Reference Guide

Evergreen Family Instruction Set Architecture

Instructions and Microcode

November 2011

Revision 1.1a

Advanced Micro Devices, Inc.
One AMD Place
P.O. Box 3453
Sunnyvale, CA 94088-3453
www.amd.com

For AMD Accelerated Parallel Processing:

URL:	**developer.amd.com/appsdk**
Developing:	**developer.amd.com/**
Support:	**developer.amd.com/appsdksupport**
Forum:	**developer.amd.com/openclforum**

Contents

Figures

Tables

Preface

About This Document

This document describes the instruction set and the microcode formats native to the Evergreen family of processors that are accessible to programmers and compilers.

The document specifies the instructions (including the format of each type of instruction) and the relevant program state (including how the program state interacts with the instructions). Some instruction fields are mutually dependent; not all possible settings for all fields are legal. This document specifies the valid combinations.

Audience

This document is intended for programmers writing application and system software, including operating systems, compilers, loaders, linkers, device drivers, and system utilities. It assumes that programmers are writing compute-intensive parallel applications (streaming applications) and assumes an understanding of requisite programming practices.

Organization

This document begins with an overview of the Evergreen family of processors' hardware and programming environment (Chapter 1). Chapter 2 describes the organization of an Evergreen-family program and the program state that is maintained. Chapter 3 describes the control flow (CF) programs. Chapter 4 the ALU clauses. Chapter 5 describes fetches through a vertex cache clause. Chapter 6 describes the fetches through a texture cache clause. Chapter 7 describes memory read clauses. Chapter 8 describes data share clauses. Chapter 9 describes instruction details, first by broad categories, and following this, in alphabetic order by mnemonic. Finally, Chapter 10 provides a detailed specification of each microcode format.

Registers

The following list shows the names are used to refer either to a register or to the contents of that register.

GPRs	General-purpose registers. There are 128 GPRs, each one 128 bits wide, organized as four 32-bit values.
CRs	Constant registers. There are 512 CRs, each one 128 bits wide, organized as four 32-bit values.
AR	Address register.
loop index	A register initialized by software and incremented by hardware on each iteration of a loop.

Endian Order

The Evergreen-family architecture addresses memory and registers using little-endian byte-ordering and bit-ordering. Multi-byte values are stored with their least-significant (low-order) byte (LSB) at the lowest byte address, and they are illustrated with their LSB at the right side. Byte values are stored with their least-significant (low-order) bit (lsb) at the lowest bit address, and they are illustrated with their lsb at the right side.

Conventions

The following conventions are used in this document.

`mono-spaced font`	A filename, file path, or code.
*	Any number of alphanumeric characters in the name of a code format, parameter, or instruction.
< >	Angle brackets denote streams.
[1,2)	A range that includes the left-most value (in this case, 1) but excludes the right-most value (in this case, 2).
[1,2]	A range that includes both the left-most and right-most values (in this case, 1 and 2).
{x | y}	One of the multiple options listed. In this case, x or y.
0.0	A single-precision (32-bit) floating-point value.
1011b	A binary value, in this example a 4-bit value.
7:4	A bit range, from bit 7 to 4, inclusive. The high-order bit is shown first.
italicized word or phrase	The first use of a term or concept basic to the understanding of stream computing.

Related Documents

- *CTM HAL Programming Guide*. Published by AMD.
- *AMD Intermediate Language (IL) Reference Manual*. Published by AMD.
- *OpenGL Programming Guide*, at http://www.glprogramming.com/red/

- *Microsoft DirectX Reference Website*, at http://msdn.microsoft.com/archive/default.asp?url=/archive/en-us/directx9_c_Summer_04/directx/graphics/reference/reference.asp
- GPGPU: http://www.gpgpu.org

Differences Between the R700-Family and Evergreen-Family of Devices

The following bullets provide a brief overview of the more important differences between the R700-family and the Evergreen-family of GPUs.

- Pixel parameter interpolation is now performed in kernel code, rather than in fixed-function hardware. Parameter data is preloaded into the local data share (LDS) before a pixel wavefront launch, and the kernel uses new `INTERP_*` instructions to evaluate a primitive's vertex attribute value of each pixel.
- Texture and vertex fetch clauses are now defined by which cache (TC or VC) services the clause, rather than by fetch-type.
- Local data share (LDS) is now accessed through ALU instructions, rather than fetch instructions.
- Added support for jump-tables.
- Added the ability to write from a maximum of four streams to a maximum of four stream-out buffers.
- Added support for flexible DX11 tesselation using hull shaders (HS) and domain shaders (DS).
- Added work-group synchronization in hardware for compute shaders (CS).
- Added support to dynamically index texture resources, texture samplers, and ALU constant buffers.
- Removed the Fbuffer and the Reduction buffer.
- Added support for floating point rounding and denormal modes.
- ALU clauses can now use up to four constant buffers using the ALU_EXTENDED opcode.
- Added support for exception flag collection.
- Added single-step control of the control flow, allowing instructions to be issued through register writes to the SQ, as well as arbitrary instructions to be inserted in the execution path.

Contact Information

To submit questions or comments concerning this document, contact our technical documentation staff at: streamcomputing@amd.com.

For questions concerning AMD Accelerated Parallel Processing products, please email: streamcomputing@amd.com.

For questions about developing with AMD Accelerated Parallel Processing, please email: streamdeveloper@amd.com.

You can learn more about AMD Accelerated Parallel Processing at: http://www.amd.com/stream.

We also have a growing community of AMD Accelerated Parallel Processing users. Come visit us at the AMD Accelerated Parallel Processing Developer Forum (http://www.amd.com/streamdevforum) to find out what applications other users are trying on their AMD Accelerated Parallel Processing products.

Chapter 1
Introduction

The Evergreen family of processors implements a parallel microarchitecture that provides an excellent platform not only for computer graphics applications but also for general-purpose streaming applications. Any data-intensive application that can be mapped to a 2D matrix is a candidate for running on an Evergreen family processor.

Figure 1.1 shows a block diagram of the Evergreen family processors.

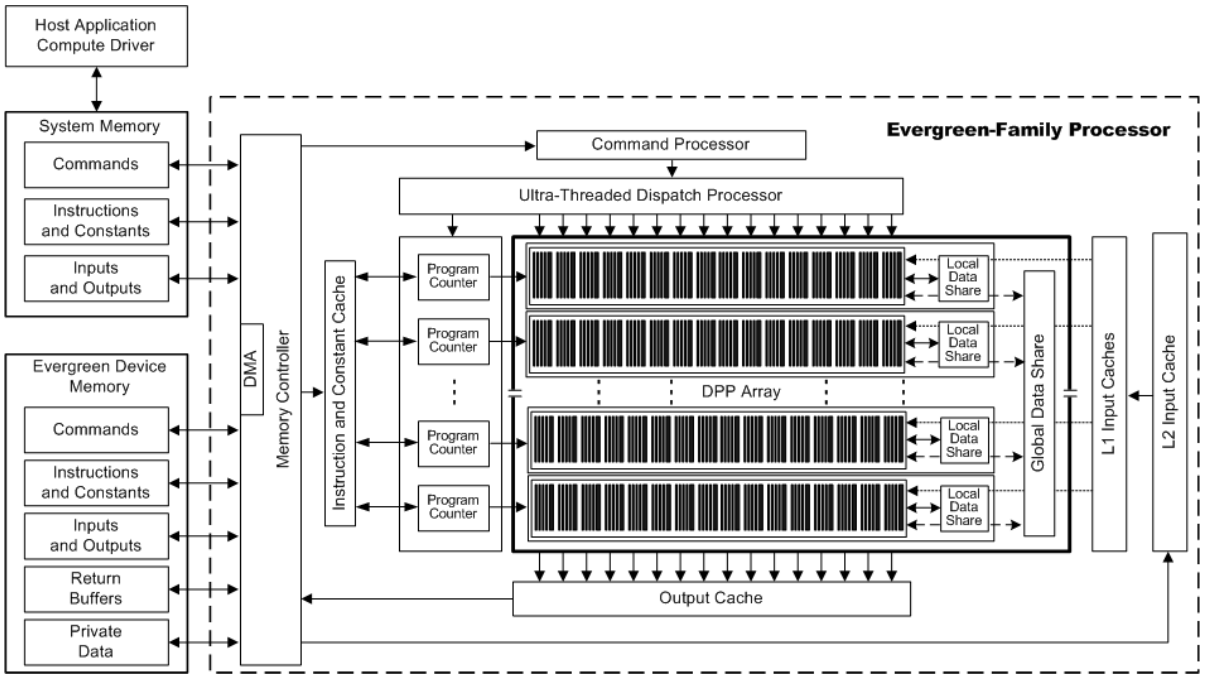


Figure 1.1 Evergreen Family Block Diagram

It includes a data-parallel processor (DPP) array, a command processor, a memory controller, and other logic (not shown). The Evergreen command processor reads commands that the host has written to memory-mapped Evergreen registers in the system-memory address space. The command processor sends hardware-generated interrupts to the host when the command is completed. The Evergreen memory controller has direct access to all Evergreen device memory and the host-specified areas of system memory. To satisfy read and write requests, the memory controller performs the functions of a direct-memory access (DMA) controller, including computing memory-address offsets based on the format of the requested data in memory.

A host application cannot write to the Evergreen device memory directly, but it can command the Evergreen to copy programs and data between system memory and Evergreen memory. For the CPU to write to GPU memory, there are two ways:

- Request the GPU's DMA engine to write it there by pointing to the location of the source data on CPU memory, then pointing at the offset in the GPU memory to which it then is written.
- Upload a kernel to run on the shaders that access the memory through the PCIe link, then process it and store it in the GPU memory.

A complete application for the Evergreen includes two parts:

- a program running on the host processor, and
- programs, called *kernels*, running on the Evergreen processor.

The Evergreen programs are controlled by host commands, which

- set Evergreen-internal base-address and other configuration registers,
- specify the data domain on which the Evergreen GPU is to operate,
- invalidate and flush caches on the Evergreen GPU, and
- cause the Evergreen GPU to begin execution of a program.

The Evergreen driver program runs on the host.

The DPP array is the heart of the Evergreen processor. The array is organized as a set of compute unit pipelines, each independent from the others, that operate in parallel on streams of floating-point or integer data. The compute unit pipelines can process data or, through the memory controller, transfer data to, or from, memory. Computation in a compute unit pipeline can be made conditional. Outputs written to memory can also be made conditional.

Host commands request a compute unit pipeline to execute a kernel by passing it:

- an identifier pair (x, y),
- a conditional value, and
- the location in memory of the kernel code.

When it receives a request, the compute unit pipeline loads instructions and data from memory, begins execution, and continues until the end of the kernel. As kernels are running, the Evergreen hardware automatically fetches instructions and data from memory into on-chip caches; Evergreen software plays no role in this. Evergreen software also can load data from off-chip memory into on-chip general-purpose registers (GPRs) and caches.

Conceptually, each compute unit pipeline maintains a separate interface to memory, consisting of index pairs and a field identifying the type of request (program instruction, floating-point constant, integer constant, boolean constant, input read, or output write). The index pairs for inputs, outputs, and constants are

specified by the requesting Evergreen instructions from the hardware-maintained program state in the pipelines.

The Evergreen family of devices can detect floating point exceptions, but does not generate interrupts. In particular, it detects IEEE floating-point exceptions in hardware; these can be recorded for post-execution analysis. The Evergreen GPU can detect floating point exceptions, but does not generate interrupts. The software interrupts shown in Figure 1.1 from the command processor to the host represent hardware-generated interrupts for signalling command-completion and related management functions.

Figure 1.2 shows a programmer's view of the dataflow for three versions of an Evergreen application. The top version (a) is a graphics application that includes a geometry shader program and a DMA copy program. The middle version (b) is a graphics application without a geometry shader and DMA copy program. The bottom version (c) is a general-purpose application. The square blocks represent programs running on the DPP array. The circles and clouds represent non-programmable hardware functions. For graphics applications, each block in the chain processes a particular kind of data and passes its result on to the next block. For general-purpose applications, only one processing block performs all computation.

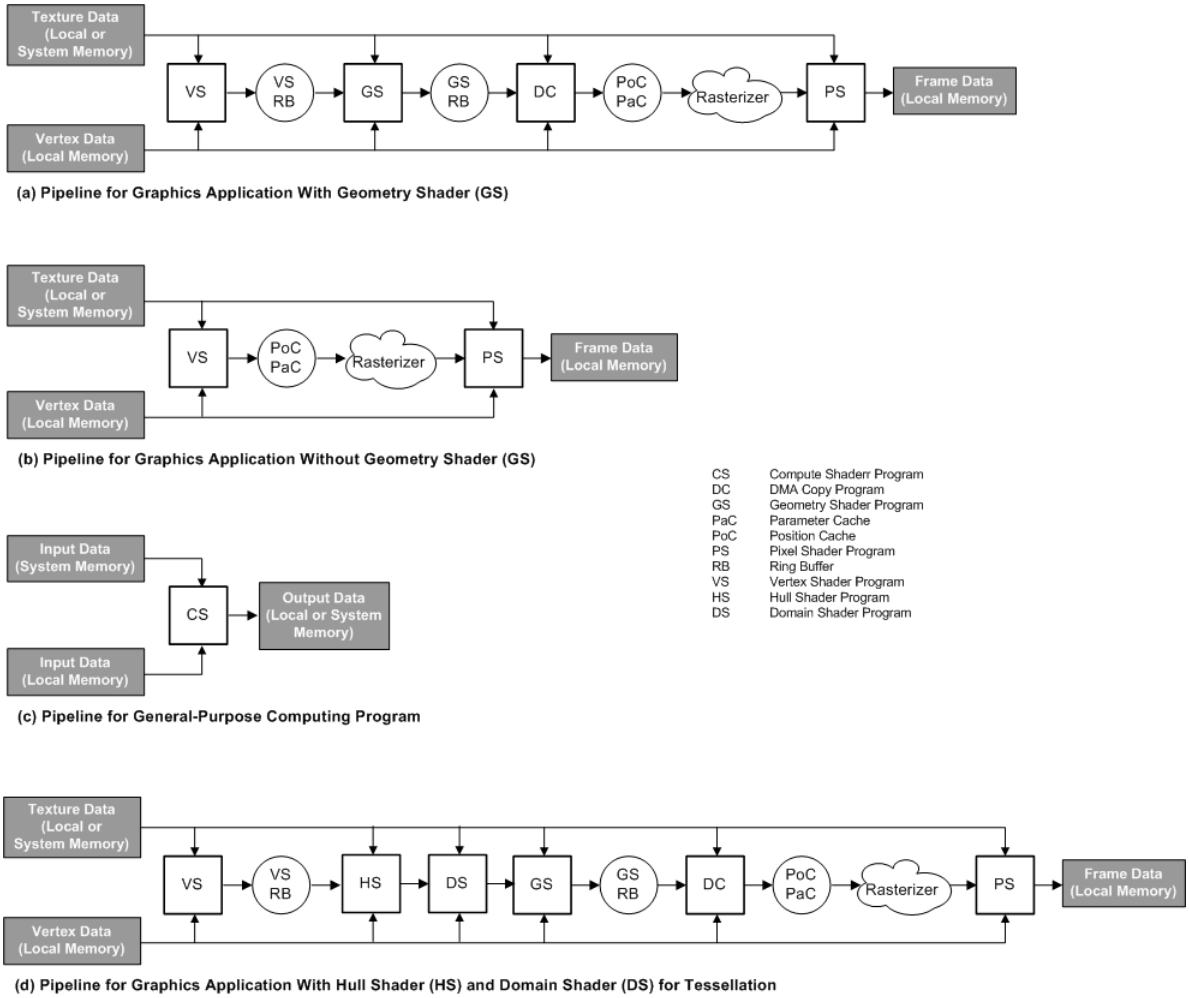


Figure 1.2 Programmer's View of Evergreen Dataflow

The dataflow sequence starts by reading 2D vertices, 2D textures, or other 2D data from local Evergreen memory or system memory; it ends by writing 2D pixels or other 2D data results to local Evergreen memory. The Evergreen processor hides memory latency by keeping track of potentially hundreds of work-items in different stages of execution, and by overlapping compute operations with memory-access operations.

Chapter 2
Program Organization and State

Evergreen programs consist of control-flow (CF) instructions, ALU instructions, instructions for fetches through a texture cache, and instructions for fetches through a vertex cache. ALU instructions can have up to three source operands and one destination operand. The instructions operate on 32-bit or 64-bit IEEE floating-point values and signed or unsigned integers. The execution of some instructions cause predicate bits to be written that affect subsequent instructions. Graphics programs typically use instructions for fetching through a vertex cache or through a texture cache for data loads, whereas general-computing applications typically use instructions for fetching through a texture cache for data loads.

2.1 Program Types

The following program types are commonly run on the Evergreen GPU (see Figure 1.2, on page 1-4):

- *Vertex Shader (VS)*—Reads vertices, processes them. If the geometry shader (GS) is active, the VS outputs its results to export shader-geometry shader (ESGS) ring buffer. If the hull shader (HS) is active, the VS outputs its results to the LDS. If neither the GS nor the HS is active, the VS outputs its results to the parameter cache and position buffer. It does not introduce new primitives. A vertex shader can invoke a *Fetch Subroutine (FS)*, which is a special global program for fetching vertex data that is treated, for execution purposes, as part of the vertex program. The FS provides driver independence between the process of fetching data required by a VS, and the VS itself.
- *Geometry Shader (GS)*—Reads primitives from the VS ring buffer, and, for each input primitive, writes zero or more primitives as output to the GS ring buffer. This program type is optional; when active, it requires a DMA copy (DC) program to be active. The GS simultaneously reads up to six vertices from an off-chip memory buffer created by the VS; it outputs a variable number of primitives to a second memory buffer.
- *DMA Copy (DC)*—Transfers data from the GS ring buffer into the parameter cache and position buffer. It is required for systems running a geometry shader.
- *Pixel Shader (PS) or Fragment Shader*—This type of program:
  - receives pixel data from the rasterizer to be shaded.

  - processes sets of pixel quads (four pixel-data elements arranged in a 2-by-2 array in neighboring lanes of a SIMD), and
  - writes output to up to eight local-memory buffers, called multiple render targets (MRTs), each of which includes a frame buffer.

- *Compute Shader (CS)*—A generic program (compute kernel) that uses an input work-item ID as an index to perform:
  - gather reads on one or more sets of input data,
  - arithmetic computation, and
  - scatter writes to one or more set of output data to memory.

  Compute shaders can write to multiple (up to eight) surfaces, which can be a mix of multiple render targets (MRTs), unordered access views (UAVs), and flat address space.
- *Hull Shader (HS)*— Receives patch data and processes it to generate new patch data along with some constant data and tesselation factors.
- Domain Shader (DS)— Fetches HS output and constant data to compute the vertex value based on U,V data from the tessellation engine. The tessellation engine generates U,V coordinates based on tessellation factors computed by the HS.

All program types accept the same instruction types, and all of the program types can run on any of the available DPP-array pipelines that support these programs; however, each kernel type has certain restrictions, which are described with that type.

2.1.1 Data Flows

The host can initialize the Evergreen GPU to run in multiple configurations. The compute shader is independent of other shaders. Pipeline configurations depend on whether tesselation is used and if the geometry shader is being used. Figure 1.2, on page 1-4 illustrates the processing order. Each type of flow is described in the following subsections. Table 2.1 shows the legal combinations for HS, LS, and DS.

Table 2.1 Data Flow When Different Shaders Stages are En/Disabled

VS	HS	DS	GS	Hardware Data Flow
on	on	on	on	Compute block -> tessellation block -> GS block LS -> HS -> TS -> ES -> GS -> VS -> PS
on	on	on	off	Compute block -> tesselation block ->\ LS -> HS -> TE -> VS -> PS In case of streamout, tessellation engine (TE) must expand the primitive to list the primitive type.
on	off	off	on	VS is treated as ES. ES -> GS -> VS -> PS
on	off	off	off	VS -> PS

2.1.2 Geometry Program Absent

Table 2.2 shows the order in which programs run when a geometry program is absent.

Table 2.2 Order of Program Execution (Geometry Program Absent)

Mnemonic	Program Type	Operates On	Inputs Come From	Outputs Go To
VS	Vertex Shader	Vertices	Vertex memory.	Parameter cache and position buffer.
PS	Pixel Shader	Pixels	Positions cache, parameter cache, and vertex geometry translator (VGT).	Local or system memory.

This processing configuration consists of the following steps.

1. The VS program sends a pointer to a buffer in device memory containing up to 64 vertex indices.
2. The Evergreen hardware groups the vectors for these vertices in its input buffers (remote memory).
3. When all vertices are ready to be processed, the Evergreen GPU allocates GPRs and work-item space for the processing of each of the 64 vertices, based on compiler-provided sizes.
4. The VS program calls the fetch subroutine (FS) program, which fetches vertex data into GPRs and returns control to the VS program.
5. The transform, lighting, and other parts of the VS program run.
6. The GPU allocates space in the position buffer and exports the vertex positions (XYZW).
7. The GPU allocates parameter-cache and position-buffer space and exports parameters and positions for each vertex.
8. The VS program exits, and the Evergreen GPU deallocates its GPR space.
9. When the VS program completes, the pixel shader (PS) program begins.
10. The Evergreen hardware assembles primitives from data in the position buffer and the vertex geometry translator (VGT), performs scan conversion and final pixel interpolation, and loads these values into GPRs.
11. The PS program then runs for each pixel.
12. The program exports data to a frame buffer, and the Evergreen GPU deallocates its GPR space.

2.1.3 Geometry Shader Present

Table 2.3 shows the order in which programs run when a geometry program is present.

Table 2.3 Order of Program Execution (Geometry Program Present)

Mnemonic	Program Type	Operates On	Inputs Come From	Outputs Go To
VS	Vertex Shader	Vertices	Vertex memory.	VS ring buffer.
GS	Geometry Shader	Primitives	VS ring buffer.	GS ring buffer.
DC	DMA Copy	Any Data	GS ring buffer.	Parameter cache or position buffer.
PS	Pixel Shader	Pixels	Positions cache, parameter cache, and vertex geometry translator (VGT).	Local or system memory.

This processing configuration consists of the following steps.

1. The Evergreen hardware loads input indices or primitive and vertex IDs from the vertex geometry translator (VGT) into GPRs.
2. The VS program fetches the vertex or vertices needed
3. The transform, lighting, and other parts of the VS program run.
4. The VS program ends by writing vertices out to the VS ring buffer.
5. The GS program reads multiple vertices from the VS ring buffer, executes its geometry functions, and outputs one or more vertices per input vertex to the GS ring buffer. The VS program can only write a single vertex per single input; the GS program can write a large number of vertices per single input. Every time a GS program outputs a vertex, it indicates to the vertex VGT that a new vertex has been output (using `EMIT_*` instructions[1]). The VGT counts the total number of vertices created by each GS program. The GS program divides primitive strips by issuing `CUT_VERTEX` instructions.
6. The GS program ends when all vertices have been output. No position or parameters is exported.
7. The DC program reads the vertex data from the GS ring buffer and transfers this data to the parameter cache and position buffer using one of the `MEM*` memory export instructions.
8. The DC program exits, and the Evergreen GPU deallocates the GPR space.
9. The PS program runs.
10. The Evergreen GPU assembles primitives from data in the position buffer, parameter cache, and VGT.
11. The hardware performs scan conversion and final pixel interpolation, and hardware loads these values into GPRs.

1. An asterisk (*) after a mnemonic string indicates that there are additional characters in the string that define variants.

12. The PS program runs.
13. When the PS program reaches the end of the data, it exports the data to a frame buffer or other render target (up to eight) using EXPORT instructions.
14. The program exits upon execution of an EXPORT_DONE instruction, and the processor deallocates GPR space.

2.1.4 Tessellation Without Geometry Shader

Table 2.4 shows the order in which programs run when a geometry program is absent.

Table 2.4 Order of Program Execution (Geometry Program Absent)

Mnemonic	Program Type	Operates On	Inputs Come From	Outputs Go To
VS	Vertex Shader	Vertices	Vertex memory.	Local data share (LDS).
HS	Hull Shader	Control points	LDS.	Tessellation factor buffer and LDS.
DS	Domain Shader	Patches	LDS.	Parameter cache and position buffer.
PS	Pixel Shader	Pixels	Positions cache, parameter cache, and vertex geometry translator (VGT).	Local or system memory.

This processing configuration consists of the following steps.

1. The VS program sends a pointer to a buffer in device memory containing up to 64 vertex indices.
2. The Evergreen hardware groups the vectors for these vertices in its input buffers (remote memory).
3. When all vertices are ready to be processed, the Evergreen GPU allocates GPRs and work-item space for the processing of each of the 64 vertices, based on compiler-provided sizes.
4. The VS program calls the fetch subroutine (FS) program, which fetches vertex data into GPRs and returns control to the VS program.
5. The transform, lighting, and other parts of the VS program run.
6. The HS takes input from the LDS and computes the new patch data and tessellation factors, which are output to the LDS.
7. The DS program allocates space in the position buffer and exports the vertex positions (XYZW).
8. The DS program allocates parameter-cache and position-buffer space and exports parameters and positions for each vertex.
9. The DS program exits, and the Evergreen GPU deallocates its GPR space.
10. When the DS program completes, the pixel shader (PS) program begins.
11. The Evergreen hardware assembles primitives from data in the position buffer and the vertex geometry translator (VGT), performs scan conversion and final pixel interpolation, and loads these values into GPRs.

12. The PS program then runs for each pixel.
13. The program exports data to a frame buffer, and the Evergreen GPU deallocates its GPR space.

2.1.5 Tessellation With Geometry Shader

Table 2.5 shows the order in which programs run when a geometry program is present.

Table 2.5 Order of Program Execution (Geometry Program Present)

Mnemonic	Program Type	Operates On	Inputs Come From	Outputs Go To
VS	Vertex Shader	Vertices	Vertex memory.	Local data share (LDS).
HS	Hull Shader	Control points	Local data share.	Tessellation factor buffer and LDS.
DS	Domain Shader	Patches	LDS.	ESGS ring buffer.
GS	Geometry Shader	Primitives	ESGS ring buffer.	GSVS ring buffer.
DC	DMA Copy	Any Data	GSVS ring buffer.	Parameter cache and position buffer.
PS	Pixel Shader	Pixels	Positions cache, parameter cache, and vertex geometry translator (VGT).	Local or system memory.

This processing configuration consists of the following steps.

1. The Evergreen hardware loads input indices or primitive and vertex IDs from the vertex geometry translator (VGT) into GPRs.
2. The VS program fetches the vertex or vertices needed
3. The transform, lighting, and other parts of the VS program run.
4. The VS program ends by writing vertices out to the VS ring buffer.
5. The HS takes input from the LDS and computes the new patch data and tessellation factors, which are output to the LDS.
6. The DS reads the address output data from the LDS, computes the vertex value based on U,V coordinates applied by the tessellation engine, and writes the vertex data output to the ESGS ring buffer.
7. The GS program reads multiple vertices from the VS ring buffer, executes its geometry functions, and outputs one or more vertices per input vertex to the GSVS ring buffer. The VS program can only write a single vertex per single input; the GS program can write a large number of vertices per single input. Every time a GS program outputs a vertex, it indicates to the vertex VGT that a new vertex has been output (using `EMIT_*` instructions[1]). The VGT counts the total number of vertices created by each GS program. The GS program divides primitive strips by issuing `CUT_VERTEX` instructions.

1. An asterisk (*) after a mnemonic string indicates that there are additional characters in the string that define variants.

8. The GS program ends when all vertices have been output. No position or parameters is exported.
9. The DC program reads the vertex data from the GSVS ring buffer and transfers this data to the parameter cache and position buffer using one of the MEM* memory export instructions.
10. The DC program exits, and the Evergreen GPU deallocates the GPR space.
11. The PS program runs.
12. The Evergreen GPU assembles primitives from data in the position buffer, parameter cache, and VGT.
13. The hardware performs scan conversion and final pixel interpolation, and hardware loads these values into GPRs.
14. The PS program runs.
15. When the PS program reaches the end of the data, it exports the data to a frame buffer or other render target (up to eight) using EXPORT instructions.
16. The program exits upon execution of an EXPORT_DONE instruction, and the processor deallocates GPR space.

2.2 Instruction Terminology

Table 2.6 summarizes some of the instruction-related terms used in this document. The instructions themselves are described in the remaining chapters.

Details on each instruction are given in Chapter 9. The register types are described in “Registers,” on page xii.

Table 2.6 Basic Instruction-Related Terms

Term	Size (bits)	Description
Microcode format	32	One of several encoding formats for all instructions. They are described in Section 3.1, “CF Microcode Encoding,” page 3-2, Section 4.1, “ALU Microcode Formats,” page 4-1, Section 6.1, “Microcode Formats for Fetches Through a Texture Cache Clause,” page 6-1, Section 5.1, “Microcode Formats for Fetches Through a Vertex Cache Clause,” page 5-2, and Chapter 10, “Microcode Formats.”
Instruction	64 or 128	Two to four microcode formats that specify: • Control flow (CF) instructions (64 bits). These include: general control flow instructions (such as branches and loops), instructions that allocate buffer space and export data, and instructions that initiate the execution of ALU, fetching through a texture cache, or fetching through a vertex cache clauses. • ALU instructions (64 bits). • Instructions for fetching through a texture cache clause (128 bits). • Instructions for fetching through a vertex cache clause (128 bits). • Data share instructions (128 bits). • Memory read instructions (128 bits). Instructions are identified in microcode formats by the _INST_ string in their field names and mnemonics. The functions of the instructions are described in Chapter 9, “Instruction Set.”
ALU Instruction Group	64 to 448	Variable-sized groups of instructions and constants that consist of: • One to five 64-bit ALU instructions. • Zero to two 64-bit literal constants. ALU instruction groups are described in Section 4.3, “ALU Instruction Slots and Instruction Groups,” page 4-3.
Literal Constant	64	Literal constants specify two 32-bit values, which can represent values associated with two elements of a 128-bit vector. These constants optionally can be included in ALU instruction groups. Literal constants are described in Section 4.3, “ALU Instruction Slots and Instruction Groups,” page 4-3.
Slot	64	An ordered position within an ALU instruction group. Each ALU instruction group has one to seven slots, corresponding to the number of ALU instructions and literal constants in the instruction group. Slots are described in Section 4.3, “ALU Instruction Slots and Instruction Groups,” page 4-3.
Clause	64 to 64x128 bits (64 128-bit words)	A set of instructions of the same type. The types of clauses are: • ALU clauses (which contain ALU instruction groups). • Clauses for fetching through a texture cache. • Clauses for fetching through a vertex cache. Clauses are initiated by control flow (CF) instructions and are described in Section 2.3, “Control Flow and Clauses,” page 2-9, and Section 3.3, “Clause-Initiation Instructions,” page 3-5.
Export	n/a	To do any of the following: • Write data from GPRs to an output buffer (a “scratch buffer,” “frame buffer,” “ring buffer,” or “stream buffer”). • Write an address for data inputs to the memory controller. • Read data from an input buffer (a “scratch buffer” or “ring buffer”) to GPRs.

Table 2.6 Basic Instruction-Related Terms (Cont.)

Term	Size (bits)	Description
Fetch	n/a	Load data, using a fetch through a vertex cache or fetch through a texture cache instruction clause. Loads are not necessarily to general-purpose registers (GPRs); specific types of loads may be confined to specific types of storage destinations.
Vertex	n/a	A heterogeneous record of data describing a vertex.
Quad	n/a	Four related pixels (for general-purpose programming: [x,y] data elements) in an aligned 2x2 space.
Primitive	n/a	A point, line segment, or polygon before rasterization. It has vertices specified by geometric coordinates. Additional data can be associated with vertices by means of linear interpolation across the primitive.
Fragment	n/a	For graphics programming: • The result of rasterizing a primitive. A fragment has no vertices; instead, it is represented by (x,y) coordinates. For general-purpose programming: • A set of (x,y) data elements.
Pixel	n/a	For graphics programming: • The result of placing a fragment in an (x,y) frame buffer. For general-purpose programming: • A set of (x,y) data elements.

2.3 Control Flow and Clauses

Each program consists of two sections:

- *Control Flow*—Control flow instructions can:
  - Initiate execution of ALU, instructions for fetching through a texture or through a vertex cache clause.
  - Export data to a buffer.
  - Control branching, looping, and stack operations.
- *Clause*—A homogeneous group of instructions; each clause comprises ALU, fetch through a texture cache clause, fetch through a vertex cache clause, global data share, local data share, or memory read instructions exclusively. A control flow instruction that initiates an ALU, a fetch through a texture cache clause, or fetch through a vertex cache clause does so by referring to an appropriate clause.

Table 2.7 provides a typical program flow example.

Table 2.7 Flow of a Typical Program

Function	Microcode Formats[1]	
	Control Flow (CF) Code	Clause Code
Start loop.	CF_DWORD[0,1]	
Initiate a fetch through a texture cache clause.	CF_DWORD[0,1]	
Fetch through a texture cache or vertex cache clause to load data from memory to GPRs.		TEX_DWORD[0,1,2]
Initiate ALU clause.	CF_ALU_DWORD[0,1]	
ALU clause to compute on loaded data and literal constants. This example shows a single clause consisting of a single ALU *instruction group* containing five ALU instructions (two quadwords each) and two quadwords of literal constants.		ALU_DWORD[0,1] ALU_DWORD[0,1] ALU_DWORD[0,1] ALU_DWORD[0,1] ALU_DWORD[0,1] LAST bit set Literal[X,Y] Literal[Z,W]
End loop.	CF_DWORD[0,1]	
Allocate space in an output buffer.	CF_ALLOC_EXPORT_DWORD0 CF_ALLOC_EXPORT_DWORD1_BUF	
Export (write) results from GPRs to output buffer.	CF_ALLOC_EXPORT_DWORD0 CF_ALLOC_EXPORT_DWORD1_BUF	

1. See Chapters 3 through 8 for more information on the microcode format and definitions.

Control flow instructions:

- constitute the main program. Jump statements, loops, and subroutine calls are expressed directly in the control flow part of the program.
- include mechanisms to synchronize operations.
- wait for a clause to complete.
- are required for buffer allocation in, and writing to, a program block's output buffer.

Some program types (VS, GS, DC, PS, LS, HS, CS) have specific control flow instructions for synchronizing with other blocks.

Each clause, invoked by a control flow instruction, is a sequential list of instructions of limited length (for the maximum length, see sections on individual clauses). Clauses contain no flow control statements, but ALU clause instructions can apply a predicate on a per-instruction basis. Instructions within a single clause execute serially. Multiple clauses of a program can execute in parallel if they contain instructions of different types and the clauses are independent of one another. (Such parallel execution is invisible to the programmer except for increased performance.)

ALU clauses contain instructions for performing operations in each of the five ALUs (ALU.[X,Y,Z,W] and ALU.Trans) including setting and using predicates, and pixel kill operations (see Section 4.8.1, “Instructions for All ALU Units,” page 4-19). Fetches through texture cache clauses contain instructions for performing texture and constant-fetch reads from memory. Fetches through vertex cache clauses are devoted to obtaining vertex data from memory. Systems without a vertex cache perform all fetches through texture cache.

A predicate is a bit that is set or cleared as the result of evaluating some condition; subsequently, it is used either to mask writing an ALU result or as a condition itself. There are two kinds of predicates, both of which are set in an ALU clause.

- The first is a single predicate local to the ALU clause itself. Once computed, the predicate can be referred to in a subsequent instruction to conditionally write an ALU result to the indicated general-purpose register(s).
- The second type is a bit in a predicate stack. An ALU clause computes the predicate bits in the stack and manipulates the stack. A predicate bit in the stack can be referred to in a control-flow instruction to induce conditional branching.

2.4 Instruction Types and Grouping

The Evergreen family of devices recognizes the following instruction types:

- control flow instructions
- clause types: ALU, fetch through texture cache, fetch through vertex cache, global data share. Memory read clauses are done in texture cache or vertex cache clauses.

There are separate instruction caches in the processor for each instruction type.

A CF program has a maximum size of 2^{28} bytes; the maximum size of each clause, however, is 128 slots for ALU clauses, and 16 instructions for TC and global data share (GDS) clauses. When a program is organized in memory, the instructions must be ordered as follows:

- All CF instructions.
- All ALU clauses.
- All fetches through texture cache or vertex cache clauses.
- All local data share clauses.
- All global data share clauses.
- All memory read clauses.
- Wavefront sync instructions.

The CPU host configures the base address of each program type before executing a program.

2.5 Program State

Table 2.8 through Table 2.11 summarize a programmer's view of the Evergreen program state that is accessible by a single work-item in an Evergreen program. The tables do not include:

- states that are maintained exclusively by Evergreen hardware, such as the internal loop-control registers,
- states that are accessible only to host software, such as configuration registers, or
- the duplication of states for many execution work-items.

The column headings in Table 2.8 through Table 2.11 have the following meanings:

- *Access by Evergreen Software*—Readable (R), writable (W), or both (R/W) by software executing on the Evergreen processor.
- *Access by Host Software*—Readable, writable, or both by software executing on the host processor. The tables do not include state objects, such as Evergreen configuration registers, that accessible only to host software.
- *Number per Work-Item*—The maximum number of such state objects available to each work-item. In some cases, the maximum number is shared by all executing work-items.
- *Width*—The width, in bits, of the state object.

Table 2.8 Control-Flow State

State	Access by Evergreen S/W	Access by Host S/W	# per Work-Item	Width (bits)	Description
Integer Constant Register (I)	R	W	1	96 (3 x 32)	The loop-variable constant specified in the CF_CONST field of the CF_DWORD1 microcode format for the current LOOP* instruction.
Loop Index (aL)	R	No	1	13	A register that is initialized by LOOP* instructions and incremented by hardware on each iteration of a loop, based on values provided in the LOOP* instruction's CF_CONST field of the CF_DWORD1 microcode format. It can be used for relative addressing of GPRs by any clause. Loops can be nested, so the counter and index are stored in the stack. ALU instructions can read the current aL index value by specifying it in the INDEX_MODE field of the ALU_DWORD0 microcode format, or in the ELEM_LOOP field of CF_ALLOC_EXPORT_DWORD1_* microcode formats. The register is 13 bits wide, but some instructions use only the low 9 bits.
Stack	No	No	Chip-Specific	Chip-Specific	The hardware maintains a single, multi-entry stack for saving and restoring the state of nested loops, pixels (valid mask and active mask, predicates, and other execution details. The total number of stack entries is divided among all executing work-items.

Table 2.9 ALU State

State	Access by Evergreen S/W	Access by Host S/W	# per Work-Item	Width (bits)	Description
General-Purpose Registers (GPRs)	R/W	No	127 minus 2 times Clause-Temporary GPRs	128 (4 x 32 bit)	Each work-item has access to up to 127 GPRs, minus two times the number of Clause-Temporary GPRs. Four GPRs are reserved as Clause-Temporary GPRs that persist only for one ALU clause (and thus are not accessible to fetch and export units). GPRs can hold data in one of several formats: the ALU can work with 32-bit IEEE floats (S23E8 format with special values), 32-bit unsigned integers, and 32-bit signed integers.
Clause-Temporary GPRs	No	Yes	4	128 (4 x 32 bit)	GPRs containing clause-temporary variables. The number of clause-temporary GPRs used by each work-item reduces the total number of GPRs available to the work-item, as described immediately above.
SIMD-Global GPRs	R/W	No	Defined by driver	128 (4 x 32 bit)	Set of GPRs that is persistent across all work-items during the execution of the kernel. Can be used to pass data between work-items.
Address Register (AR)	W	No	1	36 (4 x 9 bit)	A register containing a four-element vector of indices that are written by MOVA instructions. Hardware reads this register. The indices are used for relative addressing of a constant file (called constant waterfalling). This state only persists for one ALU clause. When used for relative addressing, a specific vector element must be selected.
Constant Registers (CRs)	R	W	512	128 (4 x 32 bit)	Registers that contain constants. Each register is organized as four 32-bit elements of a vector. Software can use either the CRs or the off-chip *constant cache*, but not both. DirectX calls these the Floating-Point Constant (F) Registers.
Index Register	W	No	2	8	A register that can be used to index into texture buffer constants, samplers, constant buffers, and random access targets. Index registers can be written from ALU clauses.
Previous Vector (PV)	R	No	1	128 (4 x 32 bit)	Registers that contain the results of the previous ALU.[X,Y,Z,W] operations. This state only persists for one ALU clause.
Previous Scalar (PS)	R	No	1	32	A register that contains the results of the previous ALU.Trans operations. This state only persists for one ALU clause.

Table 2.9 ALU State (Cont.)

State	Access by Evergreen S/W	Access by Host S/W	# per Work-Item	Width (bits)	Description
Local Data Share (LDS)	R/W Up to 32 kB.	No			Per SIMD Engine shared memory that enables an order of magnitude lower latency sharing of data between work-items of a given work-group. The application must query the runtime for the size of the local shared memory.
Global Data Share (GDS)	R/W Up to 64 kB	No			Global shared memory that enables low-latency access between all the work-items of a kernel concurrently running on the SIMD engines. This memory also enables inter-work-item atomic operations and reductions.
Predicate Register	R/W	No	1	1	A register containing predicate bits. The bits are set or cleared by ALU instructions as the result of evaluating some condition; the bits are subsequently used either to mask writing an ALU result or as a condition itself. An ALU clause computes the predicate bits in this register. A predicate bit in this register can be referred to in a control-flow instruction to induce conditional branching. This state only persists for one ALU clause. Predicate registers must be 1 bit per work-item or pixel, or 64 bits wide. Valid mask and active mask width must be the same.
Pixel State	No	No	1	192 (64 x 2 bits)	State bits that reflect each pixel's active status as conditional instructions are executed. The state can be *Active*, *Inactive-branch*, *Inactive-continue*, or *Inactive-break*.
Valid Mask	No	No	1	64	A mask indicating which pixels have been killed by a pixel-kill operation. The mask is updated when a `CF_INST_KILL` instruction is executed.
Active Mask	W (indirect)	No	1	1 bit per pixel	A mask indicating which pixels are currently executing and which are not (1 = execute, 0 = skip). This can be updated by `PRED_SET*` ALU instructions[1], but the updates do not take effect until the end of the ALU clause. `CF_ALU` instructions can update this mask with the result of the last `PRED_SET*` instruction in the clause.

