).

delta_Pixels(3 per quad): (delta_x and delta_y are 2.8 2's comp convert to flt pt distance from Ref Pixel)
IWpix = IWref + (delta_x * Iwdx) + (delta_y * Iwdy)
JWpix = JWref + (delta_x * Jwdx) + (delta_y * Jwdy)
InvWpix = InvWref + (delta_x * InvWdx) + (delta_y * InvWdy)

Wpix = 1.0 / InvWpix for all 4 pixels
Ipix = Iwpix * Wpix for all 4 pixels
Jpix =Jwpix * Wpix for all 4 pixels

All of the barycentric gradient calculations are IEEE float. The only difference between the two equations here are the delta_x and delta_y being the distance from the UL pixel of the quad instead of the distance from the reference vertex, and the initial terms here being the values at the UL pixel instead of the reference vertex.

When processing linear gradients, two special cases are created by the fact that the InvW term is overloaded with line_stipple_tex_coord information (dependent on state control). The baryc logic has to force Wpix to 1.0 for the final Iw and Jw multiplies so that the IJ terms are not corrupted by the stipple term. The InvWpix math is used to

calculate per-pixel line_stipple_tex_coord values, and those terms have to be piped around the Wpix inverse function.

2.4.1.2 Pull Model

Fixed function attribute interpolation hardware has been removed from the SPI for several generations now, with all attribute interpolation happening in the shader including our standard choices of center, centroid, or current fragment. The DX11 pull-model feature is a method that allows the shader to interpolate an attribute at any location within the pixel, and we accomplish that through the SPI by loading 1/W, I/W, and J/W as input to the PS so the shader can calculate its own gradients and then interp I, J, and W to any desired sample location. When enabled, the SPI calculates 1/W, I/W, and J/W at pixel center and loads them into VGPR along with any other enabled terms.

The SPI still performs per-pixel IJ interpolation to support pre-DX11 style attribute interpolation that takes place at a fixed set of locations. If an app only wants to use center, centroid, or fragment when sampling attributes, those IJ values can be provided by the SPI. If a pixel shader is doing "true pull model" where it is sampling attributes multiple times at locations throughout the pixel, the SPI will provide (1,I,J)/W terms as input. The pixel shader then has to calculate gradients for the */W terms, interpolate them to the desired sample location, recip 1/W, mult times I/W and J/W to get IJ at that location, and then interpolate each attribute at that location.

It is also possible that an app might use both methods, with most attributes only sampling at pixel center/centroid/fragment, but maybe a couple that need to use "true pull model". In this mode, the shader could either do everything and calculate all the IJ itself, or enable multiple IJ terms from the SPI.

As an example, say a shader wants 4 perspective-correct attributes sampled at pixel centroid, 2 non-perspective correct (linear) attributes sampled at pixel center, and 1 attribute to do "true pull model" and sample all over the place. This scenario could set up the SPI like this:
PERSP_CENTER_ENA = 0
PERSP_CENTROID_ENA = 1
PERSP_SAMPLE_ENA = 0
PERSP_PULL_MODEL_ENA=1
LINEAR_CENTER_ENA = 1
LINEAR_CENTROID_ENA = 0
LINEAR_SAMPLE_ENA = 0

With this, perspective-correct IJ sampled at pixel centroid and non-perspective correct (linear) IJ sampled at pixel center are available directly to the PS, so no extra ALU instructions are needed to calculate the IJ before those attributes interpolate. ALU and TEX gradient instructions will have to happen to calc IJ before the "true pull model" attribute can interpolate.

The app could also just configure the SPI like this (and this is true for every single pixel shader):
PERSP_CENTER_ENA = 0
PERSP_CENTROID_ENA = 0
PERSP_SAMPLE_ENA = 0
PERSP_PULL_MODEL_ENA=1
LINEAR_CENTER_ENA = 0
LINEAR_CENTROID_ENA = 0
LINEAR_SAMPLE_ENA = 0

But then there needs to be extra ALU and TEX gradient instructions to calc each set of IJ needed for attribute interpolation, not just before those attributes that want to do true pull model.

2.4.2 Scale Resolution Based on Screen Location (9.125)

The BCI impact from "scale resolution based on screen space" is that the calculation locations for the pixels of a quad shift and scale when the feature is enabled. The Upper Left pixel of a quad shifts by half a pixel during ½ scaling and it shifts by 1.5 pixels during ¼ scaling. The delta X and delta Y to the other 3 pixels of the quad will be scaled up by 4 (1/4 resolution) or 2 (1/2 resolution). The quadx, quady, and I,J,W slopes that are passed to the BCI by the SC remain unchanged.

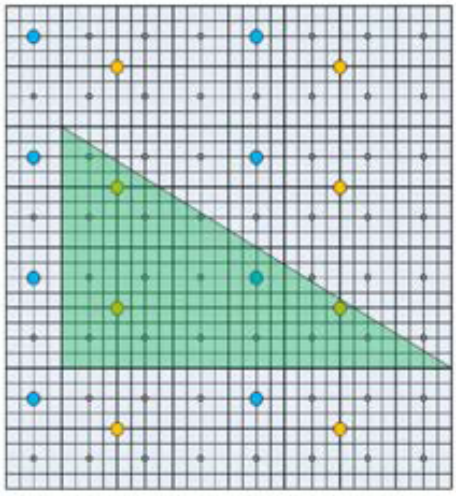
2.4.2.1 Visualizing the Scaling

In the chosen "True Scaling" approach the sample locations in the upper left pixel of a scaled quad do not map to the same locations in unscaled space. This means that all 4 of the BCI pixel evaluations are impacted when scaling.

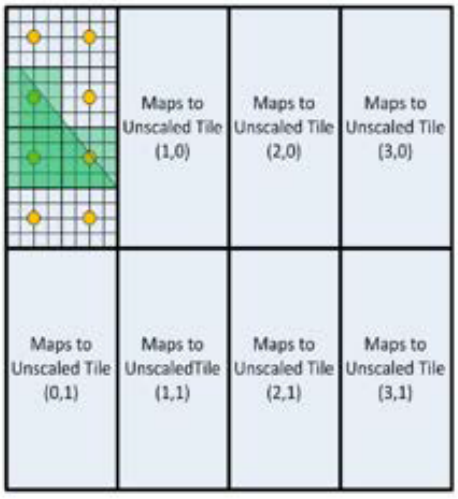
- For ½ scale factor the UL pixel center is offset by half a pixel
- For ¼ scale factor the UL pixel center is offset by 1.5 pixels
- In both scale factors, sample locations that have been specified as distances from pixel center need to scale

In the example below, scale factors of ¼ in X and ½ in Y are used. This means that the BCI would need to calculate the UL pixel-center value at a location (2.0, 1.0) pixels from the upper left corner of the quad (rather than its usual (0.5, 0.5)).

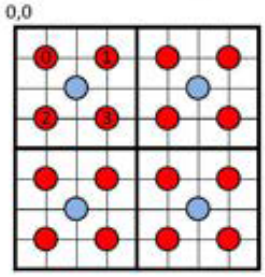
Unscaled Tile at Tile Coordinates (0,0)
Relative to the Upper Left of Supertile



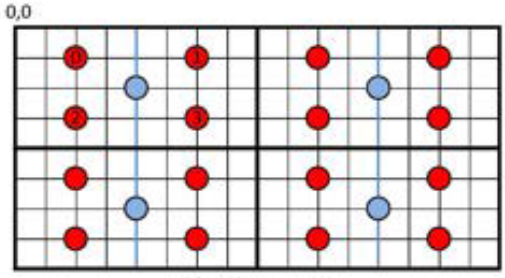
Scaled Tile (0,0). Each 2x4 pixel region maps to an entire tile in Unscaled Space.
"True Scaling" - geometry scaled relative to the upper left.



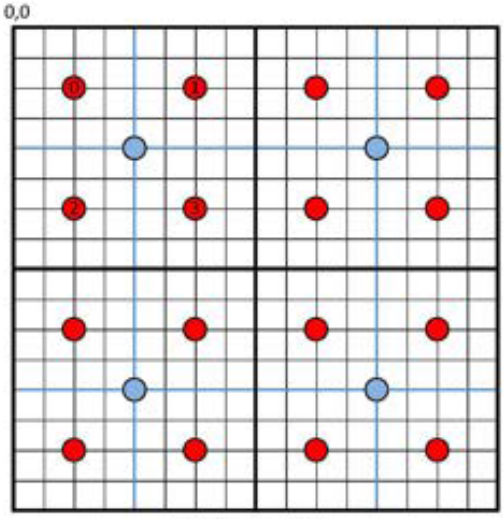
Another way to view the BCI scaling is by looking at a single unscaled input quad and seeing how the sample locations move. This example shows an incoming quad during 4xAA with a symmetric sample pattern and the implied sample location of pixel-center.



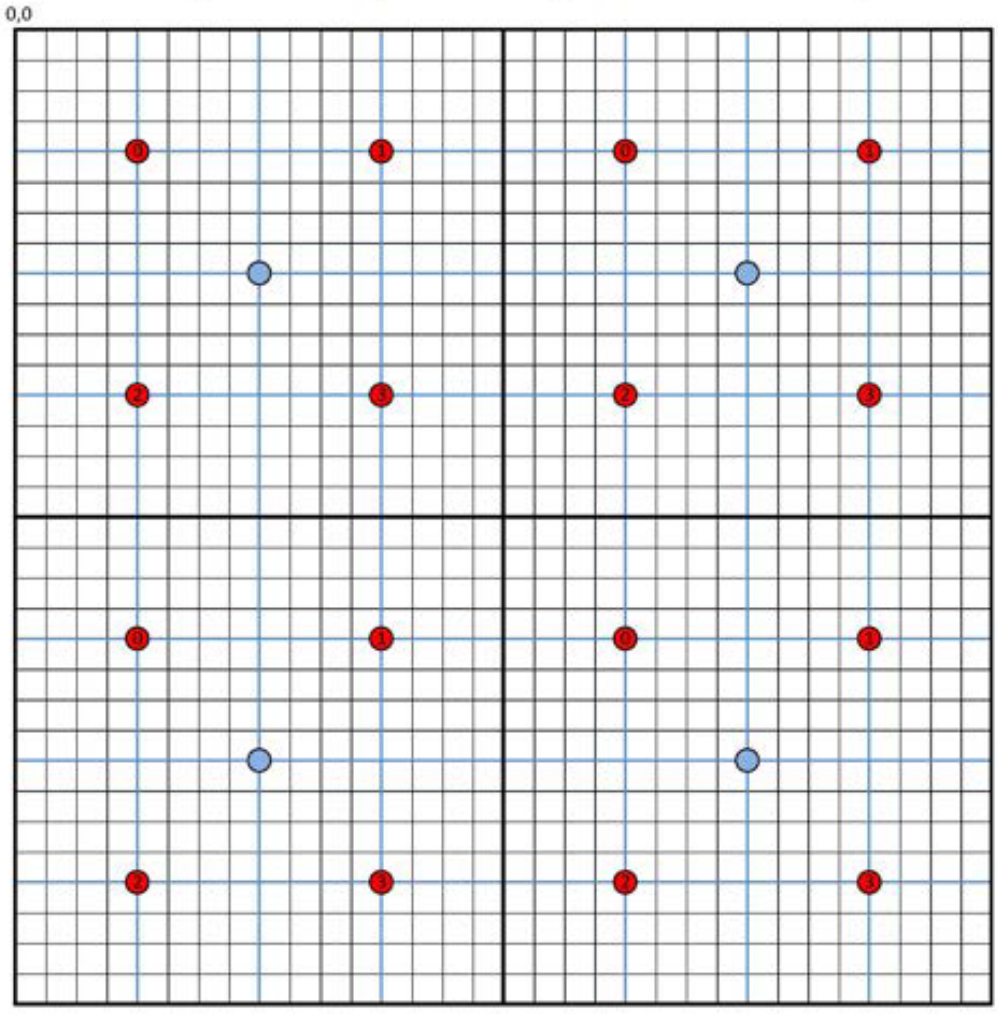
The upper left corner of the quad is considered to be (0,0) for this example and that reference point does not move as the pixel center and sample locations shift and scale. The next diagrams show how sample locations move for different scaling cases. The new "scaled pixels" are outlined in heavy black lines. The lighter blue lines represent the locations of unscaled pixel edges and are shown to help visualize how the samples are moving.



scalex=1, scaley=0



scalex=1, scaley=1



scalex=2, scaley=2

2.4.2.2 Impacts to BCI Equations

The scan converter needs to pass scale factors per quad to the SPI.

scalex[1:0] => 0 : full, 1 : half, or 2: quarter
scaley[1:0] => 0 : full, 1 : half, or 2: quarter

Assuming all pixel centers, the Upper Left (UL) calculation becomes:

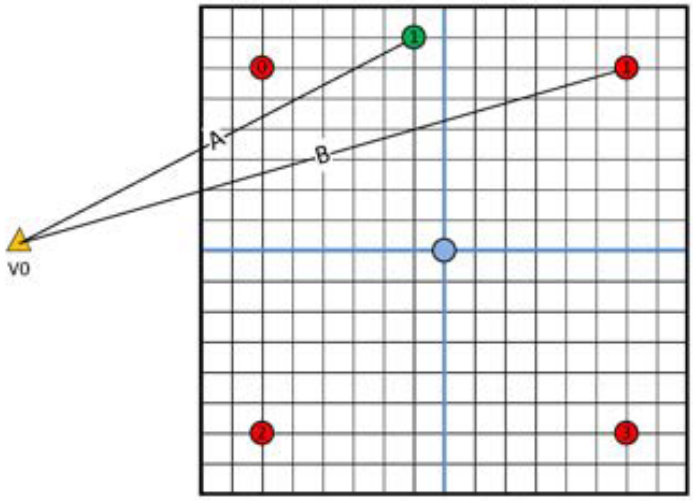
shifted_x = (scalex == 1) ? 1 : (scalex == 2) ? 2 : 0.5
shifted _y = (scaley == 1) ? 1 : (scaley == 2) ? 2 : 0.5

UL_<b> = ref_<b> + ((quad_x + shifted_x - ref_x) * <b>wdx) + ((quad_y + shifted_y - ref_y) * <b>wdy)

The other pixels are calculated as (again, assuming all pixel centers…)

UR_<b> = UL_<b> + (d<b>_dx << scalex)
LL_<b> = UL_<b> + (d<b>_dy << scaley)
LR_<b> = LL_<b> + (d<b>_dx << scalex)
(<b> is i,j, 1/w, i/w, or j/w)

The more generic form of this equation for handling any allowable sample location within each pixel is a bit more involved. The UL sample location may not be at pixel center, and arbitrary sample points in the UL pixel don't simply shift by 0.5 or 1.0 pixels when scaling. The following diagram shows the distance "A" from reference vertex V0 to unscaled UL sample ID 1 in green as compared to the distance from V0 to scaled UL sample ID 1 in red.