1. An asterisk (*) after a mnemonic string indicates that there are additional characters in the string that define variants.

Table 2.10 Fetch Through Vertex Cache Clause State

State	Access by Evergreen S/W	Access by Host S/W	# per Work-Item	Width (bits)	Description
Fetch Through Vertex Cache Clause Constants	R	W	128	84	These describe the buffer format, etc.

Table 2.11 Fetch Through Texture Cache Clause and Constant-Fetch State

State	Access by Evergreen S/W	Access by Host S/W	# per Work-Item	Width (bits)	Description
Texture Samplers	No	W	18	96	There are 18 samplers (16 for DirectX plus 2 spares) available for each of the VS, GS, PS program types, two of which are spares. A texture sampler constant is used to specify how a texture is to be accessed. It contains information such as filtering and clamping modes.
Texture Resources	No	W	160	160	There are 160 resources available for each of the VS, GS, PS program types, and 16 for FS program types.
Border Color	No	W	1	128 (4 x 32 bits)	This is stored in the texture pipeline, but is referenced in fetches through texture cache clause instructions.
Bicubic Weights	No	W	2	176	These define the weights, one horizontal and one vertical, for bicubic interpolation. The state is stored in the texture pipeline, but referenced in fetches through texture cache clause instructions.
Kernel Size for Cleartype Filtering	No	W	2	3	These define the kernel sizes, one horizontal and one vertical, for filtering with Microsoft's Cleartype™ subpixel rendering display technology. The state is stored in the texture pipeline, but referenced in fetches through texture cache clause instructions.

2.6 Data Sharing

The Evergreen family of Stream processors can share data between different work-items. Data sharing can significantly boost performance. Figure 2.1 shows the memory hierarchy that is available to each work-item.

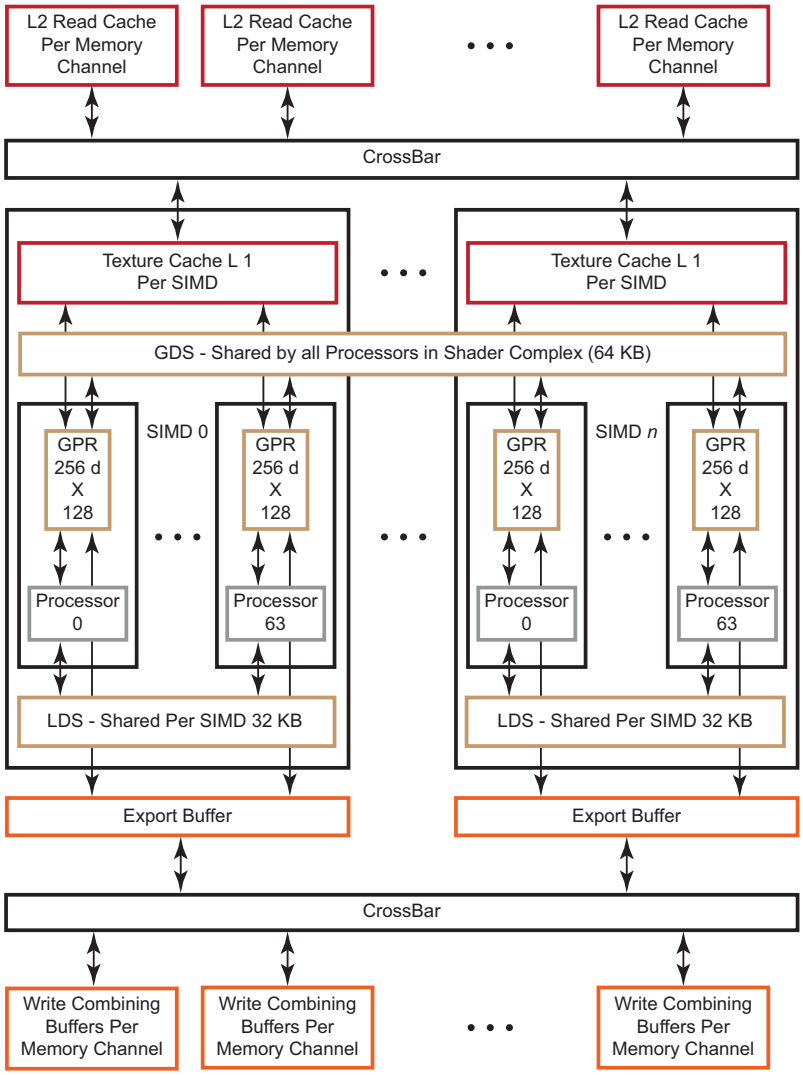


Figure 2.1 Shared Memory Hierarchy on the Evergreen Family of Stream Processors

2.6.1 Types of Shared Registers

There are two types of general-purpose registers: global shared and clause temporary.

2.6.1.1 Shared GPRs

Shared registers enable sharing of data between work-items residing in a lane of different wavefronts and that are scheduled to execute on a given SIMD. An absolute addressing mode of each source and destination operand allows sourcing data from a global (absolute-addressed) register instead of a wavefront's private (relative-addressed) registers. The maximum number shared register is 128 less two times the number of clause temp registers used. The registers put in this pool are removed from the general pool of wavefront private registers.

Each source and destination operand has an absolute addressing mode. This enables each to be accessed relative to address zero, instead of a base of the allocated pool of registers for the respective wavefront (see Figure 2.2). To use this pool, a state register must be set up defining the number of registers reserved for global usage.

The global GPRs are accessed through an index_mode (simd-global) in the ALU instruction word. This new mode interprets the src or dest GPR address as an absolute address in the range 0 to 127. This index mode works in conjunction with the src-rel/dest-rel fields, allowing the instruction to mix global and wavefront-local GPRs.

Additional index modes allow indexed addressing, where the address = GPR + offset_from_instruction or INDEX_GLOBAL_AR_X (AR.X only; see Section 4.6.1, “Relative Addressing,” page 4-6, as well as the opcode description for ALU_WORD0, page 10-23). This allows inter-work-item communication and kernel-based addressing. (This requires using a MOVA* instruction to copy the index to the AR.X register.)

This pool of global GPRs can be used to provide many powerful features, including:

- Atomic reduction variables per lane (the number depends on the number of GPRs), such as:
  - max, min, small histogram per lane,
  - software-based barriers or synchronization primitives.
- A set of constants that is unique per lane. This prevents:
  - the overhead of repeated fetches, and
  - divergent work-item execution due to constant look-up.

2.6.1.2 Clause Temporary GPRs

Clause temporary GPRs (clause temps) are a separate partition of the GPR pool that provide extra temporary registers to be used within an ALU clause, but their values are not preserved between clauses.

The GPR pool can include partitions that hold clause temporary (temp) GPRs. Clause temp GPRs prevent stalling and enable peak performance because they are stored in two sections, one for the odd, the other for the even wavefront (see Figure 2.2). Because there are two unique sections set aside for each wavefront executing on the SIMD, there is no conflict between reads and writes of clause temps between the even and odd wavefronts. When using global shared registers, both wavefronts map the registers into the same locations in memory, which can cause a conflict and a stall. This is because it takes a full instruction for the write to be visible; thus, if there are a read and a write happening on the same instruction group but from different wavefronts, there is a read/write conflict that the hardware resolves by stalling one of the wavefronts until the write is visible to the read.

The clause temp GPRs are accessed using the top GPR address locations. For example, if four clause temp register are enabled using 124, 125, 126, and 127, the address selects clause temp registers 0, 1, 2, and 3, respectively.

Clause temp registers can provide atomic (locked, uninterruptable) reduction per lane to enable higher performance between all work-items in a lane of a SIMD for the wavefronts that execute on the even or odd instruction slot.

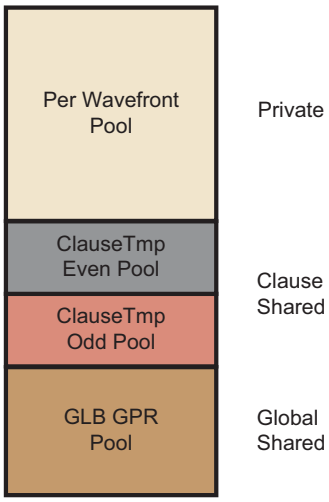


Figure 2.2 Possible GPR Distribution Between Global, Clause Temps, and Private Registers

Note that the terms even and odd refer to the ALU execution pipelines to which the scheduler arbitrarily assigns wavefronts. The first instruction slot to which a wavefront is assigned wavefront is termed odd.

Both global and clause temp shared registers require that the graphics pipeline (kernel hardware) must be flushed before changing resource allocation sizes (number of global registers, number of clause temp registers, etc.) for persistent shared use. They also require initialization prior to use. After any parallel atomic accumulation or reductions, the kernel pipeline must be flushed, followed by a special kernel that uses data sharing between lanes and/or SIMDS for a fast, on-chip final reduction. The result can be broadcast back to a global persistent register in each register file of each SIMD. The results can be used persistently across a subsequent kernel launch as a global `src` operand. This process can be very useful for a data collection pass on an image, followed by a reduction kernel, then followed by a compute kernel that uses the reduced values to alter the source image. This can be done without CPU intervention or off-chip traffic.

Physically, the GPRs are ordered from zero as: global, clause_temp, private. Note that this ordering allows a program to use the `MOV_INDEX_GLOBAL` instruction to access beyond the global registers into the clause temp registers. Global shared registers and clause temp registers must fit within the first 128 GPRs, due to ALU-instruction dest-GPR field-size limits.

SIMD-global GPRs are enabled only in the dynamic GPR mode.

2.6.2 Local Data Share (LDS)

Each SIMD has a 32 kB memory space that enables low-latency communication between work-items within a work-group, or the work-items within a wavefront; this is the local data share (LDS). This memory is configured with 32 banks, each with 256 entries of 4 bytes. The Evergreen family uses a 32 kB local data share (LDS) memory for each SIMD; this enables 128 kB of low-latency bandwidth to the processing elements. The Evergreen family of devices has full access to any LDS location for any processor. The shared memory contains 32 integer atomic units to enable fast, unordered atomic operations. This memory can be used as a software cache for predictable re-use of data, a data exchange machine for the work-items of a work-group, or as a cooperative way to enable more efficient access to off-chip memory.

2.7 Global Data Share (GDS)

The Evergreen family of devices uses a 64 kB global data share (GDS) memory that can be used by wavefronts of a kernel on all SIMDs. This memory enables 128 bytes of low-latency bandwidth to all the processing elements. The GDS is configured with 32 banks, each with 512 entries of 4 bytes each. It provides full access to any location for any processor. The shared memory contains 32 integer atomic units to enable fast, unordered atomic operations. This memory can be used as a software cache to store important control data for compute kernels, reduction operations, or a small global shared surface. Data can be preloaded from memory prior to kernel launch and written to memory after kernel completion. The GDS block contains support logic for unordered append/consume and domain launch ordered append/consume operations to buffers in memory. These dedicated circuits enable fast compaction of data or the creation of complex data structures in memory.

2.8 Device Memory

The Evergreen family of devices offers several methods for access to off-chip memory from each processing elements (PE) within each SIMD. On the primary read path, the device consists of multiple channels of L2 read-only cache that provides data to an L1 cache for each SIMD Engine. Special cache-less load instructions can force data to be retrieved from device memory during an execution of a load clause. Load requests that overlap within the clause are cached with respect to each other. The output cache is formed by two levels of cache: the first for write-combining cache (collect scatter and store operations and combine them to provide good access patterns to memory); the second is a read/write cache with atomic units that lets each processing element complete unordered atomic accesses that return the initial value. Each processing element provides the destination address on which the atomic operation acts, the data to be used in the atomic operation, and a return address for the read/write atomic unit to store the pre-op value in memory. Each store or atomic operation can be set up to return an acknowledgement to the requesting PE upon write

confirmation of the return value (pre-atomic op value at destination) being stored to device memory. This acknowledgement has two purposes:

- Enabling a PE to recover the pre-op value from an atomic operation by performing a cache-less load from its return address after receipt of the write confirmation acknowledgement
- Enabling the system to maintain a relaxed consistency model.

Each scatter write from a given PE to a given memory channel always maintains order. The acknowledgement enables one processing element to implement a fence to maintain serial consistency by ensuring all writes have been posted to memory prior to completing a subsequent write. In this manner, the system can maintain a relaxed consistency model between all parallel work-items operating on the system.

Chapter 3
Control Flow (CF) Programs

A control flow (CF) program is a main program. It directs the flow of program clauses by using control-flow instructions (conditional jumps, loops, and subroutines), and it can include memory-allocation instructions and other instructions that specify when vertex and geometry programs have completed their operations. The Evergreen hardware maintains a single, multi-entry stack for saving and restoring active masks, loop counters, and returning addresses for subroutines.

CF instructions can:

- Execute an ALU, a fetch through a texture cache clause, fetch through a vertex cache clause, or global data share clause. These operations take the address of the clause to execute, and a count indicating the size of the clause. A program can specify that a clause must wait until previously executed clauses complete, or that a clause must execute conditionally (only active pixels execute the clause, and the clause is skipped entirely if no pixels are active).
- Within an ALU clause, wavefronts within a work-group can synchronize with each other.
- Execute a DirectX9-style loop. There are two instructions marking the beginning and end of the loop. Each instruction takes the address of its paired LOOP_START and LOOP_END instructions. A loop reads from one of 32 constants to get the loop count, initial index value, and index increment value. Loops can be nested.
- Execute a DirectX10-style loop. There are two instructions marking the beginning and end of the loop. Each instruction takes an address of its paired LOOP_START and LOOP_END instructions. Loops can be nested.
- Execute a repeat loop (one that does not maintain a loop index). Repeat loops are implemented with the LOOP_START_NO_AL and LOOP_END instructions. These loops can be nested.
- Break out of the innermost loop. LOOP_BREAK instructions take an address to the corresponding LOOP_END instruction. LOOP_BREAK instructions can be conditional (executing only for pixels that satisfy a break condition).
- Continue a loop, starting with the next iteration of the innermost loop. LOOP_CONTINUE instructions take an address to the corresponding LOOP_END instruction. LOOP_CONTINUE instructions can be conditional.
- Execute a subroutine CALL or RETURN. A CALL takes a jump address. A RETURN never takes an address; it returns to the address at the top of the

stack. Calls can be conditional (only pixels satisfying a condition perform the instruction). Calls can be nested.

- Call the fetch subroutine (FS). The address field in a VC or TC control-flow instruction is unused; the address of the fetch through a vertex cache clause is global and written by the host. Thus, it makes no sense to nest these calls.
- Jump to a specified address in the control-flow program. A JUMP instruction can be conditional or unconditional.
- Perform manipulations on the current active mask for flow control (for example: executing an ELSE instruction, saving and restoring the active mask on the stack).
- Allocate data-storage space in a buffer and import (read) or export (write) addresses or data.
- Signal that the geometry shader (GS) has finished exporting a vertex, and optionally the end of a primitive strip.
- Synchronize with other wavefronts (global wave sync).

The end of the CF program is marked by setting the END_OF_PROGRAM bit in the last CF instruction in the program. The CF program terminates after the end of this instruction, regardless of whether the instruction is conditionally executed.

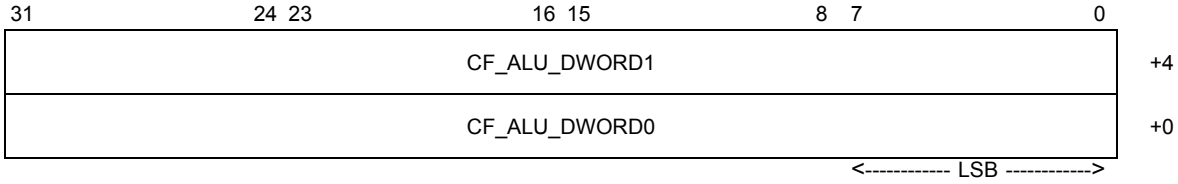
3.1 CF Microcode Encoding

The microcode formats and all of their fields are described in Chapter 10, “Microcode Formats.”. An overview of the encoding is given below. The following instruction-related terms are used throughout the remainder of this document:

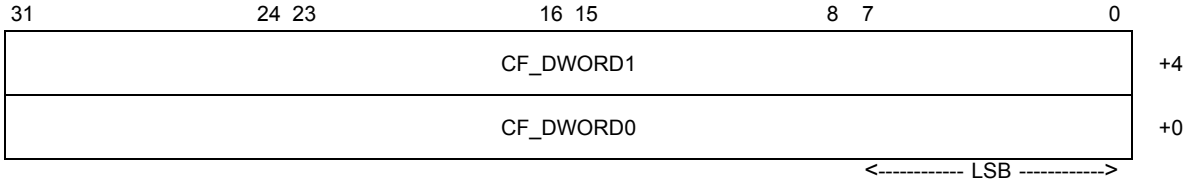
- *Microcode Format*—An encoding format whose fields specify instructions and associated parameters. Microcode formats are used in sets of two or four 32-bit doublewords (dwords). For example, the two mnemonics, CF_DWORD[0,1] indicate a microcode-format pair, CF_DWORD0 and CF_DWORD1, described in Section 10.1, “Control Flow (CF) Instructions,” page 10-2.
- *Instruction*—A computing function specified by the CF_INST field of a microcode format. For example, the mnemonic CF_INST_JUMP is an instruction specified by the CF_DWORD[0,1] microcode-format pair. All instructions have the _INST_ string in their mnemonic; for example, CF instructions have a CF_INST_ prefix. The instructions are listed in the Description columns of the microcode-format field tables in Chapter 10, “Microcode Formats”. In the remainder of this document, the CF_INST_ prefix is omitted when referring to instructions, except in passages for which the prefix adds clarity.
- *Opcode*—The numeric value of the CF_INST field of an instruction. For example, the opcode for the JUMP instruction is decimal 16 (0x10).
- *Parameter*—An address, index value, operand size, condition, or other attribute required by an instruction and specified as part of it. For example, CF_COND_ACTIVE (condition test passes for active pixels) is a field of the JUMP instruction.

The doubleword layouts in memory for CF microcode encodings are shown below, where +0 and +4 indicate the relative byte offset of the doublewords in memory, {BUF, SWIZ} indicates a choice between the strings BUF and SWIZ, and LSB indicates the least-significant (low-order) byte.

- CF microcode instructions that initiate ALU clauses use the following memory layout.



- All other CF microcode encodings use the following memory layout.



3.2 Summary of Fields in CF Microcode Formats

Table 3.1 summarizes the fields in various CF microcode formats and indicate which fields are used by the different instruction types. Each column represents a type of CF instruction. The fields in this table have the following meanings.

- *Yes*—The field is present in the microcode format and required by the instruction.
- *No*—The field is present in the microcode format but ignored by the instruction.
- *Blank*—The field is not present in the microcode format for that instruction.

For descriptions of the CF fields listed in Table 3.1, see Section 10.1, “Control Flow (CF) Instructions,” page 10-2.

Table 3.1 CF Microcode Field Summary

CF Microcode Field	CF Instruction Type					
	ALU[1]	Fetch Through Texture Cache Clause[2]	Fetch Through vertex Cache Clause[3]	Memory[4]	Branch or Loop[5]	Other[6]
CF_INST	Yes	Yes	Yes	Yes	Yes	Yes
ADDR	Yes	Yes	Yes		Note[7]	No
CF_CONST		No	No		Note[8]	Yes
POP_COUNT		No	No		Note[9]	No
COND		No	No		Yes	No
COUNT[10]	Yes	Yes	Yes		No	No
CALL_COUNT		No	No		Note	No
KCACHE_BANK[0,1]	Yes					
KCACHE_ADDR[0,1]	Yes					
KCACHE_MODE[0,1]	Yes					
VALID_PIXEL_MODE		Yes	Yes	Yes	Yes	Yes
WHOLE_QUAD_MODE	Yes	Yes	Yes	Yes	Yes	Yes
BARRIER	Yes	Yes	Yes	Yes	Yes	Yes
END_OF_PROGRAM		Yes	Yes	Yes	Yes	Yes
TYPE				Yes		
INDEX_GPR				Note[11]		
ELEM_SIZE				Yes		
ARRAY_BASE				Yes		
ARRAY_SIZE				Yes		
SEL_[X,Y,Z,W]						
COMP_MASK				Note[12]		
BURST_COUNT				Yes		
RW_GPR				Yes		
RW_REL				Yes		

1. CF ALU instructions contain the string CF_INST_ALU_.
2. CF fetch via texture cache instructions contain the string TC.
3. CF fetch via vertex cache instructions contain the string VC.
4. CF memory instructions contain the string CF_INST_MEM_.
5. CF branch or loop instructions include LOOP*, PUSH*, POP*, CALL*, RETURN*, JUMP, and ELSE.
6. CF other instructions include NOP, EMIT_VERTEX, EMIT_CUT_VERTEX, CUT_VERTEX, and KILL.
7. Some flow control instructions accept an address for another CF instruction.
8. Required if COND refers to the boolean constant, and for loop instructions that use DirectX9-style loop indexes.
9. Used by CF instructions that pop the stack. Not available to ALU clause instructions that pop the stack (see the ALU instructions for similar control).
10. COUNT has three uses: a) Call instructions use it as a ‘call-count.’ b) EMIT/EMITCUT/CUT uses it to mean ‘stream-id.’ c) ALU/TC/VC clauses use it to indicate clause length.
11. INDEX_GPR is used if the TYPE field indicates an indexed write.
12. COMP_MASK is used if the TYPE field indicates a write operation.

The following fields are available in most of the CF microcode formats.

- END_OF_PROGRAM — A program terminates after executing an instruction with the this bit set, even if the instruction is conditional and no pixels are active during the execution of the instruction. The stack must be empty when the program encounters this bit; otherwise, results are undefined when the program restarts on new data or a new program starts. Thus, instructions inside of loops or subroutines must not be marked with END_OF_PROGRAM.
- BARRIER — This expresses dependencies between instructions and allows parallel execution. If the this bit is set, all prior instructions complete before the current instruction begins. If this bit is cleared, the current instruction can co-issue with other instructions. Instructions of the same clause type never co-issue; however, instructions in a fetch through a texture cache clause and an ALU clause can co-issue if this bit is cleared. If in doubt, set this bit; results are identical whether it is set or not, but using it only when required can increase program performance.
- VALID_PIXEL_MODE — If set, instructions in the clause are executed as if invalid pixels were inactive. This field is the complement to the WHOLE_QUAD_MODE field. Set only WHOLE_QUAD_MODE or VALID_PIXEL_MODE at any one time.
- WHOLE_QUAD_MODE — If set, instructions in the clause are executed as if all pixels were active and valid. This field is the complement to the VALID_PIXEL_MODE field. Set only WHOLE_QUAD_MODE or VALID_PIXEL_MODE at any one time.

3.3 Clause-Initiation Instructions

Table 3.2 shows the clause-initiation instructions for the three types of clauses that can be used in a program. Every clause-initiation instruction contains in its microcode format an address field, ADDR (ignored for vertex clauses), that specifies the beginning of the clause in memory. ADDR specifies a quadword (64-bit) aligned address. Table 3.2 describes the alignment restrictions for clause-initiation instructions. ADDR is relative to the program base (configured in the PGM_START_* register by the host). There is also a COUNT field in the CF_DWORD1 microcode format that indicates the size of the clause. The interpretation of COUNT is specific to the type of clause being executed, as shown in Table 3.2. The actual value stored in the COUNT field is the number of slots or instructions to execute, minus one.

Table 3.2 Types of Clause-Initiation Instructions

Clause Type	CF Instructions	COUNT Meaning	COUNT Range	ADDR Alignment Restriction
ALU	ALU*[1]	Number of ALU slots[2]	[1, 128]	Varies (64-bit alignment is sufficient)
Fetch through Texture Cache	TC[3]	Number of instructions	[1, 16]	Double quadword (128-bit)
Fetch through Vertex Cache	VC*[4]	Number of instructions	[1, 16]	Double quadword (128-bit)

1. These instructions use the CF_ALU_DWORD[0,1] microcode formats, described in Section 10.1 on page 10-2.
2. See Section 4.3, “ALU Instruction Slots and Instruction Groups,” page 4-3, for a description of ALU slots.

3. These instructions use the CF_DWORD[0,1] microcode formats, described in Section 10.1, “Control Flow (CF) Instructions,” page 10-2.
4. These instructions use the CF_DWORD[0,1] microcode formats, described in Section 10.1, “Control Flow (CF) Instructions,” page 10-2.

3.3.1 ALU Clause Initiation

ALU* control-flow instructions[1] (such as ALU, ALU_BREAK, ALU_POP_AFTER, etc.) initiate an ALU clause. ALU clauses can contain OP2_INST_PRED_SET* instructions (abbreviated PRED_SET* instructions in this manual) that set new predicate bits for the processor’s control logic. The ALU control-flow instructions control how the predicates are applied for subsequent flow control.

ALU* control-flow instructions are encoded using the ALU_DWORD[0,1] microcode formats, described in Section 10.1, “Control Flow (CF) Instructions,” page 10-2. The ALU instructions within an ALU clause are described in Chapter 4, “ALU Clauses,” and Section 9.2, “ALU Instructions,” page 9-48.

ALU* control-flow instructions support locking up to four pages in the constant registers. The KCACHE_* fields control constant-cache locking for this ALU clause; the clause does not begin execution until all pages are locked, and the locks are held until the clause completes. There are two banks of 16 constants available for KCACHE locking; once locked, the constants are available within the ALU clause using special selects. See Section 4.6.4, “ALU Constants,” page 4-8, for more about ALU constants.

3.3.2 Vertex Cache Clause Initiation and Execution

The VC control-flow instruction initiates a clause that fetches data through the vertex cache, starting at the double-quadword-aligned *(*128-bit) offset in the ADDR field and containing COUNT + 1 instructions. The VC clause can contain vertex and memory fetch instructions, but not texture instructions.

The VC and TC control-flow instructions are encoded using the CF_DWORD[0,1] microcode formats, which are described in Section 10.1, “Control Flow (CF) Instructions,” page 10-2. The instructions for a fetch through a vertex cache clause are described in Chapter 5, “Fetch Through Vertex Cache Clauses,” and Section 9.3, “Instructions for Fetches Through a Vertex Cache Clause,” page 9-251.

3.3.3 Texture Cache Clause Initiation and Execution

The TC control-flow instruction initiates a fetch through a texture cache clause or a constant-fetch clause, starting at the double-quadword-aligned *(*128-bit) offset in the ADDR field and containing COUNT + 1 instructions. There is only one instruction for a fetch through a texture cache clause, and there are no special fields in the instruction for texture clause execution.

1. An asterisk (*) after a mnemonic string indicates that there are additional characters in the string that define variants.

The TC control-flow instruction is encoded using the CF_DWORD[0,1] microcode formats, which are described in Section 10.1, “Control Flow (CF) Instructions,” page 10-2. The instructions for a fetch through the texture cache clause are described in Chapter 6, “Texture Cache Clauses,” and Section 9.4, “Instructions for a Fetch Through a Texture Cache Clause,” page 9-254.

3.4 Import and Export Instructions

Importing means reading data from an input buffer (a scratch buffer, or ring buffer) to GPRs. Exporting means writing data from GPRs to an output buffer (a scratch buffer, ring buffer, or stream buffer), or writing an address for data inputs from a scratch buffer.

Exporting is done using the CF_ALLOC_EXPORT_DWORD0 and CF_ALLOC_EXPORT_DWORD1_{BUF, SWIZ} microcode formats. Two instructions, EXPORT and EXPORT_DONE, are used for normal pixel, position, and parameter-cache imports and exports. Importing is done using memory-read clauses (MEM*).

3.4.1 Normal Exports (Pixel, Position, Parameter Cache)

Most exports from a vertex shader (VS) and a pixel shader (PS) use the EXPORT and EXPORT_DONE instructions. The last export of a particular type (pixel, position, or parameter) uses the EXPORT_DONE instruction to signal hardware that the wavefront is finished with output for that type. These import and export instructions can use the CF_ALLOC_EXPORT_DWORD1_SWIZ microcode format, which provides optional swizzles for the outputs. These instructions can be used only by VS and PS threads; GS and DC threads must use one of the memory export instructions, MEM*.

Software indicates the type of export to perform by setting the TYPE field of the CF_ALLOC_EXPORT_DWORD0 microcode format equal to one of the following values:

- EXPORT_PIXEL — Pixel value output (from PS shaders). Send the output to the pixel cache.
- EXPORT_POS — Position output (from VS shaders). Send the output to the position buffer.
- EXPORT_PARAM — Parameter cache output (from VS shaders). Send the output to the parameter cache.

The RW_GPR and RW_REL fields indicate the GPR address (first_gpr) from which to read the first value or to which to write the first value (the GPR address can be relative to the loop index (aL). The value BURST_COUNT + 1 is the number of GPR outputs being written (the BURST_COUNT field stores the actual number minus one). The *N*th export value is read from GPR (first_gpr + N). The ARRAY_BASE field specifies the export destination of the first export and can take on one of the values shown in Table 3.3, depending on the TYPE field. The value increments by one for each successive export.

Table 3.3 Possible ARRAY_BASE Values

TYPE	ARRAY_BASE		Interpretation
	Field	Mnemonic	
EXPORT_PIXEL	7:0	CF_PIXEL_MRT[7,0]	Frame Buffer multiple render target (MRT), no fog.
	61	CF_PIXEL_Z	Computed Z.
EXPORT_POS	63:60	CF_POS_[3,0]	Position index of first export.
EXPORT_PARAM	31:0		Parameter index of first export.

Each memory write may be swizzled with the fields SEL_[X,Y,Z,W]. To disable writing an element, write SEL_[X,Y,Z,W] = SEL_MASK.

3.4.2 Memory Writes

All memory writes use one of the following instructions:

- MEM_SCRATCH — Scratch buffer.
- MEM_STREAM[0,3] — Stream buffer, for DirectX10 compliance, used by VS output for up to four streams.
- MEM_RING — Ring buffer, used for DC and GS output.
- MEM_EXPORT — Scatter writes.

These instructions always use the CF_ALLOC_EXPORT_DWORD1_BUF microcode format, which provides an array size for indexed operations and an element mask for writes (there is no element mask for reads from memory). No arbitrary swizzle is available; any swizzling must be done in an ALU clause. These instructions can be used by any program type.

There is one scratch buffer available for writes per program type (four scratch buffers in total). Stream buffers are available only to VS programs; ring buffers are available to GS, DC, and PS programs, and to VS programs when no GS and DC are present. Pixel-shader frame buffers use the ring buffer (MEM_RING).

The operation performed by these instructions is modified by the TYPE field, which can be one of the following:

- EXPORT_WRITE — Write to buffer.
- EXPORT_WRITE_IND — Write to buffer, using offset supplied by INDEX_GPR.

The RW_GPR and RW_REL fields indicate the GPR address (FIRST_GPR) to write the first value to (the GPR address can be relative to the loop register). The value (BURST_COUNT + 1) * (ELEM_SIZE + 1) is the number of doubleword outputs being written. The BURST_COUNT and ELEM_SIZE fields store the actual number minus one. ELEM_SIZE must be 3 (representing four doublewords) for scratch buffers, and ELEM_SIZE = 0 (doubleword) is intended for stream-out and ring buffers.

The memory address is based on the value in the ARRAY_BASE field (see Table 3.3, on page 3-8). If the TYPE field is set to EXPORT_*_IND (use_index == 1), the value contained in the register specified by the INDEX_GPR field, multiplied by (ELEM_SIZE + 1), is added to this base. The final equation for the first address in memory to write to (in doublewords) is:

```
first_mem = (ARRAY_BASE + use_index * GPR[INDEX_GPR]) * (ELEM_SIZE + 1)
```

The ARRAY_SIZE field specifies a point at which the burst is clamped; no memory is written past (ARRAY_BASE + ARRAY_SIZE) * (ELEM_SIZE + 1) doublewords. The exact units of ARRAY_BASE and ARRAY_SIZE differ depending on the memory type; for scratch buffers, both are in units of four doublewords (128 bits); for stream and ring buffers, both are in units of one doubleword (32 bits).

Indexed GPRs can stray out of bounds. If the index takes a GPR address out of bounds, then the rules specified for ALU GPR writes apply. See Section 4.6.3, “Out-of-Bounds Addresses,” page 4-7.

The Evergreen-family of GPUs supports a general memory export in which shader threads can write to arbitrary addresses within a specified memory range. This allows array-based and scatter access to memory. All threads share a common memory buffer, and there is no synchronization or ordering of writes between threads. A thread can read data that it has written and be guaranteed that previous writes from this thread have completed; however, a flush must take place before reading data from the memory-export area that another thread has written. Exports can only be written to a linear memory buffer (no tiling).

Each thread is responsible for determining the addresses it accesses.

The MEM_EXPORT instruction outputs data along with a unique dword address per pixel from a GPR, plus the global export-memory base address. Data is from one to four DWORDs.

3.4.3 Memory Reads

All memory reads use one of the following instructions:

- MEM_SCRATCH — Scratch buffer.
- MEM_EXPORT — Gather reads.

There is an element mask for reads from memory. Arbitrary swizzle is available. These instructions can be used by any program type.

There is one scratch buffer available for reads per program type (four scratch buffers in total).

The operation performed by these instructions is modified by the INDEXED field, which can be one of the following:

- INDEXED = 0 — Read from buffer.
- INDEXED = 1 — Read from buffer using offset supplied by SRC_GPR.

The DST_GPR and DST_REL fields indicate the GPR address (FIRST_GPR) to read the first value from (the GPR address can be relative to the loop register).

The memory address is based on the value in the ARRAY_BASE field (see Table 3.3, on page 3-8). If the INDEXED field is set, the value contained in the register specified by the SRC_GPR field, multiplied by (ELEM_SIZE + 1), is added to this base. The final equation for the first address in memory to read from (in doublewords) is:

```
first_mem = (ARRAY_BASE + use_index * GPR[INDEX_GPR]) * (ELEM_SIZE + 1)
```

The ARRAY_SIZE field specifies a point at which the burst is clamped; no memory is read past (ARRAY_BASE + ARRAY_SIZE) * (ELEM_SIZE + 1) doublewords. The exact units of ARRAY_BASE and ARRAY_SIZE differ depending on the memory type; for scratch buffers, both are in units of four doublewords (128 bits); for stream and ring buffers, both are in units of one doubleword (32 bits).

Indexed GPRs can stray out of bounds. If the index takes a GPR address out of bounds, then the rules specified for ALU GPR reads apply, except for a memory read in which the result is written to GPR0. See Section 4.6.3, “Out-of-Bounds Addresses,” page 4-7.

The Evergreen family supports a general memory export (read and write) in which shader threads can read from, and write to, arbitrary addresses within a specified memory range. This allows array-based and scatter access to memory. All threads share a common memory buffer, and there is no synchronization or ordering of writes between threads. A thread can read data that it has written and be guaranteed that previous writes from this thread have completed; however, a flush must take place before reading data from the memory-export area to which another thread has written.

Each thread is responsible for determining the addresses it accesses.

3.5 Synchronization with Other Blocks

Three instructions, EMIT_VERTEX, EMIT_CUT_VERTEX, and CUT_VERTEX, notify the processor’s primitive-handling blocks that new vertices are complete or primitives finished. These instructions typically follow the corresponding export operation that produces a new vertex:

- EMIT_VERTEX indicates that a vertex has been exported.
- EMIT_CUT_VERTEX indicates that a vertex has been exported and that the primitive has been cut after the vertex.
- CUT_VERTEX indicates that the primitive has been cut, but does not indicate a vertex has been exported by itself.

These instructions use the CF_DWORD[0,1] microcode formats and can be executed only by a GS program; they are invalid in other programs.

3.6 Conditional Execution

The remaining CF instructions include conditional execution and manipulation of the branch-loop states. The following subsections describes how conditional executions operate and describe the specific instructions.

3.6.1 Valid and Active Masks

Every element in the three bits that specify its state associated can be manipulated by a program.

- a one-bit *valid mask* and a 2-bit *per-pixel state*. The *valid mask* is set for any pixel that is covered by the original primitive and has not been killed by an `ALU KILL` operation.
- a two-bit *per-pixel state* that reflects the pixel's active status as conditional instructions are executed; it can take on the following states:
  - *Active*: The pixel is currently executing.
  - *Inactive-branch*: The pixel is inactive due to a branch (ALU `PRED_SET*`) instruction.
  - *Inactive-continue*: The pixel is inactive due to a `ALU_CONTINUE` instruction inside a loop.
  - *Inactive-break*: The pixel is inactive due to a `ALU_BREAK` instruction inside a loop.

Once the valid mask is cleared, it can not be restored. The per-pixel state can change during the lifetime of the program in response to conditional-execution instructions. Pixels that are invalid at the beginning of the program are put in one of the inactive states and do not normally execute (but they can be explicitly enabled, see below). Pixels that are killed during the program maintain their current active state (but they can be explicitly disabled, see below).

Branch-loop instructions can push the current pixel state onto the stack. This information is used to restore the pixel state when leaving a loop or conditional instruction block. CF instructions allow conditional execution in one of the following ways:

- Perform a *condition test* for each pixel based on current processor state:
  - The condition test determines which pixels execute the current instruction, and per-pixel state is unmodified, or
  - The per-pixel state is modified; pixels that pass the condition test are put into the active state, and pixels that fail the condition test are put into one of the inactive states, or
  - If at least one pixel passes, push the current per-pixel state onto the stack, then modify the per-pixel state based on the results of the test. If all pixels fail the test, jump to a new location. Some instructions can also pop the stack multiple times and change the per-pixel state to the result of the last pop; otherwise, the per-pixel state is left unmodified.

- Pop per-pixel state from the stack, replacing the current per-pixel state with the result of the last pop. Then, perform a *condition test* for each pixel based on the new state. Update the per-pixel state again based on the results of the test.

The condition test is computed on each pixel based on the current per-pixel state and, optionally, the valid mask. Instructions can execute in *whole quad mode* or *valid pixel mode*, which include the current valid mask in the condition test. This is controlled with the WHOLE_QUAD_MODE and VALID_PIXEL_MODE bits in the CF microcode formats, as described in the section immediately below. The condition test can also include the per-pixel state and a boolean constant, controlled by the COND field.

3.6.2 WHOLE_QUAD_MODE and VALID_PIXEL_MODE

A *quad* is a set of four pixels arranged in a 2-by-2 array, such as the pixels representing the four vertices of a quadrilateral. The *whole quad mode* accommodates instructions in which the result can be used by a gradient operation. Any instruction with the WHOLE_QUAD_MODE bit set begins execution as if all pixels were active. This takes effect before a condition specified in the COND field is applied (if available). For most CF instructions, it does not affect the active mask; inactive pixels return to their inactive state at the end of the instruction. Some branch-loop instructions that update the active mask reactivate pixels that were previously disabled by flow control or invalidation. These parameters assert whole quad mode for multiple CF instructions without setting the WHOLE_QUAD_MODE bit every time. Details for the relevant branch-loop instructions are described in Section 3.7, “Branch and Loop Instructions,” page 3-16. In general, instructions that can compute a value used in a gradient computation are executed in whole quad mode. All CF instructions support this mode.

In certain cases during whole quad mode, it can be useful to deactivate invalid pixels. This can occur in two cases:

- The program is in whole quad mode, computing a gradient. Related information not involved in the gradient calculation must be computed. As an optimization, the related information can be calculated without completely leaving whole quad mode by deactivating the invalid pixels.
- The ALU executes a KILL instruction. Killed pixels remain active because the processor does not know if the pixels are currently being used to compute a result that is used in a gradient calculation. If the recently invalidated pixels are not used in a gradient calculation, they can be deactivated.

Invalid pixels can be deactivated by entering *valid pixel mode*. Any instruction with the VALID_PIXEL_MODE bit set begins execution as if all invalid pixels were inactive. This takes effect before a condition specified in the COND field is applied (if available). For most CF instructions, it does not affect the active mask; however, as in whole quad mode, it influences the active mask for branch-loop instructions that update the active mask. These instructions can be used to permanently disable pixels that were recently activated. Valid pixel mode normally is not used to exit whole quad mode; whole quad mode normally is

exited automatically when reaching the end of scope for the branch-loop instruction that began in whole quad mode.

Instructions using the CF_DWORD[0,1] or the CF_ALLOC_EXPORT_DWORD[0,1] microcode formats have VALID_PIXEL_MODE fields. ALU clause instructions behave as if the VALID_PIXEL_MODE bit were cleared. Valid pixel mode is not the default mode; normal programs that do not contain gradient operations clear the VALID_PIXEL_MODE bit. The valid pixel mode is used only to deactivate pixels invalidated by a KILL instruction and to temporarily inhibit the effects of whole quad mode. Do not set both the WHOLE_QUAD_MODE bit and VALID_PIXEL_MODE bit.

Branch-loop instructions that pop from the stack interpret the valid pixel mode differently. If the mode is set on an instruction that pops the stack, invalid pixels are deactivated after the active mask is restored from the stack. This can make the effect of the valid pixel mode permanent for a killed pixel that is executed inside a conditional branch. By default, the per-pixel active state is overwritten with the stack contents on each pop, without regard for the current active state; however, when VALID_PIXEL_MODE is set, the invalid pixels are deactivated even though they were active going into the conditional scope.

3.6.3 The Condition (COND) Field

Instructions that use the CF_DWORD[0,1] microcode formats have a COND field that lets them be conditionally executed. The COND field can have one of the following values:

- CF_COND_ACTIVE — Pixel currently active. Non-branch-loop instructions can use only this setting.
- CF_COND_BOOL — Pixel currently active, and the boolean referenced by CF_CONST is one.
- CF_COND_NOT_BOOL — Pixel currently active, and the boolean referenced by CF_CONST is zero.

For most CF instructions, COND is used only to determine which pixels are executing that particular instruction; the result of the test is discarded after the instruction completes. Branch-loop instructions that manipulate the active state can use the result of the test to update the new active mask; these cases are described below. Non-branch-loop instructions can use only the CF_COND_ACTIVE setting. Generally, branch-loop instructions that push pixel state onto the stack push the original pixel state before beginning the instruction, and use the result of COND to write the new active state. Some instructions that pop from the stack can pop the stack first, then evaluate the condition code, and update the per-pixel state based on the result of the pop and the condition code.

Instructions that do not have a COND field behave as if CF_COND_ACTIVE were used. ALU clauses do not have a COND field; they execute pixels based on the current active mask. ALU clauses can update the active mask using PRED_SET* instructions, but changes to the active mask are not observed for the remainder of the ALU clause (however, the clause can use the predicate bits to observe the

effect). Changes to the active mask from the ALU take effect at the beginning of the next CF instruction.

3.6.4 Computation of Condition Tests

The COND, WHOLE_QUAD_MODE, and VALID_PIXEL_MODE fields combine to form the condition test results shown in Table 3.4.

Table 3.4 Condition Tests

COND	Default	WHOLE_QUAD_MODE	VALID_PIXEL_MODE
CF_COND_ACTIVE	True if and only if pixel is active.	True if and only if quad contains active pixel.	True if and only if pixel is both active and valid.
CF_COND_BOOL	True if and only if pixel is active and boolean referenced by CF_CONST is one.	True if quad contains active pixel and boolean referenced by CF_CONST is one.	True if and only if pixel is both active and valid, and boolean referenced by CF_CONST is one.
CF_COND_NOT_BOOL	True if and only if pixel is active and boolean referenced by CF_CONST is one.	True if quad contains active pixel and boolean referenced by CF_CONST is one.	True if and only if pixel is both active and valid, and boolean referenced by CF_CONST is one.

The following steps indicate how the per-pixel state can be updated during a CF instruction that does not unconditionally pop the stack:

1. Evaluate the condition test for each pixel using current state, COND, WHOLE_QUAD_MODE, and VALID_PIXEL_MODE.
2. Execute the CF instruction for pixels passing the condition test.
3. If the CF instruction is a PUSH, push the per-pixel active state onto the stack before updating the state.
4. If the CF instruction updates the per-pixel state, update the per-pixel state using the results of condition test.

ALU clauses that contain multiple PRED_SET* instructions can perform some of these operations more than once. Such clause instructions push the stack once per PRED_SET* operation.

The following steps loosely illustrate how the active mask (per-pixel state) can be updated during a CF instruction that pops the stack. These steps only apply to instructions that unconditionally pop the stack; instructions that can jump or pop if all pixels fail the condition test do not use these steps:

1. Pop the per-pixel state from the stack (can pop zero or more times). Change the per-pixel state to the result of the last POP.
2. Evaluate the condition test for each pixel using new state, COND, WHOLE_QUAD_MODE, and VALID_PIXEL_MODE.
3. Update the per-pixel state again using results of condition test.

3.6.5 Stack Allocation

Each program type has a stack for maintaining branch and other program states. The maximum number of available stack entries is controlled by a host-written register or by the hardware implementation of the processor. The minimum number of stack entries required to correctly execute a program is determined by the deepest control-flow instruction.

Each stack entry contains a number of subentries. The number of subentries per stack entry varies, based on the physical work-group width of the processor. If a processor that supports 64 thread groups per program type is configured logically to use only 48 thread groups per program type, the stack requirement for a 64-item processor still applies. Table 3.5 shows the number of subentries per stack entry, based on the physical thread-group width of the processor.

Table 3.5 Stack Subentries

	Physical Thread-Group Width of Processor			
	16	**32**	**48**	**64**
Subentries per Entry	8	8	4	4

The CALL*, LOOP_START*, and PUSH* instructions each consume a certain number of stack entries or subentries. These entries are released when the corresponding POP, LOOP_END, or RETURN instruction is executed. The additional stack space required by each of these flow-control instructions is described in Table 3.6.

Table 3.6 Stack Space Required for Flow-Control Instructions

Instruction	Stack Usage per Physical Thread-Group Width				Comments
	16	**32**	**48**	**64**	
PUSH, PUSH_ELSE when whole quad mode is not set, and ALU_PUSH_BEFORE	one subentry	one subentry	one subentry	one subentry	If a PUSH instruction is invoked, two subentries on the stack must be reserved to hold the current active (valid) masks.
PUSH, PUSH_ELSE when whole quad mode is set	one entry	one entry	one entry	one entry	
LOOP_START*	one entry	one entry	one entry	one entry	
CALL, CALL_FS	two subentries	one subentry	one subentry	one subentry	A 16-bit-wide processor needs two subentries because the program counter has more than 16 bits.

At any point during the execution of a program, if A is the total number of full entries in use, and B is the total number of subentries in use, then STACK_SIZE is calculated by:

```
A + B / (# of subentries per entry) <= STACK_SIZE
```

3.7 Branch and Loop Instructions

Several CF instructions handle conditional execution (branching), looping, and subroutine calls. These instructions use the CF_DWORD[0,1] microcode formats and are available to all thread types. The branch-loop instructions are listed in Table 3.7, along with a summary of their operations. The instructions listed in this table implicitly begin with CF_INST_.

Table 3.7 Branch-Loop Instructions

Instruction	Condition Test Computed	Push	Pop	Jump	Description
PUSH	Yes, before push.	Yes, if a pixel passes test.	Yes, if all pixels fail test.	Yes, if all pixels fail test.	If all pixels fail the condition test, pop POP_COUNT entries from the stack, and jump to the jump address; otherwise, push per-pixel state (active mask) onto stack. After the push, active pixels that failed the condition test transition to the inactive-branch state.
PUSH_ELSE	Yes, before push.	Yes, always.	No.	Yes, if all pixels fail test.	Push current per-pixel state (active mask) onto the stack, and compute new active mask. The instruction implement the ELSE part of a higher-level IF statement.
POP	Yes, before pop.	No.	Yes.	Yes	Pop POP_COUNT entries from the stack. Also, jump if condition test fails for all pixels.
LOOP_START LOOP_START_NO_AL LOOP_START_DX10	At beginning. All pixels fail if loop count is zero.	Yes, if a pixel passes test. Pushes loop state.	Yes, if all pixels fail test.	Yes, if all pixels fail test.	Begin a loop. Failing pixels go to inactive-break.
LOOP_END	At beginning. All pixels fail if loop count is one.	No.	Yes, if all pixels fail test. Pops loop state.	Yes, if any pixel *passes* test.	End a loop. Pixels that have not explicitly broken out of the loop are reactivated. Exits loop if all pixels fail condition test.
LOOP_CONTINUE	At beginning.	No.	Yes, if all pixels done with iteration.	Yes, if all pixels done with iteration.	Pixels passing test go to inactive-continue. In the event of a jump, the stack is popped back to the original level at the beginning of the loop; the POP_COUNT field is ignored.
LOOP_BREAK	At beginning.	No.	Yes, if all pixels done with iteration.	Yes, if all pixels done with iteration.	Pixels passing test go to inactive-break. In the event of a jump, the stack is popped back to the original level at the beginning of the loop; the POP_COUNT field is ignored.
JUMP	At beginning.	No.	Yes, if all pixels fail test.	Yes, if all pixels fail test.	Jump to ADDR if all pixels fail the condition test.

Table 3.7 Branch-Loop Instructions (Cont.)

Instruction	Condition Test Computed	Push	Pop	Jump	Description
ELSE	After last pop.	No.	Yes.	Yes, if all pixels are inactive after ELSE.	Pop the stack, then invert status of active or inactive-branch pixels that pass conditional test and were active on last PUSH.
CALL CALL_FS	After last pop.	Yes, if a pixel passes test. Pushes address.	Yes.	Yes, if any pixel *passes* test.	Call a subroutine if any pixel passes the condition test and the maximum call depth limit is not exceeded. POP_COUNT must be zero.
RETURN RETURN_FS	No.	No.	Yes. Pops address from stack if jump taken.	Yes, if all active pixels pass test.	Return from a subroutine.
ALU	No.	No.	No.	N/A	PRED_SET* with exec mask update puts active pixels in to the inactive-branch state.
ALU_PUSH_BEFORE	No.	Before ALU clause.	No.	N/A	Equivalent to PUSH; ALU clause.
ALU_POP_AFTER	No.	No.	Yes.	N/A	Equivalent to ALU, POP.
ALU_POP2_AFTER					Equivalent to ALU, POP, POP.
ALU_CONTINUE	No.	No.	No.	N/A	Change active pixels masked by ALU to inactive-continue. Equivalent to PUSH, ALU, ELSE, CONTINUE, POP.
ALU_BREAK	No.	No.	No.	N/A	Change active pixels masked by ALU to inactive-break. Equivalent to PUSH, ALU, ELSE, CONTINUE, POP.
ALU_ELSE_AFTER	No.	No.	Yes.	N/A	Equivalent to ALU; ELSE.

3.7.1 ADDR Field

The address specified in the ADDR field of a CF instruction is a quadword-aligned (64 bit) offset from the base of the program (host-specified PGM_START_* register). The execution continues from this offset. Branch-loop instructions typically implement conditional jumps, so execution continues either at the next CF instruction, or at the CF instruction located at the ADDR address.

3.7.2 Stack Operations and Jumps

Several stack operations are available in the CF instruction set: PUSH, POP, and ELSE. There also is a JUMP instruction that jumps if all pixels fail a condition test.

- PUSH - pushes the current per-pixel state from hardware-maintained registers onto the stack, then updates the per-pixel state based on the condition test. If all pixels fail the test, PUSH does not push anything onto the stack; instead,

it performs POP_COUNT number of pops (may be zero), then jumps to a specified address if all pixels fail the test.

- POP - pops per-pixel state from the stack to hardware-maintained registers; it pops the POP_COUNT number of entries (can be zero). POP can apply the condition test to the result of the POP, this is useful for disabling pixels that are killed within a conditional block. To disable such pixels, set the POP instruction's VALID_PIXEL_MODE bit, and set the condition to CF_COND_ACTIVE. If POP_COUNT is zero, the POP instruction simply modifies the current per-pixel state based on the result of the condition test. Pop instructions never jump.
- ELSE - performs a conceptual else operation. It starts by popping POP_COUNT entries (can be zero) from the stack. Then, it inverts the sense of active and branch-inactive pixels for pixels that are both active (as of the last surviving PUSH operation) and pass the condition test. The ELSE operation will then jump to the specified address if all pixels are inactive.
- JUMP - is used to jump over blocks of code that no pixel wants to execute. JUMP first pops POP_COUNT entries (may be zero) from the stack. It then applies the condition test to all pixels. If all pixels fail the test, it jumps to the specified address; otherwise, it continues execution on the next instruction.

3.7.3 DirectX9 Loops

DirectX9-style loops are implemented with the LOOP_START and LOOP_END instructions. Both instructions specify the DirectX9 integer constant using the CF_CONST microcode field. This field specifies the integer constant to use for the loop's trip count (maximum number of loops), beginning value (loop index initializer), and increment (step). The constant is a host-written vector, and the three loop parameters are stored as three elements of the vector. The COND field also can refer to the CF_CONST field for its boolean value. It is not be possible to conditionally enter a loop based on a boolean constant unless the boolean constant and integer constant have the same numerical address.

The LOOP_START instruction jumps to the address specified in the instruction's ADDR field if the initial loop count is zero. Software normally sets the ADDR field to the CF instruction following the matching LOOP_END instruction. If LOOP_START does not jump, hardware sets up the internal loop state. Loop-index-relative addressing (as specified by the INDEX_MODE field of the ALU_DWORD0 microcode format) is well-defined only within the loop. If multiple loops are nested, relative addressing refers to the loop register of the innermost loop. The loop register of the next-outer loop is automatically restored when the innermost loop exits.

The LOOP_END instruction jumps to the address specified in the instruction's ADDR field if the loop count is nonzero after it is decremented, and at least one pixel has not been deactivated by a LOOP_BREAK instruction. Normally, software sets the ADDR field to the CF instruction following the matching LOOP_START. The LOOP_END instruction continues to the next CF instruction when the processor exits the loop.

DirectX9-style break and continue instructions are supported. The LOOP_BREAK instruction disables all pixels for which the condition test is true. The pixels remain disabled until the innermost loop exits. LOOP_BREAK jumps to the end of the loop if all pixels have been disabled by this (or a prior) LOOP_BREAK or LOOP_CONTINUE instruction. Software normally sets the ADDR field to the address of the matching LOOP_END instruction. If at least one pixel has not been disabled by LOOP_BREAK or LOOP_CONTINUE, execution continues to the next CF instruction.

The LOOP_CONTINUE instruction disables all pixels for which the condition test is true. The pixels remain disabled until the end of the current iteration of the loop, and are re-activated by the innermost LOOP_END instruction. The LOOP_CONTINUE instruction jumps to the end of the loop if all pixels have been disabled by this (or a prior) LOOP_BREAK or LOOP_CONTINUE instruction. The ADDR field points to the address of the matching LOOP_END instruction. If at least one pixel has not been disabled by LOOP_BREAK or LOOP_CONTINUE, the program continues to the next CF instruction.

Each instruction can manipulate the stack. LOOP_START pushes the current per-pixel state and the prior loop state onto the stack. If LOOP_START does not enter the loop, it pops POP_COUNT entries (may be zero) from the stack, similar to the PUSH instruction when all pixels fail. The LOOP_END instruction evaluates the condition test at the beginning of the instruction. If all pixels fail the test, the instruction exits the loop. LOOP_END pops the loop state and one set of the per-pixel state from the stack when it exits the loop. It ignores POP_COUNT. The LOOP_BREAK and LOOP_CONTINUE instructions pop the POP_COUNT entries (may be zero) from the stack if the jump is taken.

3.7.4 DirectX10 Loops

DirectX10 loops are implemented with the LOOP_START_DX10 and LOOP_END instructions. The LOOP_START_DX10 instruction enters the loop by pushing the stack. The LOOP_END instruction jumps to the address specified in the ADDR field if at least one pixel has not yet executed a LOOP_BREAK instruction. The ADDR field points to the CF instruction following the matching LOOP_START_DX10 instruction. The LOOP_END instruction continues to the next CF instruction, at which the processor exits the loop. The LOOP_BREAK and LOOP_CONTINUE instructions are allowed in DirectX10-style loops.

Manipulations of the stack are the same for LOOP_{START_DX10,END} instructions and LOOP_{START,END} instructions.

3.7.5 Repeat Loops

Repeat loops are implemented with the LOOP_START_NO_AL and LOOP_END instructions. These loops do not push the loop index (aL) onto the stack, nor do they update aL; otherwise, they are identical to LOOP_{START,END} instructions.

3.7.6 Subroutines

The CALL and RETURN instructions implement subroutine calls and the corresponding returns. For CALL, the ADDR field specifies the address of the first CF instruction in the subroutine. The ADDR field is ignored by the RETURN instruction (the return address is read from the stack). Calls have a nesting depth associated with them that is incremented on each CALL instruction by the CALL_COUNT field. The nesting depth is restored on a RETURN instruction. If the program exceeds the maximum nesting depth (32) on the subroutine call (current nesting depth + CALL_COUNT > 32), the call is ignored. Setting CALL_COUNT to zero prevents the nesting depth from being updated on a subroutine call. Execution of a RETURN instruction when the program is not in a subroutine is illegal.

The CALL_FS instruction calls a fetch subroutine (FS) whose address is relative to the address specified in a host-configured register. The instruction also activates the fetch-program mode, which affects other operations until the corresponding RETURN instruction is reached. Only a vertex shader (VS) program can call an FS subroutine, as described in Section 2.1, “Program Types,” page 2-1.

The CALL and CALL_FS instructions can be conditional. The subroutine is skipped if and only if all pixels fail the condition test or the nesting depth exceeds 32 after the call. The POP_COUNT field typically is zero for CALL and CALL_FS.

3.7.7 ALU Branch-Loop Instructions

Several instructions execute ALU clauses:

- ALU
- ALU_PUSH_BEFORE
- ALU_POP_AFTER
- ALU_POP2_AFTER
- ALU_CONTINUE
- ALU_BREAK
- ALU_ELSE_AFTER

The ALU instruction performs no stack operations. It is the most common method of initiating an ALU clause. Each PRED_SET* operation in the ALU clause manipulates the per-pixel state directly, but no changes to the per-pixel state are visible until the clause completes execution.

The other ALU* instructions correspond to their CF-instruction counterparts. The ALU_PUSH_BEFORE instruction performs a PUSH operation before each PRED_SET* in the clause. The ALU_POP{,2}_AFTER instructions pop the stack (once or twice) at the end of the ALU clause. The ALU_ELSE_AFTER instruction pops the stack, then performs an ELSE operation at the end of the ALU clause. And the ALU_{CONTINUE,BREAK} instructions behave similarly to their CF-instruction

counterparts. The major limitation is that none of the ALU* instructions can jump to a new location in the CF program. They can only modify the per-pixel state and the stack.

3.8 Synchronizing Across Threadgroups (Global Wave Sync)

Each compute device (1 or 2 per GPU) contains 16 global wave sync (GWS) resources for implement barriers, semaphores, and other synchronization primitives. GWS resources can be shared by multiple wavefronts running on different SIMDs and on different compute devices (if multiple devices are present). This makes them more powerful than threadgroup barriers, which allow only for basic barrier-style synchronization between a set of wavefronts running on an individual SIMD. The state of each resource is described by an integer value that can be read and updated by each wavefront on the GPU. A set of GWS instructions is provided that describe for each resource specified as part of the instruction:

- the initial integer value of that resource,
- how the value of that resource is altered upon execution of the instruction (increment, decrement, no change), and
- for which resource values the execution of the instruction is stalled until another wavefront has altered the resource value.

Chapter 4
ALU Clauses

Software initiates an ALU clause with one of the CF_INST_ALU* control-flow instructions, all of which use the CF_ALU_DWORD[0,1] microcode formats. Instructions within an ALU clause, called *ALU instructions*, perform operations using the scalar ALU.[X,Y,Z,W] and ALU.Trans units, which are described in this chapter.

NOTE: For the 54xx and 55xx AMD GPU series only, the CF_INST_ALU* instructions do not save the active mask correctly. The branching can be wrong, possibly producing incorrect results and infinite loops. The three possible work-arounds are:

a. Avoid using the CF_ALU_PUSH_BEFORE, CF_ALU_ELSE_AFTER CF_ALU_BREAK, and CF_ALU_CONTINUE instructions.

b. Do not use the CF_INST_ALU* instructions when your stack depth exceeds three elements (not entries); for the 54XX series AMD GPUs, do not exceed a stack size of seven, since this GPU series has a vector size 32.

c. Do not use these instructions when your non-zero stack depth mod 4 is 0 (or mod 8 is 0, for vector size 32).

4.1 ALU Microcode Formats

ALU instructions are implemented with ALU microcode formats that are organized in pairs of two 32-bit doublewords. The doubleword layouts in memory are shown in Figure 4.1.

- +0 and +4 indicate the relative byte offset of the doublewords in memory.
- {OP2, OP3} indicates a choice between the strings OP2 and OP3 (which specify two or three source operands).
- LSB indicates the least-significant (low-order) byte.

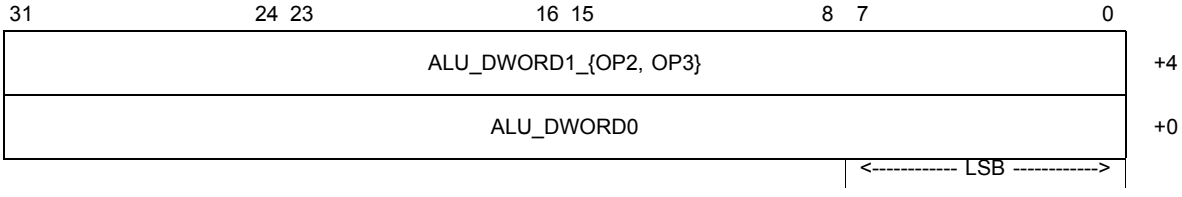


Figure 4.1 ALU Microcode Format Pair

4.2 Overview of ALU Features

An ALU *vector* is 128 bits wide and consists of four 32-bit elements. The data elements need not be related. The elements are organized in GPRs in little-endian order, as shown in Figure 4.2. Element ALU.X is the least-significant (low-order) element; element ALU.W is the most-significant (high-order) element.

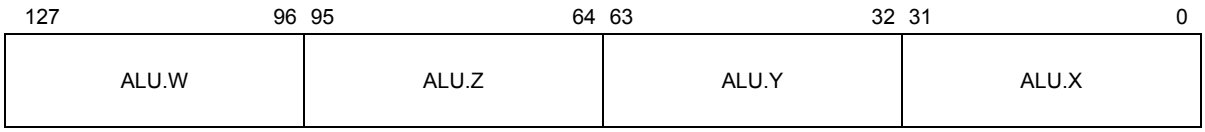


Figure 4.2 Organization of ALU Vector Elements in GPRs

The processor contains multiple sets of five scalar ALUs. Four in each set can perform scalar operations on up to three 32-bit data elements each, with one 32-bit result. The ALUs are called *ALU.X, ALU.Y, ALU.Z,* and *ALU.W* (or simply ALU.[X,Y,Z,W]). A fifth unit, called *ALU.Trans*, performs one scalar operation and additional operations for transcendental and advanced integer functions; it can replicate the result across all four elements of a destination vector. Although the processor has multiple sets of these five scalar ALUs, Evergreen software can assume that, within a given ALU clause, all instructions are processed by a single set of five ALUs.

Software issues ALU instructions in variable-length groups called *instruction groups*. These perform parallel operations on different elements of a vector, as described in Section 4.3, “ALU Instruction Slots and Instruction Groups,” page 4-3. The ALU.[X,Y,Z,W] units are nearly identical in their functions. They differ only in the vector elements to which they write their result at the end of the instruction and in certain reduction operations (see Section 4.8.2, “Instructions for ALU.[X,Y,Z,W] Units Only,” page 4-22). The ALU.Trans unit can write to any vector element and can evaluate additional functions.

ALU instructions can access 256 constants (from the constant registers) and 128 GPRs (each thread accesses its own set of 128 GPRs). Constant-register addresses and GPR addresses can be absolute, relative to the loop index (aL), or relative to an index GPR. In addition to reading constants from the constant registers, an ALU instruction can refer to elements of a literal constant that is embedded in the instruction group. Instructions also have access to two temporary registers that contain the results of the previous instruction groups. The previous vector (PV) register contains a four-element vector that is the previous result from the ALU.[X,Y,Z,W] units; the previous scalar (PS) register contains a scalar that is the previous result from the ALU.Trans unit.

Each instruction has its own set of source operands:

- SRC0 and SRC1 for instructions using the ALU_DWORD1_OP2 microcode format, and SRC0, SRC1,
- SRC2 for instructions using the ALU_DWORD1_OP3 microcode format.

An instruction group that operates on a four-element vector is specified as at least four independent scalar instructions, one for each vector element. As a result, vector operations can perform a complex mix of vector-element and constant swizzles, and even swizzles across GPR addresses (subject to read-port restrictions described in the next paragraph). Traditional floating-point and integer constants for common values (for example, 0, -1, 0.0, 0.5, and 1.0) can be specified for any source operand.

Each ALU.[X,Y,Z,W] unit writes to an instruction-specified GPR at the end of the instruction. The GPR address can be absolute, relative to the loop index, or relative to an index GPR. The ALU.[X,Y,Z,W] units always write to their corresponding vector element, but each unit can write to a different GPR address. The ALU.Trans unit can write to any vector element of any GPR address. The outputs of each ALU unit can be clamped to the range [0.0, 1.0] prior to being written, and some operations can multiply the output by a factor of 2.0 or 4.0.

4.3 ALU Instruction Slots and Instruction Groups

An ALU *instruction group* is listed in Table 2.7 on page 2-10. Each group consists of one to five ALU *instructions*, optionally followed by one or two *literal constants*, each of which can hold two vector elements. Each instruction is 64 bits wide (composed of two 32-bit microcode formats). Two elements of a literal constant are also 64 bits wide. Thus, the basic memory unit for an ALU instruction group is a 64-bit *slot*, which is a position for an ALU instruction or an associated literal constant. An instruction group consists of one to seven slots, depending on the number of instructions and literal constants. All ALU instructions occupy one slot, except double-precision floating-point instructions, which occupy either two or four slots (see Section 4.12, “Double-Precision Floating-Point Operations,” page 4-29). The ALU clause size in the CF program is specified as the total number of slots occupied by the ALU clause.

Each instruction in a group has a `LAST` bit that is set only for the last instruction in the group. The `LAST` bit delimits instruction groups from one another, allowing the Evergreen hardware to implement parallel processing for each instruction group. Each instruction is distinguished by the destination vector element to which it writes. An instruction is assigned to the ALU.Trans unit if a prior instruction in the group writes to the same vector element of a GPR, *or* if the instruction is a transcendental operation.

The instructions in an instruction group must be in instruction slots 0 through 4, in the order shown in Table 4.1. Up to four of the five instruction slots can be omitted. Also, if any instructions refer to a literal constant by specifying the `ALU_SRC_LITERAL` value for a source operand, the first, or both, of the two-element literal constant slots (slots 5 and 6) must be provided; the second of these two slots cannot be specified alone. There is no `LAST` bit for literal constants. The number of the literal constants is known from the operations specified in the instruction.

Table 4.1 Instruction Slots in an Instruction Group

Slot	Entry	Bits	Type
0	Scalar instruction for ALU.X unit	64	*src*.X and *dst*.X vector-element slot
1	Scalar instruction for ALU.Y unit	64	*src*.Y and *dst*.Y vector-element slot
2	Scalar instruction for ALU.Z unit	64	*src*.Z and *dst*.Z vector-element slot
3	Scalar instruction for ALU.W unit	64	*src*.W and *dst*.W vector-element slot
4	Scalar instruction for ALU.Trans unit	64	Transcendental slot
5	X, Y elements of literal constant (X is the first dword)	64	Constant slot
6	Z, W elements of literal constant (Z is the first dword)	64	Constant slot

Given the options described above, the size of an ALU instruction group can range from 64 bits to 448 bits, in increments of 64 bits.

4.4 Assignment to ALU.[X,Y,Z,W] and ALU.Trans Units

Assignment of instructions to the ALU.[X,Y,Z,W] and ALU.Trans units is observable by software, since it determines the values PV and PS registers hold at the end of an instruction group. In some cases, there is an unambiguous assignment to ALUs based on the instructions and destination operands. In other cases, the last slot in an instruction group is ambiguous. It can be assigned to either the ALU.[X,Y,Z,W] unit or the ALU.Trans unit.[1]

The following algorithm illustrates the assignment of instruction-group slots to ALUs. The instruction order described in Section 4.3, “ALU Instruction Slots and Instruction Groups,” page 4-3, must be observed. As a consequence, if the ALU.Trans unit is specified, it must be done with an instruction that has its LAST bit set.

```
begin
    ALU_[X,Y,Z,W] := undef;
    ALU_TRANS := undef;
    for $i = 0 to number of instructions - 1
        $elem := vector element written by instruction $i;
        if instruction $i is transcendental only instruction
            $trans := true;
        elsif instruction $i is vector-only instruction
            $trans := false;
        elsif defined(ALU_$elem) or (not
CONFIG.ALU_INST_PREFER_VECTOR and
            instruction $i is LAST)
            $trans := true;
        else
            $trans := false;
        if $trans
```

1. This ambiguity is resolved by a bit in the processor state, CONFIG.ALU_INST_PREFER_VECTOR, that is programmable only by the host. When the bit is set, ambiguous slots are assigned to ALU.Trans. When cleared (default), ambiguous slots are assigned to one of ALU.[X,Y,Z,W]. This setting applies to all thread types.

```
            if defined(ALU_TRANS)
                assert "ALU.Trans has already been allocated,
                    cannot give to instruction $i.";
            ALU_TRANS := $i;
        else
            if defined(ALU_$elem)
                assert "ALU.$elem has already been allocated,
                    cannot give to instruction $i.";
            ALU_$elem := $i;
end
```

After all instructions in the instruction group are processed, any ALU.[X,Y,Z,W] or ALU.Trans operation that is unspecified implicitly executes a NOP instruction, thus invalidating the values in the corresponding elements of the PV and PS registers.

4.5 OP2 and OP3 Microcode Formats

To keep the ALU slot size at 64 bits while not sacrificing features, the microcode formats for ALU instructions have two versions: ALU_DWORD1_OP2 (page 10-26) and ALU_DWORD1_OP3 (page 10-32). The OP2 format is used for instructions that require zero, one, or two source operands plus destination operand. The OP3 format is used for the smaller set of instructions requiring three source operands plus destination operand.

Both versions have an ALU_INST field, which specifies the instruction opcode. The ALU_DWORD1_OP2 format has a 10-bit instruction field; ALU_DWORD1_OP3 format has a five-bit instruction field. The fields are aligned so that their MSBs overlap. In the OP2 version, the ALU_INST field uses a seven-bit opcode, and the high three bits are always 000b. In the OP3 version, at least one of the high three bits of the ALU_INST field is nonzero.

4.6 GPRs and Constants

Within an ALU clause, instructions can access to up to 127 GPRs and 256 constants from the constant registers. Some GPR addresses can be reserved for *clause temporaries*. These are temporary values typically stored at GPR[124,127][1] that do not need to be preserved past the end of a clause. This gives a program access to temporary registers that do not count against its GPR count (the number of GPRs that a program can use), thus allowing more programs to run simultaneously.

For example, if the result of an instruction is required for another instruction within a clause, but not needed after the clause executes, a clause temporary can be used to hold the result. The first instruction specifies GPR[124, 127] as its destination, while the second instruction specifies GPR[124, 127] as its

1. The number of clause temporaries can be programed only by the host processor using the configuration-register field GPR_RESOURCE_MGMT_1.NUM_CLAUSE_TEMP_GPRS. A typical setting for this field is 4. If the field has N > 0, then GPR[127 – N + 1, 127] are set aside as clause temporaries.

source. After the clause executes, GPR[124, 127] can be used by another clause.

Any constant-register address can be absolute, relative to the loop index, or relative to one of four elements in the address register (AR) that is loaded by a prior MOVA* instruction in the same clause. Any GPR (source or destination) address can be absolute, relative to the loop index, or relative to the X element in the address register (AR) that is loaded by a prior MOVA* instruction in the same clause.

In addition to reading constants from the constant registers, any operand can refer to an element in a literal constant, as described in Section 4.3, “ALU Instruction Slots and Instruction Groups,” page 4-3.

Constants also can come from one of two banks of *kcache* constants that are read from memory before the clause executes. Each bank is a set of 16 constants locked into the cache for the duration of the clause by the CF instruction that started it.

4.6.1 Relative Addressing

Each instruction can use only one index for relative addressing. Relative addressing is controlled by the SRC_REL and DST_REL fields of the instruction’s microcode format. The index used is controlled by the INDEX_MODE field of the instruction’s microcode format. Each source operand in the instruction then declares whether it is absolute or relative to the common index. The index used depends on the operand type and the setting of INDEX_MODE, as shown in Table 4.2.

Table 4.2 Index for Relative Addressing

INDEX_MODE	GPR Operand	Constant Register Operand	Kcache Operand
INDEX_AR_X	AR.X	AR.X	*not valid*
INDEX_AR_Y	AR.X	AR.Y	*not valid*
INDEX_AR_Z	AR.X	AR.Z	*not valid*
INDEX_AR_W	AR.X	AR.W	*not valid*
INDEX_LOOP	Loop Index (aL)	Loop Index (aL)	Loop Index (aL)

The term *flow-control loop index* refers to the DirectX9-style loop index. Each instruction has its own INDEX_MODE control, so a single instruction group can refer to more than one type of index.

When using an AR index, the index must be initialized by a MOVA* operation that is present in a prior instruction group of the same clause. Thus, AR indexing is never valid on the first instruction of a clause.

An AR index cannot be used in an instruction group that executes a MOVA* instruction in any slot. Any slot in an instruction group with a MOVA* instruction using relative constant addressing can use only an INDEX_MODE of INDEX_LOOP.

To issue a MOVA* from an AR-relative source, the source must be split into two separate instruction groups, the first performing a MOV from the relative source into a temporary GPR, and the second performing a MOVA* on the temporary GPR.

Only one AR element can be used per instruction group. For example, it is not legal for one slot in an instruction group to use INDEX_AR_X, and another slot in the same instruction group to use INDEX_AR_Y. Also, AR cannot be used to provide relative indexing for a kcache constant; kcache constants can use only the INDEX_LOOP mode for relative indexing.

GPR clause temporaries cannot be indexed.

4.6.2 Previous Vector (PV) and Previous Scalar (PS) Registers

Instructions can read from two additional temporary registers: previous vector (PV) and previous scalar (PS). These contain the results from the ALU.[X,Y,Z,W] and ALU.Trans units, respectively, of the previous instruction group. Together, these registers provide five 32-bit elements; PV contains a four-element vector originating from the ALU.[X,Y,Z,W] output, and PS contains a single scalar value from the ALU.Trans output. The registers can be used freely in an ALU instruction group (although using one in the first instruction group of the clause makes no sense). NOP instructions do not preserve PV and PS values, nor are PV and PS values preserved past the end of the ALU clause.

4.6.3 Out-of-Bounds Addresses

GPR and constant-register addresses can stray out of bounds after relative addressing is applied. In some cases, an address that strays out of bounds has a well-defined behavior, as described below.

Assume *N* GPRs are declared per thread, and *K* clause temporaries are also declared. The GPR base address specified in SRC*_SEL must be in either the interval [0, *N* – 1] (normal clause GPR) or [128 – *K*, 127] (clause temporary), before any relative index is applied. If SRC*_SEL is a GPR address and does not fall into either of these intervals, the resulting behavior is undefined. For example, you cannot write code that generates GPR*N*[-1] to read from the last GPR in a program.

If a GPR read with base address in [0, *N* – 1] is indexed relatively, and the base plus the index is outside the interval [0, *N* – 1], the read value is always GPR0 (including instructions for fetch through a texture cache clause and fetch through a vertex cache clause, as well as imports and exports). If a GPR write with base address in [0, *N* – 1] is indexed relatively, and the base plus the index is outside the interval [0, *N* – 1], the write is inhibited (including for instructions for a fetch through a texture cache clause and a fetch through a vertex cache clause), unless the instruction is a memory read. If the instruction is a memory read, the result are written to GPR0. Relative addressing on GPR clause temporaries is illegal. Thus, the behavior is undefined if a GPR with a base address in the [128 – *K*, 127] range is used with a relative index.

A constant-register base address is always be in-bounds. If a constant-register read is indexed relatively, and the base plus the index is outside the interval [0, 255], the value read is NaN (0x7FFFFFFF).

If a kcache base address refers to a cache line that is not locked, the result is undefined. You cannot refer to kcache constants [0, 15] if the mode (as set by the CF instruction initiating the ALU clause) is KCACHE_NOP, and you cannot refer to kcache constants [16, 31] if the mode is KCACHE_NOP or KCACHE_LOCK_1. If a kcache read is indexed relatively, one cache line is locked with KCACHE_LOCK_1, and the base plus the index is outside the interval [0, 15], the value read is NaN (0x7FFFFFFF). If a kcache read is indexed relatively, two cache lines are locked, and the base plus the index is outside the interval [0, 31], the value read is NaN (0x7FFFFFFF).

4.6.4 ALU Constants

Each ALU instruction in the X,Y,Z or W slots can reference up to three constants; an instruction in the T slot can reference up to two constants. All ALU constants are 32 bits. There are four types of constants:

- DX9 ALU constants (constant file)
- DX10 ALU constants (constant cache)
- Literal constants
- Inline constants

All kernels operate exclusively in one of two modes: DX9 or DX10.

When in DX9 mode, ALU instructions have access to a constant file of 256 128-bit constants; each instruction group can reference up to four of these. These constants exist only for PS and VS kernels.

In DX10 mode, each kernel can use up to 16 constant buffers. A constant buffer is a collection of constants in memory anywhere from 1 to 4096 128-bit constants. Each ALU clause can use only two windows of 32 constants. The can be windows into the same or different constant buffers.

4.6.4.1 Constant Cache

Each ALU clause can lock up to four sets of constants into the constant cache. Each set (one cache line) is 16 128-bit constants. These are split into two groups. Each group can be from a different constant buffer (out of 16 buffers). Each group of two constants consists of either [Line] and [Line+1], or [line + loop_ctr] and [line + loop_ctr +1].

4.6.4.2 Literal Constants

Literal constants count against the total number of instructions that a clause can have. Up to four DWORD constants can be supplied and swizzled arbitrarily.

4.6.4.3 Inline Constants

Inline constants can be swizzled in to any source position and do not count against the total number of instructions in a clause or the maximum number of ALU constants in use. Literal constants supply common values: 0, 1, -1, 1.0 etc.

4.6.4.4 Statically-Indexed Constant Access

The constant-file entries can be accessed either with absolute addresses, or addresses relative to the current loop index (aL, static indirect access). In both cases, all pixels in the vector pick the same constant to use, and there is no performance penalty. Swizzling is allowed.

4.6.4.5 Dynamically-Indexed Constant Access (AR-relative, Constant Waterfalling)

To support DX9 vertex shaders, we provide dynamic indexing of constant-file constants. This means that a GPR value is used as the index into the constant file. Since the value comes from a GPR, it can be unique for each pixel. In the worst case, it may take 64 times as long to execute this instruction, since up to 64 constant-file reads can be required.

Dynamic indexing requires two instructions:

- MOVA: Move one element of a GPR into the Address Register (AR) to be used as the index value.
- *<any ALU instruction>*: Use the indices from the MOVA and perform the indirect lookup.

There is a two-instruction delay slot between loading and using the GPR index value. Hardware inserts delays if the kernel does not. The GPR indices loaded by a MOVA instruction only persist for one clause; at the end of the clause they are invalidated.

4.6.4.6 ALU Constant Buffer Sharing

ES, GS, and VS kernels can, on a per-ALU-clause basis, access their own constant buffers or those of the other shader type.

ES/VS can use their own or use GS constant buffers, and GS can use its own or ES/VS ones. This is provided for cases when the GS and VS shaders can be merged into a single hardware shader stage.

This capability is activated by setting the ALT_CONSTS bit in the SQ_CF_ALU_WORD1.

4.7 Scalar Operands

For each instruction, the operands src0, src1, and src2 are specified in the instruction's SRC*_SEL and SRC*_ELEM fields. GPR and constant-register addresses can be relative-addressed, as specified in the SRC*_REL and INDEX_MODE fields. In the OP2 microcode format, src2 is undefined.

4.7.1 Source Addresses

The data source address is specified in the SRC*_SEL field. This can refer to one of the following.

- A GPR address, GPR[0, 127], with values [0, 127].
- A kcache constant in bank 0, kcache0[0, 31], with values [128, 159]; kcache0[16, 31] are accessible only if two cache lines have been locked.
- A kcache constant in bank 1, kcache1[0, 31], with values [160, 191]; kcache1[16, 31] are accessible only if two cache lines are locked.
- A constant-register address, c[0, 255], with values [256, 511].
- The previous vector (PV) or scalar (PS) result.
- A literal constant (two constants are present if any operand uses a Z or W constant).
- A floating-point inline constant (0.0, 0.5, 1.0).
- An integer inline constant (-1, 0, 1).

If the SRC*_SEL field specifies a GPR or constant-register address, then the relative index specified by the INDEX_MODE field is added to the address if the SRC*_REL bit is set.

The definitions of the selects for PV, PS, literal constant, and the special inline constant values are given in the microcode specification. Also, the following constant values are defined to assist in encoding and decoding the SRC*_SEL field:

- ALU_SRC_GPR_BASE = 0 — Base value for GPR selects.
- ALU_SRC_KCACHE0_BASE = 128 — Base value for kcache bank 0 selects.
- ALU_SRC_KCACHE1_BASE = 144 — Base value for kcache bank 1 selects.
- ALU_SRC_CFILE_BASE = 256 — Base value for constant-register address selects.

The SRC*_ELEM field specifies from which vector element of the source address to read. It is ignored when PS is specified. If a literal constant is selected, and SRC*_ELEM specifies the Z or W element; then, both slots of the literal constant must be specified at the end of the instruction group.

4.7.2 Input Modifiers

Each input operand can be modified. The modifiers available are negate, absolute value, and absolute-then-negate; they are specified using the SRC*_NEG and SRC*_ABS fields. The modifiers are meaningful only for floating-point inputs. Integer inputs must leave these fields cleared (zero), which is the pass-through value. If the SRC*_NEG and SRC*_ABS bits are set, the absolute value is performed first. Instructions with three source operands have only the negation modifier, SRC*_NEG; absolute value, if desired, must be performed by a separate instruction with two source operands.

4.7.3 Data Flow

A simplified data flow for the ALU operands is given in Figure 4.3. The data flow is discussed in more detail in the following sections.