Sample location state settings have a format of 4b signed offset from pixel center and range from -8/16 to 7/16. Once the UL sample_id and offset is determined (center, centroid, sample) the Upper Left (UL) calculation becomes:

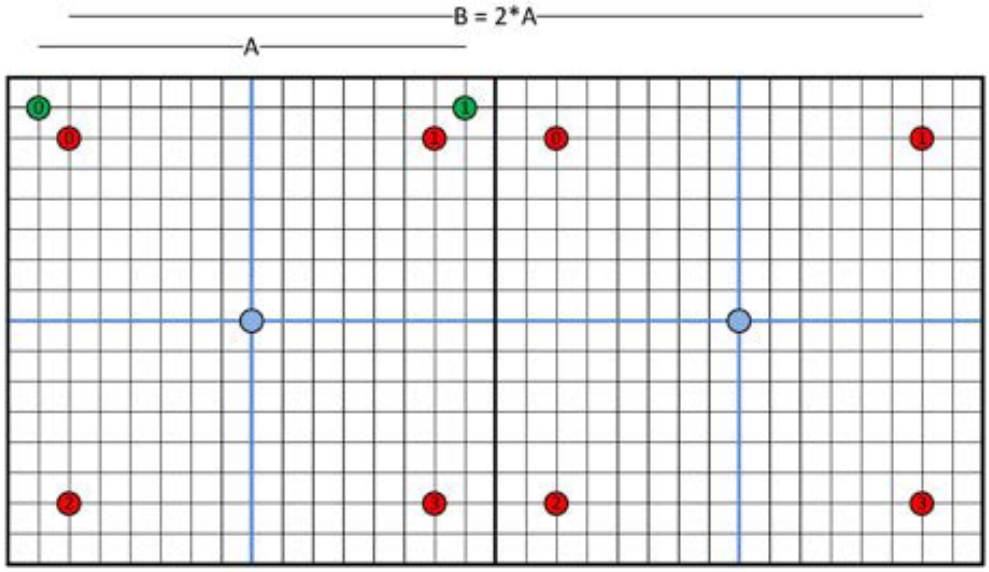
Reference Pixel (1 per quad, UL):
scaled_offset_x = (sample_offset_x << scale_x) + ((scalex == 1) ? 1 : (scalex == 2) ? 2 : 0.5)
scaled_offset _y = (sample_offset_y << scale_y) + ((scaley == 1) ? 1 : (scaley == 2) ? 2 : 0.5)

delta_x and delta_y are 16.8 2's comp convert to flt pt distance of quad UL sample location from vertex 0. quad_xy is the upper left screen space location of the current quad. If quad_xy is quantized based on scale factor then the add below for (quad + scaled_offset) can be implemented as a concatenation.
delta_x = quad_x + scaled_offset_x - ref_x
delta_y = quad_y + scaled_offset_y - ref_y

IWref = IWo + (delta_x * Iwdx) + (delta_y * Iwdy)
JWref = JWo + (delta_x * Jwdx) + (delta_y * Jwdy)
InvWref = InvWo + (delta_x * InvWdx) + (delta_y * InvWdy)

The distance between sample locations in the UL pixel and sample locations in the other three pixels do simply scale based on scale factor. The next diagram shows the distance A between unscaled UL_sample_0 and UR_sample_1 in

green as compared to the distance B between scaled UL_sample_0 and UR_sample_1 in red in a scale factor 2 scenario.



The modified distance between the UL and delta pixels is a simple scaling of the current distance calculation. This means the BCI can leave the existing distance logic between UL and UR, LL, LR and then scale the result (add 1 or 2 to the exponent).

Delta Pixels (3 per quad):
delta_x, delta_y – distance from unscaled upper left sample point to this pixel's (UR, LL, or LR) unscaled sample point

delta_Pixels(3 per quad):
delta_x and delta_y are 4.8 2's comp convert to flt pt distance from Ref Pixel
IWpix = IWref + (delta_x * scale_x * Iwdx) + (delta_y * scale_y * Iwdy)
JWpix = JWref + (delta_x * scale_x * Jwdx) + (delta_y * scale_y * Jwdy)
InvWpix = InvWref + (delta_x * scale_x * InvWdx) + (delta_y * scale_y * InvWdy)

3 END OF SPEC UPDATES, BEYOND THIS POINT INFO MAY BE OUT OF DATE

3.1.1 Support for 16 pixels per SH

This config has two independent pixel paths (packer, BCI quad-pair, ps_ctl, sc_spi interface, PSR, etc) per Shader Engine but only one Shader Array. The two ps_ctl will both request to the same resource alloc block, with waves from either packer being allowed to execute on any of the 4 SIMD of a Compute Unit. SPI will have age based arbitration between the two ps_ctl which is based on where they are with respect to the VS that produced the

prims/pixels the ps_ctl are working on. This will be done by comparing the vtx_sync_cnt (sum of vtx_sync_cnt_q and ose_vtx_sync_cnt_q in a 2 SE config) and giving priority to the ps_ctl with the higher value since that controller is working on "older" primitives. If vtx_sync_cnt_sum is equal between the two ps_ctl, priority will ping-pong between them.

If only one PS is requesting and it fits, it wins.
If both PS are requesting but only one fits, the one that fits wins.
If both PS are requesting and both fit and ages are equal, ping-pong priority decides who wins.
If both are requesting and both fit and ages are not equal, older one wins.

Because waves from either packer can launch to either SIMD pair, thus using both export busses, export arbitration for color data needs to consider which packer launched the requesting wavefronts. In a 4 DB/SH config, SPI export arbitration cannot allow two transfers to the same DB pair at the same time on the two export busses (i.e. if bus0 is exporting to DB0/1 then bus1 can only export a color if it is for DB2/3).

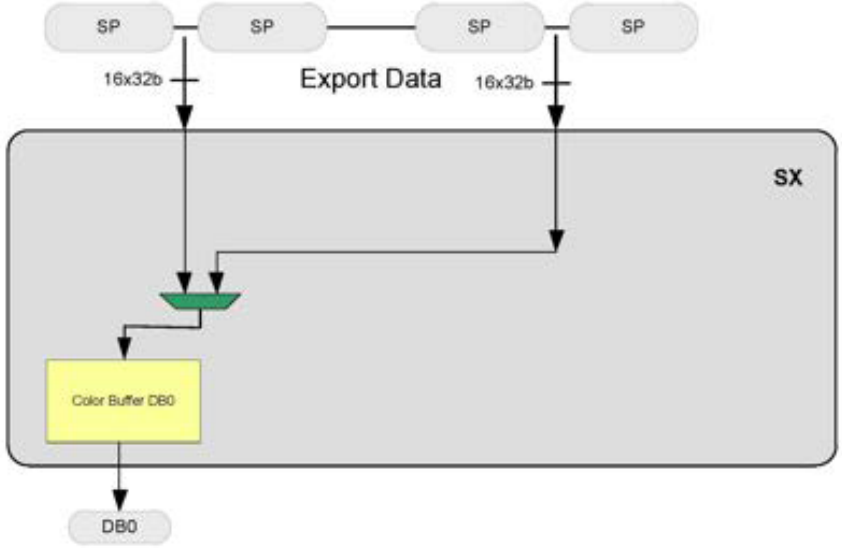


Figure 13 – Color Export Bus Arbitration, 1RB

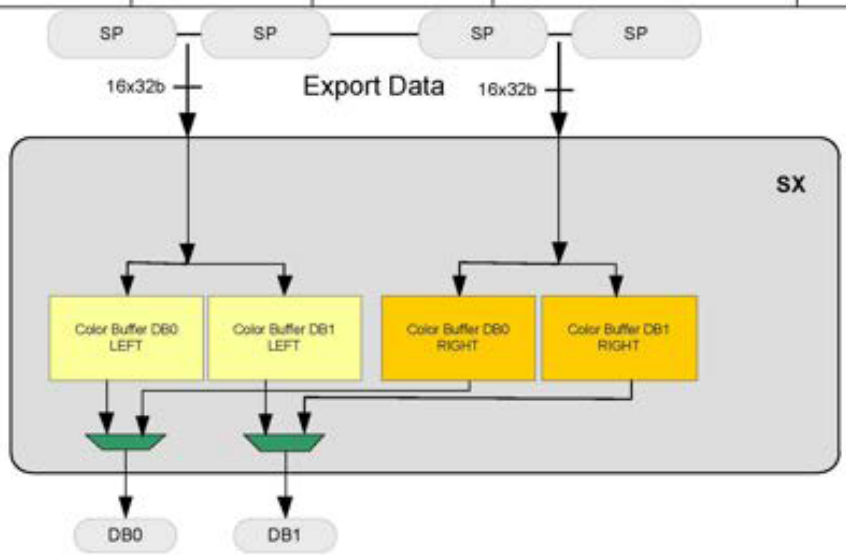


Figure 14 – Color Export Bus Arbitration, 2RB

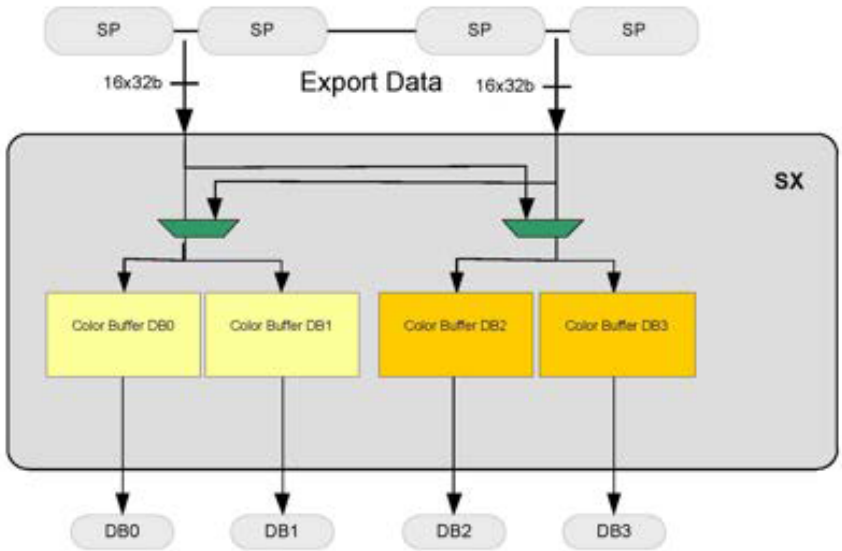


Figure 15 – Color Export Bus Arbitration, 4RB

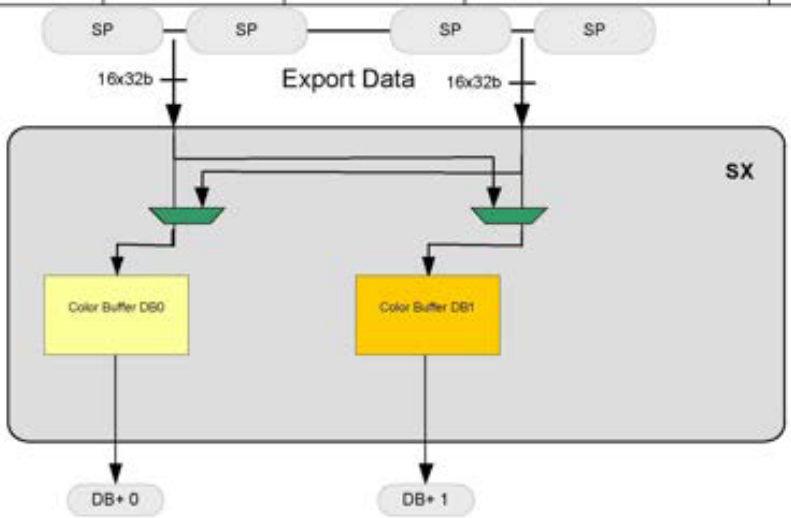


Figure 16 – Color Export Bus Arbitration, 2RB+

There is still only one 64-deep color scoreboard per SX, with waves from both packers sharing those 64 scoreboard entries. SPI will keep a single pixel_alloc fifo for the SH and will allocate color export space for all pixel waves in the same order as they allocate shader resources, regardless of pkr_id. Color buffer space is managed separately for pkr0/1, but the scoreboard space is shared. The next entry in the pixel_alloc fifo must check for space in its particular color buffers and check that there is an available scoreboard entry before servicing the pixel allocation.

For a 4DB per SX config there is one color buffer per DB in the SX and one SX_SPI_db bus per DB for freeing that space. SX_SPI_db_bus_id will always be 0 in this config because there is a one-to-one relationship between DBs and color buffers, regardless of the simd pair/export bus that processed/exported the pixel data.

3.1.2 Unique Sample Positions per Pixel

The BCI has state storage to support unique locations for all 16 samples in each of the 4 pixels of a quad. When looking up sample locations, a pixel uses it sample_id to mux the state data associated with that particular pixel.

3.2 LDS Parameter Data Loading for Pixels

For each PS, the SPI needs to copy all attributes associated with the primitives in that pixel wavefront to the LDS. The VS can export up to 32 attributes to the parameter cache, plus the SPI can generate an additional param_gen term. The SPI has to write every attribute * each primitive in the pixel wavefront. With 33 attributes, if every quad is from a unique prim, that would be 33*16 = 528 LDS writes. This would make the pixel side of processing run at its slowest pixel per clk rate.