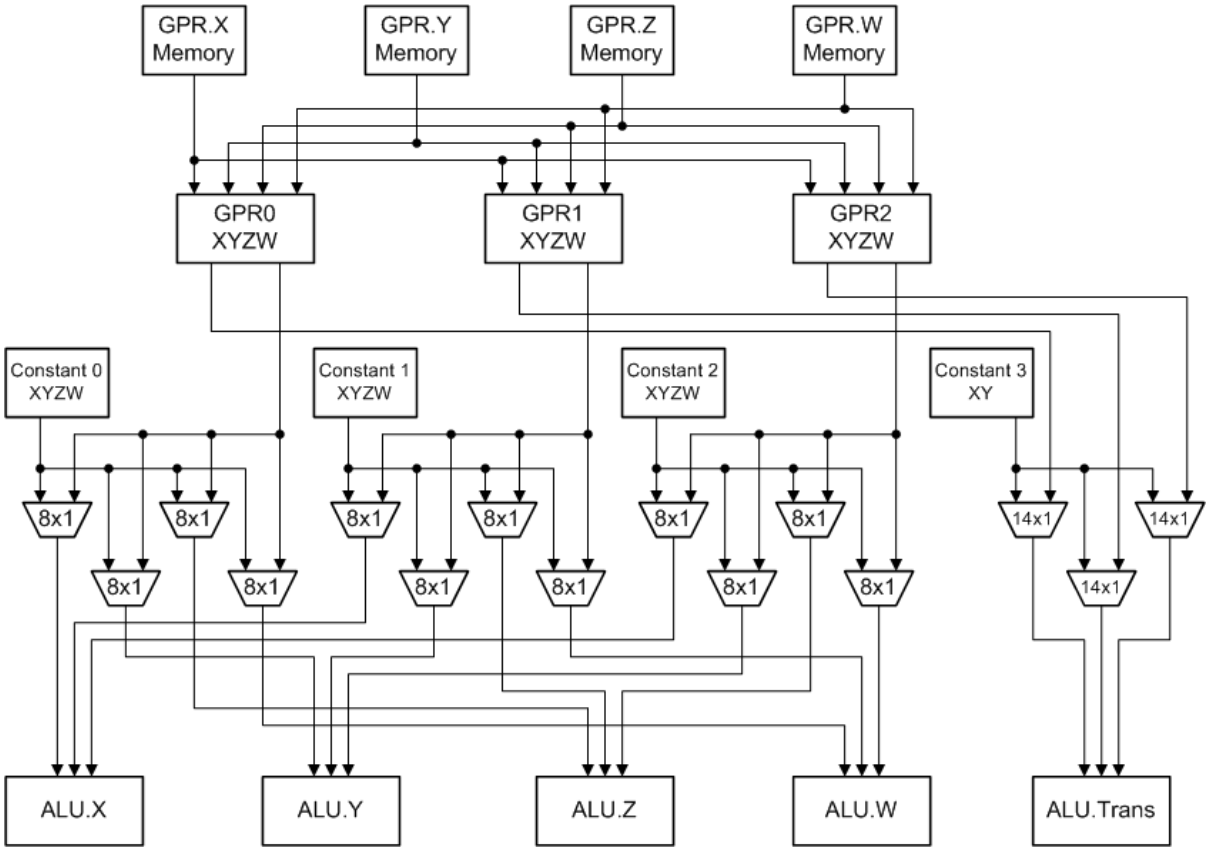


Figure 4.3 ALU Data Flow

4.7.4 GPR Read Port Restrictions

In hardware, the X, Y, Z, and W elements are stored in separate memories. Each element memory has three read ports per instruction. As a result, an instruction can refer to at most three distinct GPR addresses (after relative addressing is applied) per element. The processor automatically shares a read port for multiple operands that use the same GPR address or element. For example, all scalar src0 operands can refer to GPR2.X with only one read port. Thus, there are only 12 GPR source elements available per instruction (three for each element). Additional GPR read restrictions are imposed for both ALU.[X,Y,Z,W] and ALU.Trans, as described below.

4.7.5 Constant Register Read Port Restrictions

Software can read any four distinct elements from the constant registers in one instruction group, after relative addressing is applied. The four constants must be two pairs of constants from any address: either Cn.x,Cn.y or Cn.z,Cn.w. No more

than four distinct elements can be read from the constant file in one instruction group.

Each ALU.Trans operation can reference at most two constants of any type. For example, all of the following are legal, and the four slots shown can occur as a single instruction group:

```
GPR0.X <= C0.X + GPR0.X
GPR0.Y <= 1.0 + C1.Y // Can mix cfile and non-cfile in one
instruction group.
GPR0.Z <= C2.X + GPR0.Z // Multiple reads from cfile X bank are OK.
GPR0.W <= C3.Z + C0.X // Reads from four distinct cfile addresses
are OK.
```

4.7.6 Literal Constant Restrictions

A literal constant is fetched if any source operand refers to the literal constant, regardless of whether the operand is used by the instruction group; so, be sure to clear unused operands in instruction fields. If all operands referencing the literal refer only to the X and Y vector elements, a two-element literal (one slot) is fetched. If any operand referencing the literal refers to the Z or W vector elements, a four-element literal (two slots) is fetched. An ALU.Trans operation can reference at most two constants of any type.

4.7.7 Cycle Restrictions for ALU.[X,Y,Z,W] Units

For ALU.[X,Y,Z,W] operations, source operands src0, src1, and src2 are loaded during three cycles. At most one GPR.X, one GPR.Y, one GPR.Z and one GPR.W can be read per cycle. The GPR values requested on cycle *N* are assembled into a four-element vector, CYCLE*N*_GPR. In addition, four constant elements are sent to the pipeline from a combination of sources: the constant-register constant, a literal constant, and the special inline constants. The constant elements sent on cycle *N* are assembled into a four-element vector, CYCLE*N*_K. Collectively, these two vectors are referred to as CYCLE*N*_DATA.

The values in CYCLE*N*_DATA populate the logical operands src[0, 2]. The mapping of CYCLE[0, 2]_DATA to src[0, 2] must be specified in the microcode, using the BANK_SWIZZLE field. Read port restrictions must be respected across the instructions in an instruction group, described below. Each slot has its own BANK_SWIZZLE field, and these fields can be coordinated to avoid the read port restrictions.

For ALU.[X,Y,Z,W] operations, BANK_SWIZZLE specifies from which cycle each operand data comes from, if the operand's source is GPR data. Constant data for src*N* is always from CYCLE*N*_K. The setting, ALU_VEC_012, is the identity setting that loads operand *N* using data in CYCLE*N*_GPR.

BANK_SWIZZLE	src0	src1	src2
ALU_VEC_012	CYCLE0_GPR	CYCLE1_GPR	CYCLE2_GPR
ALU_VEC_021	CYCLE0_GPR	CYCLE2_GPR	CYCLE1_GPR
ALU_VEC_120	CYCLE1_GPR	CYCLE2_GPR	CYCLE0_GPR
ALU_VEC_102	CYCLE1_GPR	CYCLE0_GPR	CYCLE2_GPR
ALU_VEC_201	CYCLE2_GPR	CYCLE0_GPR	CYCLE1_GPR
ALU_VEC_210	CYCLE2_GPR	CYCLE1_GPR	CYCLE0_GPR

In this configuration, if an operand is referenced more than once in a scalar operation, it must be loaded in two different cycles, sacrificing two read ports. For example:

Instruction	BANK_SWIZZLE	CYCLE0_GPR	CYCLE1_GPR	CYCLE2_GPR
GPR0.X <= GPR1.X * GPR2.X + GPR1.X	ALU_VEC_012	GPR1.X	GPR2.X	GPR1.X
GPR0.Y <= GPR1.Y * GPR2.Y + GPR1.Y	ALU_VEC_012	GPR1.Y	GPR2.Y	GPR1.Y

However, as a special case, if src0 and src1 in an instruction refer to the same GPR element, only one read port is used, on the cycle corresponding to src0 in the bank swizzle. This optimization exists to facilitate squaring operations (MUL* x, x, and DOT* v, v). The following example illustrates the use of this optimization to perform square operations that do not consume more than one read port per GPR element.

Instruction	BANK_SWIZZLE	CYCLE0_GPR	CYCLE1_GPR	CYCLE2_GPR
GPR0.X <= GPR1.X * GPR1.X	ALU_VEC_012	GPR1.X	—[1]	—
GPR0.Y <= GPR1.Y * GPR1.Y	ALU_VEC_120	—[1]	GPR1.Y	—

1. src1 is shared and fetches its data on the same cycle that src0 fetches. No actual read port is used in the marked cycles.

In the above example, the swizzle selects for src0 determine on which cycle to load the shared operand. The swizzle selects for src1 are ignored. The following programming is legal, even though at first glance the bank swizzles might suggest it is not.

Instruction	BANK_SWIZZLE	CYCLE0_GPR	CYCLE1_GPR	CYCLE2_GPR
GPR0.X <= GPR1.X * GPR1.X	ALU_VEC_012	GPR1.X	—[1]	—
GPR0.Y <= GPR1.Y * GPR1.Y	ALU_VEC_102	—[1]	GPR1.Y	—
GPR0.Z <= GPR2.Y * GPR2.X	ALU_VEC_012	GPR2.Y	GPR2.X	—

1. src1 is shared and fetches its data on the same cycle that src0 fetches. No actual read port is used up in the marked cycles.

This optimization only applies when src0 and src1 share the same GPR element in an instruction. It does not apply when src0 and src2, nor when src1 and src2, share a GPR element.

Software cannot read two or more values from the same GPR vector element on a single cycle. For example, software cannot read GPR1.X and GPR2.X on cycle 0. This restriction does not apply to constant registers or literal constants. For example, the following programming is illegal.

Instruction	BANK_SWIZZLE	CYCLE0_GPR	CYCLE1_GPR	CYCLE2_GPR
GPR0.X <= GPR1.X * GPR2.X	ALU_VEC_012	invalid	GPR2.X	—
GPR0.Y <= GPR3.X * GPR1.Y	ALU_VEC_012	invalid	GPR1.Y	—
GPR0.Z <= GPR2.X * GPR1.Y	ALU_VEC_012	invalid	GPR1.Y**	—

Software can use BANK_SWIZZLE to work around this limitation, as shown below.

Instruction	BANK_SWIZZLE	CYCLE0_GPR	CYCLE1_GPR	CYCLE2_GPR
GPR0.X <= GPR1.X * GPR2.X	ALU_VEC_012	GPR1.X	GPR2.X	—
GPR0.Y <= GPR3.X * GPR1.Y	ALU_VEC_201	GPR1.Y	—	GPR3.X
GPR0.Z <= GPR2.X * GPR1.Y	ALU_VEC_102	GPR1.Y[1]	GPR2.X**	—

1. The above examples illustrate that once a value is read into CYCLEN_DATA, multiple instructions can reference that value.

The temporary registers PV and PS have no cycle restrictions. Any element in these registers can be accessed on any cycle. Constant operands can be accessed on any cycle.

4.7.8 Cycle Restrictions for ALU.Trans

The ALU.Trans unit is not subject to the close tie between src*N* and cycle *N* that the ALU.[X,Y,Z,W] units have. It can opportunistically load GPR-based operands on any cycle. However, the ALU.Trans unit must share the GPR read ports used by the ALU.[X,Y,Z,W] units. If one of the ALU.[X,Y,Z,W] units loads an operand that an ALU.Trans operand needs, it is possible to load the ALU.Trans operand on the same cycle. If not, the ALU.Trans hardware must find a cycle with an unused read port to load its operand.

The ALU.Trans slot also has a BANK_SWIZZLE field, but it interprets the field differently from ALU.[X,Y,Z,W]. The BANK_SWIZZLE field is used to determine from which of CYCLE[0, 2]_GPR each src[0, 2] operand gets its data. It can have one of the following values:

BANK_SWIZZLE	src0	src1	src2
ALU_SCL_210	CYCLE0_DATA	CYCLE1_DATA	CYCLE2_DATA
ALU_SCL_122	CYCLE1_DATA	CYCLE2_DATA	CYCLE2_DATA
ALU_SCL_212	CYCLE2_DATA	CYCLE1_DATA	CYCLE2_DATA
ALU_SCL_221	CYCLE2_DATA	CYCLE2_DATA	CYCLE1_DATA

Multiple operands in ALU.Trans can read from the same cycle (this differs from the ALU.[X,Y,Z,W] case). Not all possible permutations are available. If needed,

the unspecified permutations can be obtained by applying an appropriate inverse mapping on the ALU.[X,Y,Z,W] slots.

Here is an example illustrating how ALU.Trans operations can use free read ports from GPR instructions (in all of the following examples, the last instruction in an instruction group is always an ALU.Trans operation):

Instruction	BANK_SWIZZLE	CYCLE0_GPR	CYCLE1_GPR	CYCLE2_GPR
`GPR0.X <= GPR1.X * GPR2.X`	`ALU_VEC_012`	GPR1.X	GPR2.X	—
`GPR0.Y <= GPR3.X * GPR1.Y`	`ALU_VEC_210`	—	GPR1.Y	GPR3.X
`GPR1.X <= GPR3.Z * GPR3.W`	`ALU_SCL_221`	—	—	GPR3.[ZW]

When an operand is used by one of the ALU.[X,Y,Z,W] units, it also can be used to load an operand into the ALU.Trans unit:

Instruction	BANK_SWIZZLE	CYCLE0_GPR	CYCLE1_GPR	CYCLE2_GPR
`GPR0.X <= GPR1.X * GPR2.X`	`ALU_VEC_210`	—	GPR2.X	GPR1.X
`GPR0.Y <= GPR3.X * GPR1.Y`	`ALU_VEC_012`	GPR3.X	GPR1.Y	—
`GPR1.X <= GPR1.X * GPR1.Y`	`ALU_SCL_210`	—	GPR1.Y	GPR1.X

Any element in PV or PS registers can be accessed by ALU.Trans; generally, it is loaded as soon as possible. PV or PS register values can be loaded on any cycle, but when constant operands are present, the available bank swizzles can be constrained (see Section 4.7.8.1, “Bank Swizzle with Constant Operands”).

4.7.8.1 Bank Swizzle with Constant Operands

If the transcendental operation uses a single constant operand (any type of constant), the remaining GPR operands must not be loaded on cycle 0. The instruction group:

```
GPR0.X <= GPR1.X * GPR2.Y + CFILE0.Z
```

can use any of the following bank swizzles.

- `ALU_SCL_210` — no operand loaded on cycle 0
- `ALU_SCL_122`
- `ALU_SCL_212` — synonymous with 210 swizzle in this case
- `ALU_SCL_221`

However, the instruction group

```
GPR0.X <= CFILE0.Z * GPR1.X + GPR2.Y
```

can use only the following swizzles.

- `ALU_SCL_122`
- `ALU_SCL_212`

- ALU_SCL_221

Similarly, when a single constant operand is used, no PV or PS operand can be loaded on cycle 0. The instruction group

```
GPR0.X <= CFILE0.Z * PV.X + PS
```

can use only one of the following swizzles.

- ALU_SCL_122
- ALU_SCL_212
- ALU_SCL_221

If the transcendental operation uses *two* constant operands (any types of constants), then the remaining GPR operand must be loaded on cycle 2. The instruction group

```
GPR0.X <= CFILE0.X * CFILE0.Y + GPR1.Z
```

can use only one of the following bank swizzles.

- ALU_SCL_122
- ALU_SCL_212 — synonymous with 122 swizzle in this case

Similarly, when two constant operands are used, any PV or PS operand must be loaded on cycle 2. The instruction group

```
GPR0.X <= CFILE0.X * CFILE0.Y + PV.Z
```

can use only one of the following bank swizzles:

- ALU_SCL_122
- ALU_SCL_212 — synonymous with 122 swizzle in this case

The transcendental operation cannot reference constants in all three of its operands.

4.7.9 Read-Port Mapping Algorithm

This section describes the algorithm that determines what combinations of source operands are permitted in a single instruction. For this algorithm, let

- HW_GPR[0,1,2]_[X,Y,Z,W] store addresses for the [0, 2] GPR read port reservations
- HW_CFILE[0,1,2,3]_ADDR represent a constant-register address, and
- HW_CFILE[0,1,2,3]_ELEM represent an element (X, Y, Z, W) for the [0, 3] constant-register read port reservation.

For simplicity, this algorithm ignores relative addressing; if relative addressing is used, address references below are *after* the relative index is applied.

The function, cycle_for_bank_swizzle(*$swiz, $sel*), returns the cycle number that the operand *$sel* must be loaded on, according to the bank swizzle *$swiz*. The return value is shown in Table 4.3.

Table 4.3 Example Function's Loading Cycle

$*swiz*	$*sel* == 0	$*sel* == 1	$*sel* == 2
ALU_VEC_012	0	1	2
ALU_VEC_021	0	2	1
ALU_VEC_120	1	2	0
ALU_VEC_102	1	0	2
ALU_VEC_201	2	0	1
ALU_VEC_210	2	1	0
ALU_SCL_210	2	1	0
ALU_SCL_122	1	2	2
ALU_SCL_212	2	1	2
ALU_SCL_221	2	2	1

4.7.9.1 Initialization Execution

The following procedure is executed on initialization.

```
procedure initialize
begin
   HW_GPR[0,1,2]_[X,Y,Z,W] := undef;
   HW_CFILE[0,1,2,3]_ADDR := undef;
   HW_CFILE[0,1,2,3]_ELEM := undef;
end
```

4.7.9.2 Reserving GPR Read

The following procedure reserves the GPR read for address *$sel* and vector element *$elem* on cycle number *$cycle*.

```
procedure reserve_gpr($sel, $elem, $cycle)
   if !defined(HW_GPR$cycle _$elem)
      HW_GPR$cycle_$elem := $sel;
   elsif HW_GPR$cycle_$elem != $sel
      assert "Another instruction has already used GPR read port
$cycle
         for vector element $elem";
end
```

4.7.9.3 Reserving Constant File Read

The following procedure reserves the constant file read for address *$sel* and vector element *$elem*.

```
begin
  $resmatch := undef;
  $resempty := undef;
  for $res in {1, 0}
    if !defined(HW_CONST$res_ADDR)
      $resempty := $res;
    elsif HW_CONST$res_ADDR == $sel and HW_CONST$res_CHAN == int($chan/2)
      $resmatch := $res;
  if defined($resmatch)
    // Read for this scalar component already reserved, nothing to do here.
  elsif defined($resempty)
    HW_CONST$resempty_ADDR := $sel;
    HW_CONST$resempty_CHAN := int($chan/2);
  else
    assert "All CONST read ports are used, cannot reference C$sel, channel pair ($chan/2).";
end
```

4.7.9.4 Execution for Each ALU.[X,Y,Z,W] Operation

The following procedure is executed for each ALU.[X,Y,Z,W] operation specified in the instruction group.

```
procedure check_vector
begin
    for $src in {0, ..., number_of_operands(ALU_INST)}
        $sel := SRC$src_SEL;
        $elem := SRC$src_ELEM;
        if isgpr($sel)
            $cycle := cycle_for_bank_swizzle(BANK_SWIZZLE, $src);
            if $src == 1 and $sel == SRC0_SEL and $elem == SRC0_ELEM
                // Nothing to do; special-case optimization,
                    second source uses first source's reservation
            else
                reserve_gpr($sel, $elem, $cycle);
        elsif isconst($sel)
            // Any constant, including literal and inline constants
            if iscfile($sel)
                reserve_cfile($sel, $elem);
        else
            // No restrictions on PV, PS
end
```

4.7.9.5 Execution of ALU.Trans Operation

The following procedure is executed for an ALU.Trans operation, if it is specified in the instruction group. The ALU.Trans unit tries to reuse an existing reservation whenever possible. The constant unit cannot use cycle 0 for GPR loads if one constant operand is specified; it must use cycle 2 for GPR load if two constant operands are specified.

```
procedure check_scalar
begin
    $const_count := 0;
    for $src in {0, ..., number_of_operands(ALU_INST)}
        $sel := SRC$src_SEL;
        $elem := SRC$src_ELEM;
        if isconst($sel)
```

```
            // Any constant, including literal and inline constants
            if $const_count >= 2
               assert "More than two references to a constant in
transcendental operation.";
            $const_count++;
            if iscfile($sel)
               reserve_cfile($sel, $elem);
      for $src in {0, ..., number_of_operands(ALU_INST)}
         $sel := SRC$src_SEL;
         $elem := SRC$src_ELEM;
         if isgpr($sel)
            $cycle := cycle_for_bank_swizzle(BANK_SWIZZLE, $src);
            if $cycle < $const_count
               assert "Cycle $cycle for GPR load conflicts with
constant
                     load in transcendental operation.";
            reserve_gpr($sel, $elem, $cycle);
         elsif isconst($sel)
            // Constants already processed
         else
            // No restrictions on PV, PS
end
```

4.8 ALU Instructions

This section gives a brief summary of ALU instructions. See Section 9.2, “ALU Instructions,” page 9-48, for details about the instructions.

4.8.1 Instructions for All ALU Units

The instructions shown in Table 4.4 are valid for all ALU units: ALU.[X,Y,Z,W] units and ALU.Trans units. All of the instruction mnemonics in this table have an OP2_INST_ or OP3_INST_ prefix that is not shown here.

Table 4.4 ALU Instructions (ALU.[X,Y,Z,W] and ALU.Trans Units)

Mnemonic	Description
Integer Operations	
AND_INT	Logical bit-wise AND.
ASHR_INT	Scalar arithmetic shift right. The sign bit is shifted into the vacated locations. src1 is interpreted as an unsigned integer. The component-wise shift right of each 32-bit value in src0 by an unsigned integer bit count is provided by the LSB 5 bits (0-31 range) in src1.selected_component, inserting 0.
CNDE_INT	Integer conditional move equal based on integer (either signed or unsigned).
CNDE	Conditional move equal based on floating point compare of first argument being equal to 0.0.
CNDGE_INT	Integer conditional move greater than or equal based on signed integer values.
CNDGE	Conditional move equal based on floating point compare of first argument being greater than, or equal to, 0.0.
CNDGT_INT	Integer conditional move greater than based on signed integer values.

Table 4.4 ALU Instructions (ALU.[X,Y,Z,W] and ALU.Trans Units) (Cont.)

Mnemonic	Description
CNDGT	Conditional move equal based on floating point compare of first argument being greater than 0.0.
KILLE_INT	Integer pixel kill equal. Set kill bit.
KILLGE_INT	Integer pixel kill greater or equal. Set kill bit.
KILLGE_UINT	Unsigned integer pixel kill greater or equal. Set kill bit.
KILLGT_INT	Integer pixel kill greater than. Set kill bit.
KILLGT_UINT	Unsigned integer pixel kill greater than. Set kill bit.
KILLNE_INT	Integer pixel kill not equal. Set kill bit.
LSHL_INT	Scalar logical shift left. Zero is shifted into the vacated locations. src1 is interpreted as an unsigned integer. If src1 is > 31, the result is 0x0.
LSHR_INT	Scalar logical shift right. Zero is shifted into the vacated locations. src1 is interpreted as an unsigned integer. If src1 is > 31, the result is 0x0.
MAX_INT	Integer maximum based on signed integer elements.
MAX_UINT	Integer maximum based on unsigned integer elements.
MIN_INT	Integer minimum based on signed integer elements.
MIN_UINT	Integer minimum based on signed unsigned integer elements.
MOV	Single-operand move.
NOP	No operation.
NOT_INT	Logical bit-wise NOT.
OR_INT	Logical bit-wise OR.
PREDE_INT	Integer predicate set equal. Update predicate register.
PRED_SETE_PUSH_INT	Integer predicate counter increment equal. Update predicate register.
PRED_SETGE_INT	Integer predicate set greater than or equal. Update predicate register.
PRED_SETGE_PUSH_INT	Integer predicate counter increment greater than or equal. Update predicate register.
PRED_SETGE_UINT	Unsigned integer predicate set greater than or equal. Update predicate register.
PRED_SETGT_INT	Integer predicate set greater than. Updates predicate register.
PRED_SETGT_PUSH_INT	Integer predicate counter increment greater than. Update predicate register.
PRED_SETGT_UINT	Unsigned integer predicate set greater than. Updates predicate register.
PRED_SETLE_INT	Integer predicate set if less than or equal. Updates predicate register.
PRED_SETLE_PUSH_INT	Predicate counter increment less than or equal. Update predicate register.
PRED_SETLT_INT	Integer predicate set if less than. Updates predicate register.
PRED_SETLT_PUSH_INT	Predicate counter increment less than. Update predicate register.
PRED_SETNE_INT	Scalar predicate set not equal. Update predicate register.
PRED_SETNE_PUSH_INT	Predicate counter increment not equal. Update predicate register.
SETE_INT	Integer set equal based on signed or unsigned integers.
SETGE_INT	Integer set greater than or equal based on signed integers.
SETGE_UINT	Integer set greater than or equal based on unsigned integers.
SETGT_INT	Integer set greater than based on signed integers.
SETGT_UINT	Integer set greater than based on unsigned integers.
SETNE_INT	Integer set not equal based on signed or unsigned integers.

Table 4.4 ALU Instructions (ALU.[X,Y,Z,W] and ALU.Trans Units) (Cont.)

Mnemonic	Description
SUB_INT	Integer subtract based on signed or unsigned integer elements.
XOR_INT	Logical bit-wise XOR.
Floating-Point Operations	
ADD	Floating-point add.
ADD_64	Floating-point 64-bit add.
CEIL	Floating-point ceiling function.
FLOOR	Floating-point floor function.
FRACT	Floating-point fractional part of src1.
KILLE	Floating-point kill equal. Set kill bit.
KILLGE	Floating-point pixel kill greater than equal. Set kill bit.
KILLGT	Floating-point pixel kill greater than. Set kill bit.
KILLNE	Floating-point pixel kill not equal. Set kill bit.
MAX	Floating-point maximum.
MAX_DX10	Floating-point maximum. DX10 implies slightly different handling of NaNs.
MIN	Floating-point minimum.
MIN_DX10	Floating-point minimum. DX10 implies slightly different handling of NaNs.
MUL	Floating-point multiply. 0*anything = 0.
MUL_IEEE	IEEE Floating-point multiply. Uses IEEE rules for 0*anything.
MULADD	Floating-point multiply-add (MAD).
MULADD_D2	Floating-point multiply-add (MAD), followed by divide by 2.
MULADD_M2	Floating-point multiply-add (MAD), followed by multiply by 2.
MULADD_M4	Floating-point multiply-add (MAD), followed by multiply by 4.
MULADD_IEEE	Floating-point multiply-add (MAD). Uses IEEE rules for 0*anything.
MULADD_IEEE_D2	IEEE Floating-point multiply-add (MAD), followed by divide by 2. Uses IEEE rules for 0*anything.
MULADD_IEEE_M2	IEEE Floating-point multiply-add (MAD), followed by multiply by 2. Uses IEEE rules for 0*anything.
MULADD_IEEE_M4	IEEE Floating-point multiply-add (MAD), followed by multiply by 4. Uses IEEE rules for 0*anything.
PRED_SET_CLR	Predicate counter clear. Update predicate register.
PRED_SET_INV	Predicate counter invert. Update predicate register.
PRED_SET_POP	Predicate counter pop. Updates predicate register.
PRED_SET_RESTORE	Predicate counter restore. Update predicate register.
PRED_SETE	Floating-point predicate set equal. Update predicate register.
PRED_SETE_PUSH	Predicate counter increment equal. Update predicate register.
PRED_SETGE	Floating-point predicate set greater than equal. Update predicate register.
PRED_SETGE_PUSH	Predicate counter increment greater than equal. Update predicate register.
PRED_SETGT	Floating-point predicate set greater than. Update predicate register.
PRED_SETGT_PUSH	Predicate counter increment greater than. Update predicate register.
PRED_SETNE	Floating-point predicate set not equal. Update predicate register.
PRED_SETNE_PUSH	Predicate counter increment not equal. Update predicate register.

Table 4.4 ALU Instructions (ALU.[X,Y,Z,W] and ALU.Trans Units) (Cont.)

Mnemonic	Description
RNDNE	Floating-point Round-to-Nearest-Even Integer.
SETE	Floating-point set equal.
SETE_DX10	Floating-point equal based on floating-point arguments. The result, however, is integer.
SETGE	Floating-point set greater than equal.
SETGE_DX10	Floating-point greater than or equal based on floating-point arguments. The result, however, is integer.
SETGT	Floating-point set greater than.
SETGT_DX10	Floating-point greater than based on floating-point arguments. The result, however, is integer.
SETNE	Floating-point set not equal.
SETNE_DX10	Floating-point not equal based on floating-point arguments. The result, however, is integer.
TRUNC	Floating-point integer part of src0.

4.8.1.1 KILL and PRED_SET* Instruction Restrictions

Only a pixel shader (PS) program can execute a pixel kill (KILL) instruction. This instruction is illegal in other program types. A KILL instruction is the last instruction in an ALU clause, because the remaining instructions executed in the clause do not reflect the updated valid state after the kill operation. Two KILL instructions cannot be co-issued.

The term PRED_SET* is any instruction that computes a new predicate value that can update the local predicate or active mask. Two PRED_SET* instructions cannot be co-issued. Also, PRED_SET* and KILL instructions cannot be co-issued. Behavior is undefined if any of these co-issue restrictions are violated.

4.8.2 Instructions for ALU.[X,Y,Z,W] Units Only

The instructions shown in Table 4.5 can be used only in a slot in the instruction group that is destined for one of the ALU.[X,Y,Z,W] units. None of these instructions are legal in an ALU.Trans unit. All of the instruction names in Table 4.5 are preceded by OP2_INST_.

Table 4.5 ALU Instructions (ALU.[X,Y,Z,W] Units Only)

Mnemonic	Description
Reduction Operations	
ADD_INT	Integer add based on signed or unsigned integer elements.
CUBE	Cubemap instruction. It takes two source operands (SrcA = Rn.zzxy, SrcB = Rn.yxzz). All four vector elements must share this instruction. Output clamp and modifier do not affect FaceID in the resulting W vector element.

Table 4.5 ALU Instructions (ALU.[X,Y,Z,W] Units Only) (Cont.)

Mnemonic	Description
DOT4	Four-element dot product. The result is replicated in all four vector elements. All four vector elements must share this instruction. Only the PV.X register element holds the result; the processor is responsible for selecting this swizzle code in the bypass operation.
DOT4_IEEE	Four-element dot product.The result is replicated in all four vector elements. Uses IEEE rules for 0*anything. All four ALU.[X,Y,Z,W] instructions must share this instruction. Only the PV.X register element holds the result; the processor is responsible for selecting this swizzle code in the bypass operation.
FLT32_TO_FLT64	Floating-point 32-bit convert to 64-bit floating-point.
FLT64_TO_FLT32	Floating-point 64-bit convert to 32-bit floating-point.
FRACT_64	Positive fractional part of a 64-bit floating-point value.
FREXP_64	Split double-precision floating-point into fraction and exponent.
LDEXP_64	Combine separate fraction and exponent into double-precision.
MAX4	Four-element maximum.The result is replicated in all four vector elements. All four vector elements must share this instruction. Only the PV.X register element holds the result, and the processor is responsible for selecting this swizzle code in the bypass operation.
MUL_64	Floating-point multiply, 64-bit.
MULADD_64	Floating-point multiply-add, 64-bit.
PRED_SETE_64	Floating-point predicate set if equal, 64-bit.
PRED_SETGE_64	Floating-point predicate set if greater than or equal, 64-bit.
PRED_SETGT_64	Floating-point predicate set, if greater than, 64-bit.
Non-Reduction Operations	
MOVA	Round floating-point to the nearest integer in the range [-256, +255], and copy to address register (AR) and to a GPR.
MOVA_FLOOR	Truncate floating-point to the nearest integer in the range [-256, +255], and copy to address register (AR) and to a GPR.
MOVA_INT	Clamp signed integer to the range [-256, +255], and copy to address register (AR) and to a GPR.

4.8.2.1 Reduction Instruction Restrictions

When any of the reduction instructions (DOT4, DOT4_IEEE, CUBE, and MAX4) is used, it must be executed on all four elements of a single vector. Reduction operations compute only one output; so, ensure that the values in the OMOD and CLAMP fields are the same for all four instructions.

4.8.2.2 MOVA* Restrictions

All MOVA* instructions shown in Table 4.5 write vector elements of the address register (AR). They do not need to execute on all of the ALU.[X,Y,Z,W] operands at the same time. One ALU.[X,Y,Z,W] unit can execute a MOVA* operation while other ALU.[X,Y,Z,W] units execute other operations. Software can issue up to four MOVA instructions in a single instruction group to change all four elements of the AR register. A MOVA* instruction issued in ALU.X writes AR.X, regardless of any GPR write mask used.

Predication is allowed on any MOVA* instruction.

MOVA* instructions must not be used in an instruction group that uses AR indexing in any slot (even slots that are not executing MOVA*, and even for an index not being changed by MOVA*). To perform this operation, split it into two separate instruction groups: the first performing a MOV with GPR-indexed source into a temporary GPR, and the second performing the MOVA* on the temporary GPR.

MOVA* instructions produce undefined output values. To inhibit the GPR destination write, clear the WRITE_MASK field for any MOVA* instruction. Do not use the corresponding PV vector element(s) in the following ALU instruction group.

4.8.3 Instructions for ALU.Trans Units Only

The instructions in Table 4.6 are legal only in an instruction-group slot destined for the ALU.Trans unit. If any of these instructions is executed, the instruction-group slot is allocated to the ALU.Trans unit immediately. An ALU.Trans operation must be specified as the last instruction slot in an instruction group; so, using one of these instructions effectively marks the end of the instruction group.

Table 4.6 ALU Instructions (ALU.Trans Units Only)

Mnemonic	Description
Integer Operations	
FLT_TO_INT	Floating-point input is converted to a signed integer value using truncation. If the value does fit in 32 bits, the low-order bits are used.
FLT_TO_UINT	Convert floating point to integer.
INT_TO_FLT	The input is interpreted as a signed integer value and converted to a floating-point value.
MULHI_INT	Scalar multiplication. The arguments are interpreted as signed integers. The result represents the high-order 32 bits of the multiply result.
MULHI_UINT	Scalar multiplication. The arguments are interpreted as unsigned integers. The result represents the high-order 32 bits of the multiply result.
MULLO_INT	Scalar multiplication. The arguments are interpreted as signed integers. The result represents the low-order 32 bits of the multiply result.
MULLO_UINT	Scalar multiplication. The arguments are interpreted as unsigned integers. The result represents the low-order 32 bits of the multiply result.
RECIP_INT	Scalar integer reciprocal. The argument is interpreted as a signed integer. The result is interpreted as a fractional signed integer. The result for 0x0 is undefined.
RECIP_UINT	Scalar unsigned integer reciprocal. The argument is interpreted as an unsigned integer. The result is interpreted as a fractional unsigned integer. The result for 0x0 is undefined.
UINT_TO_FLT	The input is interpreted as an unsigned integer value and converted to a float.
Floating-Point Operations	
COS	Scalar cosine function. Valid input domain [-PI, +PI].
EXP_IEEE	Scalar Base2 exponent function.
LOG_CLAMPED	Scalar Base2 log function.
LOG_IEEE	Scalar Base2 log function.

Table 4.6 ALU Instructions (ALU.Trans Units Only) (Cont.)

Mnemonic	Description
MUL_LIT	Scalar multiply. The result is replicated in all four vector elements. It is used primarily when emulating a LIT operation (Blinn's lighting equation). Zero times anything is zero. Instruction takes three inputs.
MUL_LIT_D2	MUL_LIT operation, followed by divide by 2.
MUL_LIT_M2	MUL_LIT operation, followed by multiply by 2.
MUL_LIT_M4	MUL_LIT operation, followed by multiply by 4.
RECIP_CLAMPED	Scalar reciprocal.
RECIP_FF	Scalar reciprocal.
RECIP_IEEE	Scalar reciprocal.
RECIPSQRT_CLAMPED	Scalar reciprocal square root.
RECIPSQRT_FF	Scalar reciprocal square root.
RECIPSQRT_IEEE	Scalar reciprocal square root.
SIN	Scalar sin function. Valid input domain [-PI, +PI].
SQRT_IEEE	Scalar square root. Useful for normal compression.

4.8.3.1 ALU.Trans Instruction Restrictions

At most one of the transcendental and integer instructions shown in Table 4.6 can be specified in a given instruction group, and it must be specified in the last instruction slot.

4.9 ALU Outputs

The following subsections describe the output modifiers, destination registers, predicate output, NOP instruction, and MOVA instructions.

4.9.1 Output Modifiers

Each ALU output passes through an output modifier before being written to the PV and PS registers and the destination GPRs. This output modifier works for floating-point outputs only.

The first part of the output modifier is to scale the result by a factor of 2.0 (either multiply or divide) or 4.0 (multiply only). For instructions with two source operands, this output modifier is specified in the instruction's OMOD field. For instructions with three source operands, the modifier is specified as part of the opcode. As a result, it is available only for certain instructions. The modifier works with floating-point values only; it is not valid for integer operations. For non-reduction operations, each instruction can specify a different value for OMOD. Reduction operations compute only one output. Each instruction for a reduction operation must use the same OMOD value (for instructions with two source operands).

The second part of the output modification is to clamp the result to [0.0, 1.0]. This is controlled by the instruction's CLAMP field. The clamp modifier works only with

floating-point values; it is not valid, and should be disabled, for integer operations. For non-reduction operations, each instruction can specify a different value for CLAMP. Reduction operations only compute one output. Each instruction for a reduction operation must use the same CLAMP value.

4.9.2 Destination Registers

The results are written to PV or PS registers and to the destination GPR specified in the DST_GPR field of the instruction. The destination GPR can be relative to an index. To enable this, set the DST_REL bit, and specify an appropriate INDEX_MODE. The INDEX_MODE parameter is shared with the input operands for the instruction. If the resulting GPR address is not in [0, GPR_COUNT – 1], which are the declared GPRs for this thread, and are not in [127 – *N* + 1, 127], which are the *N* temporary GPRs, then no GPR write is performed; only PV and PS registers are updated.

Instructions with two source operands have a write mask, WRITE_MASK, that determines if the result is written to a GPR. The PV or PS registers result is updated even if WRITE_MASK is 0. Instructions with three source operands have no write mask; however, you can specify an out-of-bounds GPR destination to inhibit their write. For example, if the thread is using four clause temporaries and less than 124 GPRs, it is safe to use DST_GPR = 123 to ignore the result. Otherwise, you must sacrifice one of the temporary GPRs for instructions with three source operands. The PV or PS registers result is updated for instructions with three source operands even if the destination GPR address is invalid.

Two instructions running on the ALU.[X,Y,Z,W] units cannot write to the same GPR element. However, it is possible for ALU.Trans to write to the same GPR element as one of the operations running in ALU.[X,Y,Z,W]. This can be done either explicitly, as in:

```
GPR0.X <= GPR1.X
...
GPR0.X <= GPR2.X
```

or implicitly via relative addressing. If the ALU.Trans unit and one of the ALU.[X,Y,Z,W] units try to write to the same GPR element, the transcendental operation dominates, and the ALU.Trans result is written to the GPR element. This affects the GPR write only; the PV register reflects only the vector result.

4.9.3 Predicate Output

Instructions with two source operands that affect the internal predicate have two additional bits: UPDATE_PRED and UPDATE_EXEC_MASK. The UPDATE_PRED bit determines whether to write the updated predicate results internally (only valid until the end of the clause). If UPDATE_PRED is set, the new predicate takes effect on the next ALU instruction group. The UPDATE_EXEC_MASK bit determines whether to send the new predicate result back to the CF program. The active mask persists across clauses and is used by the CF program, but does not take affect until the end of the current ALU clause. UPDATE_PRED and

UPDATE_EXEC_MASK must be cleared for instructions that do not compute a new predicate result.

4.9.4 NOP Instruction

NOP instructions perform no writes to GPRs, and they invalidate PV and PS registers.

4.9.5 MOVA Instructions

MOVA* instructions update the constant register and AR. They are not designed to write values into the GPR registers. Writing to PV and PS registers and any write to a GPR has undefined results. It is strongly recommended that software clear the WRITE_MASK bit for any MOVA* instruction, and does not attempt to use the corresponding PV or PS register value in the following instruction. At most one MOVA instruction can be present in an instruction group.

4.10 Predication and Branch Counters

The processor maintains one predicate bit per pixel within an ALU clause. This predicate initially reflects the active Mask from the processor. The predicate can be updated during the ALU clause using various PRED_SET* and stack operations. The predicate bit does not persist past the end of an ALU clause. To carry a predicate across clauses, an ALU instruction group can update the active Mask that is used for subsequent clauses, as described in Section 4.9.3.

Each instruction can be conditioned on the predicate, using the instruction's PRED_SEL field. Different instructions in the same instruction group can be predicated differently. The predicate condition can be one of three values:

- PRED_SEL_OFF — Always execute the instruction.
- PRED_SEL_ZERO — Execute the instruction if the pixel's predicate bit is currently zero.
- PRED_ZEL_ONE — Execute the instruction if the pixel's predicate bit is currently one.

If an instruction is disabled by the predicate bit, then no GPR value is written, the PV and PS registers are not updated. Also, the PRED_SET*, MOVA, and KILL instructions, which have an effect on non-register state, have no effect for that pixel. An instruction that modifies the ALU predicate (for example: PRED_SET*) can choose to update the predicate bit using UPDATE_PRED, and it can separately choose to send a new active Mask based on the *computed* predicate using UPDATE_EXEC_MASK. An instruction can compute a new predicate and choose to update *only* the processor's active Mask. In this case, the processor sees the computed predicate, not the old predicate that persists.

Instruction groups that do not compute a new predicate result must clear the UPDATE_PRED and UPDATE_EXEC_MASK fields of their instructions. At most one

instruction in an instruction group can be a PRED_SET* instruction; thus, at most one instruction can have either of these bits set.

In addition to predicates, flow control relies on maintenance of branch counters. Branch counters are maintained in normal GPRs and are manipulated by the various predicate operations. Software can inhibit branch-counter updating by simply disabling the GPR write for the operation, using the instruction's WRITE_MASK field.

4.11 Adjacent-Instruction Dependencies

Register write or read dependencies can exist between two adjacent ALU instruction groups. When an ALU instruction group writes to a GPR, the value is not immediately available for reading by the next instruction group. In most cases, the processor avoids stalling by detecting when the second instruction group references a GPR written by the first instruction group, then substituting the dependent register read with a reference to the previous ALU.[X,Y,Z,W] or ALU.Trans result (in the PV or PS registers). If the write is predicated, a special override is used to ensure the value is read from the original register or PV or PS depending on the previous predication. A compiler does not need to do anything special to enable this behavior. However, there are cases where this optimization is not available, and the compiler must either insert a NOP or otherwise defer the dependent register read for one instruction group.