With each quad received from the SC, the SPI gets prim boundary (flag for first quad of a prim) and per-vertex param cache base information (where in the parameter cache to read the attribute data for the vertices that created the quad). The SPI stores each of the base pointers for each unique prim and total prim count to know how many primitives exist in the pixel wavefront.

The SPI can request two primitives of attribute data per clock out of the parameter cache and write that data to the LDS. There are some cases that can't request at the rate of 2 primitives per clock. This is because those primitives are made up of vertices, the attributes of each vertex are in a specific parameter cache bank, and if consecutive primitives have vertices in the same bank then the SPI can't read that data at the same time (this is known as a bank conflict). Consequently, the SPI can send only one read address to the PC per bank.

For each attribute going to the LDS, the SPI cycles through each primitive in the pixel wavefront, writing either one or two prims every clock. Each attribute is 384 bits, made up of 3 128 bit terms. If the SPI is processing 2 primitives per clock, then 2*384 bits of data is being transferred per clock.

LDS Attribute Terms:

V0, value of attribute at the reference vertex 0 of the primitive
V10, value of attribute at vertex 1 – vertex 0
V20, value of attribute at vertex 2 – vertex 0

The PC performs the difference to avoid needing additional PS instructions for interpolation.

Figure 17 – LDS Logical Layout

3.2.1 Organization of Data in the Parameter Cache

The Vertex Shader attribute output is written to the PC with the format shown in the following diagram.

Line	Bank 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0 – A0	V0	V1	V2													V15
1 – A1	V0	V1	V2													V15
2 – A2	V0	V1	V2													V15
3 – A0	V16	V17	V18													V31
4 – A1	V16	V17	V18													V31
5 – A2	V16	V17	V18													V31
6 – A0	V32	V33	V34													V47
7 – A1	V32	V33	V34													V47
8 – A2	V32	V33	V34													V47
9 – A0	V48	V49	V50													V63
10 – A1	V48	V49	V50													V63
11 – A2	V48	V49	V50													V63

Figure 18 – Parameter Cache Data Organization

The parameter cache is 16 banks wide, matching the width of wavefronts being executed in the shader array. Each bank is written by one of the 16 VS threads exporting on a given phase. Data is packed by vertex into the storage such that the attributes of a given vertex are written to sequential addresses of that vert's destination bank.

The SPI sends the base address for each VS wave to the parameter cache for each write on the expaddr interface (base_addr0 = pc_base). The parameter cache offsets each attribute write by the attribute number received from the SQ on the export command interface. The parameter cache offsets each phase write by the number of enabled attributes specified by SPI on the expaddr interface (base_addr1 = vs_export_count).

In the example above, pc_base would be 0 and the second phase of attribute2 would write at (0 + (phase*num_attr) + attr_num) = (0 + 1*3 + 2) = 5.
In the case of half-pack waves, the writes for phase2 and phase3 would have 0 write-masks so those locations will not be written into the parameter cache.

3.2.2 VS-PS Remapping

The SPI provides support for the driver implementation of Vertex Shader (VS) output to Pixel Shader (PS) input re-mapping. The SPI can load up to 32 normal parameters from the parameter cache into the LDS. For each pixel shader input, the driver can specify the attribute number that should be read from the parameter cache. This specified attribute number is added to the primitive's parameter cache pointers to determine the read address sent to the PC. If a PS input parameter has no matching VS output then the driver can set a bit for that parameter instructing the SPI to instead load a selectable "default" value for the current parameter.

3.2.3 Flat Shading

Flat shading means that all pixels of a primitive should get the same parameter value from a provoking vertex.

Using LDS read instructions to move the interpolated data is the expected method for "constant shading" in DX10. In DX10, interpolants must declare shading as constant and there is no global renderstate disable, so therefore the compiler knows exactly which interpolants to read directly from the LDS.

There is a flat_shade disable in DX9, but we don't have to preserve NAN/INF/integer terms exactly in DX9, so always using interp instructions is fine there. When flat shading these type attributes, SPI uses provoking vtx from PA to swizzle the param cache pointers before reading the attribute data so that the constant shading term is loaded to P0. SPI drives the correct mux_select to PC so that P10 and P20 are forced to 0 in the param cache before sending to the LDS. When the interp instruction is executed the operation will be P0 + 0 + 0, resulting in P0 for all pixels.

3.2.4 Point Sprite Override

per-channel selects of 0,1,s,t, param cache value on the LDS loads.
The SPI will send the appropriate mux_sel bits to the PC to tell it to override the param data before sending it to the LDS. When the pixel shader sources the LDS, it will get the override param data and will get the correct (S,T,1,0) in the destination VGPR.

3.2.5 PARAM_GEN

Pixel Attribute loading from the parameter cache into the LDS can include an SPI generated parameter "param_gen" ST value. This data is typically used for anti-aliasing of points or lines. The LDS input term is loaded with W=T, Z=S, Y=0, X=0. If PARAM_GEN is set, lds_load will write the param_gen term into LDS address (NUM_INTERP).

3.2.6 Support Deeper Parameter Cache and Avoid Duplicate Data

GFXIP_7 offers support for removing the duplication of the data in the parameter caches by keeping the same amount of memories (in order to have the read ports available) but splitting the allocation between even and odd parameters. PC0 block will still contain 64 memories but will only store the X and Y components for 16 vertices and 2 parameters (1 even param and one odd param). PC1 will do the same but for Z and W components. Then to guarantee there are no conflicts for parameter reads, SE0_SPI will read even parameters on even phases and odd parameters on odd phases and SE1_SPI will read even parameters on odd phases and odd parameters on even phases. The difference engine pipeline will be inserted after the parameters are read and PC0 will have the diff engine for SE0 while PC1 will have the diff engine for SE1. SPI allocs space for an even number of attributes in param cache, rounding up VS_EXPORT_COUNT when necessary.

3.2.6.1 Performance

If the VS exports an odd number of attributes then the final attribute should be exported twice, once to each of the even and odd halves of the param cache. From a VS point of view, this means enabling an additional export term and then writing the final real attribute to the extra term.

If a PS linked to a VS with an odd number of exports also has an odd number of input attributes and sources one more attribute from the even param cache than the odd param cache, any PS Input attribute that sources the final real VS export (OFFSET == final VS export) should have its DUPlicate bit set so the LDS write controller knows the attribute can be read on either param cache phase.

> **SPI_PS_INPUT_CNTL_*.DUP** – "DUPlicate" bit that tags a PS Input Attribute as having been duplicated in both even and odd param cache halves so that it can be read on either phase.

If the VS exports only a single term it still needs to be duplicated to the odd param cache so that PS waves can read the attribute from either bank on consecutive clocks. If a VS with odd num_exports has knowledge that a specific attribute may or may not be used by different linked PS, that attribute should be duplicated. The conditional attribute could be moved to the last spot in the parameter cache, or it could be duplicated in place. If duplicated in place, the OFFSET field for subsequent PS inputs needs to be adjusted accordingly by the driver.

As an example, if VS attribute ID 2 is conditionally used by different PS and it is moved to the last spot in the parameter cache, the parameter cache data and corresponding PS input settings could look like this:

PC0 Even	PC1 Odd
0	1
3	4
2	2

PS Input	Semantic	Offset	DUP
0	2	4	1
1	4	3	
2	3	2	
3	0	0	
4	1	1	

If that same case instead duplicates attribute ID 2 in place, the parameter cache data and PS input settings would look like this:

PC0 Even	PC1 Odd
0	1
2	2
3	4

PS Input	Semantic	Offset	DUP
0	2	2	1
1	4	5	
2	3	4	
3	0	0	
4	1	1	

The LDS write controller will have two separate attribute machines, one for even OFFSET attributes and one for odd OFFSET attributes. The even machine always handles even OFFSETs and the odd machine always handles odd OFFSETs. Since DUP will only ever be set for a PS input sourcing the final real VS export when there are an odd total number of exports, only the even LDS machine will need to handle DUP attributes.

If the even machine reaches an attribute marked with DUP, it is allowed to generate reads to both halves of the parameter cache and will do so in successive clocks. Any DUP read from the even machine will override reads from the odd machine, blocking the odd machine until the DUP attribute is complete.

3.3 Pixel Shader VGPR initialization

Two SPI registers, SPI_PS_INPUT_ENA and SPI_PS_INPUT_ADDR, control the enabling of IJ calculations and specifying of VGPR initialization. SPI_PS_INPUT_ENA is used to determine what gradients are enabled for setup, whether per-pixel Z is enabled, what terms are calculated and/or passed through the baryc logic, and what is loaded into VGPR for PS. SPI_PS_INPUT_ADDR can be used to manipulate the VGPR destination of terms that are enabled by INPUT_ENA, typically providing a way to maintain consistent VGPR addressing when terms are removed from INPUT_ENA. It is valid to set a bit in ADDR when the corresponding bit in ENA is not set, but if the ENA bit is set then the corresponding bit in ADDR must also be set. These two registers contain an identical set of fields and consist of the following:

Field Name	IJ / VGPR Terms	VGPR Dest with Full Load
PERSP_SAMPLE_ENA	PERSP_SAMPLE I	VGPR0
	PERSP_SAMPLE J	VGPR1
PERSP_CENTER_ENA	PERSP_CENTER I	VGPR2
	PERSP_CENTER J	VGPR3
PERSP_CENTROID_ENA	PERSP_CENTROID I	VGPR4
	PERSP_CENTROID J	VGPR5
PERSP_PULL_MODEL_ENA	PERSP_PULL_MODEL I/W	VGPR6
	PERSP_PULL_MODEL J/W	VGPR7
	PERSP_PULL_MODEL 1/W	VGPR8
LINEAR_SAMPLE_ENA	LINEAR_SAMPLE I	VGPR9
	LINEAR_SAMPLE J	VGPR10
LINEAR_CENTER_ENA	LINEAR_CENTER I	VGPR11
	LINEAR_CENTER J	VGPR12
LINEAR_CENTROID_ENA	LINEAR_CENTROID I	VGPR13
	LINEAR_CENTROID J	VGPR14
LINE_STIPPLE_TEX_ENA	LINE_STIPPLE_TEX	VGPR15
POS_X_FLOAT_ENA	POS_X_FLOAT	VGPR16
POS_Y_FLOAT_ENA	POS_Y_FLOAT	VGPR17
POS_Z_FLOAT_ENA	POS_Z_FLOAT	VGPR18
POS_W_FLOAT_ENA	POS_W_FLOAT	VGPR19
FRONT_FACE_ENA	FRONT_FACE	VGPR20
ANCILLARY_ENA	RTA_Index[26:16], Sample_Num[11:8], Prim Typ[1:0]	VGPR21
SAMPLE_COVERAGE_ENA	SAMPLE_COVERAGE	VGPR22
POS_FIXED_PT_ENA	Position (Y[16], X[16])	VGPR23

The above table shows VGPR destinations for PS when all possible terms are enabled. If PS_INPUT_ADDR == PS_INPUT_ENA, then PS VGPRs pack towards VGPR0 as terms are disabled.

Field Name	ENA	ADDR	IJ / VGPR Terms	VGPR Dest
PERSP_SAMPLE_ENA	1	1	PERSP_SAMPLE I	VGPR0
			PERSP_SAMPLE J	VGPR1
PERSP_CENTER_ENA	1	1	PERSP_CENTER I	VGPR2
			PERSP_CENTER J	VGPR3
PERSP_CENTROID_ENA	0	0	PERSP_CENTROID I	X
			PERSP_CENTROID J	X
PERSP_PULL_MODEL_ENA	0	0	PERSP_PULL_MODEL I/W	X
			PERSP_PULL_MODEL J/W	X
			PERSP_PULL_MODEL 1/W	X
LINEAR_SAMPLE_ENA	0	0	LINEAR_SAMPLE I	X
			LINEAR_SAMPLE J	X
LINEAR_CENTER_ENA	0	0	LINEAR_CENTER I	X
			LINEAR_CENTER J	X
LINEAR_CENTROID_ENA	0	0	LINEAR_CENTROID I	X
			LINEAR_CENTROID J	X
LINE_STIPPLE_TEX_ENA	0	0	LINE_STIPPLE_TEX	X
POS_X_FLOAT_ENA	1	1	POS_X_FLOAT	VGPR4
POS_Y_FLOAT_ENA	1	1	POS_Y_FLOAT	VGPR5
POS_Z_FLOAT_ENA	0	0	POS_Z_FLOAT	X
POS_W_FLOAT_ENA	0	0	POS_W_FLOAT	X
FRONT_FACE_ENA	0	0	FRONT_FACE	X
ANCILLARY_ENA	0	0	Ancil Data	X
SAMPLE_COVERAGE_ENA	0	0	SAMPLE_COVERAGE	X
POS_FIXED_PT_ENA	0	0	Position (Y[16], X[16])	X

However, if PS_INPUT_ADDR != PS_INPUT_ENA then the VGPR destination of enabled terms can be manipulated.