Application software does not need to do anything special in any of the following cases. These are cases in which the processor explicitly detects a dependency and optimizes the instruction-group pair to avoid a stall.

- Write to R*N* or R*N*[LOOP_INDEX], followed by read from R*M* or R*M*[LOOP_INDEX]; *N* may or may not equal *M*.
- Write to R*N*[GPR_INDEX], followed by read from R*M*[gpr_index]; *N* may or may not equal *M*.

Application software also does not need to do anything special in the following cases. In these cases, the processor does nothing special, but the pairing is legal because there is no aliasing or dependency.

- Write to R*N*, followed by read from R*M*[GPR_INDEX]. The compiler ensures *N* != *M* + GPR_INDEX.
- Write to R*N*[LOOP_INDEX], followed by read from R*M*[GPR_INDEX]. The compiler ensures *N* + loop_index != *M* + GPR_INDEX.
- Write to R*N*[GPR_INDEX], followed by read from R*M*. The compiler ensures *N* + GPR_INDEX != *M*.
- Write to R*N*[GPR_INDEX], followed by read from R*M*[LOOP_INDEX]. The compiler ensures *N* + GPR_INDEX != *M* + LOOP_INDEX.

To illustrate, the following example instruction-group pairs are legal.

```
R1 = R0;
R2 = R1;// rewritten to R2 = PV/PS.
```

```
    R2 = R0;
    R2 = R1 predicated;
    R3 = R2;// rewritten to R3 = PV/PS, override for R2.
    R1[gpr_index] = R0;
    R2 = R1[gpr_index];// rewritten to R2 = PV/PS.
    R2[gpr_index] = R0;
    R2[gpr_index] = R1 predicated;
    R3 = R2[gpr_index];// rewritten to R3 = PV/PS, override for
R2[GPR_INDEX].
    R1[gpr_index] = R0;// compiler guarantees GPR_INDEX != 0.
    R2 = R1;// never a dependent read.
    R1[loop_index] = R0;// LOOP_INDEX might be 0.
    R2 = R1;// can be dependent, the processor will detect if it is.
```

The following example instruction-group pairs are illegal.

```
    R1[gpr_index] = R0;// GPR_INDEX might be zero.
    R2 = R1;// can be dependent, the processor doesn't catch this.
    R1[gpr_index] = R0;// GPR_INDEX can equal loop_index.
    R2 = R1[loop_index];// can be dependent, the processor doesn't catch
this.
```

4.12 Double-Precision Floating-Point Operations

Unless otherwise stated in this document, floating-point operations and operands are single-precision. There are, however, some double-precision floating-point instructions. These double-precision instructions support higher precision calculations and conversion between single- and double-precision formats. Basic add, multiply, and multiply-add operations are implemented using the IEEE 754 round-to-nearest mode. Note that double-precision floating-point (DPFP) is not available on all R7xx products; therefore, check the specifications of your particular product to determine if DPFP is supported.

The mnemonics and 64-bit operands of double-precision instructions contain the suffix `_64`. The instructions occupy either two or four slots in an instruction group (Section 4.3, "ALU Instruction Slots and Instruction Groups," page 4-3), as specified in their descriptions in Section 9.2, "ALU Instructions," page 9-48. All source operands are double-precision numbers, except 32-bit operands in format-conversion operations. Source operands are stored in GPRs as a 32-bit high (most-significant) doubleword and a 32-bit low (least-significant) doubleword, in elements ALU.[X,Y] and/or elements ALU.[Z,W]. The result of a double-precision operation is also stored similarly, but the order of doublewords is usually inverted with respect to the source operands.

4.13 Wavefront Synchronization Within a Work-Group

Wavefronts within a work-group can synchronize with each other to share data or to provide mutually exclusive access to shared data. Two synchronization methods area available:

- `inst_group_barrier` (`sync_barrier`) – when a wavefront reaches this point, it remains inactive until all wavefronts in the group reach it, then all become active.
- `inst_group_seq_begin/end` (`sync_wave_sequential`) – when a wavefront reaches this point, it waits until all wavefronts arrive. Then, the first wavefront

in the work-group is activated. It proceeds until it executes group_seq_end. The second wavefront then is activated (the first wave continues). This is repeated for all wavefronts in the work-group.

When a wavefront executes an instruction, the wavefront becomes inactive and remains inactive until all other wavefronts in the work-group have completed execution of the instruction. At this point, all wavefronts become active again.

4.13.1 ALU Rounding and Denormals

The default rounding modes for a kernel and default denorm handling are set in the SQ_PGM_RESOURCES2 register. This applies to all ALU instructions. Within an ALU clause, the kernel can include a SET_MODE instruction, which temporarily overrides the rounding and denorm modes. SET_MODE affects other instructions in the current group (x,y,z,w,t) and stays in effect either until the next SET_MODE or until the end of the clause, when it returns to the default register value.

Note that the SET_MODE instruction changes all round and denorm values; it is not possible to override some and leave others at the default setting.

Round modes:

- Round to nearest
- Round toward 0 (truncate)
- Round toward +infinity
- Round toward -infinity

Denormal handling:

single_denorm_flush_input	(on/off)
single_denorm_force_underflow_to_zero	(on/off)
double_denorm_flush_input	(on/off)
double_denorm_force_underflow_to_zero	(on/off)

4.13.2 Floating-Point Flags

This feature exists only on GPUs that support double-precision floating-point operations.

Each floating point operation generates a six-bit flag per work-item (combined result of x,y,z,w); this indicates which (if any) floating-point exception occurred during this instruction group. These flags can be moved to, and from, GPRs.

The flags are:

[5]	[4]	[3]	[2]	[1]	[0]
Inexact	Underflow	Overflow	Division by Zero	Denormal	Invalid Operation

These flags are cleared to zero automatically at the beginning of each ALU clause. Flags are set by every instruction, and the flags are accumulated by

logically ORing the flags from the current instruction with the results from previous ones.

Two ALU instructions operate on these flags:

OP2_INST_STORE_FLAGS: copies the flag values into the destination GPR.

OP2_INST_LOAD_STORE_FLAGS: copies the flag values into destination GPR, and copies the value from source GPR into the flags, overwriting the previous flag values.

Chapter 5
Fetch Through Vertex Cache Clauses

Software initiates a fetch through a vertex cache clause with the VC or TC control-flow instructions, both of which use the CF_DWORD[0,1] microcode formats. Instructions for fetches through a vertex cache within the clause use the VTX_DWORD0, 1, and 2, and VTX_WORD2 microcode formats, with a fourth (high-order) doubleword of zeros.

A fetch through a vertex cache clause consists of instructions that fetch vertices from a vertex buffer based on GPR addresses or memory read instructions. The clause can be at most 16 instructions long. Vertex fetches using the semantic table use the VTX_WORD1_SEM microcode format to specify the eight-bit semantic ID. The semantic table indicates the ID of the GPR to which the data is written. All other vertex fetches use the VTX_WORD1_GPR microcode format, which specifies the destination GPR directly.

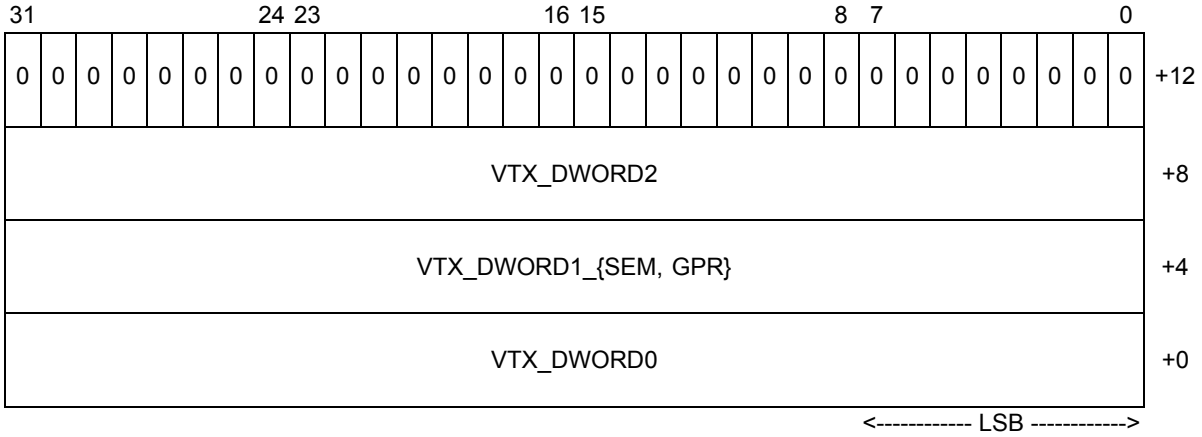
Each instruction for a fetch through a vertex cache clause has a BUFFER_ID field that specifies the buffer containing the constants for a fetch through a vertex cache clause, and an OFFSET field for the offset at which reading of the value in the buffer is to begin. The instruction uses the SRC_REL bit to determine whether to use the SRC_GPR specified in the instruction (bit is cleared), or (if the bit is set) to use SRC_GPR + the loop index (aL). The result of non-semantic fetches is written to DST_GPR. The DST_REL bit determines if the address is absolute or relative to the loop index (aL). Semantic fetches determine the destination GPR by reading the entry in the semantic table that is specified by the instruction's SEMANTIC_ID field. The source index and the four-element result from memory can be swizzled.

The source value can be fetched from any element of the source GPR using the instruction's SRC_SEL_X field. Unlike texture instructions, the SRC_SEL_X field cannot be a constant; it must refer to a vector element of a GPR. The destination swizzle is specified in the DST_SEL_[X,Y,Z,W] fields; the swizzle can write any of the fetched elements, the value 0.0, or the value 1.0. To disable an element write, set the DST_SEL_[X,Y,Z,W] fields to the SEL_MASK value

Individual instructions for fetches through a vertex cache clause cannot be predicated; predicated fetches through vertex cache clauses must be done at the CF level by making the instruction for a fetch through a vertex cache clause conditional. All vertex instructions in the clause are executed with the conditional constraint specified by the CF instruction.

5.1 Microcode Formats for Fetches Through a Vertex Cache Clause

Microcode formats for fetches through a vertex cache clause are organized in 4-tuples of 32-bit doublewords. Figure 5.1 shows the doubleword layouts in memory. The +0, +4, +8, and +12 indicate the relative byte offset of the doublewords in memory; {SEM, GPR} indicates a choice between the strings SEM and GPR; LSB indicates the least-significant (low-order) byte; and the high-order doubleword is padded with zeros.



5.2 Constant Sharing

ES, GS, and VS kernels can, on a per-clause basis, select to either access their own constant buffers or those of the other shader type.

ES/VS can use their own or GS constants, and GS can use its own or ES/VS ones. This is provided for cases when the GS and VS shaders can be merged into a single hardware shader stage.

This capability is activated by setting the `ALT_CONSTS` bit in the `SQ_VTX_WORD2`.

Chapter 6
Texture Cache Clauses

Software initiates a fetch through a texture cache clause with the TC control-flow instruction, which uses the CF_WORD[0 1] microcode formats. Instructions for a fetch through a texture cache clause use the TEX_WORD[0,1,2] microcode formats, with a fourth (high-order) doubleword of zeros.

A fetch through a texture cache clause consists of instructions that lookup texture elements, called *texels*, based on a GPR address. Texture instructions are used for both fetches through a texture cache clause and constant-fetch operations. A texture clause can be at most eight instructions long.

Each texture instruction has a RESOURCE_ID field, which specifies an ID for the buffer address, size, and format to read, and a SAMPLER_ID field, which specifies an ID for filter and other options. The instruction reads the texture coordinate from the SRC_GPR. The SRC_REL bit determines if the address is absolute or relative to the loop index (aL). The result is written to the DST_GPR. The DST_REL bit determines if the address is absolute or relative to the loop index (aL). Both the fetch coordinate and the resulting four-element data from memory can be swizzled. The source elements for the swizzle are specified with the SRC_SEL_[X,Y,Z,W] fields; a source element also can use the swizzle constants 0.0 and 1.0. The destination elements for the swizzle are specified with the DST_SEL_[X,Y,Z,W] fields; it can write any of the fetched elements, the value 0.0, or the value 1.0. To disable an element write, set the DST_SEL_[X,Y,Z,W] fields to the SEL_MASK value.

Individual texture instructions cannot be predicated; predicated fetches through a texture cache clause must be done at the CF level, by making the texture-clause instruction conditional. All texture instructions in the clause are executed with the conditional constraint specified by the CF instruction.

6.1 Microcode Formats for Fetches Through a Texture Cache Clause

Microcode formats for fetches through a texture cache clause are organized in 4-tuples of 32-bit doublewords. Figure 6.1 shows the doubleword layouts in memory, in which +0, +4, +8, and +12 indicate the relative byte offset of the doublewords in memory; LSB indicates the least-significant (low-order) byte; and the high-order doubleword is padded with zeros.

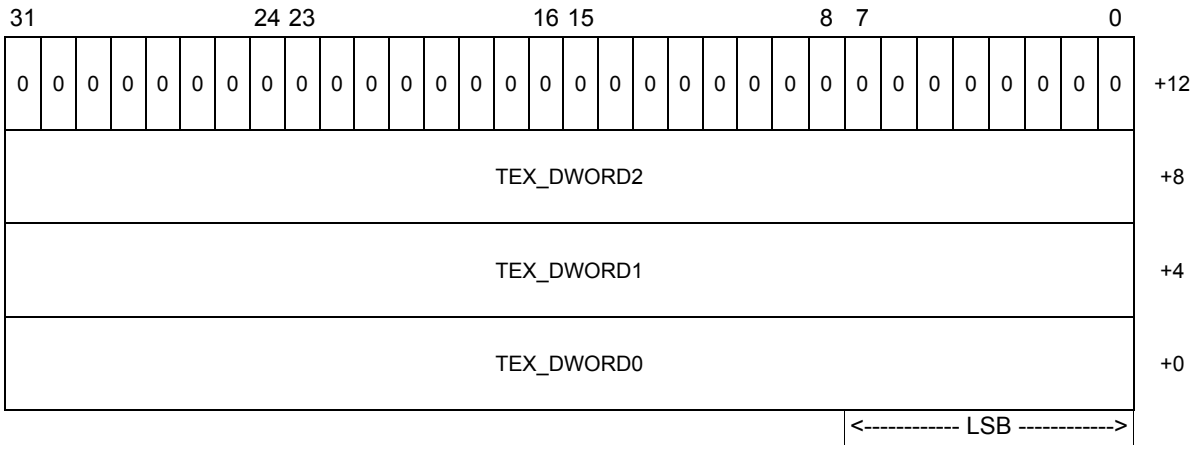


Figure 6.1 Microcode-Format 4-Tuple for Fetches Through a Texture Cache Clause

6.2 Constant-Fetch Operations

The buffer ID space, specified in the RESOURCE_ID field of the TEX_WORD0 microcode format, is eight bits wide, allowing a constant and a fetch through a texture cache clause to coexist in the same ID space. The two types of fetches differ according to the manner in which their resources are organized.

6.3 FETCH_WHOLE_QUAD and WHOLE_QUAD_MODE

The processor executes pixel threads in groups of four, called *quads*. Sometimes the edge of a primitive (such as a triangle) cuts through a quad so that some pixels in the quad are outside the primitive. The threads executing these pixels are placed in the invalid state.

The following two features are sometimes helpful when computing the inputs to gradient operations:

- Instructions for fetch through a texture cache clause contain a bit (FETCH_WHOLE_QUAD) if this bit is set the fetches from invalid pixels are still executed.
- Within a quad, some pixels may have the active Mask set to execute while others may be set to skip. Normally the pixels which are set to skip, to do NOT execute instructions, however if the WHOLE_QUAD_MODE bit is set, the all four thread in the quad execute if at least one pipeline is set to execute.

6.4 Constant Sharing

ES, GS, and VS kernels can, on a per-clause basis, either their own texture and sampler constants or those of the other shader type.

ES/VS can use their own or use GS constant buffers, and GS can use its own or ES/VS ones. This is for cases when the GS and VS shaders can be merged into a single hardware shader stage.

This capability is activated by setting the ALT_CONSTS bit in the SQ_TEX_WORD0.

Chapter 7
Memory Read Clauses

Software initiates a memory read clause with the VC or TC control-flow instructions, both of which use CF_DWORD[0,1] microcode formats. Memory read instructions within the clause use the MEM_RD_DWORD[0,1,2], with a fourth double-dword of zeros.

A memory-read clause consists of instructions that fetch data from one of three types of buffers:

- Scratch
- Reduction
- Scatter (general read/write)

Reads from these buffer types can be intermixed within a clause, and the clause can consist of up to 16 memory read instructions. Memory read instructions can be in the same clause as instructions for fetches through texture or vertex cache clauses, but not in the same clause as global data share instructions.

Many of the instruction word fields are identical to those of an instruction for a fetch through a vertex cache clause. See Chapter 5, “Fetch Through Vertex Cache Clauses,” for more detail. The fields that differ are:

- elem_size
- uncached
- array_base
- array_size
- indexed

Uncached is described in the next section.

The other four are identical to the fields with the same name in the EXPORT instructions that write to those memory buffers.

7.1 Memory Address Calculation

Scratch:

```
memory_address = Array_base + (burst_counter + Indexed*SRC_GPR) * vectorsize * (elemsize+1)
                 + component_offset + ThreadInWavefront*vectorsize*(elemsize+1)
```

Before this calculation, SRC_GPR is clamped to the range: [0, array_size-1].

Vectorsize is the number of threads in a wavefront. ThreadInWavefront is a constant value unique to each thread: 0...63. The component_offset is 0 for x, 1 for y, 2 for z, and 3 for w.

Scatter:

```
memory address = Array_base + SRC_GPR + (burst_counter * (elemsize+1))
```

Array-size is not used.

7.2 Cached and Uncached Reads

Memory read instructions have a bitfield that controls whether to use or bypass the on-chip memory cache: MEM_RD_DWORD0.UNCACHED. This bit must be set whenever a kernel writes data to a buffer and reads it back within the same invocation of the kernel. It can only be cleared when data written to memory has been flushed to memory before the kernel is executed.

7.3 Burst Memory Reads

Burst memory reads allow up to 16 consecutive locations to be read into up to 16 consecutive GPRs. The burst count is specified in the MEM_RD_DWORD0.BURST_CNT field. For each iteration of the burst, the DST_GPR is incremented by 1, and the ARRAY_BASE is incremented by (elemsize + 1) * vectorSize.

Chapter 8
Data Share Operations

Local data share (LDS) is a very low-latency, RAM scratchpad for temporary data with at least one order of magnitude higher effective bandwidth than direct, uncached global memory. It permits sharing of data between work-items in a work-group, as well as holding parameters for pixel shader parameter interpolation. Unlike read-only caches, the LDS permits high-speed write-to-read re-use of the memory space (full gather/read/load and scatter/write/store operations).

8.1 Overview

Figure 8.1 shows the conceptual framework of the LDS is integration into the memory of AMD GPUs using OpenCL.

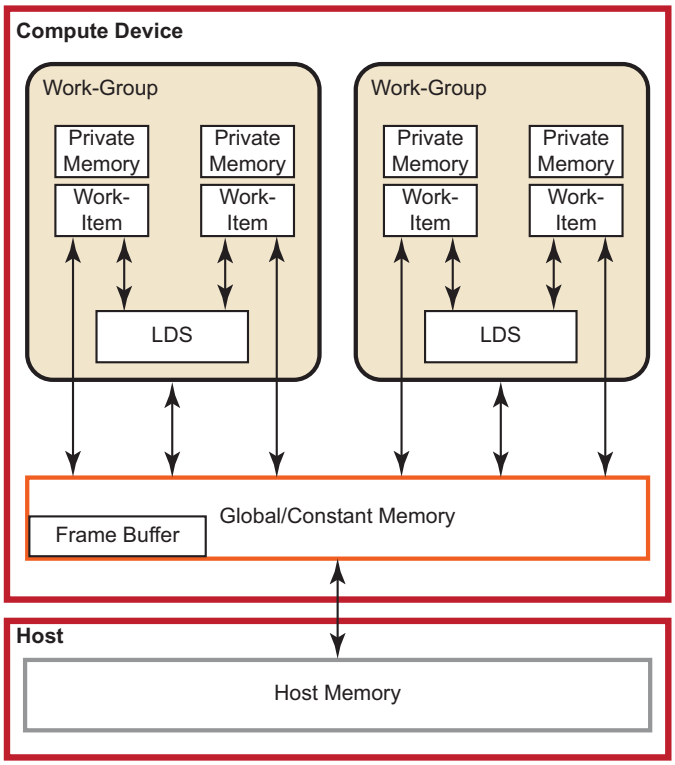


Figure 8.1 High-Level Memory Configuration

Physically located on-chip, directly next to the ALUs, the LDS is approximately two orders of magnitude faster than global memory (assuming no bank conflicts).

There are 32 kB memory per compute unit, segmented into 32 or 16 banks (depending on the GPU type) of 1 k dwords (for 32 banks) or 2 k dwords (for 16 banks). Each bank is a 256x32 two-port RAM (1R/1W per clock cycle). Dwords are placed in the banks serially, but all banks can execute a store or load simultaneously. Wavefronts are split over two or four banks, depending on the GPU. One work-group can request up to 32 kB memory. Reads across wavefront are dispatched over four cycles in waterfall.

The high bandwidth of the LDS memory is achieved not only through its proximity to the ALUs, but also through simultaneous access to its memory banks. Thus, it is possible to concurrently execute 16 write or read instructions, each nominally 32-bits; extended instructions, read2/write2, can be 64-bits each. If, however, more than one access attempt is made to the same bank at the same time, a bank conflict occurs. In this case, for indexed and atomic operations, hardware prevents the attempted concurrent accesses to the same bank by turning them into serial accesses. This decreases the effective bandwidth of the LDS. For maximum throughput (optimal efficiency), therefore, it is important to avoid bank conflicts. For direct reads/writes, the developer must avoid bank conflicts by selecting strides. A knowledge of request scheduling and address mapping is key to achieving this.

8.2 Dataflow in Memory Hierarchy

Figure 8.2 is a conceptual diagram of the dataflow within the memory structure.

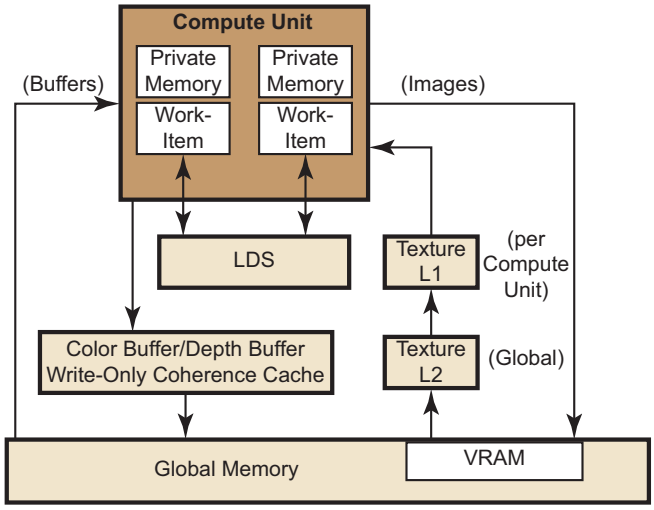


Figure 8.2 Memory Hierarchy Dataflow

To load data into LDS from global memory, it is read from global memory and placed into the work-item's registers; then, a store is performed to LDS. Similarly, to store data into global memory, data is read from LDS and placed into the work-item's registers, then placed into global memory. To make effective use of the LDS, an algorithm must perform many operations on what is transferred between global memory and LDS.

LDS atomics are performed in the LDS hardware. (Thus, although ALUs are not directly used for these operations, latency is incurred by the LDS executing this function.) If the algorithm does not require write-to-read reuse (the data is read only), it usually is better to use the image dataflow (see right side of Figure 8.2) because of the cache hierarchy.

Actually, buffer reads may use L1 and L2. When caching is not used for a buffer, reads from that buffer bypass L2; however, there is data sharing in L1 within a clause. After a buffer read, the line is invalidated; then, on the next read, it is read again (from the same wavefront or from a different clause). After a buffer write, the changed parts of the cache line are written to memory.

Buffers and images are written through the CB/DB cache, but this is flushed immediately after an image write.

The data in private memory is first placed in registers. If more private memory is used than can be placed in registers, or dynamic indexing is used on private arrays, the overflow data is placed (spilled) into scratch memory. Scratch memory is a private subset of global memory, so performance can be dramatically degraded if spilling occurs.

Global memory can be in the high-speed GPU memory (VRAM) or in the host memory, which is accessed by the PCIe bus. A work-item can access global memory either as a buffer or a memory object. Buffer objects are generally read and written directly by the work-items. Data is accessed through the L2 and L1 data caches on the GPU, but immediately invalidated at the end of a clause. Thus, data reuse is available within a wavefront for a given clause. This limited form of caching provides read coalescing among work-items in a wavefront. Similarly, writes are executed through the “fast-path” (depth buffer or DB) or “complete-path” (color buffer or CB), which have write-only caches that are invalidated, and all update bits are sent to memory at the end of a clause. The DB is the raw, high-speed, 32-bit only data write path. The CB is used for format conversion and atomics.

Global atomic operations are executed through the complete-path; the CB caches perform the atomic. Atomic operations in which the return value is not used (“fire-and-forget”) can be pipelined, and the work-item does not have to wait for the atomic to complete before continuing. If the return value is used, the work-item must wait for the atomic to complete, the line to be flushed, and a read from global memory.

Image objects are limited to read-only or write-only (no concurrent r/w). Thus, on reads, the data is cached through the L2 and L1 data caches; on writes, the data is cached through the CB/DB buffers.

8.3 LDS Access

The LDS is accessed using only ALU instructions. These instructions can direct the ALUs to read out up to three dwords per thread (1 address + 2 data) and store them into the LDS. The data is read from the GPRs similar to a MOVA,

then transferred to the LDS input-queue (IQ). The address and data are then written into the RAMs over as many cycles are necessary to avoid write port conflicts. There are three ways to read data out of the LDS in one of three ways:

- Direct
- Parameter
- Indexed or atomic

8.3.1 Direct Reads

The address comes directly from the instruction and has a uniform stride from thread to thread in the wave. Reads bypass the output queue and are directed straight to the shader processors. These reads can be part of any ALU operation, but must not cause LDS RAM read-port conflicts (bank conflicts), regardless of active mask.

Direct reads have an address and stride in the instruction word and can read src0 and src1 from the LDS so long as they do not conflict on LDS banks with themselves or each other (based on the strides and ignoring the active mask). Any ALU instruction in slot XYZW (not T) then can select src0 or src1 instead of a GPR or constant as a source operand by selecting src0 or src1 through the ALU_WORD0 instruction (see page 10-23). Direct reads require many extra instruction fields. These are held in literal-constant 0 (xy). When an instruction uses `lds_directa` or `lds_directb`, Literal0 exists and defines the data on the A and B busses; it is not available as a constant. Each pixel receives a unique value (unless stride = 0).

The instruction's bank-swizzle must be set to place src0 on cycle 0, and src1 on cycle 1.

Literal0 holds:

A Bus

[12:0] `offset_a` — Dword offset

[19:13] `stride_a` — Dword stride. Work-items in a wavefront must not conflict. Legal values are: 0, 1, 3, … (any odd number). A single direct fetch reads 16 dwords per cycle out of 32 banks.

[21:20] reserved.

[22] `thread_rel_a` — Add (wavefront_in_work-group * vectorsize * stride) to `offset_a`.

[31:23] reserved.

B Bus

[44:32] `offset_b` — Dword offset

[51:45] `stride_b` — Dword stride. Work-items in a wavefront must not conflict. Legal values are: 0, 1, 3, ... (any odd number). A single direct fetch reads 16 dwords per cycle out of 32 banks.

[53:52] reserved.

[54] thread_rel_b — add (wavefront_in_work-group * vectorsize * stride) to offset_b.

[62:55] reserved.

[63] direct_read_32 — Read 32 dwords of src0 in one cycle, then read 32 dwords of src1 in the next cycle, then read the last 32 dwords of src0 in next cycle, and finally read the last 32 dwords of src1 in fourth cycle (rather than 16 dwords of src0 and src1 in same cycle). In this mode, the legal strides are: 0, 1, or any number that is not an even multiple of 4.

8.3.2 Parameter Reads (Into Interpolation Instructions)

Parameter values are read directly out of the LDS for parameter interpolation. Unlike normal direct reads, the data is grouped so the four values sent to each shader processor are shared by the four pixels in the quad (since they all come from the same primitive). The data can represent either XY or ZW of [P0, P1-P0, and P2-P0]. Direct parameter reads are available only to the following instructions: interp_xy and interp_zw, interp_x, and interp_z. This generally is used in the graphics, not the OpenCL paths.

Parameter interpolation instructions interpolate two parameters (either XY or ZW) in one set of four vector instructions. The parameter AB select must be the same in all slots. The source-GPR varies when selecting either barycentric input register I or J. The output of interp_xy appears in the X and Y slots; ZW in the Z and W slots. Other slots must be masked-out. The bank swizzle must be 210 (read the I/J GPR in the third cycle). Interp_xy or _zw must occupy all four instruction slots, even if only interpolating one parameter (write-mask out the other one if unused). These four slots interpolate two parameters.

```
X:  interp_xy (210) <dstgpr>.x, <srcIgpr>.*, param_(0..32)  // X = ytmp + paramA*J
Y:  interp_xy (210) <dstgpr>.y, <srcJgpr>.*, param_(0..32)  // Ytmp = paramA+paramB*I
Z:  interp_xy (210) <dstgpr>.z, <srcIgpr>.*, param_(0..32)  // result send to Y
W:  interp_xy (210) <dstgpr>.w, <srcJgpr>.*, param_(0..32)
```

8.3.3 LDS Parameters

The LDS has two busses to send parameter data to the shader processors (A and B bus). Each shader processor can read from one or the other bus, but not both. The two busses send parameters for two different primitives. Interpolation can run at full rate if no cycle has more than two unique primitives among the four pixel quads across the four shader processors. If there are more than two primitives, the SQ waterfalls the operation over two cycles.

Parameter interpolation can occur in two ways:

- Direct Parameter Reads: The shader program reads parameter data directly from the LDS at a fixed offset and stride per instruction. Each read returns two of the three parameters (XYZW or P0, P1-P0, or P2-P0). All of the addressing is handled in hardware.

- General Parameter Interpolation: This is the more flexible bus slower mode, in which the shader program computes the parameter's address per pixel in LDS and uses indexed-LDS-reads to retrieve the data. In this case, each pixel only receives one dword on each of the src0 and src1 busses per instruction.

Parameters are stored in LDS memory. Each parameter is 12 dwords: xyzw of P0, P1-P0, and P2-P0. They are stored in the order shown in Figure 8.3. Parameter data is always four-dword aligned. In general interpolation mode, the pixel shader must calculate the LDS address to read parameters:

LDS Address = `param_start_offset` + (attr# * NumPrimsInVec * 12dwords) + Prim#*12 + attributeoffset(0,4,8 = p0xyzw,p10xyzw,p21xyzw)

`Param_start_offset` is available as an inline-ALU-constant and is the value: `LDS_ALLOC_PS` (see Figure 8.3).

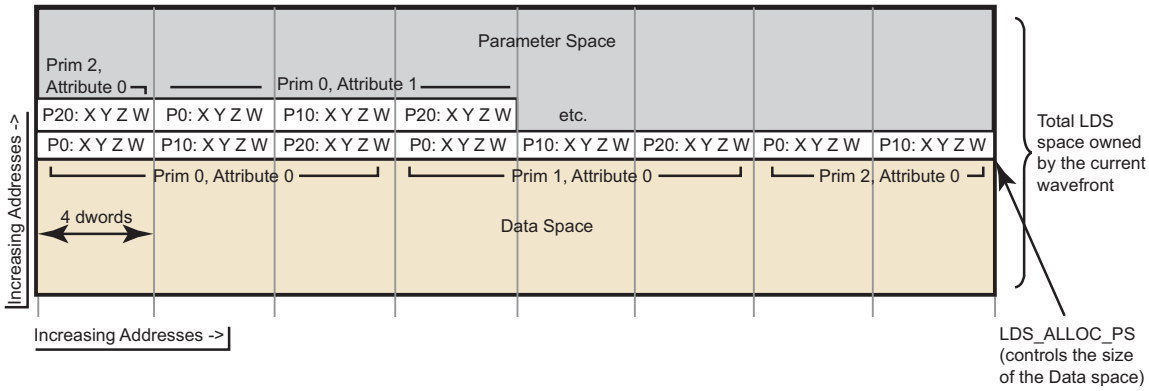


Figure 8.3 LDS Layout with Parameters and Data Share

8.3.4 Indexed and Atomic Reads

For indexed reads, the address comes from a GPR (see Figure 8.4). First, for each work-item, an `lds_indexed_op` instruction reads the LDS address from the GPR into the input queue. Then, the LDS performs the reads over as many cycles as necessary to avoid conflicts, and places the data into the output queue. Last, a separate ALU instruction uses the data from the output queue as an ALU source (via `src_sel` in the microcode).

For indexed writes, the address and data come from the GPRs. First an `lds_indexed_op` instruction reads GPR values (address and data) from the shader processor into the input queue. Then, the LDS performs the writes over as many cycles as necessary to avoid conflicts.

Atomic operations are a variant of the indexed read where the data after the GPRs is placed into the input queue. As data is read out from the LDS banks, it also passes through the atomic math unit (AMU) and is written back into the LDS. Optionally, the value before the arithmetic can be placed into the output queue to be read by a later ALU instruction.

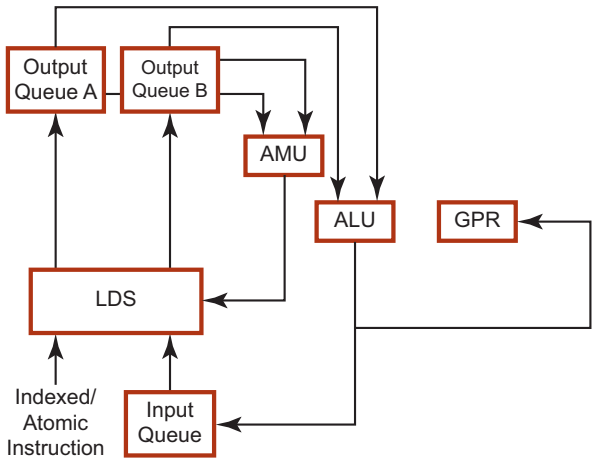


Figure 8.4 LDS Dataflow

Indexed and atomic instructions read the LDS and place data either into the AMU, in which case the result of the arithmetic manipulation is written back into the LDS, or it is placed in one of the two available output queues. If it is written back into the LDS, it requires another ALU instruction to get the data out. From the output queues, data is pulled by an ALU instruction into the ALU as a source (either src0 or src1) for the operation specified by the instruction.

Indexed reads cause data to be loaded into the output queue. This queue is logically split into an A and B queue. Any ALU instruction can select to use the head of the A or B queue as a source operand (much like direct LDS reads). The instruction's `src_sel` uses `lds_oqa` or `lds_oqb` to select this, and the FIFO is optionally popped after the entire XYZWT instruction group is executed. It is illegal for a shader to leave any data in the output queue at the end of the clause. Within an instruction group (xyzwt), all usages of the output queue must be of the same type (all pop or not pop; A and B must be the same.).

Note: this instruction does not require use of the literal constant.

8.4 Examples

8.4.1 LDS_READ dst

One GPR (dst) holds the LDS address from which to read. The value from the GPR is sent to the LDS unit; the LDS reads its memory at that address and places that value into output queue A.

8.4.2 LDS_WRITE dst, src0

One GPR (dst) holds the LDS address. Another GPR (src0) holds the data to write into LDS at that address. The two GPR values (address and data) are read out of the GPRs and sent to the LDS. When the LDS receives the address and data, it writes the data (src0) into the LDS at address (dst).

8.4.3 LDS_ADD dst, src0

One GPR (dst) holds the LDS address. Another GPR (src0) holds the data to add to the data already in LDS at that address. The two GPR values (address and data) are read out of the GPRs and sent to the LDS. After receiving the address, LDS reads its memory at that address, adds the data value (from the src0 GPR), and writes the result into LDS at the specified address (dst).

8.4.4 LDS_ADD_RTN dst, src0

One GPR (dst) holds the LDS address. Another GPR (src0) holds the data to add to the data already in LDS at that address. The two GPR values (address and data) are read out of the GPRs and sent to the LDS. When the LDS receives the address, it reads the LDS memory at that address, writes the value read from memory into the output queue (OQA), adds the data value from the src0 GPR, and writes the result into LDS at the dst address.

8.4.5 LDS_READ2 QAB, src0, src1

Two addresses are sent to LDS for lookups; their values are returned on the LDS stack. This works on all AMD GPUs, but only those with 32 LDS banks can take advantage of this. The following pseudo-code demonstrates this concept:

```
lid =local thread id
val1 = lds[lid]
val2 = lds[lid + stride]
```

When compiled, this is:

```
MOV r0.x, lid
ADD_INT r1.x r0.x, stride
LDS_READ2 QAB, r0.x, r1.x
```

Since there are no bank conflicts (`lid` is unique for thread in a wavefront), it is possible to achieve the peak LDS read bandwidth. Peak bandwidth requires that there are no bank conflicts across 32 threads. For a single wavefront of 64 threads, the first 32 threads are grouped, and their LDS addresses are checked for conflicts. If no conflicts exist, the read is executed in one cycle. If conflicts exist, then multiple cycles are taken. Similarly, the threads in the second half of the wavefront are checked for conflicts.

8.5 Performance and Optimization

See Chapter 4, “OpenCL Performance and Optimization,” in the *ATI Stream Computing OpenCL Programming Guide*.

Chapter 9
Instruction Set

This section describes the instruction set used by assemblers. The instructions grouped by the clauses in which they are used. Within each grouping, they are listed alphabetically, by mnemonic. All of the instructions have mnemonic prefixes, such as CF_INST_, OP2_INST_, or OP3_INST_. In this section's instruction list, only the portion of the mnemonic following the prefix is shown, although the full prefix is described in the text. The opcode and microcode formats for each instruction are also given. The microcode formats are described in Chapter 10, "Microcode Formats", where the instructions are ordered by their microcode formats, rather than alphabetically by mnemonic. That chapter also defines the microcode field-name acronyms.

9.1 Control Flow (CF) Instructions

The CF instructions mnemonics begin with CF_INST_ in the CF_INST field of their microcode formats.

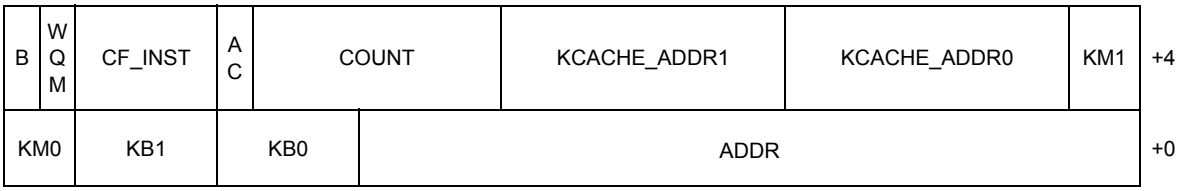
Initiate ALU Clause

Instructions **ALU**

Description Initiates an ALU clause. If the clause issues PRED_SET* instructions, each PRED_SET* instruction updates the active state but does not perform a stack operation.

The ALU instructions within an ALU clause are described in Section Chapter 4, "ALU Clauses," page 4-1 and Section 9.2, "ALU Instructions," page 9-48.

Microcode



Format CF_ALU_WORD0 (page 10-8) and CF_ALU_WORD1 (page 10-9).

Instruction Field CF_INST == CF_INST_ALU, opcode 8 (0x8).

Initiate ALU Clause, Loop Break

Instructions **ALU_BREAK**

Description Initiates an ALU clause. If the clause issues PRED_SET* instructions, each PRED_SET* instruction causes a break operation on the unmasked pixels. The instruction takes the address to the corresponding LOOP_END instruction.

ALU_BREAK is equivalent to PUSH, ALU, ELSE, CONTINUE, and POP.

The ALU instructions within an ALU clause are described in Section Chapter 4, “ALU Clauses,” page 4-1 and Section 9.2, “ALU Instructions,” page 9-48.

Microcode

B	WQM	CF_INST	AC	COUNT	KCACHE_ADDR1	KCACHE_ADDR0	KM1	+4
KM0	KB1	KB0	ADDR					+0

Format CF_ALU_WORD0 (page 10-8) and CF_ALU_WORD1 (page 10-9).

Instruction Field CF_INST == CF_INST_ALU_BREAK, opcode 14 (0xE).

Initiate ALU Clause, Continue Unmasked Pixels

Instructions **ALU_CONTINUE**

Description Initiates an ALU clause. If the clause issues PRED_SET* instructions, each PRED_SET* instruction causes a continue operation on the unmasked pixels. The instruction takes an address to the corresponding LOOP_END instruction.

ALU_CONTINUE is equivalent to PUSH, ALU, ELSE, CONTINUE, and POP.

The ALU instructions within an ALU clause are described in Section Chapter 4, “ALU Clauses,” page 4-1 and Section 9.2, “ALU Instructions,” page 9-48.

Microcode

B	WQM	CF_INST	AC	COUNT	KCACHE_ADDR1	KCACHE_ADDR0	KM1	+4
KM0	KB1	KB0	ADDR					+0

Format CF_ALU_WORD0 (page 10-8) and CF_ALU_WORD1 (page 10-9).

Instruction Field CF_INST == CF_INST_ALU_CONTINUE, opcode 13 (0xD).