Field Name	ENA	ADDR	IJ / VGPR Terms	VGPR Dest
PERSP_SAMPLE_ENA	1	1	PERSP_SAMPLE I	VGPR0
			PERSP_SAMPLE J	VGPR1
PERSP_CENTER_ENA	1	1	PERSP_CENTER I	VGPR2
			PERSP_CENTER J	VGPR3
PERSP_CENTROID_ENA	0	1	PERSP_CENTROID I	VGPR4 skipped
			PERSP_CENTROID J	VGPR5 skipped
PERSP_PULL_MODEL_ENA	0	1	PERSP_PULL_MODEL I/W	VGPR6 skipped
			PERSP_PULL_MODEL J/W	VGPR7 skipped
			PERSP_PULL_MODEL 1/W	VGPR8 skipped
LINEAR_SAMPLE_ENA	0	0	LINEAR_SAMPLE I	X
			LINEAR_SAMPLE J	X
LINEAR_CENTER_ENA	0	0	LINEAR_CENTER I	X
			LINEAR_CENTER J	X
LINEAR_CENTROID_ENA	0	1	LINEAR_CENTROID I	VGPR9 skipped
			LINEAR_CENTROID J	VGPR10 skipped
LINE_STIPPLE_TEX_ENA	0	1	LINE_STIPPLE_TEX	VGPR11 skipped
POS_X_FLOAT_ENA	1	1	POS_X_FLOAT	VGPR12
POS_Y_FLOAT_ENA	1	1	POS_Y_FLOAT	VGPR13
POS_Z_FLOAT_ENA	0	0	POS_Z_FLOAT	X
POS_W_FLOAT_ENA	0	0	POS_W_FLOAT	X
FRONT_FACE_ENA	0	0	FRONT_FACE	X
ANCILLARY_ENA	0	0	Ancil Data	X
SAMPLE_COVERAGE_ENA	0	0	SAMPLE_COVERAGE	X
POS_FIXED_PT_ENA	0	0	Position (Y[16], X[16])	X

Restrictions on programming of SPI_PS_INPUT_ENA

1) At least one of these must be enabled:
   PERSP_SAMPLE, PERSP_CENTER, PERSP_CENTROID, PERSP_PULL_MODEL
   LINEAR_SAMPLE, LINEAR_CENTER, LINEAR_CENTROID, LINE_STIPPLE
2) No POS_W_FLT w/o one of PERSP_{SAMPLE, CENTER, CENTROID, or PULL_MODEL}

3.4 Vertex/Pixel Synchronization

The SPI is responsible for synchronizing the submission of pixel waves only after the required vertex waves have completed to ensure parameter data will be in the parameter caches before loading to the LDS. Pixel shaders depend on at least one VS wavefront to be complete before PS execution can start. A PS in the SPI cannot be dependent on a VS wavefront that is also pending in the SPI. In order to generate pixels as a result of a vertex shader, the SPI must have received the VS waveDone message confirming that all of the vertex attribute data has been written to the parameter cache. There can't be a pixel wavefront in the SPI which is dependent on a vertex wavefront in the SPI because in order for the SPI to get pixels generated by a vertex shader, that vertex shader has to have been sent to the SQ to do the vertex shading, export the position and parameters, send the positions over to the SX, through the PA, create the primitives, scan for pixels in the SC, then pixels enter into the SPI. Therefore, the SPI cannot have PS-VS dependency inside of the SPI; the VS has to already have been issued. The synchronization that does happen is the SPI has to make sure that before it lets any pixels associated with a VS start shading, that the VS is completed which means written all of its export data out to the parameter cache. The design encourages the vertex shader to export position early so the latency is minimized through the SX-PA-SC path to get the primitives rasterized as soon as possible.

3.5 Combined Data Flow

The combined data flow diagram, Figure 19, shows all of the input controllers together with the VSR write arbitration, resource allocation, and shader write controllers shared between all of the controllers.

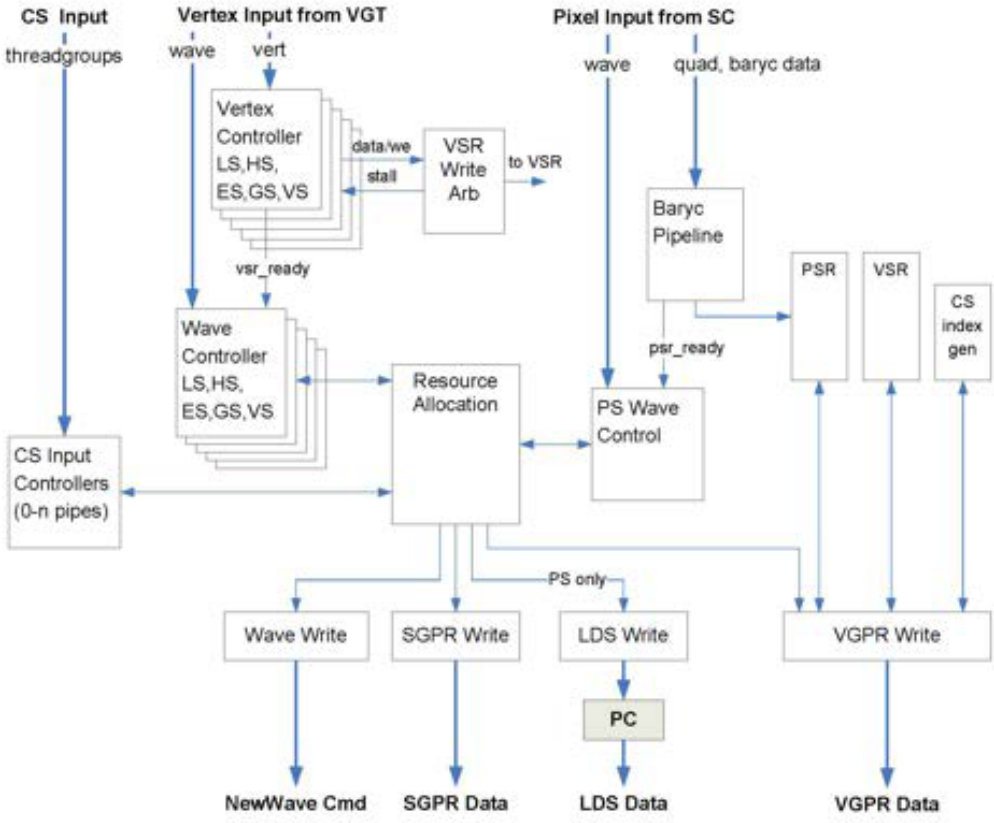


Figure 19 – Combined Data Flow

3.6 Resource Allocation

The SPI manages the following resources as part of wavefront launching (see online reg spec for ranges and units) VGPR, SGPR, wavebuffer, on-chip LDS, barriers, scratch, off-chip LDS, parameter cache and position buffer (VS only), color export buffer (PS only).
GDS command and data credits are also managed as GDS export requests are made.

3.6.1 CU and SIMD Assignment

The SPI is responsible for Compute Unit (CU) and SIMD assignment for all shader types. PS, VS, GS, ES can select between sending a wave to all 4 SIMD of a CU before stepping to the next CU or stepping CU with each wavefront (VS, GS, ES only when not part of a group that must go to a specific CU). Each LS threadgroup always steps to the next CU, and HS has to be sent to the same CU as its parent LS.

3.6.1.1 SIMD Assignment for Work Distribution and Input Bandwidth

In order to efficiently utilize ALU resources and shader input busses the SPI needs to distribute waves across SIMD as they are allocated. For ALU utilization, waves should be distributed across all the SIMD in a given CU as successive waves and/or threadgroups are sent to that specific CU. For input bandwidth utilization, waves need to be distributed across all SIMD as they are allocated – even as wave and/or threadgroup allocation moves from one CU to the next. These two desires can sometimes be at odds with each other and we need schemes that can give us acceptable behavior for both requirements.

Take an example of a compute dispatch with 2 waves per threadgroup on a 4 CU system. If we were only concerned with ALU distribution the following pattern would be acceptable:

CU	WAVE 0	WAVE 1	CU	WAVE 0	WAVE 1	CU	WAVE 0	WAVE 1	CU	WAVE 0	WAVE 1
0	SIMD0	SIMD2	0	SIMD1	SIMD3	0	SIMD0	SIMD2	0	SIMD1	SIMD3
1	SIMD0	SIMD2	1	SIMD1	SIMD3	1	SIMD0	SIMD2	1	SIMD1	SIMD3
2	SIMD0	SIMD2	2	SIMD1	SIMD3	2	SIMD0	SIMD2	2	SIMD1	SIMD3
3	SIMD0	SIMD2	3	SIMD1	SIMD3	3	SIMD0	SIMD2	3	SIMD1	SIMD3

This pattern launches the two waves from the first threadgroup to CU0, SIMD0 and 2, the two waves from the second threadgroup to CU1, SIMD0 and 2, etc. When the fourth threadgroup launch wraps back around to CU0, waves go to SIMD1 and 3. Work is distributed nicely across all SIMD in each given CU, but this pattern is not good for input bandwidth utilization because there are long sequences of successive waves that do not distribute across all SIMD. In each CU the pattern is (0,2,1,3), but the launch-order sequence is (0,2,0,2,0,2,0,2,1,3,1,3,etc). A better pattern that attacks both utilization problems looks like the following, with waves distributing across all SIMD both for launch order across CU and within each CU.

CU	WAVE 0	WAVE 1	CU	WAVE 0	WAVE 1	CU	WAVE 0	WAVE 1	CU	WAVE 0	WAVE 1
0	SIMD0	SIMD2	0	SIMD1	SIMD3	0	SIMD0	SIMD2	0	SIMD1	SIMD3
1	SIMD1	SIMD3	1	SIMD0	SIMD2	1	SIMD1	SIMD3	1	SIMD0	SIMD2
2	SIMD0	SIMD2	2	SIMD1	SIMD3	2	SIMD0	SIMD2	2	SIMD1	SIMD3
3	SIMD1	SIMD3	3	SIMD0	SIMD2	3	SIMD1	SIMD3	3	SIMD0	SIMD2

This example also illustrates how the two utilization solutions can be at odds with each other. When the fourth threadgroup launch wraps from CU3 back to CU0 the input bandwidth preference would be to allocate to SIMD0,2 but the ALU utilization preference is to allocate to SIMD1,3.

There are several register fields that control how SPI distributes compute work to CU and/or SIMD.

COMPUTE_RESOURCE_LIMITS - FORCE_SIMD_DIST
0 = Try to balance input bandwidth as threadgroups walk CU
1 = Force equal SIMD distribution within a CU, ignoring input bandwidth concerns

COMPUTE_RESOURCE_LIMITS - SIMD_DEST_CNTL
0 = adjust preferred SIMD if there's a conflict with previous start for target CU
1 = don't adjust and always prefer DEST SIMD

COMPUTE_RESOURCE_LIMITS - CU_GROUP_COUNT
Number of threadgroups to attempt to send to a CU before moving on to the next CU

If FORCE_SIMD_DIST is set to 1 then SPI will always pick back up on the SIMD where it left off the last time it sent a threadgroup to the destination CU. The setting of SIMD_DEST_CNTL is ignored when FORCE_SIMD_DIST is 1.
When force FORCE_SIMD_DIST is 0, SPI tries to balance input bandwidth by following a SIMD pattern of (0,2,1,3) even as it walks through CU. SIMD_DEST_CNTL can be used to tweak this behavior to either guarantee input bandwidth (1 = preferred for purely input limited cases), or attempt to also balance SIMD distribution within a CU (0 = preferred for cases that are not input limited).

3.6.2 GPR Management

The SPI provides V/SGPR resource management for each SIMD (see online register spec for ranges and units). VGPR and SGPR are allocated for each wavefront that launches to the Shader Array.

3.6.3 LDS Management

SPI provides resource management for LDS space for each CU.

Each CS first_wave allocates COMPUTE_PGM_RSRC2.LDS_SIZE.
Each LS first_wave allocates SPI_SHADER_PGM_RSRC2_LS.LDS_SIZE.
Each ES first_subgrp allocates SPI_SHADER_PGM_RSRC2_ES.LDS_SIZE.

Each PS wave allocates:
(num_ps_input_attributes * 12 * num_prims_in_wavefront) +
SPI_SHADER_PGM_RSRC2_PS.EXTRA_LDS_SIZE

The "* 12" is due to P0, P10, P20 each of which is 4 dwords (XYZW). The maximum number of prims in a wave is 16.

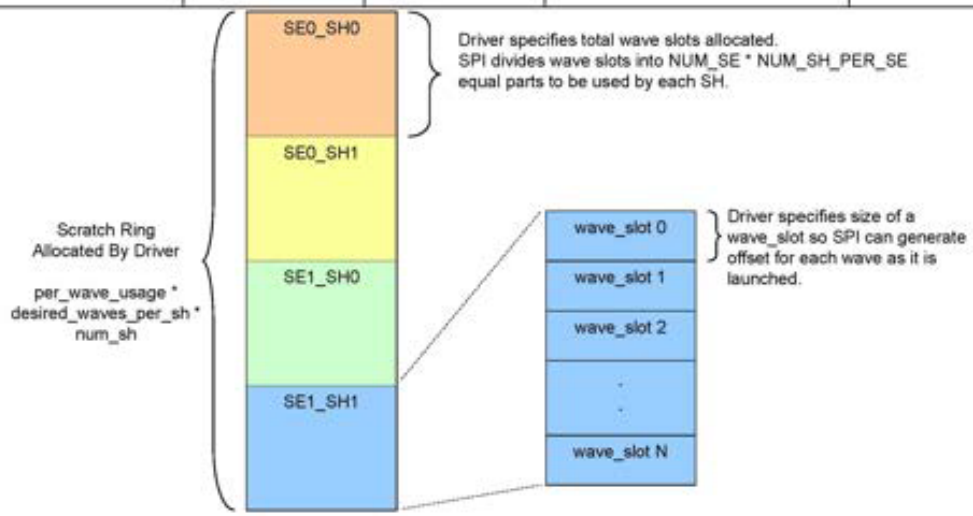
If there are 32 interpolants and 16 prims, the result is 12 * 32 * 16 = 6144 dwords.
Including param_gen, it becomes 12 * 33 * 16 = 6336 dwords, which is the maximum required LDS space for PS attribute data.

3.6.4 Wave Buffer

Provide resource management for wave buffer entries for each SIMD. Each allocated wavefront consumes one entry in its destination SIMD.

3.6.5 Scratch

The SPI provides scratch resource management, also known as the temp buffer or temp ring, for all shader types. Scratch management uses a scheme where the driver allocates temp space based on a desired number of in-flight waves in the system. The SPI will divide the driver-allocated ring into equal chunks per shader array, and also implement a management scheme that allows GFX types to share a common ring. There will be one set of resource management in the SPI shared between GFX types, and one for each of the CS pipes.