Initiate ALU Clause, Stack Push and Else After

Instructions **ALU_ELSE_AFTER**

Description Initiates an ALU clause. If the clause issues PRED_SET* instructions, each PRED_SET* instruction causes a stack push first, then updates the hardware-maintained active state, then performs an ELSE operation to invert the pixel state after the clause completes execution.

The instruction can be used to implement the ELSE part of a higher-level IF statement.

The ALU instructions within an ALU clause are described in Section Chapter 4, “ALU Clauses,” page 4-1 and Section 9.2, “ALU Instructions,” page 9-48.

Microcode

B	WQM	CF_INST	AC	COUNT	KCACHE_ADDR1	KCACHE_ADDR0	KM1	+4
KM0	KB1	KB0	ADDR					+0

Format CF_ALU_WORD0 (page 10-8) and CF_ALU_WORD1 (page 10-9).

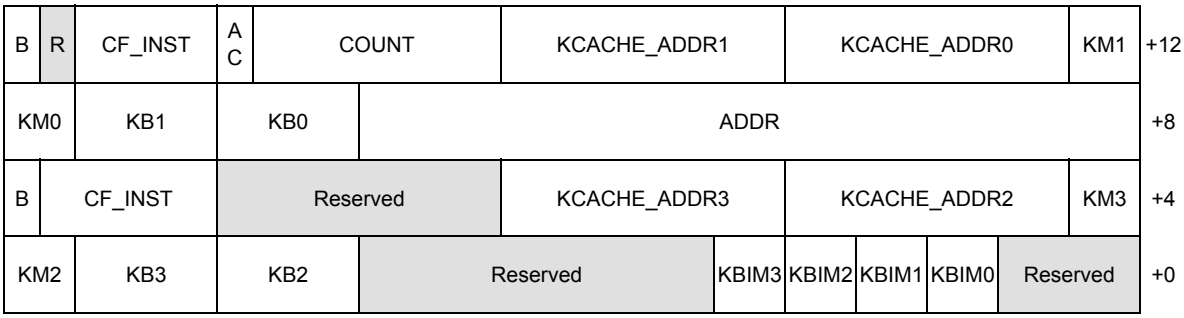
Instruction Field CF_INST == CF_INST_ALU_ELSE_AFTER, opcode 15 (0xF).

ALU Clause Instruction Extension

Instructions **ALU_EXTENDED**

Description ALU clause instruction extension for constant buffers and four constant buffers per clause. This is the first half of the ALU instruction pair. It defines constant buffers 2 and 3, and index-select for all four constant buffers.

Microcode



Format `CF_ALU_WORD0` (page 10-8), `CF_ALU_WORD1` (page 10-9), `CF_ALU_WORD0_EXT` (page 10-11), and `CF_ALU_WORD1_EXT` (page 10-13).

Instruction Field `CF_INST == CF_INST_ALU_EXTENDED`, opcode 12 (0xC).

Initiate ALU Clause, Pop Stack After

Instructions **ALU_POP_AFTER**

Description Initiates an ALU clause, and pops the stack after the clause completes execution.

The ALU instructions within an ALU clause are described in Section Chapter 4, “ALU Clauses,” page 4-1 and Section 9.2, “ALU Instructions,” page 9-48.

Microcode

B	WQM	CF_INST	AC	COUNT	KCACHE_ADDR1	KCACHE_ADDR0	KM1	+4
KM0	KB1	KB0	ADDR					+0

Format CF_ALU_WORD0 (page 10-8) and CF_ALU_WORD1 (page 10-9).

Instruction Field CF_INST == CF_INST_ALU_POP_AFTER, opcode 10 (0xA).

Initiate ALU Clause, Pop Stack Twice After

Instructions **ALU_POP2_AFTER**

Description Initiates an ALU clause, and pops the stack twice after the clause completes execution.

The ALU instructions within an ALU clause are described in Section Chapter 4, “ALU Clauses,” page 4-1 and Section 9.2, “ALU Instructions,” page 9-48.

Microcode

B	WQM	CF_INST	AC	COUNT	KCACHE_ADDR1	KCACHE_ADDR0	KM1	+4
KM0	KB1	KB0	ADDR					+0

Format CF_ALU_WORD0 (page 10-8) and CF_ALU_WORD1 (page 10-9).

Instruction Field CF_INST == CF_INST_ALU_POP2_AFTER, opcode 11 (0xB).

Initiate ALU Clause, Stack Push Before

Instructions **ALU_PUSH_BEFORE**

Description Initiates an ALU clause. If the clause issues PRED_SET* instructions, the first PRED_SET* instruction causes a stack push and an update of the hardware-maintained active execution state. Subsequent PRED_SET* instructions only update the execution state.

The ALU instructions within an ALU clause are described in Section Chapter 4, “ALU Clauses,” page 4-1 and Section 9.2, “ALU Instructions,” page 9-48.

Microcode

B	WQM	CF_INST	AC	COUNT	KCACHE_ADDR1	KCACHE_ADDR0	KM1	+4
KM0	KB1	KB0	ADDR					+0

Format CF_ALU_WORD0 (page 10-8) and CF_ALU_WORD1 (page 10-9).

Instruction Field CF_INST == CF_INST_ALU_PUSH_BEFORE, opcode 9 (0x9).

Call Subroutine

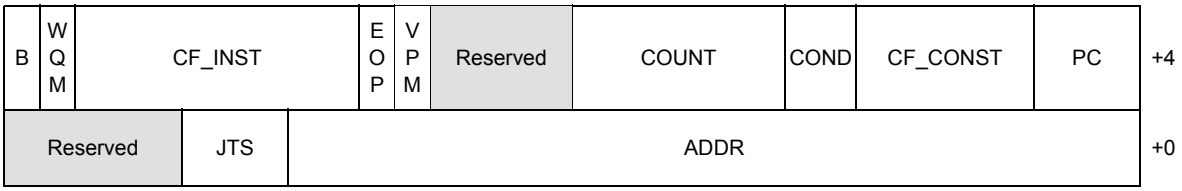
Instructions **CALL**

Description Execute a subroutine call (push call variables onto stack). The ADDR field specifies the address of the first CF instruction in the subroutine.

Calls can be conditional (only pixels satisfying a condition perform the instruction). A CALL_COUNT field specifies the amount by which to increment the call nesting counter. This field is interpreted in the range [0,31]. The instruction is skipped if the current nesting depth + CALL_COUNT > 32. CALLs can be nested. Setting CALL_COUNT to zero prevents the nesting depth from being updated on a subroutine call.

The POP_COUNT field must be zero for CALL.

Microcode



Format CF_WORD0 (page 10-3) and CF_WORD1 (page 10-5).

Instruction Field CF_INST == CF_INST_CALL, opcode 18 (0x12).

Call Fetch Subroutine

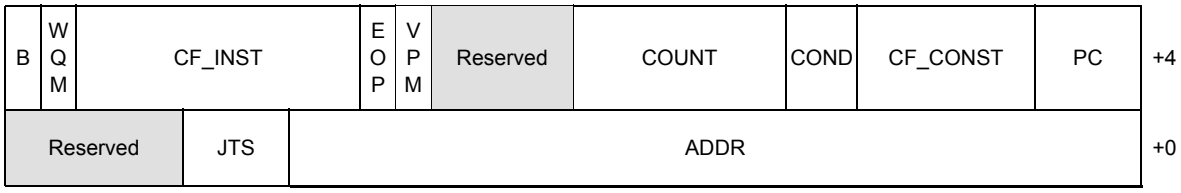
Instructions **CALL_FS**

Description Execute a fetch subroutine (FS) with an address relative to the address specified in a host-configured register. The instruction also activates the fetch-program mode, which affects other operations until the corresponding RETURN instruction is reached. Only a vertex shader (VS) program can call an FS subroutine, as described in Section 2.1, “Program Types,” page 2-1.

Calls can be conditional (only pixels satisfying a condition perform the instruction). A CALL_COUNT field specifies the amount by which to increment the call nesting counter. This field is interpreted in the range [0,31]. The instruction is skipped if the current nesting depth + CALL_COUNT > 32. The subroutine is skipped if and only if all pixels fail the condition test or the nesting depth exceeds 32 after the call.

The POP_COUNT field must be zero for CALL_FS.

Microcode



Format CF_WORD0 (page 10-3) and CF_WORD1 (page 10-5).

Instruction Field CF_INST == CF_INST_CALL_FS, opcode 19 (0x13).

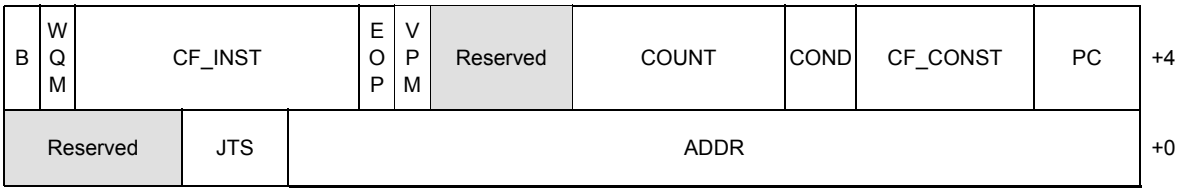
End Primitive Strip, Start New Primitive Strip

Instructions **CUT_VERTEX**

Description Emit an end-of-primitive strip marker. The next emitted vertex starts a new primitive strip. Indicates that the primitive strip has been cut, but does not indicate that a vertex has been exported by itself.

Available only to the Geometry Shader (GS).

Microcode



Format CF_WORD0 (page 10-3) and CF_WORD1 (page 10-5).

Instruction Field CF_INST == CF_INST_CUT_VERTEX, opcode 23 (0x17).

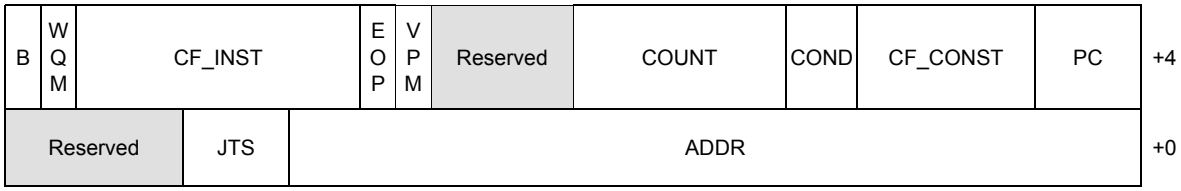
Else

Instructions **ELSE**

Description Pop POP_COUNT entries (can be zero) from the stack, then invert the status of active and branch-inactive pixels for pixels that are both active (as of the last surviving PUSH operation) and pass the condition test. Control then jumps to the specified address if all pixels are inactive.

The operation can be conditional.

Microcode



Format CF_WORD0 (page 10-3) and CF_WORD1 (page 10-5).

Instruction Field CF_INST == CF_INST_ELSE, opcode 13 (0xD).

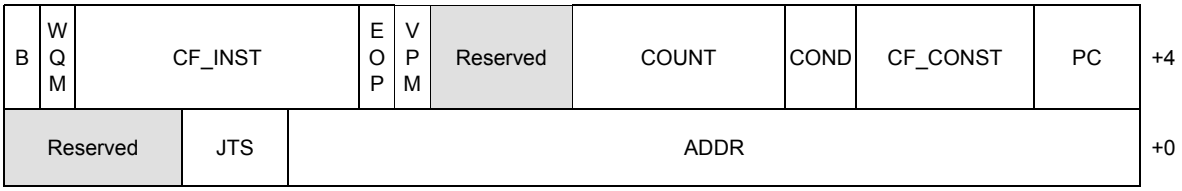
Emit Vertex, End Primitive Strip

Instructions **EMIT_CUT_VERTEX**

Description Emit a vertex and an end-of-primitive strip marker. The next emitted vertex starts a new primitive strip. Indicates that a vertex has been exported and that the primitive strip has been cut after the vertex. The instruction must follow the corresponding export operation that produces a new vertex.

Available only to the Geometry Shader (GS).

Microcode



Format CF_WORD0 (page 10-3) and CF_WORD1 (page 10-5).

Instruction Field CF_INST == CF_INST_EMIT_CUT_VERTEX, opcode 22 (0x16).

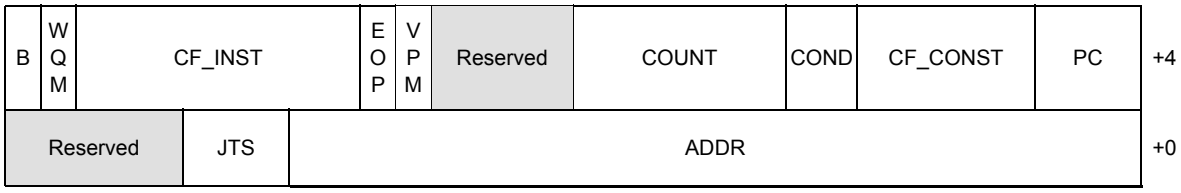
Vertex Exported to Memory

Instructions **EMIT_VERTEX**

Description Signal that a geometry shader (GS) has finished exporting a vertex to memory. Indicates that a vertex has been exported. The instruction must follow the corresponding export operation that produces a new vertex.

Available only to the Geometry Shader (GS).

Microcode



Format `CF_WORD0` (page 10-3) and `CF_WORD1` (page 10-5).

Instruction Field `CF_INST == CF_INST_EMIT_VERTEX`, opcode 21 (0x15).

Export from VS or PS

Instructions **EXPORT**

Description Export from a vertex shader (VS) or a pixel shader (PS). Used for normal pixel, position, and parameter-cache exports. The instruction supports optional swizzles for the outputs. The instruction can be used only by VS and PS programs; GS and DC programs must use one of the CF memory-export instructions, MEM*.

Microcode

B	MRK	CF_INST	EOP	VPM	BC	Reserved	SEL_W	SEL_Z	SEL_Y	SEL_X	+4
ES		INDEX_GPR	RR		RW_GPR	TYPE	ARRAY_BASE				+0

Format CF_ALLOC_EXPORT_WORD0 (page 10-14) and either CF_ALLOC_EXPORT_WORD1_BUF (page 10-19) or CF_ALLOC_EXPORT_WORD1_SWIZ (page 10-21).

Instruction Field CF_INST == CF_INST_EXPORT, opcode 83 (0x53).

Export Last Data

Instructions **EXPORT_DONE**

Description Export the last of a particular data type from a vertex shader (VS) or a pixel shader (PS). Used for normal pixel, position, and parameter-cache exports. The instruction supports optional swizzles for the outputs. The instruction can be used only by VS and PS programs; GS and DC programs must use one of the CF memory-export instructions, MEM*.

Microcode

B	MRK	CF_INST	EOP	VPM	BC	Reserved	SEL_W	SEL_Z	SEL_Y	SEL_X	+4
ES		INDEX_GPR	RR	RW_GPR		TYPE	ARRAY_BASE				+0

Format CF_ALLOC_EXPORT_WORD0 (page 10-14) and either CF_ALLOC_EXPORT_WORD1_BUF (page 10-19) or CF_ALLOC_EXPORT_WORD1_SWIZ (page 10-21).

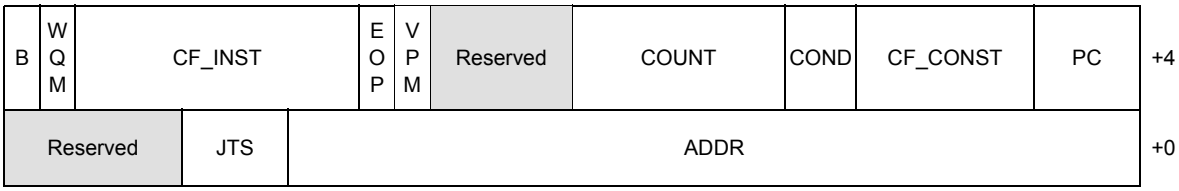
Instruction Field CF_INST == CF_INST_EXPORT_DONE, opcode 84 (0x54).

Global Data Share

Instructions GDS

Description Executes a global data share (GDS) clause containing 1-16 GDS instructions. This clause type is used to write data to the Tesselation Factor (TF) buffer or to transfer data between GPRs and the GDS.

Microcode



Format CF_WORD0 (page 10-3) and CF_WORD1 (page 10-5).

Instruction Field CF_INST == CF_INST_GDS, opcode 3 (0x3).

Global Wavefront Barrier

Instructions **GWS_BARRIER**

Description Stalls execution until the number of waves indicated by VALUE and VAL_INDEX_MODE are waiting on the resource selected by RESOURCE and RES_INDEX_MODE.

Microcode

B	WQM	CF_INST				EOP	VPM	Reserved	COUNT	COND	CF_CONST	PC	+4
GWS_OP	RIM	VIM	S	Reserved		RESOURCE		Reserved	VALUE				+0

Format CF_GWS_WORD0 (page 10-4) and CF_WORD1 (page 10-5).

Instruction Field GWS_OPCODE == GWS_BARRIER, opcode 2 (0x2).

Global Wavefront Resource Initialization

Instructions **GWS_INIT**

Description Initializes the value of the resource selected by the RESOURCE and REX_INDEX_MODE fields to the quantity described by the fields VALUE, SIGN, and VAL_INDEX_MODE.

Microcode

B	WQM	CF_INST				EOP	VPM	Reserved	COUNT	COND	CF_CONST	PC	+4
GWS_OP	RIM	VIM	S	Reserved		RESOURCE		Reserved	VALUE				+0

Format CF_GWS_WORD0 (page 10-4) and CF_WORD1 (page 10-5).

Instruction Field GWS_OPCODE == GWS_INIT, opcode 3 (0x3); CF_INST == CF_INST_GLOBAL_WAVE_SYNC, opcode 30 (0x1E).

Global Wavefront Sync Semaphore P

Instructions **GWS_SEMA_P**

Description Performs an atomic semaphore “P” operation on the resource selected by the RESOURCE and RES_INDEX_MODE fields. Execution of this instruction is stalled until the semaphore has a positive value. The semaphore value is then decremented by one before execution can continue.

Microcode

B	WQM	CF_INST				EOP	VPM	Reserved	COUNT	COND	CF_CONST	PC	+4
GWS_OP	RIM	VIM	S	Reserved		RESOURCE		Reserved	VALUE				+0

Format CF_GWS_WORD0 (page 10-4) and CF_WORD1 (page 10-5).

Instruction Field GWS_OPCODE == GWS_SEMA_P, opcode 1 (0x1).

Global Wavefront Sync Semaphore V

Instructions **GWS_SEMA_V**

Description Performs an atomic semaphore “V” operation on the resource selected by the RESOURCE and RES_INDEX_MODE fields. The value of the resource is incremented by one on execution of this instruction.

Microcode

B	WQM	CF_INST				EOP	VPM	Reserved	COUNT		COND	CF_CONST	PC	+4
GWS_OP	RIM	VIM	S	Reserved		RESOURCE		Reserved	VALUE					+0

Format CF_GWS_WORD0 (page 10-4) and CF_WORD1 (page 10-5).

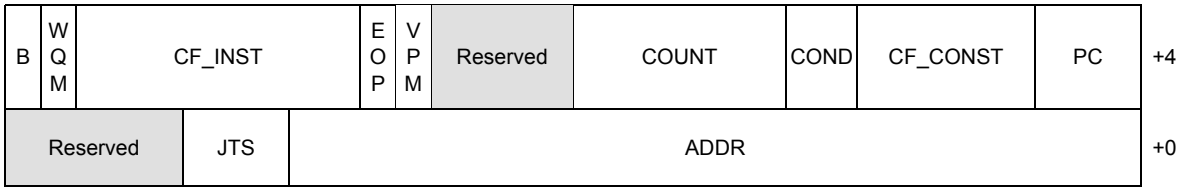
Instruction Field GWS_OPCODE == GWS_SEMA_V, opcode 0 (0x0).

Halt Wavefront Execution

Instructions **HALT**

Description Halts the execution of the wavefront. The drive then can read the internal state of the wavefront and then reenable the wavefront through a register write.

Microcode



Format `CF_WORD0` (page 10-3) and `CF_WORD1` (page 10-5).

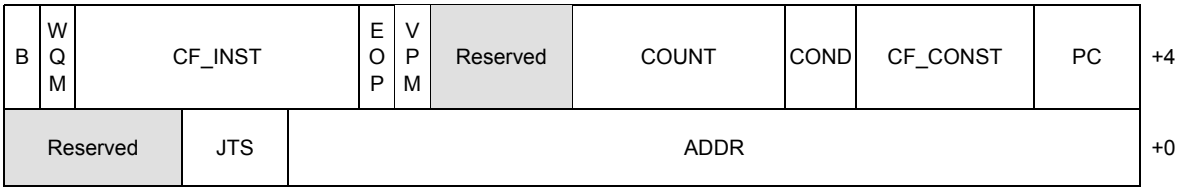
Instruction Field `CF_INST == CF_INST_HALT`, opcode 31 (0x1F).

Jump to Address

Instructions **JUMP**

Description Jump to a specified address, subject to an optional condition test for pixels. It first pops POP_COUNT entries (can be zero) from the stack to. Then it applies the condition test to all pixels. If all pixels fail the test, then it jumps to the specified address. Otherwise, it continues execution on the next instruction. The instruction cannot be used to leave an if/else, subroutine, or loop operation.

Microcode



Format CF_WORD0 (page 10-3) and CF_WORD1 (page 10-5).

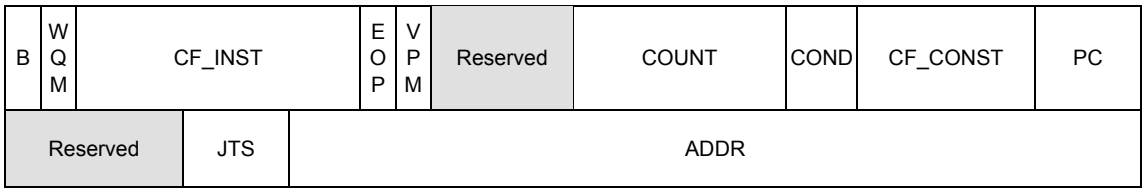
Instruction Field CF_INST == CF_INST_JUMP, opcode 10 (0x10).

Jump Table

Instructions **JUMPTABLE**

Description Executes a jump through a jump table. This instruction is followed by a series of up to 256 jump instructions forming the jump table. The index into the table comes from either a loop-constant or a GPR through the index registers. The instruction after the last jump table entry must be indicated by the ADDR field. If no pixels are enabled after the condition test, execution continues at this address.

Microcode



Format `CF_WORD0` (page 10-3) and `CF_WORD1` (page 10-5).

Instruction Field `CF_INST == CF_INST_JUMPTABLE`, opcode 29 (0x1E).

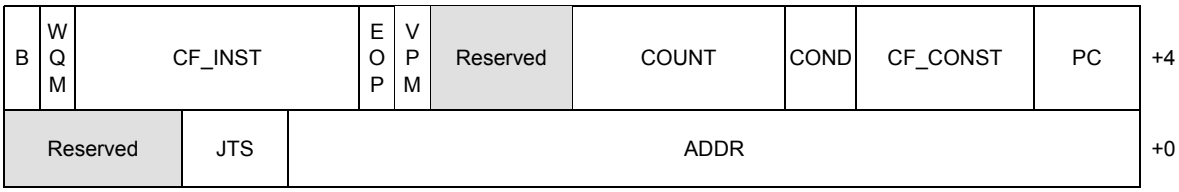
Kill Pixels Conditional

Instructions **KILL**

Description Kill (prevent rendering of) pixels that pass a condition test. Jump if all pixels are killed. Only a pixel shader (PS) can execute this instruction; the instruction is illegal in other program types. Ensure that the KILL instruction is the last instruction in an ALU clause, because the remaining instructions executed in the clause do not reflect the updated valid state after the kill operation. Two KILL instructions cannot be co-issued.

Killed pixels remain active because the processor does not know if the pixels are currently involved in computing a result that is used in a gradient calculation. If the recently invalidated pixels are not involved in a gradient calculation they can be deactivated. The valid pixel mode (VALID_PIXEL_MODE bit) is used to deactivate pixels invalidated by a KILL instruction.

Microcode



Format CF_WORD0 (page 10-3) and CF_WORD1 (page 10-5).

Instruction Field CF_INST == CF_INST_KILL, opcode 24 (0x18).

Break Out Of Innermost Loop

Instructions **LOOP_BREAK**

Description Break out of an innermost loop. The instructions disables all pixels for which a condition test is true. The pixels remain disabled until the innermost loop exits. The instruction takes an address to the corresponding LOOP_END instruction. In the event of a jump, the stack is popped back to the original level at the beginning of the loop; the POP_COUNT field is ignored.

If all pixels have been disabled by this (or a prior) LOOP_BREAK or LOOP_CONTINUE instruction, LOOP_BREAK jumps to the end of the loop and pops POP_COUNT entries (can be zero) from the stack. If at least one pixel has not been disabled by LOOP_BREAK or LOOP_CONTINUE yet, execution continues to the next instruction.

Microcode

B	WQM	CF_INST	EOP	VPM	Reserved	COUNT	COND	CF_CONST	PC	+4
Reserved		JTS	ADDR							+0

Format CF_WORD0 (page 10-3) and CF_WORD1 (page 10-5).

Instruction Field CF_INST == CF_INST_LOOP_BREAK, opcode 9 (0x9).

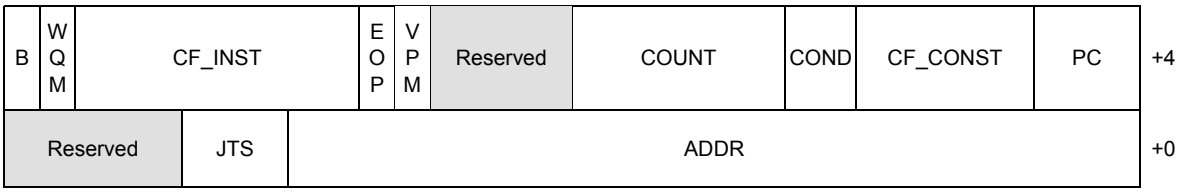
Continue Loop

Instructions **LOOP_CONTINUE**

Description Continue a loop, starting with the next iteration of the innermost loop. Disables all pixels for which a condition test is true. The pixels remain disabled until the end of the current iteration of the loop, and they are re-activated by the innermost LOOP_END.

Control jumps to the end of the loop if all pixels have been disabled by this (or a prior) LOOP_BREAK or LOOP_CONTINUE instruction. In the event of a jump, the stack is popped back to the original level at the beginning of the loop; the POP_COUNT field is ignored. The ADDR field points to the address of the matching LOOP_END instruction. If at least one pixel hasn't been disabled by LOOP_BREAK or LOOP_CONTINUE instruction, the program continues to the next instruction.

Microcode



Format CF_WORD0 (page 10-3) and CF_WORD1 (page 10-5).

Instruction Field CF_INST == CF_INST_LOOP_CONTINUE, opcode 8 (0x8).

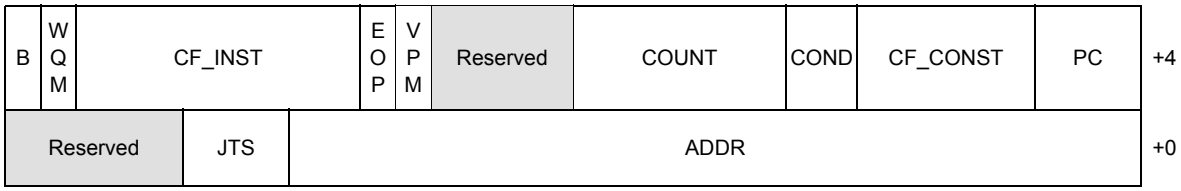
End Loop

Instructions **LOOP_END**

Description Ends a loop if all pixels fail a condition test. Execution jumps to the specified address if the loop counter is non-zero after it is decremented, and at least one pixel ha not been deactivated by a LOOP_BREAK instruction. Software normally sets the ADDR field to the CF instruction following the matching LOOP_START instruction. Execution continues to the next CF instruction if the loop is exited.

LOOP_END pops loop state and one set of per-pixel state from the stack when it exits the loop. It ignores POP_COUNT.

Microcode



Format CF_WORD0 (page 10-3) and CF_WORD1 (page 10-5).

Instruction Field CF_INST == CF_INST_LOOP_END, opcode 5 (0x5).

Start Loop

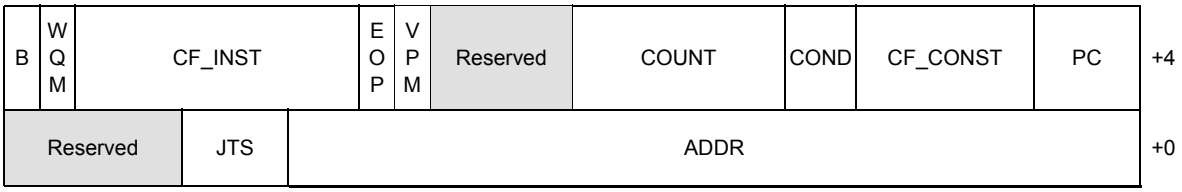
Instructions **LOOP_START**

Description Begin a loop. The instruction pushes the internal loop state onto the stack. A condition test is computed. All pixels fail the test if the loop count is zero. Pixels that fail the test become inactive. If all pixels fail the test, the instruction does not enter the loop, and it pops POP_COUNT entries (can be zero) from the stack.

The instruction reads one of 32 constants, specified by the CF_CONST field, to get the loop's trip count (maximum number of loop iterations), beginning value (loop index initializer), and increment (step), which are maintained by hardware. The instruction jumps to the address specified in the instruction's ADDR field if the initial loop index value is zero. Software normally sets the ADDR field to the instruction following the matching LOOP_END instruction. Control jumps to the specified address if the initial loop count is zero. If LOOP_START does not jump, it sets up the hardware-maintained loop state.

Loop register-relative addressing is well-defined only within the loop. If multiple loops are nested, relative addressing refers to the state of the innermost loop. The state of the next-outer loop is automatically restored when the innermost loop exits.

Microcode



Format CF_WORD0 (page 10-3) and CF_WORD1 (page 10-5).

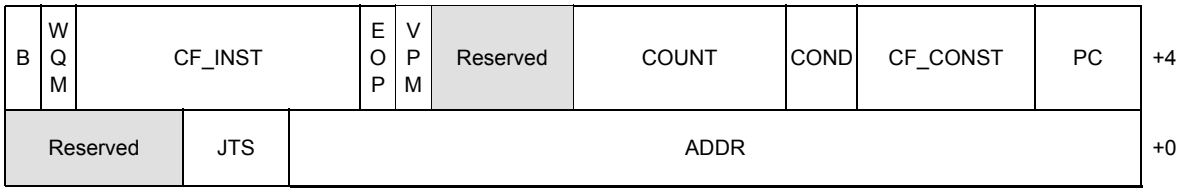
Instruction Field CF_INST == CF_INST_LOOP_START, opcode 4 (0x4).

Start Loop (DirectX 10)

Instructions **LOOP_START_DX10**

Description Enters a DirectX10 loop by pushing control-flow state onto the stack. Hardware maintains the current break count and depth-of-loop nesting. Stack manipulations are the same as those for LOOP_START.

Microcode



Format CF_WORD0 (page 10-3) and CF_WORD1 (page 10-5).

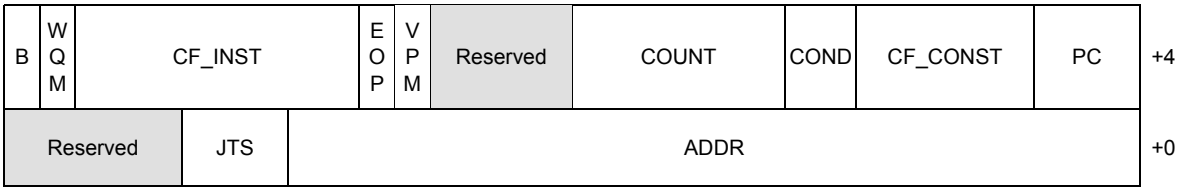
Instruction Field CF_INST == CF_INST_LOOP_START_DX10, opcode 6 (0x6).

Enter Loop If Zero, No Push

Instructions **LOOP_START_NO_AL**

Description Same as LOOP_START but does not push the loop index (aL) onto the stack or update the aL. Repeat loops are implemented with LOOP_START_NO_AL and LOOP_END.

Microcode



Format CF_WORD0 (page 10-3) and CF_WORD1 (page 10-5).

Instruction Field CF_INST == CF_INST_LOOP_START_NO_AL, opcode 7 (0x7).

Access Scatter Buffer

Instructions **MEM_EXPORT**

Description Performs a memory read or write on the scatter buffer. This instruction is legal with a TYPE of: read, read-indexed, write, write-indexed. Indexed is the expected common use. Used only for writes.

The 13-bit ARRAY_BASE field is valid and is added to the base address for each pixel (units of DWORD).

The ARRAY_SIZE field is unused. Set it to zero.

The ES field is supported, allowing 1,2,3,4 DWORDs written per export. Burst read/write is allowed and in this case, the address is incremented by “elemsize” DWORDs.

The address in the INDEX_GPR is a DWORD address, no matter how much data is exported.

Address Calculation & Clamping SP supplies a 32-bit integer address offset per pixel (assume zero if no EA export).

```
Per pixel DWORD address =
{BASE_reg,6'h0} + clamp({ARRAY_SIZE,6'h0}, (BC increment counter *elemsize +
INDEX_GPR + ARRAY_BASE))
```

Microcode

B	MRK	CF_INST	EOP	VPM	BC	Reserved	SEL_W	SEL_Z	SEL_Y	SEL_X	+4
ES	INDEX_GPR		RR	RW_GPR		TYPE	ARRAY_BASE				+0

Format CF_ALLOC_EXPORT_WORD0 (page 10-14) and either CF_ALLOC_EXPORT_WORD1_BUF (page 10-19) or CF_ALLOC_EXPORT_WORD1_SWIZ (page 10-21).

Instruction Field CF_INST == CF_INST_MEM_EXPORT, opcode 85 (0x55).

Export Combined Address And Data

Instructions **MEM_EXPORT_COMBINED**

Description Write two consecutive dwords of data per thread to scatter memory. GPRs hold: X = data0, Y = data1, Z = unused, W = address.

Data1 can be masked out by setting `comp_mask.y` = 0.

Burst_count must be zero. Indexed writes are not allowed.

Microcode

B	M	CF_INST	EOP	VPM	BURST_COUNT	COMP_MASK	ARRAY_SIZE	+4
ES		INDEX_GPR	RR	RW_GPR		TYPE	ARRAY_BASE	+0

Format `CF_ALLOC_EXPORT_WORD0` (page 10-14) and `CF_ALLOC_EXPORT_WORD1_BUF` (page 10-19).

Instruction Field `CF_INST == CF_INST_MEM_EXPORT_COMBINED`, opcode 91 (0x5B).

Export To UAV

Instructions **MEM_RAT**

Description Write 1-4 consecutive dwords to a UAV. Apply the RAT_INST to combine data written from the kernel with data existing in the UAV in memory.

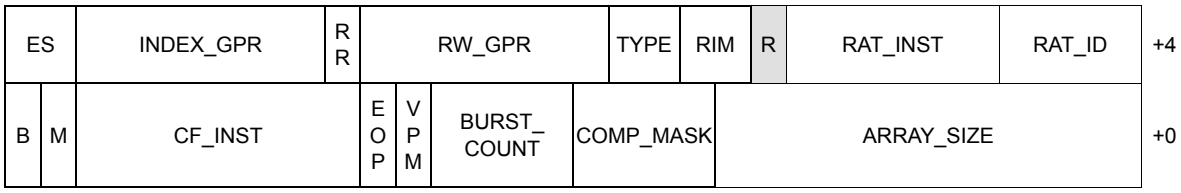
Index GPR: X = addr0, Y = addr1, Z = addr2, W = unused.

RW GPR: Four dwords of data for RAT_INST_STORE_TYPED.

RW GPR: X = data, Y = returnAddr, Z = CmpData for other RAT_INSTs.

See the RAT_INST list on page 16.

Microcode



Format CF_ALLOC_EXPORT_WORD1_BUF (page 10-19) and CF_ALLOC_EXPORT_WORD0_RAT (page 10-16).

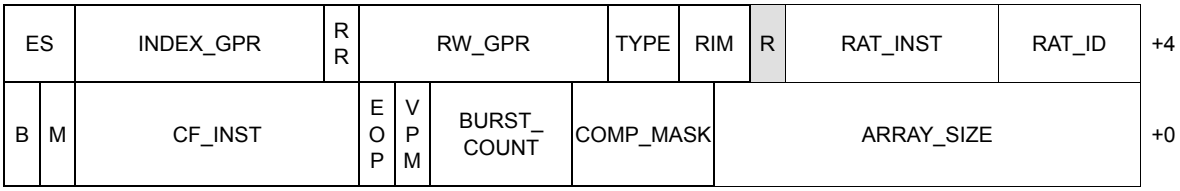
Instruction Field CF_INST == CF_INST_MEM_RAT, opcode 86 (0x56).

Export To UAV Without Caching

Instructions **MEM_RAT_CACHELESS**

Description Write data to a UAV surface, bypassing on-chip caches. Only RAT_STORE and RAT_READ opcodes are allowed; no atomic operations.

Microcode



Format CF_ALLOC_EXPORT_WORD0_RAT (page 10-16) and CF_ALLOC_EXPORT_WORD1_BUF (page 10-19).

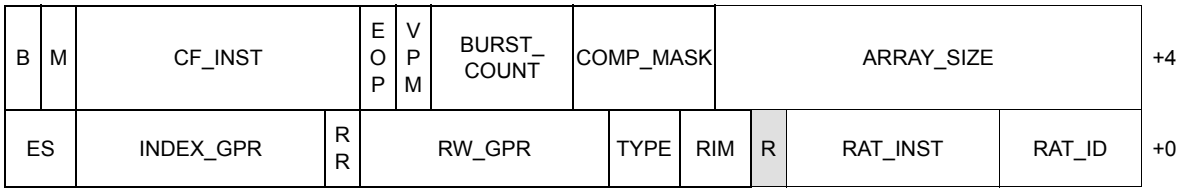
Instruction Field CF_INST == CF_INST_MEM_RAT_CACHELESS, opcode 87 (0x57).

Export To UAV Of Combined Address And Data Without Caching

Instructions **MEM_RAT_COMBINED_CACHELESS**

Description Export to a Random Access Target - reduced functionality (via DB). Combined Address and Data in one export (data = x, data = y; address = w). Must be non-indexed-write, and no burst-writes.

Microcode



Format `CF_ALLOC_EXPORT_WORD1_BUF` (page 10-19) and `CF_ALLOC_EXPORT_WORD0_RAT` (page 10-16).

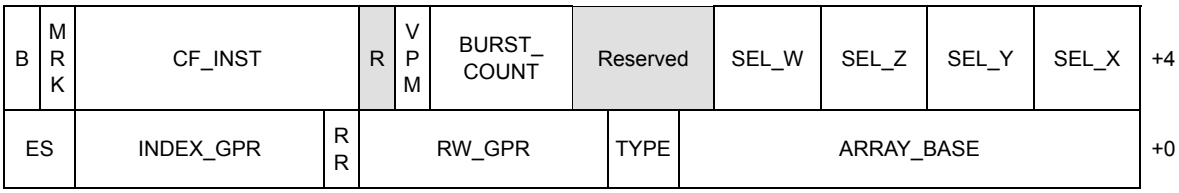
Instruction Field `CF_INST == CF_INST_MEM_RAT_COMBINED_CACHELESS`, opcode 92 (0x5C).

Export To UAV Without Caching

Instructions **MEM_RING**
MEM_RING1
MEM_RING2
MEM_RING3

Description Write to the respective ring (either 1, 2, or 3). Currently applies only to GSVS ring.

Microcode



Format CF_ALLOC_EXPORT_WORD0 (page 10-14) and CF_ALLOC_EXPORT_WORD1_BUF (page 10-19).

Instruction Field CF_INST == CF_INST_MEM_RING, opcode 82 (0x52).
CF_INST == CF_INST_MEM_RING1, opcode 88 (0x58).
CF_INST == CF_INST_MEM_RING2, opcode 89 (0x59).
CF_INST == CF_INST_MEM_RING3, opcode 90 (0x5A).

Memory Write On Stream

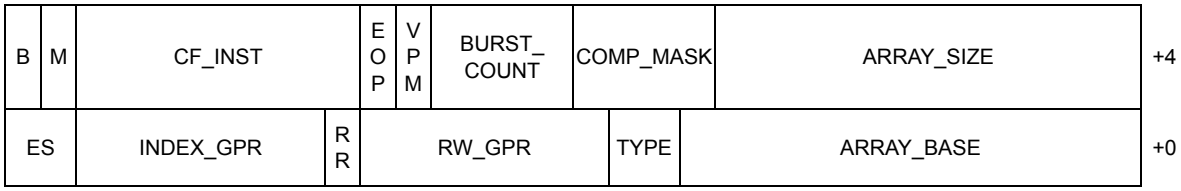
Instructions

MEM_STREAM0_BUF0
MEM_STREAM0_BUF1
MEM_STREAM0_BUF2
MEM_STREAM0_BUF3
MEM_STREAM1_BUF0
MEM_STREAM1_BUF1
MEM_STREAM1_BUF2
MEM_STREAM1_BUF3
MEM_STREAM2_BUF0
MEM_STREAM2_BUF1
MEM_STREAM2_BUF2
MEM_STREAM2_BUF3
MEM_STREAM3_BUF0
MEM_STREAM3_BUF1
MEM_STREAM3_BUF2
MEM_STREAM3_BUF3

Description

Perform a memory write on the respective buffer 0 of the respective stream. Used for DirectX stream-out operations. Write data to one of four buffers for one of the four input streams.

Microcode



Format

CF_ALLOC_EXPORT_WORD0 (page 10-14) and CF_ALLOC_EXPORT_WORD1_BUF (page 10-19).