In this scheme, the driver will program the following 8-state register fields:

SPI_TMPRING_SIZE
WAVES[11:0] – Total size of allocated region in number of waves, max is 32 per CU. wave_slots are not tied directly to CU, but the max number of waves we want in flight is a function of the number of CU in the system.
WAVESIZE[24:12] – Amount of space used by each wave in dwords, format is [20:8] since each wave is 64 threads (6 bits). The API specs temp space in terms of 4 dword (component) vectors per thread up to a max of 4K 4-component vectors (16K * 64 threads = 1M dwords per wave), plus the driver needs some additional space. The current register size supports a range of 0→(2M-1) dwords.

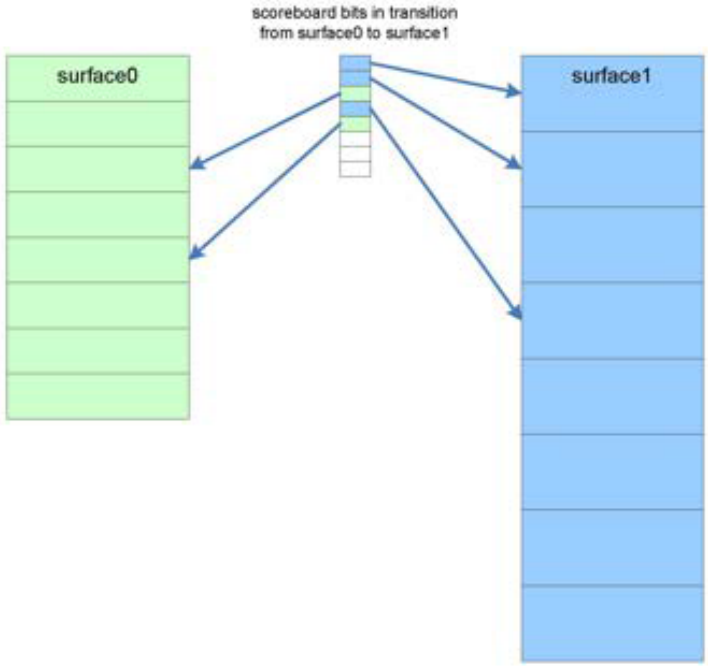
The physical base of the TMPRING will be specified as a resource, either loaded as user-data or fetched by the SQ. The SPI will provide a wave-specific offset as an SGPR term which the shader uses along with the resource to generate physical addresses. This means shaders that spill to scratch require two additional SGPR dwords for the resource and offset.

The SPI will divide SPI_TMPRING_WAVES into equal chunks per SH and maintain separate management for each SH. Temp space will be managed with an allocation scheme that allows out-of-order deallocation so waves can free their space as soon as the shader completes.

Each shader stage will have 1 persistent state bit (PGM_RSRC2_*S.SCRATCH_EN) specifying whether the shader uses temp space or not. If a shader does not use temp space, it will not allocate a wave_slot.

The driver has to check bound shaders for a draw/dispatch versus SPI_TMPRING_WAVESIZE for that cmd buffer ring to make sure it is large enough. If not, the driver needs to either allocate a new temp ring or reorganize the existing ring by changing WAVES and WAVESIZE. If the currently allocated memory region is reorganized (change WAVES and WAVESIZE but keep same resource), a PARTIAL_FLUSH through PS is required to protect the space until all pending work is done with temp. If a new temp ring is allocated then there is no need to flush. The resource (persistent user_data) can change to point to the new surface, WAVESIZE can change so the SPI generates the correct offset for waves launched using the new surface, and the management logic will just transition to using the new settings as each shader stage reaches the new draw/dispatch. There may be some period of time when the old and new surfaces are not fully used since the SPI scoreboard will contain a mix of

waves from the two temp surfaces. The scoreboard and alloc logic do not care which surface contains the allocated temp space since that info is passed to the shader program through the resource and offset SGPR values. The SPI only needs to know which temp_wave_slot was used by a given wavefront so the correct scoreboard bit can be cleared when the wave completes.



The TMPRING will typically be configured to some default size providing enough space for typical shader usage, and only needs to change when a shader with a very large TMPRING usage is bound.

3.6.6 Barrier

Barrier resources have a fixed pool of 16 in each CU and are used to synchronize multiple wavefronts in a threadgroup. Only HS and CS need barriers in the SQ because those are the only shader types that can share data between threads through the LDS. If a threadgroup consists of only one wavefront (64 threads or less), no barrier resource is allocated by the SPI.

3.6.7 Bulky CS Threadgroups

CS persistent state includes a bit that can mark the dispatch as "bulky". SPI manages a single bulky slot per CU that is consumed whenever a bulky threadgroup allocates to that CU. If the bulky slot is in use, a new CS request marked as bulky will not fit on that CU. Even single-wave threadgroups can be marked as bulky, and only one of those is allowed on a CU at a given time. Only one bulky allocation is allowed on a CU, but other types (including !bulky CS) can still allocate to that CU if other resources are satisfied.

3.6.8 Position Buffer and Parameter Cache

Parameter Cache and position buffer space are managed for VS only. VS waves allocate parameter and position space unless a given wave is only processing as stream-out and not rasterizing or is a "null wave" that is only sent to deallocate resources at the end of a group. In the non-allocating cases the shader won't be allowed to write to either of these buffers and space is not wasted. Parameter cache is deallocated on the PS side when the final primitive using vertices from the VS has issued all of its pixels. The SC passes along a token to the SPI indicating that the parameter cache space can be de-allocated. Position buffer is deallocated as the PA drains positions from the SX. There is a signal sent from the SX to SPI indicating that position buffer space is freeing.

The register PA_CL_VS_OUT_CNTL register is snooped by the SPI for position buffer calculation:

```
vs_position_count =
        (1 + PA_CL_VS_OUT_CNTL_GET_VS_OUT_MISC_VEC_ENA(data) +
        PA_CL_VS_OUT_CNTL_GET_VS_OUT_CCDIST0_VEC_ENA(data) +
        PA_CL_VS_OUT_CNTL_GET_VS_OUT_CCDIST1_VEC_ENA(data)) * 64
```

GPU__SX__POS_EXPORT_REG_BUFFER_SIZE defines the physical size of the position buffer and the default setting of SPI_SX_EXPORT_BUFFER_SIZES.POSITION_BUFFER_SIZE. This register field can be used to limit the amount of position space that the SPI allows to be in use at any given time.

There is one logical parameter cache for the entire chip. VS waves will be sent to all Shader Engines, and each SE is allowed to use and must manage (1/num_SE) of the param cache. The register SPI_VS_OUT_CONFIG is used to determine the amount of space to allocate.

```
pc_alloc_space = (((vs_export_count/GPU__GC__PC_PTR_WIDTH)+1) * ((vs_half_pack) ? 2 : 4))
```

When VS_HALF_PACK is set the VGT will create partial VS waves every 32 vertices (instead of a full 64), only filling the wave half full. This means each wave only needs half as much parameter cache space.

When the parameter cache storage is two parameters wide (PC_PTR_WIDTH = 2) the equation will round up to the nearest even value and then divide by 2. Allocation always starts on an the even bank so there will be wasted space for odd vs_export_count settings, although that wasted odd slot can be used to help performance of reads through the DUPLICATE functionality described in a previous section.

GPU__SX__PARAMETER_CACHE_DEPTH defines the physical size of the parameter cache. SPI_CONFIG_CNTL_1.PC_LIMIT_ENABLE/SIZE can be used to artificially limit the amount of parameter cache space that the SPI allows to be in use at any given time.

3.6.8.1 Late VS Allocation

The SPI supports position and param cache allocation after shader resource alloc, similar to PS color buffer alloc, allowing VS to start execution without having pos/pc space for exports. This means we can have more VS in flight than fit in pos/pc space and that means we provide more latency hiding for VS fetching and pre-export ALU.

VS late alloc is an attempt to deal with cases that are currently bottlenecked by the number of VS waves that can be in flight. From Evergreen through SI, VS waves have to alloc both parameter cache and position buffer space before launching, which means those resources can limit the number of VS waves in flight. There are cases where the majority of the VS latency is before any pos or param export (ie fetch a bunch of data initially, process it, then output), and late VS alloc can help hide that initial latency by allowing VS waves to launch without having their export space. Those VS waves can start and fetch their data and only stall if they reach an export instruction before their space has been allocated.

Ideally, LATE_ALLOC_VS should only be set high enough to keep PS fed with work - any higher than that and newer VS are just taking up resources that could be applied to older PS. In other words, if a given draw is not VS latency limited then LATE_VS_ALLOC won't help (and could potentially hurt) performance. Vertex shaders

with lots of latency before their first export, either fetches or ALU instructions, will be the best candidates to take advantage of late alloc.

And finally, care must be taken when setting LATE_ALLOC_VS > 0 since it can cause a deadlock with PS. VS are allowed to launch without having export space and those VS consume shared shader resources (GPR, scratch) until they are able to export and complete. The pos/param resources VS is waiting on are freed by PS, so PS have to be able to make progress in order for those VS to alloc, export, and complete. If late alloc VS take so much of the shared resources that PS cannot alloc and make progress, we will deadlock. This is only expected to happen when VS + PS resource usage is very large relative to resources available: > 3/4? of VGPR or SGPR, or scratch is enabled with a small total scratch pool, only a couple of CU present (small config), etc. The higher the total VS/PS resource usage is relative to total resources available, the smaller LATE_ALLOC_VS should be set.

Similar to the LS/PS LDS deadlock scenario, late alloc VS SIMD deadlock can be avoided by guaranteeing there is at least one CU that can run PS but not VS (using PGM_RSRC3 CU_EN settings). CU resource deadlock can also be avoided using reservations on a single CU to guarantee there are resources available to PS that VS cannot use. Scratch pool deadlock can be avoided by making sure that LATE_ALLOC_VS is always less than SPI_TMPRING_SIZE.WAVES when VS uses scratch.

3.6.9 Allocation Priority

The 3-ring arbitration priority scheme from SI will be extended to handle the new HP3D and multiple asynchronous compute pipes. The participating pipes will be any HP3D task (LS, HS, ES, GS, VS, PS), GFX task (LS, HS, ES, GS, VS, PS, CS), and four of the eight Compute Pipes presented by the pipe pair arbitration. The Compute Pipes presented will have one of the following pipe priorities, determined by the CPF_SPI_pipeN_priority interface for each pipe:

CS_HIGH - typically above HP3D
CS_MEDUIM - typically between HP3D and GFX
CS_LOW - below GFX

To resolve a tie between multiple compute pipes of the same pipe priority level, a least recently issued (totem pole) circuit will be employed. Each time a pipe is selected to issue any work to the shader core, the pipe will be moved to bottom of the least recently issued circuit and thus make that pipe the lowest priority of its PRIORITY until some other pipe of the same PRIORITY issues a wave.

Coming out of reset, the least recently issued list will be P0 ➔ P7 with pipe 0 the most favored initially for the given pipe priority. The diagram below illustrates the pipe arbitration. Of the five priority levels of CS HIGH, HP3D, CS MEDUIM, GFX, CS_LOW, from highest to lowest priority levels the best winner will be chosen. NOTE: If there are graphics task in HP3D, the HP3D pipe arbitration can win, but the post graphics shader type arbitration could result in a GFX wave selection. Due to the pipelining of HP3D and GFX in the same physical pipeline, there are cases where GFX or HP3D could be more important to the priority winner.

The following table shows the totem pole arrangement from left to right. The Pn where n is the compute pipe providing the work group and the (-, H, M, L) , - No work, H – pipe priority High, M – pipe priority Medium, L – pipe priority Low. For each time period the four out of eight CS pipes that survive pipe pair arbitration are shown in red, and the underlined pipeline is the one that pipe arbitration will select from the 6 competing pipelines.

Time Red pipes considered for launch, underline pipe selected for launch

Least Recently Issued H→L Priority List for P0-P7

T0	P0(-)	P1(H)	P2(-)	P3(-)	P4(-)	P5(-)	P6(-)	P7(-)	HP3D(-)	GFX(-)
T1	P0(-)	P2(-)	P3(-)	P4(M)	P5(H)	P6(-)	P7(-)	P1(H)	HP3D(-)	GFX(-)
T2	P0(M)	P2(L)	P3(L)	P4(M)	P5(H)	P6(M)	P7(M)	P1(H)	HP3D(-)	GFX(-)
T3	P0(M)	P2(L)	P3(L)	P4(M)	P6(M)	P7(M)	P1(H)	P5(-)	HP3D(-)	GFX(-)
T4	P0(M)	P2(L)	P3(L)	P4(M)	P6(M)	P7(M)	P5(-)	P1(H)	HP3D(-)	GFX(-)
T5	P2(L)	P3(L)	P4(M)	P6(M)	P7(M)	P5(-)	P1(H)	P0(M)	HP3D(-)	GFX(-)
T6	P2(L)	P3(L)	P4(M)	P6(M)	P7(M)	P5(-)	P0(M)	P1(H)	HP3D(X)	GFX(-)
T7	P2(L)	P3(L)	P4(M)	P6(M)	P7(M)	P5(L)	P0(M)	P1(-)	HP3D(-)	GFX(X)
T8	P2(L)	P3(L)	P4(M)	P6(M)	P5(L)	P0(M)	P1(-)	P7(M)	HP3D(-)	GFX(X)
T9	P2(L)	P3(L)	P4(M)	P5(L)	P0(M)	P1(-)	P7(-)	P6(M)	HP3D(-)	GFX(X)
T10	P2(L)	P3(L)	P5(L)	P0(M)	P1(L)	P7(-)	P6(-)	P4(M)	HP3D(-)	GFX(X)

The current programmable based priority selection machine enables flexible priority selection between the pipelines of the system. It can be setup to get a fixed priority or revolving priority between the pipes either fine grain or coarse grain. The following non-context configuration registers set by privileged OS/LLD during setup.

Register to specify duration of 4 sequential time periods

SPI_ARB_CYCLES_0.TS0_DURATION (16 bits)
SPI_ARB_CYCLES_0.TS1_DURATION (16 bits)
SPI_ARB_CYCLES_1.TS2_DURATION (16 bits)
SPI_ARB_CYCLES_1.TS3_DURATION (16 bits)
Granularity is 16, 64, 128, or 256 clocks sclks depending on DUR_MULT.
Range 16ns to 16ms at 1GHZ clock

Register to specify priority level ordering of each time period

SPI_ARB_PRIORITY.PIPE_ORDER_TS0 (3 bits)

Prioritization orders for Time slices

0x0 – CS_H, HP3D, CS_M, GFX, CS_L
0x1 – HP3D, CS_H, CS_M, GFX, CS_L
0x2 – HP3D, CS_H, GFX, CS_M, CS_L
0x3 – HP3D, GFX, CS_H, CS_M, CS_L
0x4 – CS_H, CS_M, CS_L, HP3D, GFX
0x5 – CS_M, CS_L, HP3D, GFX, CS_H
0x6 – CS_L, HP3D, GFX, CS_H, CS_M

SPI_ARB_PRIORITY.PIPE_ORDER_TS1 (3 bits), Same encoding as TS0
SPI_ARB_PRIORITY.PIPE_ORDER_TS2 (3 bits), Same encoding as TS0
SPI_ARB_PRIORITY.PIPE_ORDER_TS3 (3 bits), Same encoding as TS0

SPI_ARB_PRIORITY.TS0_DUR_MULT (2 bits)
SPI_ARB_PRIORITY.TS1_DUR_MULT (2 bits)
SPI_ARB_PRIORITY.TS2_DUR_MULT (2 bits)
SPI_ARB_PRIORITY.TS3_DUR_MULT (2 bits)
Number of sclks used to increment duration count: 0-16, 1-64, 2-128, 3-256.

Hardware Queue Descriptor (HQD) Register to specify priority of a given pipe

CP_HQD_PIPE_PRIORITY. PIPE_PRIORITY (2 bits)

0x0 - CS Low
0x1 - CS Medium
0x2 - CS High

Example 0 - Fixed priority order

TS0_Duration = TS1_Duration = TS2_Duration = TS3_Duration = 16
Pipe_Order_TS0 = Pipe_Order_TS1 = Pipe_Order_TS2 = Pipe_Order_TS3 = 2
Result in priority 2 selection from highest to lowest
HP3D pipe always selected if present
Any CS_H job surviving to final pipe arbitration
Any GFX task ready to go
Any CS_M job surviving to final pipe arbitration
And last any CS_L

Example 1 - Alternate HP3D and CS_H as higher priority 50/50

TS0_Duration = TS1_Duration = TS2_Duration = 16
TS3_Duration = 48
Pipe_Order_TS0 = Pipe_Order_TS1 = Pipe_Order_TS2 = 2
Pipe_Order_TS3 = 0

50% of the time Result in priority 2 selection order from highest to lowest
HP3D pipe always selected if present
Any CS_H job surviving to final pipe arbitration
Any GFX task ready to go
Any CS_M job surviving to final pipe arbitration
And last any CS_L
50% of the time Result in priority 0 selection order from highest to lowest
Any CS_H job surviving to final pipe arbitration
HP3D pipe always selected if present
Any CS_M job surviving to final pipe arbitration
Any GFX task ready to go
And last any CS_L

Example 2 - Picking between Graphics Stages

SPI_CONFIG_CNTL.GPR_WRITE_PRIORITY applies across all graphics requests, regardless of HP3D or GFX.

If GPR_WRITE_PRIORITY = Low -> High (LS, HS, ES, GS, VS, PS), and the current PIPE_ORDER is 0x1 = HP3D, CS_H, CS_M, GFX, CS_L and we have req from
P0_High
P3_Med
P7_High
LS_HP3D
VS_GFX
PS_GFX

Then the compute work will lose to graphics because there is an HP3D request and HP3D is highest priority given the current PIPE_ORDER. But the final winner in this example will be PS because it has the highest GPR_WRITE_PRIORITY.

If duration and ordering are all desired to be constant, then setting up this machine once provides a constant priority between CS_H, HP3D, CS_M , GFX, CS_L pipes.

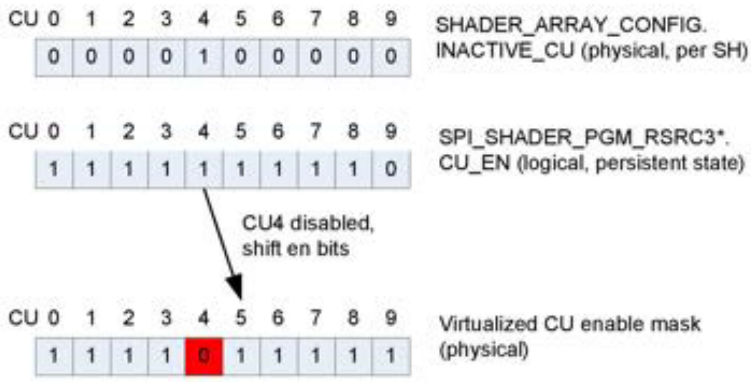
At reset or change in programming the machine will restart in order continuously:
TS0 → TS1 → TS2→ TS3 → TS0 etc….

The registers can be programmed at any time without idling the shader core.

3.6.10 Virtualization of Compute Unit Masks

The programmable SPI_SHADER_PGM_RSRC3.CU_EN registers provide a logical representation of the CU in a given config, with bits 0 to N-1 representing the N possible CU that can be enabled in that config. Depending on which physical CU are disabled by the SHADER_ARRAY_CONFIG regs, the SPI will shift the logical CU_EN settings to create a physical enable mask consistent with the current config. When CU_EN masks are virtualized the SPI doesn't need unique settings per SH for CU masking, but each SH has to virtualize based on its own SHADER_ARRAY_CONFIG setting.

As an example, take a part that has 10 physical CU and one disabled CU (CU4). The CU_EN settings are shifted based on the config setting, with each bit above the disabled CU moving up to an enabled CU.



The SPI supports any number from 0 to (NUM_CU – 1) disabled CU.

3.6.11 Resource Reservations

In order to support configs with one operational CU (either due to config, harvesting, or clk/power gating), we need to solve the problem of deadlock between TS and PS in the LDS. SI parts could do this by setting a reservation on half of the LDS for PS, but that means no other types are allowed to use that LDS including CS. SI resource reservation works by only allowing the one specified type to use the reserved resources. For GFXIP_7, the desire is to be able to set a reservation and allow multiple types to use that reserved space. This will be done by replacing the TYPE_A/B fields of SPI_RESOURCE_RESERVE with a new field that is a one-hot mask, one bit for each pipe, gfx stage, etc to specify which types are allowed to use the reservation. The SPI registers for controlling resource reservations are shown below.

SPI_RESOURCE_RESERVE_CU_0 - n

VGPR	3:0	= 0-8 blocks of 16 VGPR per SIMD; (64 VGPRs/CU)
SGPR	7:4	= 0-8 blocks of 32 SGPR per SIMD; (128 SGPRs/CU)
LDS	11:8	= 0-8 blocks of 4Kbytes LDS

WAVES 14:12 = 0-5 blocks of 1 per SIMD (4 Waves/CU)
BARRIERS 18:15 = 0-8 barriers per CU

SPI_RESOURCE_RESERVE_EN_CU_0 - n

EN	0:0	0x0	Enable the reservations
TYPE_MASK	15:1	0x0	1 = PS, 2 = VS, 3 = GS, 4 = ES, 5 = HS, 6 = LS, 7 = DXCS, 8-15 = CS0-CS7
QUEUE_MASK	23:16	0x0	16=QueueSlot0, 17=QueueSlot1, 18=QueueSlot2, . . . , 23=QueueSlot7 for all enabled compute pipes in the type mask
RESERVE_SPACE_ONLY	24:24		Mode bit for reserve type use of reservation space 0 - Use both the available reserved and non-reserved space 1 - Use only the available reserved space

The RESERVE_SPACE_ONLY feature is only honored for compute only reservations. If the TYPE_MASK includes any GFX_*, (PS, VS, GS, ES, HS, LS,CS) task in the reservation this bit will be forced to 0 and prevent the use of RESERVE_SPACE_ONLY feature.

Similar to per-type CU_EN regs, reservation settings are also virtualized such that the registers are logical and hardware maps them to physical CU based on the current SHADER_ARRAY_CONFIG.

3.6.12 Multiplier for Resource Limits

WCL_PIPE_PERCENT_{GFX/HP3D}, WCL_PIPE_PERCENT_CS{0-7} – 5 bit value where 0=1/32, 1=2/32 . . . 31=1.0 becomes a multiplier of the pipeline wave in-flight registers so that a scheduling thread can provide asynchronous or synchronous control of the wave limit distribution across pipelines.

Graphics wave limits are specified as SPI_SHADER_PGM_RSRC3.WAVE_LIMIT[5:0]. WAVE_LIMIT has a granularity of 16 and a setting of 0 disables the limit.

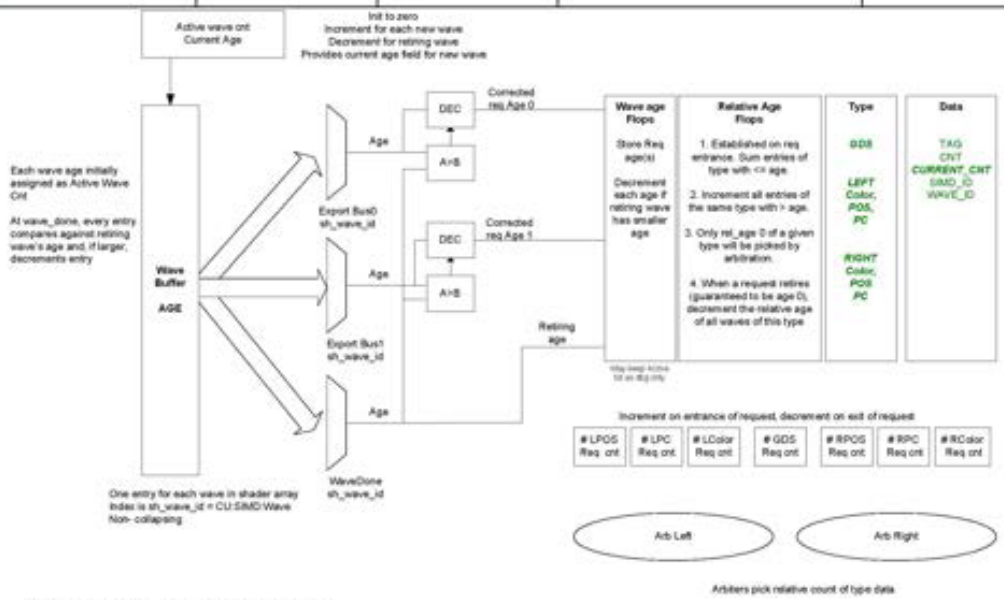
00 = no limit,
01 = up to 16 waves allowed,
02 = up to 32 waves allowed,
---
31 = up to 496 waves allowed

Compute wave limits are specified as COMPUTE_RESOURCE_LIMITS.WAVES_PER_SH[9:0], which has a granularity of 1 and a setting of 0 disables the limit. 0 = no limit, 1 = 1 wave allowed, 1023 = 1023 waves.

When PIPE_PERCENT is multiplied with WAVE_LIMIT the result should not be allowed to truncate or round to 0, which would effectively disable the wave limit. SPI will make sure that if WAVE_LIMIT > 0, the minimum allowed multiplied result will be 1 wave.

3.7 Export Arbitration

Export requests are made on 1 or possibly multiple export requests busses. They must simply be added to the export request buffer (ERB) that is dedicated to the bus making the request. The slot in the buffer is uniquely identified by the cu-id, tag-id provided with the request. The tags are divided evenly across the export busses. E.g. if there are 12 tags available with 2 export busses, then tag-ids 0-5 are reserved for export bus 0 and tags 6-11 are reserved for export bus 1.



The arbitration rules are as follows:

1. Arbitration occurs to grant access to all export busses (2 or 4) for a given 4 phase cycle.
2. There are 4 export types (Position (POS), Parameter Cache (PC), GDS and Color (COL))
3. Arbitartion grants occur as follows:
   a. 2 busses: Simd 0/1 granted on phase 0. Simd 2/3 granted on phase 2.
   b. 4 busses: Simd N granted on phase N (N=0,1,2,3)
4. A fixed, but programmable priority, based on export type is maintained. E.G. Color can be assigned to a higher priority that GDS.
5. Within the POS, PC, and COL types relative wave ages (per export bus) are maintained. This means the export request is assigned an age based upon the age of its request wave, and not based upon the order the request was received. Older exports are prioritized over younger.
6. GDS request type is maintained in request order age, and a single order for all request busses is maintained.
7. GDS type (and only GDS) also requires an allocation of GDS resources be performed before issuing a GDS grant. The resources are a GDS "CMD" input buffer, and a GDS "DATA" input buffer. The arbiter maintains a count of available space (decrementing such space for each grant, and incrementing the space available under control of a "free" bus from the GDS. For each GDS export to be granted the Arbiter requires 1 CMD space, and 4 DATA spaces so each command granted will decrement the values accordingly. There is a GDS_CMD_FREE and a GDS_DATA_FREE. Each of these adds 1 to its respective counter.

```
cmd_count == cfg_specific_cmd_count_default;
data_count == cfg_specific_data_count_default;

If ( (cmd_count >= 1) && (data_count >= 4))
    Grant_gds_requests <= TRUE;
Else
    Grant_gds_requests <= FALSE;
```

```
If (gds_export_granted)
    Cmd_count <= cmd_count - 1;
    Data_count <= data_count – 4;

If (gds_spi_cmd_free)
    Cmd_count <= cmd_count + 1;

If (gds_spi_data_free)
    Data_count <= data_count + 1;
```

8. Conflict rules: Because the resource being exported to may have a single write port, it is illegal for certain combinations to occur on the (multiple) export grant busses for a given 4-phase cycle:
   a. At most one bus can grant a POS request
   b. At most one bus can grant either a PC or a GDS request. Some single SH configs have 2 busses from SX to param cache, allowing SPI to grant a PC export on both busses.
   c. Configs with one packer per SH have no restrictions on color buffer grants. Configs with two packers per SH can only allow exports destined for a given DB pair on one export bus at a time.
   d. PC grants are further constrained as there is a single logical write port to the PC shared by all shaders in a system.
      This is accomplished by having each Wave Buffer in a system skew its PC grants, by component, by a unique amount from all other wave buffers in the system.
9. To provide fairness, each arbiter cycle rotates priorities amongst the export busses. The bus with highest priority chooses its preferred export. The next bus chooses its preferred export – but in light of the previous busses choice, and so on.