Instruction Field

CF_INST == CF_INST_MEM_STREAM0_BUF0, opcode 64 (0x40).
CF_INST == CF_INST_MEM_STREAM0_BUF1, opcode 65 (0x41).
CF_INST == CF_INST_MEM_STREAM0_BUF2, opcode 66 (0x42).
CF_INST == CF_INST_MEM_STREAM0_BUF3, opcode 67 (0x43).
CF_INST == CF_INST_MEM_STREAM1_BUF0, opcode 68 (0x44).
CF_INST == CF_INST_MEM_STREAM1_BUF1, opcode 69 (0x45).
CF_INST == CF_INST_MEM_STREAM1_BUF2, opcode 70 (0x46).
CF_INST == CF_INST_MEM_STREAM1_BUF3, opcode 71 (0x47).
CF_INST == CF_INST_MEM_STREAM2_BUF0, opcode 72 (0x48).
CF_INST == CF_INST_MEM_STREAM2_BUF1, opcode 73 (0x49).
CF_INST == CF_INST_MEM_STREAM2_BUF2, opcode 74 (0x4A).
CF_INST == CF_INST_MEM_STREAM2_BUF3, opcode 75 (0x4B).
CF_INST == CF_INST_MEM_STREAM3_BUF0, opcode 76 (0x4C).
CF_INST == CF_INST_MEM_STREAM3_BUF1, opcode 77 (0x4D).
CF_INST == CF_INST_MEM_STREAM3_BUF2, opcode 78 (0x4E).
CF_INST == CF_INST_MEM_STREAM3_BUF3, opcode 79 (0x4F).

Access Scratch Buffer

Instructions **MEM_WR_SCRATCH**

Description Perform a memory write on the scratch buffer.

Microcode

B	MRK	CF_INST	EOP	VPM	BC	Reserved	SEL_W	SEL_Z	SEL_Y	SEL_X	+4
ES		INDEX_GPR	RR		RW_GPR	TYPE	ARRAY_BASE				+0

Format `CF_ALLOC_EXPORT_WORD0` (page 10-14) and either `CF_ALLOC_EXPORT_WORD1_BUF` (page 10-19) or `CF_ALLOC_EXPORT_WORD1_SWIZ` (page 10-21).

Instruction Field `CF_INST == CF_INST_MEM_WR_SCRATCH`, opcode 80 (0x50).

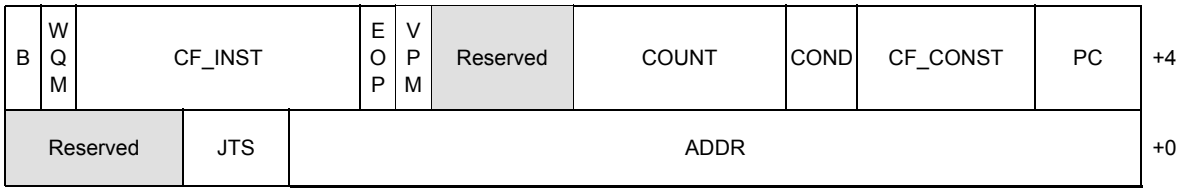
No Operation

Instructions **NOP**

Description No operation. It ignores all fields in the CF_WORD[0,1] microcode formats, except the CF_INST, BARRIER, and END_OF_PROGRAM fields. The instruction does not preserve the current PV or PS value in the slot in which it executes. Instruction slots that are omitted implicitly execute NOPs in the corresponding ALU. As a consequence, slots that are unspecified do not preserve PV or PS for the next instruction. To preserve PV or PS and perform no other operation in an ALU clause, use a MOV instruction with a disabled write mask.

See the ALU version of NOP on page 9-168.

Microcode



Format CF_WORD0 (page 10-3) and CF_WORD1 (page 10-5).

Instruction Field CF_INST == CF_INST_NOP, opcode 0 (0x0).

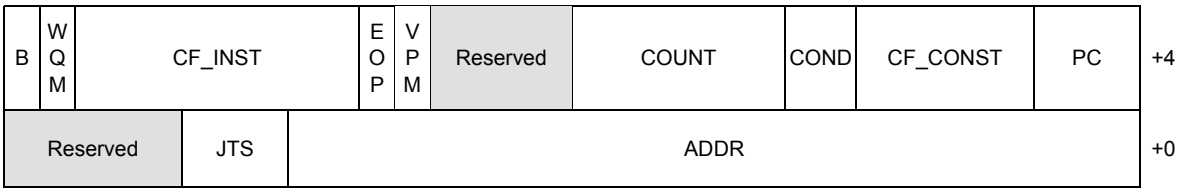
Pop From Stack

Instructions **POP**

Description Pops POP_COUNT number of entries (can be zero) from the stack. POP can apply a condition test to the result of the pop. This is useful for disabling pixels that are killed within a conditional block. To disable such pixels, set the POP instruction's VALID_PIXEL_MODE bit and set the condition to CF_COND_ACTIVE. If POP_COUNT is zero, POP simply modifies the current per-pixel state based on the result of the condition test.

POP instructions never jump.

Microcode



Format CF_WORD0 (page 10-3) and CF_WORD1 (page 10-5).

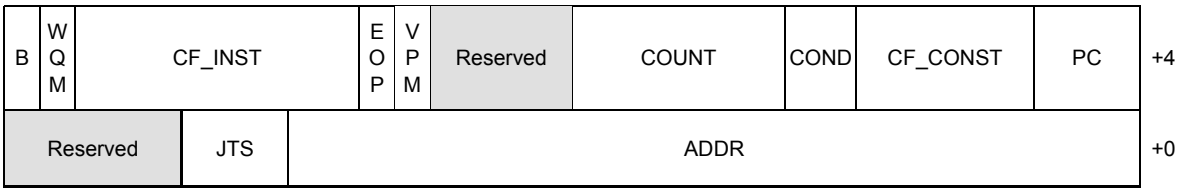
Instruction Field CF_INST == CF_INST_POP, opcode 14 (0xE).

Push State To Stack

Instructions **PUSH**

Description If all pixels fail a condition test, pop POP_COUNT entries from the stack and jump to the specified address. Otherwise, push the current per-pixel state (active mask) onto the stack. After the push, active pixels that failed the condition test transition to the inactive-branch state in the new active mask.

Microcode



Format CF_WORD0 (page 10-3) and CF_WORD1 (page 10-5).

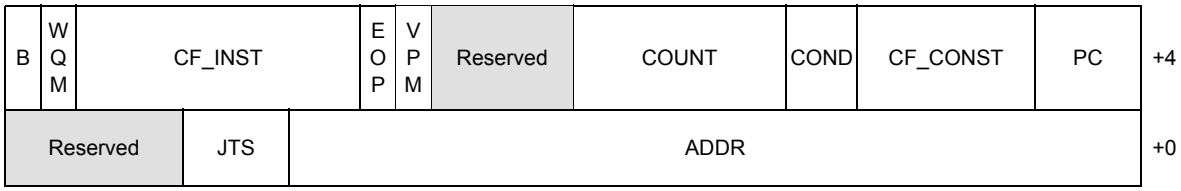
Instruction Field CF_INST == CF_INST_PUSH, opcode 11 (0xB).

Return From Subroutine

Instructions **RETURN**

Description Return from subroutine. Pops the return address from the stack to program counter. Paired only with the CALL instruction. The ADDR field is ignored; the return address is read from the stack.

Microcode



Format CF_WORD0 (page 10-3) and CF_WORD1 (page 10-5).

Instruction Field CF_INST == CF_INST_RETURN, opcode 20 (0x14).

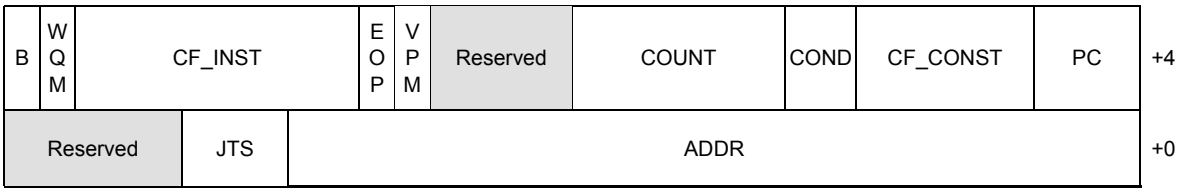
Initiate Fetch Clause Through Texture Cache

Instructions **TC**

Description Initiates a fetch clause which will be serviced by the Texture Cache (TC) hardware. This clause can contain texture, vertex or constant fetches.

ADDR specifies the double-quadword-aligned offset to the first instruction in the clause that contains COUNT+1 instructions. The instructions within a fetch through a texture cache clause are described in Section Chapter 6, “Texture Cache Clauses,” page 6-1 and Section 9.4, “Instructions for a Fetch Through a Texture Cache Clause,” page 9-254.

Microcode



Format CF_WORD0 (page 10-3) and CF_WORD1 (page 10-5).

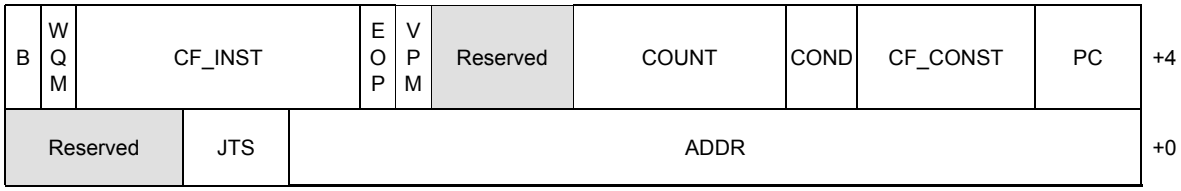
Instruction Field CF_INST == CF_INST_TC, opcode 1 (0x1).

Fetch Clause Through Texture Cache With ACK

Instructions **TC_ACK**

Description Execute a TC clause and provides an ACK when the clause has completed. This can be used with CF_INST_WAIT_ACK to cause the shader to sleep until the TC_ACK clause has returned all of the fetch data to GPRs. All previous TC/VC/GDS requests must have completed if this instruction is issued without BARRIER_BEFORE being set.

Microcode



Format CF_WORD0 (page 10-3) and CF_WORD1 (page 10-5).

Instruction Field CF_INST == CF_INST_TC_ACK, opcode 27 (0x1B).

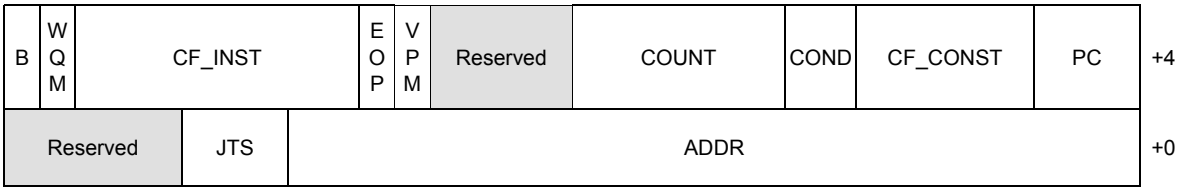
Initiate Clause of Vertex or Constant Fetches Through Vertex Cache

Instructions VC

Description Execute a clause of vertex or constant fetches which is serviced by the Vertex Cache (VC).

ADDR specifies the double-quadword-aligned offset to the first instruction in the clause that contains COUNT+1 instructions. The instructions within a fetch through a texture cache clause are described in Section Chapter 6, “Texture Cache Clauses,” page 6-1 and Section 9.4, “Instructions for a Fetch Through a Texture Cache Clause,” page 9-254.

Microcode



Format CF_WORD0 (page 10-3) and CF_WORD1 (page 10-5).

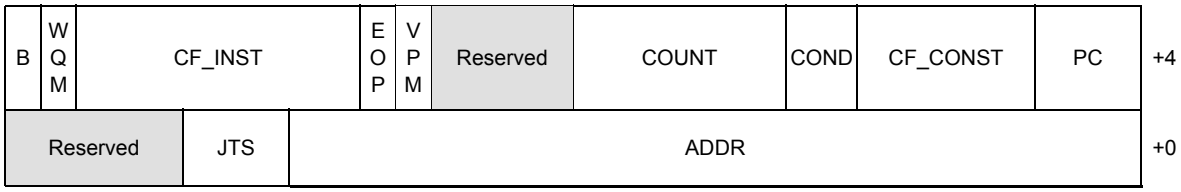
Instruction Field CF_INST == CF_INST_VC, opcode 2 (0x2).

Fetch Clause Through Vertex Cache With ACK

Instructions **VC_ACK**

Description Execute a fetch clause through the Vertex Cache, which produces an ACKnowledge when the clause has completed. This used in conjunction with the CF_INST_WAIT_ACK instruction to cause the kernel to wait for the clause to complete before proceeding.

Microcode



Format CF_WORD0 (page 10-3) and CF_WORD1 (page 10-5).

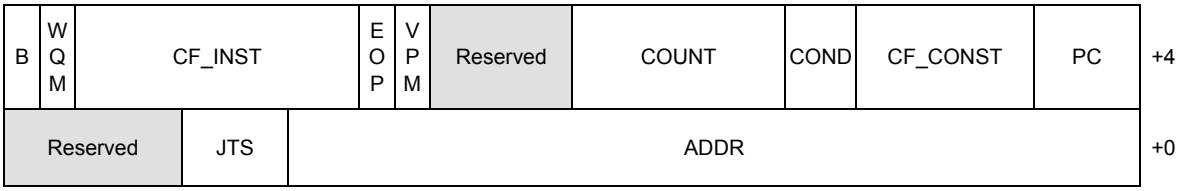
Instruction Field CF_INST == CF_INST_VC_ACK, opcode 28 (0x1C).

Wait for Write or Fetch-Read ACKs

Instructions **WAIT_ACK**

Description Wait for write-acks or fetch-read-acks to return before proceeding. Wait if the number of outstanding acks is greater than the value in the ADDR field.

Microcode



Format CF_WORD0 (page 10-3) and CF_WORD1 (page 10-5).

Instruction Field CF_INST == CF_INST_WAIT_ACK, opcode 26 (0x1A).

9.2 ALU Instructions

All of the instructions in this section have a mnemonic that begins with OP2_INST_ or OP3_INST_ in the ALU_INST field of their microcode formats.

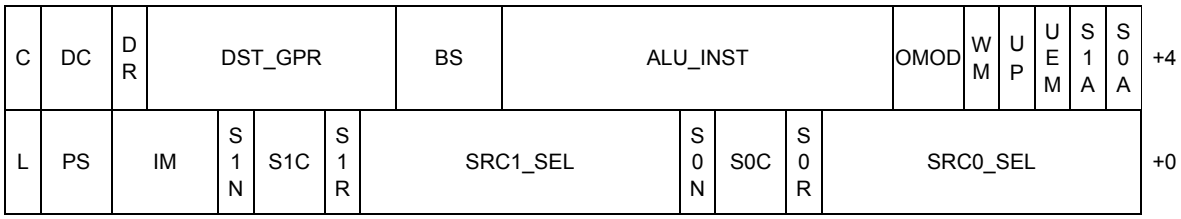
Floating-Point Add

Instructions **ADD**

Description Floating-point add.

```
dst = src0 + src1;
```

Microcode



Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_ADD, opcode 0 (0x0).

Add Floating-Point, 64-Bit

Instructions **ADD_64**

Description Floating-point 64-bit add. Adds two double-precision numbers in the YX or WZ elements of the source operands, src0 and src1, and outputs a double-precision value to the same elements of the destination operand. No carry or borrow beyond the 64-bit values is performed. The operation occupies two slots in an instruction group.

```
dst = src0 + src1;
```

Table 9.1 Result of ADD_64 Instruction

	src1								
src0	**-inf**	**-F[1]**	**-denorm**	**-0**	**+0**	**+denorm**	**+F[1]**	**+inf**	**NaN[2]**
-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf	NaN64	src1 (NaN64)
-F[1]	-inf	-F	src0	src0	src0	src0	+-F or +0	+inf	src1 (NaN64)
-denorm	-inf	src1	-0	-0	+0	+0	src1	+inf	src1 (NaN64)
-0	-inf	src1	-0	-0	+0	+0	src1	+inf	src1 (NaN64)
+0	-inf	src1	+0	+0	+0	+0	src1	+inf	src1 (NaN64)
+denorm	-inf	src1	+0	+0	+0	+0	src1	+inf	src1 (NaN64)
+F[1]	-inf	+-F or +0	src0	src0	src0	src0	+F	+inf	src1 (NaN64)
+inf	NaN64	+inf	+inf	+inf	+inf	+inf	+inf	+inf	src1 (NaN64)
NaN	src0 (NaN64)	src0 (NaN64)	src0 (NaN64)	src0 (NaN64)	src0 (NaN64)	src0 (NaN64)	src0 (NaN64)	src0 (NaN64)	src0 (NaN64)

1. F is a finite floating-point value.
2. NaN64 = 0xFFF8000000000000. An NaN64 is a propagated NaN value from the input listed.

These properties hold true for this instruction:

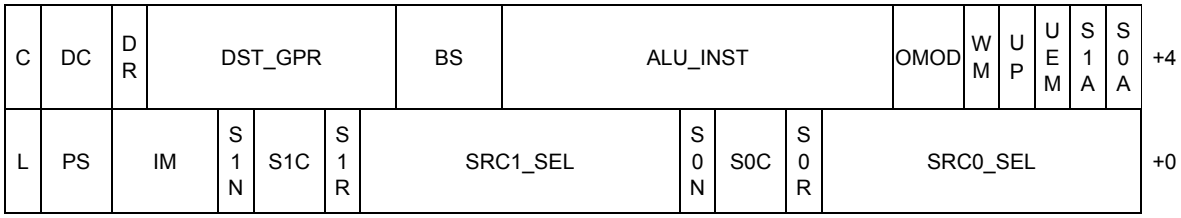
```
(A + B) == (B + A)
(A – B) == (A + -B)
A + -A = +zero
```

Add Floating-Point, 64-Bit (Cont.)

Coissue ADD_64 is a two-slot instruction. The following coissues are possible.

- A single ADD_64 instruction in slots 2 and 3, and any valid instructions in slots 0, 1, and 4.
- A single ADD_64 instruction in slots 0 and 1, and any valid instructions in slots 2, 3, and 4.
- Two ADD_64 instructions in slots 0, 1, 2, and 3,and any valid instruction in slot 4.

Microcode



Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_ADD_64, opcode 203 (0xCB).

Add Floating-Point, 64-Bit (Cont.)

Example The following example coissues two ADD_64 instructions in slots 0 and 1, and 2 and 3.

```
Input data:

Input data  3.0 (0x4008000000000000)
Input data  6.0 (0x4018000000000000)
Input data 12.0 (0x4028000000000000)

mov ra.h, l(0x40080000)   //high dword (Input 1)
mov rb.l, l(0x00000000)   //low  dword

mov rc.h, l(0x40180000)   //high dword (Input 2)
mov rd.l, l(0x00000000)   //low  dword

mov rg.h, l(0x40180000)   //high dword (Input 3)
mov rh.l, l(0x00000000)   //low  dword

mov ri.h, l(0x40280000)   //high dword (Input 4)
mov rj.l, l(0x00000000)   //low  dword

Issue instructions:

ADD_64  re.x ra.h rc.h; //can be any vector element
ADD_64  rf.y rb.l rd.l; //can be any vector element
ADD_64  rk.z rg.h ri.h; //can be any vector element
ADD_64  rl.w rh.l rj.l; //can be any vector element

Result:

Input 1 + Input 2  = 3.0 +  6.0 = 9.0  (0x4022000000000000)
Input 3 + Input 4  = 6.0 + 12.0 = 18.0 (0x4032000000000000)

re.x = 0x00000000    (LSB of Input1 and Input2 add result)
rf.y = 0x40220000    (MSB of Input1 and Input2 add result)
rk.z = 0x00000000    (LSB of Input3 and Input4 add result)
rl.w = 0x40320000    (MSB of Input3 and Input4 add result)
```

Input Modifiers Input modifiers (Section 4.7.2, “Input Modifiers,” page 4-10) can be applied to the source operands during the destination X channel (slot 0) or Z channel (slot 2). These slots contain the sign bits of the sources.

Output Modifiers Output modifiers (Section 4.9.1, “Output Modifiers,” page 4-25) can be applied to the destination during the destination X channel (slot 0) or Z channel (slot 2).

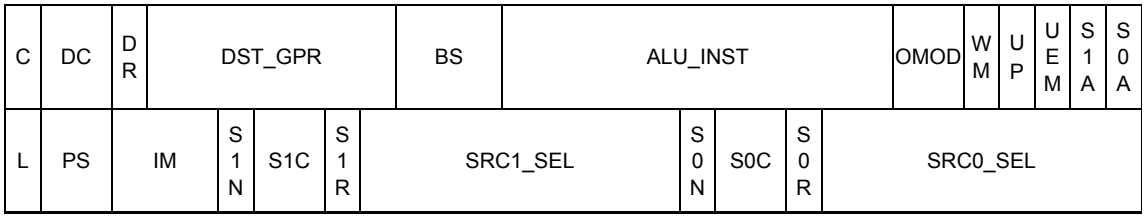
Add Integer

Instructions **ADD_INT**

Description Integer add, based on signed or unsigned integer operands.

```
dst = src0 + src1;
```

Microcode



Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_ADD_INT, opcode 52 (0x34).

Dependent Add

Instructions **ADD_PREV**

Description Add src0 to the previous channel's result. The previous channel opcode must result in a 32-bit, single-precision floating-point.

The output modifier and clamping on the w/z slot is not allowed (results are undefined).

Do not use in w or z channels.

The previous channel y is the w channel's FP32 result.
The previous channel x is the z channel's FP32 result.

```
dst = src0 + prev_channel_result
```

Microcode

C	DC	DR	DST_GPR			BS	ALU_INST					OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL				+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_ADD_PREV, opcode 211 (0xD3).

Output Carry Bit of Unsigned Integer ADD

Instructions **ADDC_UINT**

Description Output carry bit of unsigned integer ADD.

```
If (src0 + src1 > 0xFFFFFFFF) {
    dst = 0x00000001;
}
Else {
    dest = 0;
}
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST						OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL							+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_ADDC_UINT; opcode 82 (0x52).

AND Bitwise

Instructions **AND_INT**

Description Logical bit-wise AND.

```
dst = src0 & src1;
```

Microcode

C	DC	DR	DST_GPR							BS	ALU_INST											OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM							S1N	S1C	S1R	SRC1_SEL						S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_AND_INT, opcode 48 (0x30).

Scalar Arithmetic Shift Right

Instructions **ASHR_INT**

Description Scalar arithmetic shift right. The sign bit is shifted into the vacated locations. The five lsb of `src1` are interpreted as an unsigned integer. If `src1` is > 31, the result is either 0x0 or -0x1 (0xFFFFFFFF), depending on the sign of `src0`.

```
dst = src0 >> (src1 & 0x1F)
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST					OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL						+0

Format `ALU_WORD0` (page 10-23) and `ALU_WORD1_OP2` (page 10-26).

Instruction Field `ALU_INST == OP2_INST_ASHR_INT`; opcode 21 (0x15).

Count Bits Set 32 Accumulate

Instructions **BCNT_ACCUM_PREV_INT**

Description Count number of bits set in src0 and add to previous channel's BCNT_INT result. This works only if BCNT_INT is in the previous channel. Not legal if used in channel w. The previous channel z is channel w's BCNT_INT result. The previous channel y is channel z's BCNT_INT result. The previous channel x is channel y's BCNT_INT result.

```
count = 0;
    for (i = 0 to 31) {
    count = count + src0[i];
}
dst = count + prev_channel_bcnt_int_dst;
```

See MBCNT_32LO_ACCUM_PREV_INT, s, for intended usage.

Microcode

C	DC	D R	DST_GPR						BS	ALU_INST									OMOD	W M	U P	U E M	S 1 A	S 0 A	+4
L	PS	IM							S 1 N	S1C	S 1 R	SRC1_SEL						S 0 N	S0C	S 0 R	SRC0_SEL				+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_BCNT_ACCUM_PREV_INT, opcode 182 (0xB6).

Count Bits Set

Instructions **BCNT_INT**

Description DX11 count bits set. Counts the number of bits set in src0.

```
count = 0;
    for (i = 0 to 31) {
    count = count + src0[i];
}
dst = count;
```

See MBCNT_32LO_ACCUM_PREV_INT, page 139, for intended usage.

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST						OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL							+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_BCNT_INT, opcode 170 (0xAA).

Signed Integer Bitfield Extract

Instructions **BFE_INT**

Description DX11 signed bitfield extract. src0 = input data, src1 = offset, and src2 = width. The bit position offset is extracted through offset + width from the input data. All bits remaining after dst are stuffed with replications of the sign bit.

```
If (src2[4:0] == 0) {
    dst = 0;
}
Else if (src2[4:0] + src1[4:0] < 32) {
    dst = (src0 << (32-src1[4:0] - src2{4:0])) >>> (32 - src2[4:0])
}
Else {
    dst = src0 >>> src1[4:0]
```

Microcode

C	DC	DR	DST_GPR		BS	ALU_INST	SN	S2C	SR	SRC2_SEL			+4
L	PS	IM		S1N	S1C	S1R	SRC1_SEL		S0N	S0C	S0R	SRC0_SEL	+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP3 (page 10-32).

Instruction Field ALU_INST == OP3_INST_BFE_INT, opcode 5 (0x5).

Unsigned Integer Bitfield Extract

Instructions **BFE_UINT**

Description DX11 unsigned bitfield extract. src0 = input data, scr1 = offset, and src2 = width. Bit position offset is extracted through offset + width from input data.

```
If (src2[4:0] == 0) {
    dst = 0;
}
Else if (src2[4:0] + src1[4:0] < 32) {
    dst = (src0 << (32-src1[4:0] - src2{4:0])) >> (32 - src2[4:0])
}
Else {
    dst = src0 >> src1[4:0]
```

Microcode

C	DC	DR	DST_GPR		BS	ALU_INST	SN	S2C	SR	SRC2_SEL			+4
L	PS	IM		S1N	S1C	S1R	SRC1_SEL		S0N	S0C	S0R	SRC0_SEL	+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP3 (page 10-32).

Instruction Field ALU_INST == OP3_INST_BFE_UINT, opcode 4 (0x4).

Bitfield Insert

Instructions **BFI_INT**

Description Bitfield insert used after BFM to implement DX11 bitfield insert.

src0 = bitfield mask (from BFM)

src 1 & src2 = input data

This replaces bits in src2 with bits in src1 according to the bitfield mask.

```
dst = (src1 & src0) | (src2 & -src0)
```

Microcode

C	DC	DR	DST_GPR		BS	ALU_INST	SN	S2C	SR	SRC2_SEL		+4
L	PS	IM	S1N	S1C	S1R	SRC1_SEL	S0N	S0C	S0R	SRC0_SEL		+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP3 (page 10-32).

Instruction Field ALU_INST == OP3_INST_BFI_INT, opcode 6 (0x6).

Bitfield Mask

Instructions **BFM_INT**

Description Bitfield mask used before BFI to implement DX11 bitfield insert. This creates a bitfield mask suitable for src0 input to BFI.

src0[4:0] = bitfield width

src1[4:0] = bitfield offset

```
dst = (((1 << src0[4:0]) -1) << src1[4:0])
```

Microcode

C	DC	DR	DST_GPR	BS	ALU_INST	OMOD	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1C	S1R	SRC1_SEL	S0N	S0C	S0R	SRC0_SEL	+0

Format `ALU_WORD0` (page 10-23) and `ALU_WORD1_OP2` (page 10-26).

Instruction Field `ALU_INST == OP2_INST_BFM_INT`, opcode 160 (0xA0).

Dword Reversal

Instructions **BFREV_INT**

Description Reverses the DX11 bits. `src0` = input data.

```
res = 0
    for (i=0 to 31){
        res[i]= src0[31- i];
}
dst = res;
```

Microcode

C	DC	DR	DST_GPR				BS	ALU_INST				OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL				+0

Format `ALU_WORD0` (page 10-23) and `ALU_WORD1_OP2` (page 10-26).

Instruction Field `ALU_INST == OP2_INST_BFREV_INT`, opcode 81 (0x51).

Bit Align

Instructions **BIT_ALIGN_INT**

Description Right-shifts 64 bits into a 32-bit GPR.

```
dst = ({src0, src1} >> src2[4:0]) & 0xFFFFFFFF;
```

Microcode

C	DC	DR	DST_GPR			BS	ALU_INST	SN	S2C	SR	SRC2_SEL		+4
L	PS	IM		S1N	S1C	S1R	SRC1_SEL	S0N	S0C	S0R	SRC0_SEL		+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP3 (page 10-32).

Instruction Field ALU_INST == OP3_INST_BIT_ALIGN_INT, opcode 12 (0xC).

Byte Align

Instructions **BYTE_ALIGN_INT**

Description Right-shifts eight bytes into a four-byte GPR.

```
dst = ({src0, src1} >> (8 * src2[1:0])) & 0xFFFFFFFF;
```

Microcode

C	DC	DR	DST_GPR		BS	ALU_INST	SN	S2C	SR	SRC2_SEL			+4
L	PS	IM		S1N	S1C	S1R	SRC1_SEL		S0N	S0C	S0R	SRC0_SEL	+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP3 (page 10-32).

Instruction Field ALU_INST == OP3_INST_BYTE_ALIGN_INT, opcode 13 (0xD).

Floating-Point Ceiling

Instructions **CEIL**

Description Floating-point ceiling.

```
dst = TRUNC(src0);
If ( (src0 > 0.0f) && (src0 != dst) ) {
    dst += 1.0f;
}
```

Microcode

C	DC	DR	DST_GPR	BS	ALU_INST	OMOD	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1C	S1R	SRC1_SEL	S0N	S0C	S0R	SRC0_SEL	+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_CEIL, opcode 18 (0x12).

Floating-Point Conditional Move If Equal

Instructions **CNDE**

Description Floating-point conditional move if equal.

Compares the first source operand with floating-point zero, and copies either the second or third source operand to the destination operand based on the result. Execution can be conditioned on a predicate set by the previous ALU instruction group. If the condition is not satisfied, the instruction has no effect, and control is passed to the next instruction.

The instruction specifies which one of four data elements in a four-element vector is operated on, and the result can be stored in any of the four elements of the destination GPR. Operands can be accessed using absolute addresses, or an index in a GPR or the address register (AR).

A fog value can be exported by merging a transcendental ALU result into the low-order bits of the vector destination. The active mask and predicate bit can be updated by the result.

```
If (src0 == 0.0f) {
    dst = src1;
}
Else {
    dst = src2;
}
```

Microcode

C	DE	DR	DST_GPR		BS	ALU_INST (**11000**)	S2N	S2E	S2R	SRC2_SEL		+4
L	PS	IM		S1N	S1E	S1R	SRC1_SEL		S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP3 (page 10-32).

Instruction Field ALU_INST == OP3_INST_CNDE, opcode 25 (0x19).

Integer Conditional Move If Equal

Instructions **CNDE_INT**

Description Integer conditional move if equal, based on signed or unsigned integer operand. Compare “CNDE” on page 67.

```
If (src0 == 0x0) {
    dst = src1;
}
Else {
    dst = src2;
}
```

Microcode

C	DE	DR	DST_GPR			BS	ALU_INST (**11000**)	S2N	S2E	S2R	SRC2_SEL		+4
L	PS	IM		S1N	S1E	S1R	SRC1_SEL		S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP3 (page 10-32).

Instruction Field ALU_INST == OP3_INST_CNDE_INT, opcode 28 (0x1C).

Floating-Point Conditional Move If Greater Than Or Equal

Instructions **CNDGE**

Description Floating-point conditional move if greater than or equal. Compare “CNDE” on page 67.

```
If (src0 >= 0.0f) {
    dst = src1;
}
Else {
    dst = src2;
}
```

Microcode

C	DE	DR	DST_GPR			BS	ALU_INST (**11000**)	S2N	S2E	S2R	SRC2_SEL		+4
L	PS	IM		S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL		+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP3 (page 10-32).

Instruction Field ALU_INST == OP3_INST_CNDGE, opcode 27 (0x1B).

Integer Conditional Move If Greater Than Or Equal

Instructions **CNDGE_INT**

Description Integer conditional move if greater than or equal, based on signed integer operand. Compare “CNDE” on page 67.

```
If (src0 >= 0x0) {
    dst = src1;
}
Else {
    dst = src2;
}
```

Microcode

C	DE	DR	DST_GPR		BS	ALU_INST (**11000**)	S2N	S2E	S2R	SRC2_SEL		+4
L	PS	IM		S1N	S1E	S1R	SRC1_SEL		S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP3 (page 10-32).

Instruction Field ALU_INST == OP3_INST_CNDGE_INT, opcode 30 (0x1E).

Floating-Point Conditional Move If Greater Than

Instructions **CNDGT**

Description Floating-point conditional move if greater than. Compare “CNDE” on page 67.

```
If (src0 > 0.0f) {
    dst = src1;
}
Else {
    dst = src2;
}
```

Microcode

C	DE	DR	DST_GPR			BS	ALU_INST **(11000)**	S2N	S2E	S2R	SRC2_SEL		+4
L	PS	IM		S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL		+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP3 (page 10-32).

Instruction Field ALU_INST == OP3_INST_CNDGT, opcode 26 (0x1A).

Integer Conditional Move If Greater Than

Instructions **CNDGT_INT**

Description Integer conditional move if greater than, based on signed integer operand. Compare “CNDE” on page 67.

```
If (src0 > 0x0) {
    dst = src1;
}
Else {
    dst = src2;
}
```

Microcode

C	DE	DR	DST_GPR		BS	ALU_INST (**11000**)	S2N	S2E	S2R	SRC2_SEL			+4
L	PS	IM		S1N	S1E	S1R	SRC1_SEL		S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP3 (page 10-32).

Instruction Field ALU_INST == OP3_INST_CNDGT_INT, opcode 29 (0x1D).

Double-Precision Floating-Point Conditional Move If Not Equal

Instructions **CNDNE_64**

Description Compares the src0 with floating-point zero, and copies either src1 or src2 to the destination operand based on the result.

```
If (src0 != 0.0f) {
    dst = src1;
    }
Else {
    dst = src2;
    }
```

The instruction specifies which one of two data elements in a four-element vector is operated on (the two dwords of a double-precision floating-point number), and the result can be stored in the wz or yx elements of the destination GPR.

Microcode

C	DC	DR	DST_GPR		BS	ALU_INST	SN	S2C	SR	SRC2_SEL		+4
L	PS	IM	S1N	S1C	S1R	SRC1_SEL	S0N	S0C	S0R	SRC0_SEL		+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP3 (page 10-32).

Instruction Field ALU_INST == OP3_INST_CNDNE_64, opcode 9 (0x9).

Scalar Cosine

Instructions cos

Description Input must be normalized from radians by dividing by 2*PI. The valid input domain is [-256, +256], which corresponds to an un-normalized input domain [-512*PI, +512*PI]. Out-of-range input results in float 1.

```
dst = ApproximateCos(src0);
```

Microcode

C	DC	DR	DST_GPR						BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM				S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_COS, opcode 142 (0x8E).

Cube Map

Instructions **CUBE**

Description Cubemap, using two operands (src0 = Rn.zzxy, src1 = Rn.yxzz). This reduction instruction must be executed on all four elements of a single vector. Reduction operations compute only one output, so the values in the output modifier (OMOD) and output clamp (CLAMP) fields must be the same for all four instructions. OMOD and CLAMP do not affect the Direct3D FaceID in the resulting W vector element.

This instruction is not available in the ALU.Trans unit.

```
dst.W = FaceID;
dst.Z = 2.0f * MajorAxis;
dst.Y = S cube coordinate;
dst.X = T cube coordinate;
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_CUBE, opcode 192 (0xC0).

Variable-Length Dot Product

Instructions **DOT**

Description Variable-length dot product. Cannot be used in transcendental or w channel. All other channels require MUL or DOT opcode in the previous channel. Channels z, y, and x add the result from the previous channel to this channel's MUL result. Previous channel results are not normalized. Each channel's result is written to a GPR (the x channel is not broadcast).

Do not use in the w channel.

Previous channel z is w channel's MUL or DOT result.
Previous channel y is z channel's MUL or DOT result.
Previous channel x is y channel's MUL or DOT result.

```
dst.n = src0 * src1 + dst.(n+1)
```

Example: Two dot 2's, results in z and x

```
w z  y x
mul dot mul dot
```

Example: One dot2, result in y

```
w z  y x
* mul dot *
```

Example: Dot3, result in y

```
w z  y x
mul dot dot *
```

Example: Dot3, result in x

```
w z  y x
* mul dot dot
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL				S0N	S0C	S0R	SRC0_SEL							+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_DOT, opcode 208 (0xD0).

Variable-Length Dot Product With IEEE Rules

Instructions **DOT_IEEE**

Description Variable-length dot product with IEEE rules for 0*anything.

Cannot be used in transcendental or w channel. All other channels require MUL or DOT opcode in previous channel. Channels z, y, and x add the result from the previous channel to this channel's MUL result. Previous channel results are not normalized. Each channel's result is written to a GPR (the x channel is not broadcast).

Do not use in the w element.

Previous z channel is w channel's MUL or DOT result.
Previous y channel is z channel's MUL or DOT result.
Previous x channel is y channel's MUL or DOT result.

```
dst.n = src0 * src1 + dst.(n+1)
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST						OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL					+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_DOT_IEEE, opcode 175 (0xAF).

Four-Channel Dot Product

Instructions **DOT4**

Description Four-channel dot product. This reduction instruction must be executed on all four channels of a single vector. Reduction operations compute only one output, so the values in the output modifier (OMOD) and output clamp (CLAMP) fields must be the same for all four instructions.

Only the PV.X register channel holds the result of this operation, and the processor selects this swizzle code in the bypass operation.

This instruction is not available in the ALU.Trans unit.

```
dst = srcA.W * srcB.W +
srcA.Z * srcB.Z +
srcA.Y * srcB.Y +
srcA.X * srcB.X;
```

Microcode

C	DC	DR	DST_GPR				BS	ALU_INST						OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_DOT4, opcode 190 (0xBE).

Four-Channel Dot Product, IEEE

Instructions **DOT4_IEEE**

Description Four-channel dot product that uses IEEE rules for zero times anything. This reduction instruction must be executed on all four channels of a single vector. Reduction operations compute only one output, so the values in the output modifier (OMOD) and output clamp (CLAMP) fields must be the same for all four instructions.

Only the PV.X register channel holds the result of this operation, and the processor selects this swizzle code in the bypass operation.

This instruction is not available in the ALU.Trans unit.

```
dst = srcA.W * srcB.W +
srcA.Z * srcB.Z +
srcA.Y * srcB.Y +
srcA.X * srcB.X;
```

Microcode

C	DC	DR	DST_GPR				BS	ALU_INST						OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_DOT4_IEEE, opcode 191 (0xBF).

Scalar Base-2 Exponent, IEEE

Instructions **`EXP_IEEE`**

Description Scalar base-2 exponent.

```
If (src0 == 0.0F) {
    dst = 1.0F;
}
Else {
    dst = Approximate2ToX(src0);
}
```

Microcode

C	DC	DR	DST_GPR	BS	ALU_INST	OMOD	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1C	S1R	SRC1_SEL	S0N	S0C	S0R	SRC0_SEL	+0

Format `ALU_WORD0` (page 10-23) and `ALU_WORD1_OP2` (page 10-26).

Instruction Field `ALU_INST == OP2_INST_EXP_IEEE`, opcode 129 (0x81).

Find First Bit Signed High

Instructions **FFBH_INT**

Description Find the first bit set in a positive integer from the MSB, or find the first bit clear in a negative integer from the MSB.

```
If (src0 == 0 or src0 == 0xFFFFFFFF) {
    dst = 0xFFFFFFFF;
}
Else {
    count = 0;
    sign = src0[31];
    while (src0[31];
        count = count + 1;
        src0 = src0 << 1;
    }
    dst = count;
}
```

Microcode

C	DC	DR	DST_GPR							BS	ALU_INST											OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM				S1N	S1C	S1R	SRC1_SEL							S0N	S0C	S0R	SRC0_SEL									+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_FFBH_INT, opcode 173 (0xAD).

Find First Bit Unsigned High

Instructions **FFBH_UINT**

Description Find the first bit set in an unsigned integer from the MSB.

```
If (src0 == 0) {
    dst = 0xFFFFFFFF;
}
Else {
    count = 0;
    while (src0[31] == 0) {
        count = count + 1;
        src0 = src0 << 1;
    }
    dst = count;
}
```

Microcode

C	DC	D R	DST_GPR							BS	ALU_INST									OMOD	W M	U P	U E M	S 1 A	S 0 A	+4
L	PS	IM					S 1 N	S1C		S 1 R	SRC1_SEL						S 0 N	S0C	S 0 R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_FFBH_UINT, opcode 171 (0xAB).

Find First Bit Signed Low

Instructions **FFBL_INT**

Description Find the first bit set in an integer from the LSB.

```
If (src0 == 0) {
    dst = 0xFFFFFFFF;
}
Else {
    count = 0;
    while (src0[0] == 0) {
        count = count + 1;
        src0 = src0 >> 1;
    }
    dst = count;
}
```

Microcode

C	DC	DR	DST_GPR			BS	ALU_INST						OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL					+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_FFBL_INT, opcode 172 (0xAC).

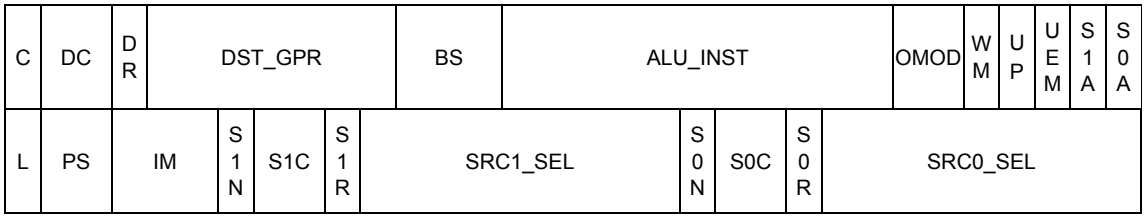
Floating-Point Floor

Instructions **FLOOR**

Description Floating-point floor.