Two Busses:
bus 0, bus1
bus1, bus0
Four Busses
Bus0, bus1, bus2, bus3
Bus1, bus2, bus3, bus0
Bus2, bus3, bus0, bus1
Bus3, bus0, bus1, bus2

3.7.1 Maintaining GDS order

To keep GDS in strict order across both export busses, a fifo is used. The fifo is written for each export request, providing the index into the export request buffer for each such request (If there are two busses, the fifo is wide enough to record two indices at a time).

This FIFO when not empty enters into arbitration . The aribiter may thus choose a GDS request as a winner. If so, the index is present to the export control block (which contains the export request memory). The relevant data is looked up for the winning grant, and otherwise the grant will act much as a grant on a non-GDS type.

3.7.2 Export Granting

When the arbiter selects a given export for granting, the index of the export is used to read the associated data from the export request buffer. The index is used to regenerate a cu-id and a tag-id. If this is the final grant of a given request then (req mask will be zero after the update), then the "done" flag is asserted.

When the done flag is asserted the request is complete and counters, etc may be updated to reflect this.

3.8 Persistent State

The SPI supports persistent state management for graphics shader stages, both for HP3D and GFX. For each shader stage, there is storage for one whole set of state at the shader launch point preceded by fifo storage for incremental updates. The update fifos provide storage for some number of words that could either cover a lot of draw/dispatch calls with small state changes or a few with large state changes. For this class of state, the persistent set is used until all waves using the set have been launched, at which time the wavefront launch is stalled while the persistent set is updated with the incremental state changes. Once all updates have been applied (up until the next DRAW_INITIATOR) the shader stage is allowed to launch waves from the new state set. Each shader stage can do its respective update independently and likely at different times.

The independent persistent state per stage decouples the shader stage state from other stages and pipelines. This results in better use of the remaining 7 sets of context state (8th set will be clear state) by minimizing how often the shared state sets are needed. The shader stage state can be many small state changes or a few large state changes. The update and final persistent storage will be doubled so the interrupted GFX state has a place to be saved while HP3D work is processed. The diagram below shows the fifos and storage for persistent state and staging.

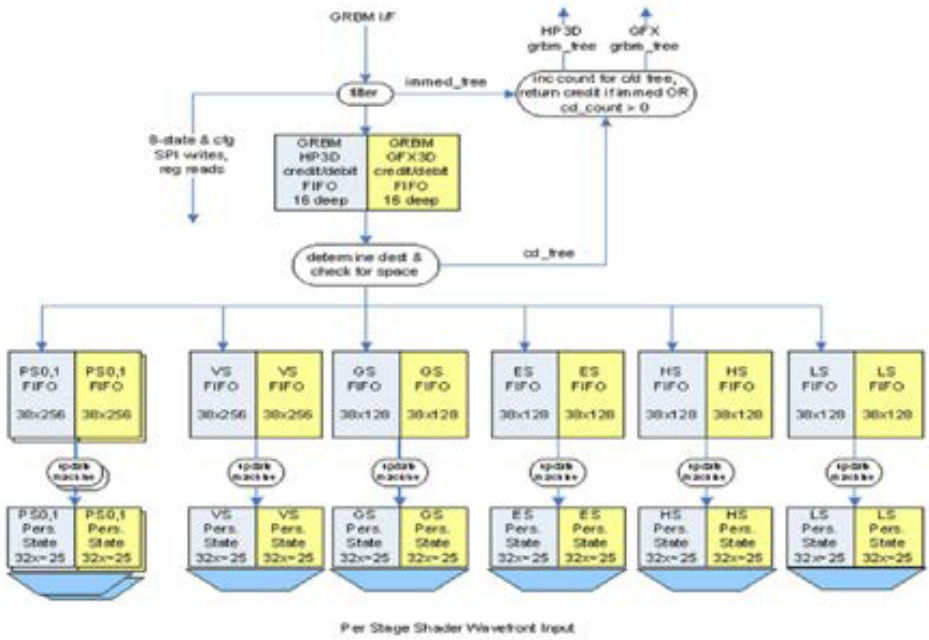


Figure 20 – Persistent State Update FIFOs

3.9 Partial Flush Events

SPI provides support for partial pipe flushes through event synchronization with the CP. The only way to change static resource allocation and config state is to either flush the whole pipe or use partial flush events. The difference between doing a PS_PARTIAL_FLUSH and a full pipe flush is that the PS guarantees that the pixel

shader has finished, but does not guarantee that all data has been written out to memory. The latency savings is the difference between waiting for memory writes to complete.

3.10 Wave/Event Ordering

SPI needs to maintain wave and event ordering for certain functionality, such as fence/flush event signaling and VS_done-to-PS_fpos synchronization. Since waves can complete their shader program and pop off the wave buffer out of order, SPI needs another method for remembering order. To accomplish this the SPI will keep an event_wave_crawler for each shader type that gets pushed for every wave and for events of interest to that type. A 'done' status bit is kept for each crawler spot and the crawler cannot advance past a wave until that spot's done bit is set. This allows waves to complete and dealloc shared resources out of order while also keeping events and waves in order.

3.11 Event Collection

SPI must collect certain event_id across graphics shader types and notify the CP when the events are done. These events are end_of_pipe (EOP) type events and CONTEXT_DONE. For CONTEXT_DONE, SPI must see the event across all of LS,HS,ES,GS,VS,PSn before signaling the CP. For EOP events, SPI must see the event across all of LS,HS,ES,GS,VS,PSn and DX11-CS before signaling CP. EOP events include CACHE_FLUSH_TS, CACHE_FLUSH_AND_INV_TS, BOTTOM_OF_PIPE_TS, FLUSH_AND_INV_DB_DATA_TS, and FLUSH_AND_INV_CB_DATA_TS.

3.12 H/V (horizontal/vertical) Pixel Picker (for Debug and Performance Analysis)

Registers for controlling this feature, SPI_{P0/P1}_TRAP_SCREEN:

PSBA_LO	This is the pre-shader base address [39:8] This specifies the address in memory of the shader program that will be invoked by pixels of interest.
PSMA_LO	This is the pre-shader memory address [39:8] – This specified the address in memory used to store the data structure used by the pixel picker pre-shader. The memory can contain constants, atomic variables and an append region.
GPR_MIN.VGPR_MIN	Number of Vector General Purpose Registers (VGPR) needed for the pre-shader. If this is larger than the number of VGPRs needed by the native shader, the SPI uses this setting for any wavefront that uses the pre-shader. Note: The pre-shader is responsible for preserving VGPRs if it plans to resume the native shader.
GPR_MIN.SGPR_MIN	Number of Scalar General Purpose Registers (SGPR) needed for the pre-shader in addition to the 16 extra trap registers. If this is larger than the number of SGPRs needed by the native shader, the SPI uses this setting for any wavefront that uses the pre-shader. Note: The pre-shader is responsible for preserving SGPRs if it plans to resume the native shader and these are in addition to the 16 extra trap registers.

SPI also shadows the privileged SC register controlling locking of the TRAP_SCREEN settings by a privileged client.

PA_SC_P3D_TRAP_SCREEN_HV_LOCK DISABLE_NON_PRIV_WRITES	Disables writes to P3D (P0) TRAP_SCREEN regs by non-privileged clients.
PA_SC_HP3D_TRAP_SCREEN_HV_LOCK DISABLE_NON_PRIV_WRITES	Disables writes to HP3D (P1) TRAP_SCREEN regs by non-privileged clients.

SPI gets new trap_mask bits from SC_SPI_pc_prim bus, 4 bits per quad, and builds it into a 64 bit mask for the wavefront. If any of the bits are 1 it means that the Pixel Picker feature is enabled and the pixel represented by the set bit was detected by the SC as a pixel of interest (POI) for the Pixel Picker feature.
A non-zero trap_mask for the wavefront will cause the SPI to allocate an extra 16 SGPRs (the trap SGPRs) for the wave. If the SPI_TRAP_SCREEN_VGPR_MIN register value is larger than normal pixel shader's VGPR requirements, the SPI chooses the larger allocation. If the SPI_TRAP_SCREEN_SGPR_MIN register value is larger than normal pixel shader SGPR requirements, the SPI chooses the larger allocation prior to the bump of the extra 16 trap SGPRs.
Next, the SPI sets up the wavefront to run the pre-shader prior to the normal shader by loading the extra necessary state as follows:

Program Count (PC)	←	Pre-shader base address (PSBA)
(traptemps 6, 7)	←	Pre-shader memory address (PSMA)
(traptemps 8, 9)	←	Pre-shader 64 bit pixel-of-interest mask
(traptemps 10,11)	←	Normal pixel shader Program Base Address (PBA) for continuance

Priv = 1 ← pre-shader is a special trap handler in privileged mode that has access to an extra 16 SGPRs and potentially the indicated additional VGPRs/SRGPRs. Other traps will be blocked until preshader completes.

Additionally the following is required if a generic trap routine exists:
TBA ← trap base address
TMA ← trap memory address
TrapEnable = 1

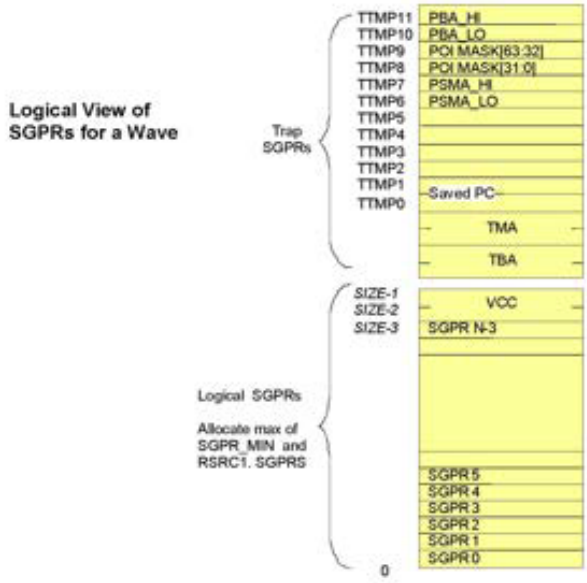


Figure 21 – Persistent State Update FIFOs

3.13 Wavefront Lifetime Status Counters

The overall intent of this feature is to:

- Measure the maximum wavefront lifetime for each task (up to 21 total, including HP3D [LS, HS, ES, GS, VS, PS], GFX [CS, LS, HS, ES, GS, VS, PS], CS [Pipe 0-7]) with a minimum granularity of 1024 GPU clocks.
- Provide a feedback mechanism (interrupt through SQG) if any wavefront lifetime exceeds the maximum programmable latency value

A new physical CLK_CNT[13:0] will be provided to count SCLKS and provide a lower speed clock to the wavefront counter update process.

For each physical wavefront that can be active in the system at a time, storage for a 31 bit (WF_LIFETIME_CNT[30:0]) wave lifetime count, a 1 bit START flag, and 1 bit ACTIVE flag will be provided. There will be 40 instances (10 Waves/SIMD * 4 SIMD/CU) for each CU in the system. The wave type and source pipeline will be obtained from existing storage in the current wave buffer. The counts stored in these locations will have a saturate at the max possible value. The START flag will be used to reset the respective count to zero on the first update after a wavefront has started. The ACTIVE flag will be used to indicate the counter should be incremented and that a test against max value and limit value for the type needs to be done.

The solution will be controlled and monitored based on the following registers:

Register Name	R/W	Description
WF_LIFETIME_EN	R/W	Enable for the Wavefront lifetime counter feature. 0 ➔ CLK_CNT is disabled from counting 1 ➔ CLK_CNT is enabled to count
WF_LIFETIME_SAMPLE_PERIOD[3:0]	R/W	The hardware will add a counter (CLK_CNT[13:0]) that will increment on per core clock (SCLK, period of 1.25ns) if WF_LIFETIME_EN == 1. This register will indicate the number of clocks (in units of 1024 SCLKS) required for CLK_CNT[13:0] to count to trigger a process to increment each active WF_LIFETIME_CNT and test against limit registers. This counter controls the time unit granularity used to measure the lifetime of each wavefront. The range of settings will be 0 (1024 SCLKs) ➔ 15 (16,384 SCLKs) in units of 1024 SCLKs. Based on a SCLK running at 800MHz, the range of the period settings supported by this counter is ~1.28us to 19.2us.
WF_LIFETIME_LIMIT[31:0]	R/W	A 32-bit render state limit register per pipeline (HP3D, GFX, CS P0 – P7), 10 in total. This render state register has two fields: [31] - EN_WARN - When set, indicates that the warning mechanism for waverfronts initiated from this ring/pipe has been enabled. The GPU will generate an interrupt to the host for the first wavefront of each type that exceeds the programmable maximum lifetime value. [30:0] – MAX_CNT – The WF_LIFETIME_CNT[30:0] will be tested against this limit, if exceeded • Capture max time for task type in MAX_STATUS • Send interrupt if EN_WARN is set and this is the first occurrence for task type since status was last read.
WF_LIFETIME_STATUS[31:0]	Read Only	MAX_CNT[30:0] – worst case wave lifetime duration since last read. INT_GENERATED[31] –hardware bit to prevent issuing more

		than one interrupt per wave type since last status poll. Cleared when WF_LIFETIME_STATUS is read. There will be one instances of this register for each task (21 total) HP3D (LS, HS, ES, GS, VS, PS), GFX (CS, LS, HS, ES, GS, VS, PS), CS (P0-7) Register will capture the largest WF_LIFETIME_CNT for each task type. • The first time a wavefront exceeds its limit register (WF_LIFETIME_LIMIT), if the EN_WARN flag is set an interrupt will be generated. • Subsequent interrupts for the offending wave type will be mask until the status register has been read. (Hidden state per MAX_STATUS register will enable interrupt mask) The purpose of this registers is to record the maximum running value for a given waverfront type for each pipe/ring. The user can read this set of registers to learn the maximum running values at any given time.

This is how the above registers are used:

- For each tick of the SCLK, if (WF_LIFETIME_EN == 1) CLK_CNT is incremented by 1
- At each CLK_CNT == WF_LIFETIME_SAMPLE_PERIOD, initiate wave lifetime counts process
  Reset CLK_CNT = 0;
  For each wave
  - If the wavefront has START flag == 1 in wave buffer
    - Zero the count and clear start flag
  - Else if ACTIVE flag in wave buffer is set
    - Increment WF_LIFETIME_CNTs by 1
  - If new WF_LIFETIME_CNT > WF_LIFETIME_STATUS.MAX_CNT of it's type's
    - Set corresponding WF_LIFETIME_STATUS.MAX_CNT = WF_LIFETIME_CNT
  - If new WF_LIFETIME_CNT > WF_LIFETIME_LIMIT.MAX_CNT (limit register per type)
    - If !WF_LIFETIME_STATUS.INT_GENERATED & WF_LIFETIME_LIMIT.EN_WARN
      - Generate interrupt for task
      - Set hidden WF_LIFETIME_STATUS.INT_GENERATED = 1
      - Set WF_LIFETIME_STATUS(type) = WF_LIFETIME_CNT

  End For wave

- Read of a WF_LIFETIME_STATUS register will set fields MAX_CNT=0 and INT_GENERATED=0
- When a wavefront is created the START flag and ACTIVE flag will be set
- At the completion/exit of the wavefront the active flag will be cleared, preventing false readings/usage of a stale count

4 Performance

CS - desired performance of launching 16 threads per SE when VGPR and SGPR load times allow
PS - desired performance of launching 16 threads per SE when VGPR, SGPR, and LDS load times allow
Vertex types – desired performance of 1 vertex per clock when VGPR and SGPR load times allow

The SPI's goal will be to efficiently use a single VGPR write port phase to load the required input VGPR and saturate the use of the wires to the SP.
The SPI's goal will be to efficiently use all four phases of the single SGPR write bus to load the required input SGPR and saturate the use of the wires to the SQ.

4.1 Barycentric Calculation

There will be 4 quads worth of barycentric logic (2 per packer) to enable 16 pixels per clock. Any fully covered quad pair row should calculate center/centroid IJ in one clk instead of two if both are enabled.

4.2 Parameter Cache Read

Read parameter data for attribute interpolation from the parameter cache storage in the SX at a rate of two primitives per clock, assuming there are no bank select conflicts between the two primitives, and direct the writing of that data to the LDS.

4.3 GPR Loading

Sustain loading 1 vertex component per clock per-SE for VS, GS, ES, HS, and LS from the VGT to the SP VGPRs for peak vertex rates.

Sustain 1 CS wavefront issue every 4 clocks (16 threads per clk) when writing < 3 VGPR components and < 17 SGPR dwords. For multi-SE configs, this can only be achieved for ordered threadgroups > 1 wave for async compute due to the signaling that must occur between SPI to maintain threadgroup launch order.

Sustain 1 PS wavefront issue every 4 clocks (1 from each packer every 8 clocks), 16 threads per clk, when writing < 3 VGPR components and < 17 SGPR dwords.

4.4 Pixel

Sustain 8 pixels per clock per packer with one IJ and two attributes when there are no more than 4 primitives per pixel vector and cylindrical wrap is disabled and there are no parameter cache pointer conflicts. Performance should scale efficiently with each additional interpolant and/or GPR load.

4.5 Graphics Balanced Throughput Cases

Figure 22 shows the balanced throughput case of VS and PS only where a test provides precisely the amount of data required to demonstrate the peak performance of the design. This example assumes a config that can provide 4 quads per clk to the SPI with 2 VGPR input busses to load data. Each VS wave takes 64 clocks to build and a minimum of 4 clocks to send through the SPI and into the VGPRs. Each PS wave takes a minimum of 8 clocks, but with 2 input busses that is equivalent to 1 wave every 4 clocks. If every VS creates exactly 15 PS, the pipe is perfectly balanced through the SPI into the VGPR.

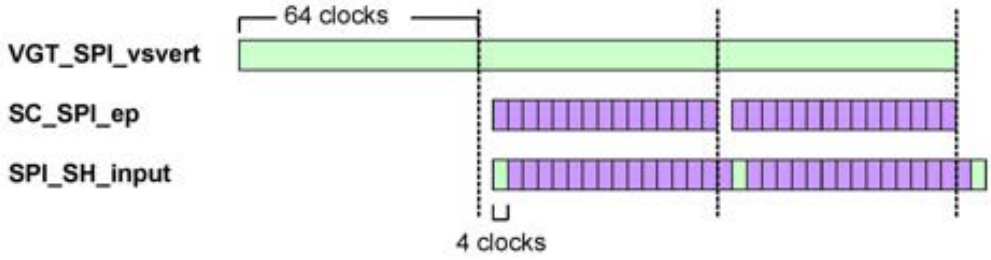


Figure 22 - Performance, Balanced Throughput Case, VS-PS

Figure 23 shows the balanced case when Geometry Shading is enabled. ES,GS,VS take 64 clocks to build, 4 clocks to issue, each PS takes 4 clocks. If every ES creates 1 GS, every GS creates 1 VS, every VS creates 13 PS, the pipe is perfectly balanced through the SPI.

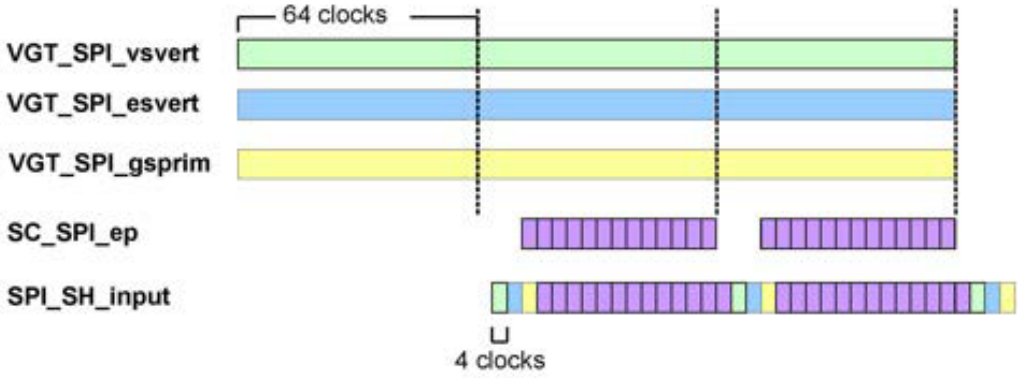


Figure 23 – Performance, Balanced Throughput Case, ES-GS-VS-PS

This illustrates why multiple buffers exist for the types arriving from the VGT. In the described peak mode, with all of these types arriving in parallel, there is a wavefront for 3 different types arriving at the same time. The SPI can only send the wavefront to the GPR one at a time. If the SPI was single buffered, the other two that didn't win first would have to stall the VGT inputs until the first was issued to the SQ. The SPI does indeed have multiple buffers to prevent this bottleneck from happening.

Figure 24 shows the balanced throughput case for Tessellation along with Geometry shading. LS, HS, ES, GS, VS, PS - 5 wave types coming from the VGT every 64 clocks.

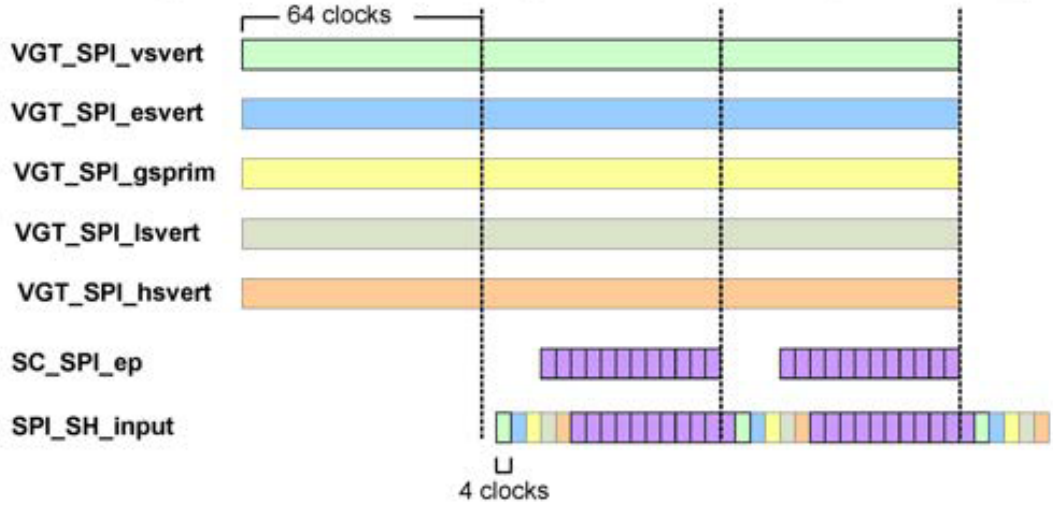


Figure 24 – Performance, Balanced Throughput Case, LS-HS-ES-GS-VS-PS

4.6 Performance Counters

The list of SPI perfcounter selects is defined in …/doc/design/blocks/spi/perfcount.xls

4.6.1 Performance Counter Binning

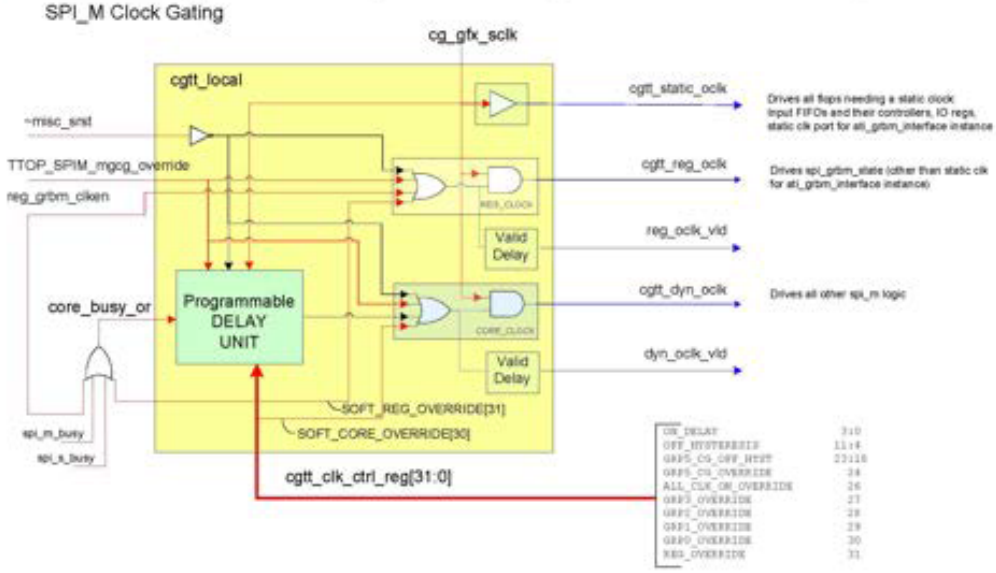
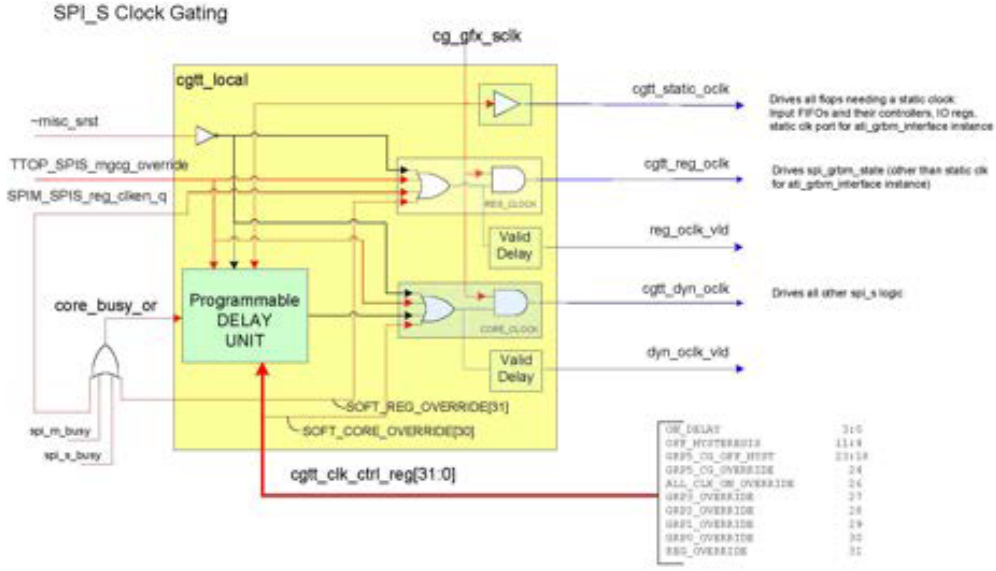
SPI_PERFCOUNTER_BINS.*_MIN/_MAX define the range for which an associated counter should increment. The BIN* at the end of a perfcount select determines which BIN setting that particular count looks at, and the range of BIN for each count is given in spi.blk. For instance, if one of the perfcount selects is set to SPI_PTR_BUFF_PRIM_BIN0, BIN0_MIN is set to 0, and BIN0_MAX is set to 4, then the count should increment for each wave that contains between 1 and 5 primitives.

5 Clock Gating

The SPI acts as the CGTS (shader core) clock gating master controller. When the SPI starts receiving new shader work for processing, the clock gate state machine begins the process of turning on the ALU clocks. The SPI must see "all_clocks_on" go active from the clock gate SM before launching wavefronts.

The SPI also includes CGTT medium grain clock gating that allows the SPI clocks to turn off when the block is idle.

SPI_M Clock Gating



SPI_S Clock Gating