```
dst = TRUNC(src0);
If ( (src0 < 0.0f) && (src0 != dst) ) {
    dst += -1.0f;
}
```

Microcode



Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_FLOOR, opcode 20 (0x14).

Floating-Point To Signed Integer

Instructions **FLT_TO_INT**

Description Floating-point input is converted to a signed integer value using truncation. Channels 0-3 use the 32-bit round mode state; channel 4 uses truncation. If the value does fit in 32 bits, the low-order bits are used. Special case number handling:

+inf -> max_int

-inf -> max_in

NaN & -Nan & 0 & -0 -> 0

```
dst = (int)src0
```

Microcode

C	DC	DR	DST_GPR				BS	ALU_INST					OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL					+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_FLT_TO_INT, opcode 80 (0x50).

Float to Signed Integer Using FLOOR

Instructions **FLT_TO_INT_FLOOR**

Description Float input is converted to a signed integer value using FLOOR. Float magnitudes too great to be represented by an integer float (unbiased exponent > 30) saturate to max_int or -max_int. This does not match the opcode FLT_TO_INT with round toward zero, which handles NaNs differently.

Special case number handling:

inf & NaN -> max_int

-inf & -NaN -> -max_int

0 & -0 -> 0

```
dst = (int)(FLOOR)src0
```

Microcode

C	DC	DR	DST_GPR			BS	ALU_INST				OMOD	WM	UP	UEM	S1A	S0A		+4
L	PS	IM		S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL					+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_FLT_TO_INT_FLOOR, opcode 177 (0xB1).

Convert Float Input to Signed Integer Value

Instructions **FLT_TO_INT_RPI**

Description Float input is converted to a signed integer value using round to positive infinity tiebreaker for 0.5. Float magnitudes too great to be represented by an integer float (unbiased exponent > 30) saturate to max_int or -max_int. This does not match the opcode FLT_TO_INT, which rounds and handles NaNs differently.

Special case number handling:

inf & NaN -> max_int

-inf & -NaN -> max_int

0 & -0 -> 0

```
dst = (int)(FLOOR)(src0 + 0.5)
```

Equivalently,

```
If (frac(arg) == 0.5)
    dst = (int)Ceil(arg);
Else
    dst = (int)Round_ne(arg);
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM				S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL							+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_FLT_TO_INT_RPI, opcode 176 (0xB0).

Floating-Point To Unsigned Integer

Instructions **FLT_TO_UINT**

Description Converts input to an unsigned integer value using truncation. Positive float magnitudes too great to be represented by an unsigned integer float (unbiased exponent > 31) saturate to `max_uint`.

Special number handling:

-inf & NaN & 0 & -0 -> 0

Inf -> max_uint

```
dst = (int)src0
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL						+0

Format `ALU_WORD0` (page 10-23) and `ALU_WORD1_OP2` (page 10-26).

Instruction Field `ALU_INST == OP2_INST_FLT_TO_UINT`; opcode 154 (0x9A).

Float to Unsigned Conversion of Four Floating Point Inputs

Instructions **FLT_TO_UINT4**

Description Float to unsigned conversion of four floating point inputs to packed eight-bit unsigned integer values. Uses all four vector channels. The 32-bit result is replicated to all four vector output channels.

```
Result = (flt_to_uint(src0.w) & 0xFF) << 24)) +
(flt_to_uint(src0.z) & 0xFF) << 16)) +
(flt_to_uint(src0.y) & 0xFF) << 8)) +
(flt_to_uint(src0.x) & 0xFF))) ;
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST						OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL					+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_FLT_TO_UINT4, opcode 174 (0xAE).

16-Bit Floating-Point to 32-Bit Floating-Point

Instructions **FLT16_TO_FLT32**

Description Conversion of 16-bit floating-point to 32-bit floating-point. src0 = input float16. Supports input and output modifiers. Denorms cannot be created because F16 does not have a large enough range. Float 16 denorms are accepted.

```
dst = FLT16_32(src0[15:0])
```

Microcode

C	DC	DR	DST_GPR			BS	ALU_INST					OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM		S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL					+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_FLT16_TO_FLT32, opcode 163 (0xA3).

Floating-Point 32-Bit To Floating-Point 16-Bit

Instructions **FLT32_TO_FLT16**

Description Floating-point 32-bit conversion to 16-bit floating-point.

This supports input modifiers. It creates FLt16 denorms when appropriate. This is not compatible with output modifiers because of the creation of denorms.

```
dst[15:0] = FLT32_16(src0)
```

Microcode

C	DC	DR	DST_GPR			BS	ALU_INST					OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM		S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL					+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_FLT32_TO_FLT16, opcode 162 (0xA2).

Input Modifiers Input modifiers (Section 4.7.2, on page 4-10) can be applied to the source operands during the destination X channel (slot 0) or Z channel (slot 2). These slots contain the sign bits of the sources.

Output Modifiers Output modifiers (Section 4.9.1, on page 4-25) can be applied to the destination during the destination X channel (slot 0) or Z channel (slot 2).

Floating-Point 32-Bit To Floating-Point 64-Bit

Instructions **FLT32_TO_FLT64**

Description Floating-point 32-bit convert to 64-bit floating-point. The instruction converts `src0.X` or `src0.Z` to a 64-bit double-precision floating-point value and places the result in `dst.YX` or `dst.ZW`, respectively. If the source value does fit in 32 bits, the low-order bits are used. Using values outside the specified range produces undefined results.

A 32-bit NaN source is handled specially. The sign is copied, the mantissa is copied into bits [52:30], and the exponent is forced to 0x7FF. The result for a NaN source is a NaN with the same sign, and the single-precision mantissa is the MSB of the double-precision mantissa.

```
dst = src0;

mant  = mantissa(src0)
exp   = exponent(src0)
sign  = sign(src0)

e = exp + (1023-127);

if (exp==0xFF)       //src0 is inf or a NaN
{
    If (mant!=0x0)    //src0 is a NaN
    {
        dst = {sign, 0x7FF, {mant,29'b0}};   //29 low-order bits are zero
    }
    else               //src0 is inf
    {
        dst = (sign) ? 0xFFF0000000000000 : 0x7FF0000000000000;
    }
}
else if (exp==0x0)    //src0 is zero or a denorm
{
    dst = (sign) ? 0x8000000000000000 : 0x0;
}
else                     //src0 is a valid floating-point value
{
    m = mant<<29;
    m |= (e << 52);
    m |= (sign << 63);

    dst = m;
}
```

Table 9.2 Result of FLT32_TO_FLT64 Instruction

src0										
-inf	**-F[1]**	**-1.0**	**-denorm**	**-0**	**+0**	**+denorm**	**+1.0**	**+F[1]**	**+inf**	**NaN**
-inf	-F	-1.0	-0.0	-0.0	+0.0	+0.0	+1.0	+F	+inf	NaN[2]

1. F is a finite floating-point value.
2. The hardware propagates a 32-bit input NaN to the output. So if the input is a 32-bit -/+ signaling NaN, the output is a 64-bit -/+ signaling NaN. A 32-bit -/+ quiet NaN returns a 64 bit -/+ quiet NaN. A 32-bit 0xFFC00000 NaN returns a 64 bit NaN64 (0xFFF8000000000000).

Coissue `FLT32_TO_FLT64` is a two-slot instruction. The following coissue scenarios are possible:.

- A single `FLT32_TO_FLT64` instruction in slots 0 and 1, and any valid instructions in slots 2, 3, and 4.
- A single `FLT32_TO_FLT64` instruction in slots 2 and 3, and any valid instructions in slots 0, 1, and 4.
- Two `FLT32_TO_FLT64` instructions in slots 0, 1, 2, and 3,and any valid instruction in slot 4.

Floating-Point 32-Bit To Floating-Point 64-Bit (Cont.)

Microcode

C	DC	DR	DST_GPR							BS	ALU_INST												OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM								S1N	S1C	S1R	SRC1_SEL							S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_FLT32_TO_FLT64,

for scalar operations used on vector or transcendental unit: opcode 29 (0x1D)
for vector operations on vector units only: opcode 206 (0xCE).

Example The following example coissues two FLT32_TO_FLT64 instructions in slots 0 and 1, and 2 and 3:

```
Input data:

Input data  0.5f (0x3F000000)
Input data  1.0f (0x3F800000)

mov ra.h, l (0x3F000000)    //Input 1
mov rb.l     //Don't care

mov rc.h, l(0x3F800000)     //Input 2
mov rd.l     //Don't care

Issue instructions:

FLT32_TO_FLT64 re.x ra.h   //can be any vector element
FLT32_TO_FLT64 rf.y rb.l   //Don't care
FLT32_TO_FLT64 rg.z rc.h   //can be any vector element
FLT32_TO_FLT64 rh.w rd.l   //Don't care

Result:

flt32_to_flt64(0.5f) = 0.5 (0x3FE0000000000000)
flt32_to_flt64(1.0f) = 1.0 (0x3FF0000000000000)

re.x = 0x00000000 (LSB of output)
rf.y = 0x3FE00000 (MSB of output)
rg.z = 0x00000000 (LSB of output)
rh.w = 0x3ff00000 (MSB of output)
```

Input Modifiers Input modifiers (Section 4.7.2, on page 4-10) can be applied to the source operands during the destination X channel (slot 0) or Z channel (slot 2). These slots contain the sign bits of the sources.

Output Modifiers Output modifiers (Section 4.9.1, on page 4-25) can be applied to the destination during the destination X channel (slot 0) or Z channel (slot 2).

Floating-Point 64-Bit To Floating-Point 32-Bit

Instructions **FLT64_TO_FLT32**

Description Floating-point 64-bit convert to 32-bit floating-point. The instruction converts `src0.YX` or `src0.WZ` to a 32-bit single-precision floating-point value in `dst.X` or `dst.Z`, respectively. If the result does fit in 32 bits, the low-order bits are used.

```
dst = src0;

mant  = mantissa(src0)
exp   = exponent(src0)
sign  = sign(src0)

if (exp==0x7FF)     //src0 is inf or a NaN
{
    if (mant==0x0)   //src0 is a NaN
    {
        dst = (sign) ? 0xFFC00000 : 0x7FC00000;
    }
    else              //src0 is inf
    {
        dst = (sign) ? 0xFF800000 : 0x7F800000;
    }
}
else if (exp==0x0)  //src0 is zero or a denorm
{
    dst = (sign) ? 0x80000000 : 0x0;
}
else                 //src0 is a valid floating-point value
{
    dst = src0;
}
```

Table 9.3 Result of FLT64_TO_FLT32 Instruction

src0											
-NaN	**-inf**	**-F[1]**	**-1.0**	**-denorm**	**-0**	**+0**	**+denorm**	**+1.0**	**+F[1]**	**+inf**	**+NaN**
0xFFC00000	-inf	-F	-1.0	-0.0	-0.0	+0.0	+0.0	+1.0	+F	+inf	0x7FC00000

1. F is a finite floating-point value.

Coissue `FLT64_TO_FLT32` is a two-slot instruction. The following coissues are possible.

- A single `FLT64_TO_FLT32` instruction in slots 0 and 1, and any valid instructions in slots 2, 3, and 4.
- A single `FLT64_TO_FLT32` instruction in slots 2 and 3, and any valid instructions in slots 0, 1, and 4.
- Two `FLT64_TO_FLT32` instructions in slots 0, 1, 2, and 3,and any valid instruction in slot 4.

Floating-Point 64-Bit To Floating-Point 32-Bit (Cont.)

Microcode

C	DC	D R	DST_GPR							BS	ALU_INST										OMOD	W M	U P	U E M	S 1 A	S 0 A	+4
L	PS	IM					S 1 N	S1C	S 1 R	SRC1_SEL					S 0 N	S0C	S 0 R	SRC0_SEL								+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_FLT64_TO_FLT32,
for scalar operations used on vector or transcendental unit: opcode 28 (0x1C)
for vector operations on vector units only: opcode 205 (0xCD)

Example The following example coissues two FLT64_TO_FLT32 instructions in slots 0 and 1, and 2 and 3.

```
Input data:

Input data  1.0 (0x3FF0000000000000)
Input data  2.0 (0x4000000000000000)

mov ra.h, l(0x3FF00000)   //high dword (Input 1)
mov rb.l, l(0x00000000)   //low  dword

mov rc.h, l(0x40000000)   //high dword (Input 2)
mov rd.l, l(0x00000000)   //low  dword

Issue instructions:

FLT64_TO_FLT32  re.x  ra.h //can be any vector element
FLT64_TO_FLT32  rf.y  rb.l //can be any vector element
FLT64_TO_FLT32  rg.z  rc.h //can be any vector element
FLT64_TO_FLT32  rh.w  rd.l //can be any vector element

Result:

flt64_to_flt32(1.0) = 1.0f (0x3F800000)
flt64_to_flt32(2.0) = 2.0f (0x40000000)

re.x  = 0x3F800000 (1.0f)
rf.y  = 0                   //Always 0
rg.z  = 0x40000000 (2.0f)
rh.w  = 0                 //Always 0
```

Input Modifiers Input modifiers (Section 4.7.2, on page 4-10) can be applied to the source operands during the destination X channel (slot 0) or Z channel (slot 2). These slots contain the sign bits of the sources.

Output Modifiers Output modifiers (Section 4.9.1, on page 4-25) can be applied to the destination during the destination X channel (slot 0) or Z channel (slot 2).

Fused Single-Precision Multiply-Add

Instructions **FMA**

Description Fused single-precision multiply-add. Only for double-precision parts.

```
dst = src0 * src1 + src2
```

Microcode

C	DC	DR	DST_GPR		BS	ALU_INST	SN	S2C	SR	SRC2_SEL		+4
L	PS	IM	S1N	S1C	S1R	SRC1_SEL	S0N	S0C	S0R	SRC0_SEL		+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP3 (page 10-32).

Instruction Field ALU_INST == OP3_INST_FMA, opcode 7 (0x7).

Double-Precision Floating-Point Fused Multiply-Add

Instructions **FMA_64**

Description Adds the src2 to the product of the src0 and src1. A single round is performed on the sum - the product of src0 and src1 is not truncated or rounded.

```
dst = (src0 * src1) + src2
```

The instruction specifies which one of two data elements in a four-element vector is operated on (the two dwords of a double precision floating point number), and the result can be stored in the wz or yx elements of the destination GPR.

Microcode

C	DC	DR	DST_GPR			BS	ALU_INST	SN	S2C	SR	SRC2_SEL	+4
L	PS	IM		S1N	S1C	S1R	SRC1_SEL	S0N	S0C	S0R	SRC0_SEL	+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP3 (page 10-32).

Instruction Field ALU_INST == OP3_INST_FMA_64, opcode 10 (0xA).

Floating-Point Fractional

Instructions **FRACT**

Description Floating-point fractional part of source operand.

`dst = src0 - FLOOR(src0);`

Microcode

C	DC	DR	DST_GPR	BS	ALU_INST	OMOD	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1C	S1R	SRC1_SEL	S0N	S0C	S0R	SRC0_SEL	+0

Format `ALU_WORD0` (page 10-23) and `ALU_WORD1_OP2` (page 10-26).

Instruction Field `ALU_INST == OP2_INST_FRACT`, opcode 16 (0x10).

Floating-Point Fractional, 64-Bit

Instructions **FRACT_64**

Description Gets the positive fractional part of a 64-bit floating-point value located in `src0.YX` or `src0.WZ`, and places the result in `dst.YX` or `dst.WZ`, respectively.

```
dst = src0;

mant  = mantissa(src0)
exp   = exponent(src0)
sign  = sign(src0)

if (exp==0x7FF)      //src0 is an inf or a NaN
{
    If (mant==0x0)    //src0 is NaN
    {
        dst = src0;
    }
    else              //src0 is inf
    {
        dst = NaN64;
    }
}
else if (exp==0x0)  //src0 is zero or a denorm
{
    dst = 0x0;
}
else                  //src0 is a float
{
    dst = src0 - floor(src0);
}
```

Table 9.4 Result of FRACT_64 Instruction

src0										
-inf	**-F[1]**	**-1.0**	**-denorm**	**-0**	**+0**	**+denorm**	**+1.0**	**+F[1]**	**+inf**	**NaN**
NaN64	[+0.0,+1.0)	+0	+0	+0	+0	+0	+0	[+0.0,+1.0)*	NaN64	NaN64

1. F is a finite floating-point value.

Coissue `FRACT_64` is a two-slot instruction. The following coissues are possible:.

- A single `FRACT_64` instruction in slots 0 and 1, and any valid instructions in slots 2, 3, and 4.
- A single `FRACT_64` instruction in slots 2 and 3, and any valid instructions in slots 0, 1, and 4.
- Two `FRACT_64` instructions in slots 0, 1, 2, and 3,and any valid instruction in slot 4.

Microcode

C	DC	DR	DST_GPR			BS	ALU_INST			OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM		S1N	S1C	S1R	SRC1_SEL		S0N	S0C	S0R	SRC0_SEL				+0

Format `ALU_WORD0` (page 10-23) and `ALU_WORD1_OP2` (page 10-26).

Floating-Point Fractional, 64-Bit (Cont.)

Instruction Field `ALU_INST == OP2_INST_FRACT_64`, opcode 198 (0xC6).

Example The following example coissues two `FRACT_64` instructions in slots 0 and 1, and 2 and 3.

```
Input data:

Input data 8.814369 (0x4021A0F4F077BCA7)
Input data 13.113172 (0x402A39F1A0AC1721)

mov ra.h, l(0x4021A0F4) //high dword (Input 1)
mov rb.l, l(0xF077BCA7) //low dword

mov rc.h, l(0x402A39F1) //high dword (Input 2)
mov rd.l, l(0xA0AC1721) // low dword

Issue instructions:

FRACT_64 re.x ra.h //can be any vector element
FRACT_64 rf.y rb.l //can be any vector element
FRACT_64 rg.z rc.h //can be any vector element
FRACT_64 rh.w rd.l //can be any vector element

Result:

fract64(0x4021A0F4F077BCA7) = fract64(8.814369)  = 0x3FEA0F4F077BCA70
(0.814369)
fract64(0x402A39F1A0AC1721) = fract64(13.113172) = 0x3FBCF8D0560B9080
(0.113172)


re.x = 0x077BCA70 (LSB of output)
rf.y = 0x3FEA0F4F (MSB of output)
rg.z = 0x560B9080 (LSB of output)
rh.w = 0x3FBCF8D0 (MSB of output)
```

Input Modifiers Input modifiers (Section 4.7.2, on page 4-10) can be applied to the source operands during the destination X channel (slot 0) or Z channel (slot 2). These slots contain the sign bits of the sources.

Output Modifiers Output modifiers (Section 4.9.1, on page 4-25) can be applied to the destination during the destination X channel (slot 0) or Z channel (slot 2).

Split Double-Precision Floating_Point Into Fraction and Exponent

Instructions **FREXP_64**

Description Splits the double-precision floating-point value in src0.YX into separate fraction (mantissa) and exponent values. The exponent is output as a signed integer to dst.YX. The fraction, in the range (-1.0f, -0.5f] or [0.5f, 1.0f), is output as a sign-extended double-precision value to dst.WZ.

```
dst = src0;

frac_src0  = fraction(src0)
exp_src0   = exponent(src0)
sign_src0  = sign(src0)
frac_dst   = fraction(dst)
exp_dst    = exponent(dst)

if (exp_src0==0x7FF)                //src0 is inf or NaN
{
    exp_dst = 0xFFFFFFFF;
    if (frac_src0==0x0)              //src0 is inf
    {
        frac_dst = 0xFFF8000000000000;
    }
    else                            //src0 is a NaN
    {
        frac_dst = src0;
    }
}
else if (exp_dst==0x0)             //src0 is zero or denorm
{
    exp_dst = 0x0;
    frac_dst = {sign_src0,0x0};
}
else                                  //src0 is a float
{
    frac_dst = {sign_src0, 0x3fe, frac_src0}; // double from (-1, -0.5] to
[0.5, 1)
    exp_dst  = exp_src0 - 1023 + 1;           // convert to 2's complement
}
```

Table 9.5 Result of FREXP_64 Instruction

	src0			
dst	-inf or +inf	-0 or +0	-denorm or +denorm	NaN
frac_dst	NaN64[1]	{sign_src0,0}	{sign_src0,0}	src0
exp_dst	0xFFFFFFFF	0	0	0xFFFFFFFF

1. NaN64 = 0xFFF8000000000000.

Coissue The instruction uses four slots in an instruction group. A single FREXP_64 instruction must be issued in slots 0, 1, 2, or 3. Slot 4 can contain any other valid instruction.

Split Double-Precision Floating_Point Into Fraction and Exponent (Cont.)

Microcode

C	DC	DR	DST_GPR			BS	ALU_INST				OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM		S1N	S1C	S1R	SRC1_SEL		S0N	S0C	S0R	SRC0_SEL					+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_FREXP_64, opcode 196 (0xC4).

Example The following example issues one FREXP_64 instruction in each of slots 0, 1, 2, and 3.

```
For src0 = 3.0 (0x4008000000000000):

mov ra.h , l(0x40080000)  //high dword (Input)
mov rb.l , l(0x00000000)  //low  dword

Issue instructions:

FREXP_64  rc.x ra.h;   //Can be any vector element in any GPR
FREXP_64  rd.y rb.l;   //Can be any vector element in any GPR
FREXP_64  re.z         //Don't care about source operand (not used)
FREXP_64  rf.w         //Don't care about source operand (not used)

Result:

rc.x = 0x0         (All bits are always zero)
rd.y = 2           (Exponent 0.75*2^2 = 3.0)
re.z = 0x0         (LSB of mantissa)
rf.w = 0x3FE80000  {s,0x3FE, MSB of mantissa}
```

Input Modifiers Input modifiers (Section 4.7.2, on page 4-10) can be applied to the source operand during the destination X channel (slot 0). This slot contains the sign bit of the source.

Output Modifiers The instruction does not take output modifiers.

Group Barrier

Instructions **GROUP_BARRIER**

Description Creates a synchronization point between all of the threads in a work-group. Every thread in the work-group must execute this instruction before any thread is allowed to proceed past this instruction. Once all threads have reached this instruction, they can proceed.

It is illegal to execute this instruction within dynamic flow control.

Microcode

C	DC	DR	DST_GPR			BS	ALU_INST					OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM		S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL					+0

Format `ALU_WORD0` (page 10-23) and `ALU_WORD1_OP2` (page 10-26).

Instruction Field `ALU_INST == OP2_INST_GROUP_BARRIER`, opcode 84 (0x54).

Begin of Group Sequence

Instructions **GROUP_SEQ_BEGIN**

Description Creates a synchronization point between all threads in a work-group and protects a section of kernel code, allowing only one thread at a time to execute that code. Every thread in the thread group must execute this instruction before any thread can proceed past this instruction. Once all threads have reached this instruction, the first thread in the work-group can proceed; it alone continues executing its kernel until it executes the GROUP_SEQ_END instruction, which signals that the next thread can begin executing. This first thread continues executing its kernel while the second one executes. All threads that executed GROUP_SEQ_BEGIN must eventually execute GROUP_SEQ_END.

Microcode

C	DC	DR	DST_GPR				BS	ALU_INST					OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL					+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_GROUP_SEQ_BEGIN, opcode 85 (0x55).

End Group Sequence

Instructions **GROUP_SEQ_END**

Description Marks the end of a critical section of code. See GROUP_SEQ_BEGIN for details.

Microcode

C	DC	DR	DST_GPR							BS	ALU_INST										OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM					S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL									+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_GROUP_SEQ_END, opcode 86 (0x56).

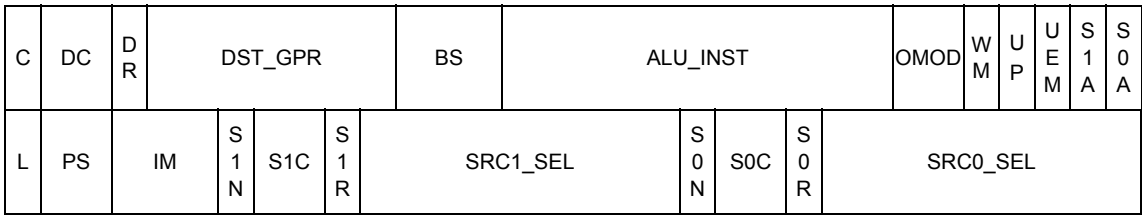
Integer To Floating-Point

Instructions **INT_TO_FLT**

Description Integer to floating-point. The input is interpreted as a signed integer value and converted to a floating-point value.

```
dst = (float) src0
```

Microcode



Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_INT_TO_FLT, opcode 155 (0x9B).

Read Parameter Data From LDS for P0

Instructions **INTERP_LOAD_P0**

Description Read parameter data from LDS and write it into GPRs for P0. Each primitive has parameters at the three vertices: P0, P1, and P2. The SRC0 argument contains the parameter number (0 – 32).

Microcode

C	DC	DR	DST_GPR							BS	ALU_INST											OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM							S1N	S1C	S1R	SRC1_SEL							S0N	S0C	S0R	SRC0_SEL						+0

Format `ALU_WORD0` (page 10-23) and `ALU_WORD1_OP2` (page 10-26).

Instruction Field `ALU_INST == OP2_INST_INTERP_LOAD_P0`, opcode 224 (0xE0).

Read Parameter Data from LDS for P1 - P0

Instructions **INTERP_LOAD_P10**

Description Read parameter data from LDS and write it into GPRs for (P1 – P0). Each primitive has parameters at the three vertices: P0, P1, and P2. The SRC0 argument contains the parameter number (0 – 32).

Microcode

C	DC	DR	DST_GPR						BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM				S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_INTERP_LOAD_P10, opcode 225 (0xE1).

Read Parameter Data from LDS for P2 - P0

Instructions **INTERP_LOAD_P20**

Description Read parameter data from LDS and write it into GPRs for (P2 – P0). Each primitive has parameters at the three vertices: P0, P1, and P2. The SRC0 argument contains the parameter number (0 – 32).

Microcode

C	DC	DR	DST_GPR							BS	ALU_INST											OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM							S1N	S1C	S1R	SRC1_SEL							S0N	S0C	S0R	SRC0_SEL						+0

Format `ALU_WORD0` (page 10-23) and `ALU_WORD1_OP2` (page 10-26).

Instruction Field `ALU_INST == OP2_INST_INTERP_LOAD_P20`, opcode 226 (0xE2).

Interpolation of the X Channel

Instructions **INTERP_X**

Description This opcode must be present in the x and y vector channels

Channel y – MULADD.

Channel x – DOT, using result of channel y MULADD.

Channel x is the result of interpolating the x parameters.

```
dst.x = P0.x + P1.x * i + P2.x * j
```

Note: The red + indicates this addition is done using the DOT path. The MULADD result is not rounded or normalized before entering this ADD, so the result is not an IEEE compliant ADD.

Microcode

C	DC	DR	DST_GPR				BS	ALU_INST						OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_INTERP_X, opcode 216 (0xD8).

Interpolation for X,Y Channels

Instructions **INTERP_XY**

Description This opcode must be present in all vector channels

Channel w – MULADD

Channel z – DOT, using result of channel w MULADD

Channel y – MULADD

Channel x – DOT, using result of channel y MULADD

Channel z result is interpolated y channel, and is muxed onto the channel y output.

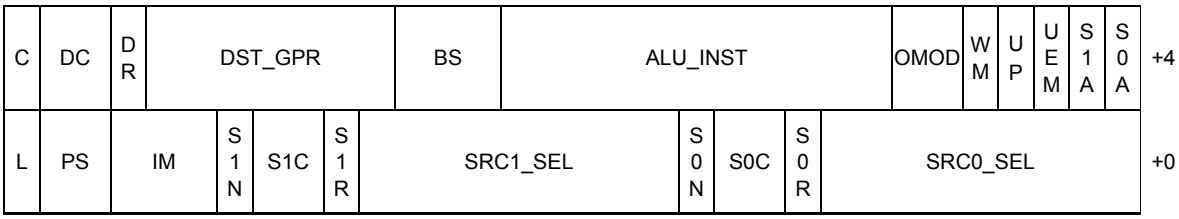
Channel x result is interpolated x channel.

```
dst.x = P0.x + P1.x * i + P2.x *j

dst.y = P0.y + P1.y * i + P2.x *j
```

Note: The red + indicates this addition is done using the DOT path. The MULADD result is not rounded or normalized before entering this ADD, so the result is not an IEEE compliant ADD.

Microcode



Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_INTERP_XY, opcode 214 (0xD6).

Interpolation of the Z Channel

Instructions **INTERP_Z**

Description This opcode must be present in the w and z vector channels.

Channel w – MULADD.

Channel z – DOT, using result of channel w MULADD.

Channel z is the result of interpolating the z parameters.

```
dst.z = P0.z + P1.z * i + P2.z * j
```

Note: The red + indicates this addition is done using the DOT path. The MULADD result is not rounded or normalized before entering this ADD, so the result is not an IEEE compliant ADD.

Microcode

C	DC	DR	DST_GPR			BS	ALU_INST					OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL				+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_INTERP_Z, opcode 217 (0xD9).

Interpolation of the Z, W Channels

Instructions **INTERP_ZW**

Description This opcode must be present in all vector channels.

Channel w – MULADD.

Channel z – DOT, using result of channel w MULADD.

Channel y – MULADD.

Channel x – DOT, using result of channel y MULADD.

Channel z is the result of interpolating the z parameters, and is muxed onto the channel y output.

Channel x result is interpolated x channel, and is muxed onto the channel w output.

```
dst.x = P0.x + P1.x * i + P2.x * j

dst.w = P0.w + P1.w * i + P2.w * j
```

Note: The red + indicates this addition is done using the DOT path. The MULADD result is not rounded or normalized before entering this ADD, so the result is not an IEEE compliant ADD.

Microcode

C	DC	DR	DST_GPR				BS	ALU_INST						OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM				S1N	S1C	S1R	SRC1_SEL				S0N	S0C	S0R	SRC0_SEL				+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_INTERP_ZW, opcode 215 (0xD7).

Floating-Point Pixel Kill If Equal

Instructions **KILLE**

Description Floating-point pixel kill if equal. Set kill bit. Ensure that the KILL* instruction is the last instruction in an ALU clause, because the remaining instructions executed in the clause do not reflect the updated valid state after the kill operation. Only a pixel shader (PS) can execute this instruction; the instruction is ignored in other program types.

```
If (src0 == src1) {
    dst = 1.0f;
    Killed = TRUE;
}
Else {
    dst = 0.0f;
}
```

Microcode

C	DC	DR	DST_GPR						BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM				S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_KILLE, opcode 44 (0x2C).

Integer Kill If Equal

Instructions **KILLE_INT**

Description Integer kill if equal. Set kill bit. Ensure that the KILL* instruction is the last instruction in an ALU clause, because the remaining instructions executed in the clause do not reflect the updated valid state after the kill operation. Only a pixel shader (PS) can execute this instruction; the instruction is ignored in other program types.

```
If (src0 == src1) {
    dst = 1.0f;
    Killed = TRUE;
}
Else {
    dst = 0.0f;
}
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM				S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL					+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_KILLE_INT, opcode 70 (0x46).

Floating-Point Pixel Kill If Greater Than Or Equal

Instructions **KILLGE**

Description Floating-point pixel kill if greater than or equal. Set kill bit. Ensure that the KILL* instruction is the last instruction in an ALU clause, because the remaining instructions executed in the clause do not reflect the updated valid state after the kill operation. Only a pixel shader (PS) can execute this instruction; the instruction is ignored in other program types.

```
If (src0 >= src1) {
    dst = 1.0f;
    Killed = TRUE;
}
Else {
    dst = 0.0f;
}
```

Microcode

C	DC	DR	DST_GPR						BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM				S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_KILLGE, opcode 46 (0x2E).

Integer Kill If Greater Than Or Equal

Instructions **KILLGE_INT**

Description Unsigned integer kill if greater than or equal. Set kill bit. Ensure that the KILL* instruction is the last instruction in an ALU clause, because the remaining instructions executed in the clause do not reflect the updated valid state after the kill operation. Only a pixel shader (PS) can execute this instruction; the instruction is ignored in other program types.

```
If (src0 >= src1) {
    dst = 1.0f;
    Killed = TRUE;
}
Else {
    dst = 0.0f;
}
```

Microcode

C	DC	DR	DST_GPR				BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL					+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_KILLGE_INT, opcode 72 (0x48).

Unsigned Integer Kill If Greater Than Or Equal

Instructions **KILLGE_UINT**

Description Unsigned integer kill if greater than or equal. Set kill bit. Ensure that the KILL* instruction is the last instruction in an ALU clause, because the remaining instructions executed in the clause do not reflect the updated valid state after the kill operation. Only a pixel shader (PS) can execute this instruction; the instruction is ignored in other program types.

```
If (src0 >= src1) {
    dst = 1.0f;
    Killed = TRUE;
}
Else {
    dst = 0.0f;
}
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL				S0N	S0C	S0R	SRC0_SEL							+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_KILLGE_UINT; opcode 65 (0x41).

Floating-Point Pixel Kill If Greater Than

Instructions **KILLGT**

Description Floating-point pixel kill if greater than. Set kill bit. Ensure that the KILL* instruction is the last instruction in an ALU clause, because the remaining instructions executed in the clause do not reflect the updated valid state after the kill operation. Only a pixel shader (PS) can execute this instruction; the instruction is ignored in other program types.

```
If (src0 > src1) {
    dst = 1.0f;
    Killed = TRUE;
}
Else {
    dst = 0.0f;
}
```

Microcode

C	DC	DR	DST_GPR			BS	ALU_INST					OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL				+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_KILLGT, opcode 45 (0x2D).

Integer Kill If Greater Than

Instructions **KILLGT_INT**

Description Integer kill if greater than. Set kill bit. Ensure that the KILL* instruction is the last instruction in an ALU clause, because the remaining instructions executed in the clause do not reflect the updated valid state after the kill operation. Only a pixel shader (PS) can execute this instruction; the instruction is ignored in other program types.

```
If (src0 > src1) {
    dst = 1.0f;
    Killed = TRUE;
}
Else {
    dst = 0.0f;
}
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM				S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL					+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_KILLGT_INT, opcode 71 (0x47).

Unsigned Integer Kill If Greater Than

Instructions **KILLGT_UINT**

Description Unsigned integer kill if greater than. Set kill bit. Ensure that the KILL* instruction is the last instruction in an ALU clause, because the remaining instructions executed in the clause do not reflect the updated valid state after the kill operation. Only a pixel shader (PS) can execute this instruction; the instruction is ignored in other program types.

```
If (src0 > src1) {
    dst = 1.0f;
    Killed = TRUE;
}
Else {
    dst = 0.0f;
}
```

Microcode

C	DC	DR	DST_GPR			BS	ALU_INST					OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM		S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL					+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_KILLGT_UINT, opcode 64 (0x40).

Floating-Point Pixel Kill If Not Equal

Instructions **KILLNE**

Description Floating-point pixel kill if not equal. Set kill bit. Ensure that the KILL* instruction is the last instruction in an ALU clause, because the remaining instructions executed in the clause do not reflect the updated valid state after the kill operation. Only a pixel shader (PS) can execute this instruction; the instruction is ignored in other program types.

```
If (src0 != src1) {
    dst = 1.0f;
    Killed = TRUE;
}
Else {
    dst = 0.0f;
}
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM				S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL					+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_KILLNE, opcode 47 (0x2F).

Integer Kill If Not Equal

Instructions **KILLNE_INT**

Description Integer kill if not equal. Set kill bit. Ensure that the KILL* instruction is the last instruction in an ALU clause, because the remaining instructions executed in the clause do not reflect the updated valid state after the kill operation. Only a pixel shader (PS) can execute this instruction; the instruction is ignored in other program types.

```
If (src0 != src1) {
    dst = 1.0f;
    Killed = TRUE;
}
Else {
    dst = 0.0f;
}
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_KILLNE_INT, opcode 73 (0x49).

Combine Separate Fraction and Exponent into Double-precision

Instructions **LDEXP_64**

Description The LDEXP_64 instruction gets a 52-bit mantissa from the double-precision floating-point value in src1.YX and a 32-bit integer exponent in src0.X, and multiplies the mantissa by $2^{exponent}$. The double-precision floating-point result is stored in dst.YX.

```
dst = src1 * 2^src0

mant  = mantissa(src1)
exp   = exponent(src1)
sign  = sign(src1)

if (exp==0x7FF)             //src1 is inf or a NaN
{
    dst = src1;
}
    else if (exp==0x0)       //src1 is zero or a denorm
{
    dst = (sign) ? 0x8000000000000000 : 0x0;
}
else                           //src1 is a float
{
    exp+= src0;
   if (exp>=0x7FF)                //overflow
    {
        dst = {sign,inf};
    }
    if (src0<=0)              //underflow
    {
        dst = {sign,0};
    }

    mant |= (exp<<52);
    mant |= (sign<<63);

    dst = mant;
}
```

Table 9.6 Result of LDEXP_64 Instruction

	src0				
src1	**-/+inf**	**-/+denorm**	**-/+0**	**-/+F[1]**	**NaN**
-/+I[2]	-/+inf	-/+0	-/+0	src1 * (2^src0)	src0
Not -/+I	-/+inf	-/+0	-/+0	invalid result	src0

1. F is a finite floating-point value.
2. I is a valid 32-bit integer value.

Coissue LDEXP_64 is a two-slot instruction. The following coissues are possible:

A single LDEXP_64 instruction in slots 0 and 1, and any valid instructions in slots 2, 3, and 4.

A single LDEXP_64 instruction in slots 2 and 3, and any valid instructions in slots 0, 1, and 4.

Two LDEXP_64 instructions in slots 0, 1, 2, and 3,and any valid instruction in slot 4.

Combine Separate Fraction and Exponent into Double-precision (Cont.)

Microcode

C	DC	DR	DST_GPR				BS	ALU_INST						OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_LDEXP_64, opcode 197 (0xC5).

Example The following example coissues two LDEXP_64 instructions in slots 0 and 1, and 2 and 3.

```
Input data:

Input data (x1) 0x47F000006FC6A731
Input data (e1) 0x2C6
Input data (x2) 0xC7EFFFFEE072B19F
Input data (e2) 0x15E

mov ra.h, l(0x47F00000)    //high dword x1(Input 1)
mov rb.l, l(0x6FC6A731)    //low  dword

mov rc.h, l(0xC7EFFFFE)    //high dword x2(Input 2)
mov rd.l, l(0xE072B19F)    //low  dword

mov rj.h, l(0x2C6)         //e1
mov rk.l, l(0x15E)         //e2

Issue instructions:

LDEXP_64  re.x ra.h rj.h   //can be any vector element
LDEXP_64  rf.y rb.l rj.h   //can be any vector element
LDEXP_64  rg.z rc.h rk.l   //can be any vector element
LDEXP_64  rh.w rd.l rk.l   //can be any vector element

Result:

re.x  = 0x6FC6A731 (output LSB)
rf.y  = 0x74500000 (output MSB)
rg.z  = 0xE072B19F (output LSB)
rh.w  = 0xDDCFFFFE (output MSB)
```

Input Modifiers Input modifiers (Section 4.7.2, on page 4-10) can be applied to the src0 operand during the destination X channel (slot 0) or Z channel (slot 2). These slots contain the sign bits of the sources. The src1 operand is an integer and does not accept modifiers.

Output Modifiers Output modifiers (Section 4.9.1, on page 4-25) can be applied to the destination during the destination X channel (slot 0) or Z channel (slot 2).

Linear Interpolation

Instructions **LERP_UINT**

Description Unsigned eight-bit pixel average. Src c is similar to a round mode. If set, .5 rounds up; if cleared, .5 truncates.

```
dst = ((src0[31:24] + src1[31:24] + src2[24]) >> 1) << 24 +
((src0[23:16] + src1[23:16] + src2[16]) >>1) << 16 +
((src0[15:8] + src1[15:8] + src2[8]) >> 1) << 8 +
((src0[7:0] + src1[7:0] + src2[0]) >> 1);
```

Microcode

C	DC	DR	DST_GPR		BS	ALU_INST	SN	S2C	SR	SRC2_SEL		+4
L	PS	IM	S1N	S1C	S1R	SRC1_SEL	S0N	S0C	S0R	SRC0_SEL		+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP3 (page 10-32).

Instruction Field ALU_INST == OP3_INST_LERP_UINT, opcode 11 (0xB).

Load and Store Flags

Instructions **LOAD_STORE_FLAGS**

Description Load src0 into a working copy of the exception flags. This clears flags or restores flags from a previous clause.

Writes a working copy of the exception flags into a GPR if `gprwr` is enabled. Flags are inclusive of current VLIW.

Available only in the w channel.
`dst = exception flags`

Microcode

C	DC	DR	DST_GPR			BS	ALU_INST	SN	S2C	SR	SRC2_SEL	+4
L	PS	IM		S1N	S1C	S1R	SRC1_SEL	S0N	S0C	S0R	SRC0_SEL	+0

Format `ALU_WORD0` (page 10-23) and `ALU_WORD1_OP3` (page 10-32).

Instruction Field `ALU_INST == OP2_INST_LOAD_STORE_FLAGS`, opcode 219 (0xDB).

Scalar Base-2 Log

Instructions **LOG_CLAMPED**

Description Scalar base-2 log.

```
If (src0 == 1.0f) {
    dst = 0.0f;
}
Else {
    dst = LOG_IEEE(src0)
// clamp dst
if (dst == -INFINITY) {
    dst = -MAX_FLOAT;
}
```

Microcode

C	DC	DR	DST_GPR							BS	ALU_INST									OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM				S1N	S1C		S1R	SRC1_SEL						S0N	S0C	S0R	SRC0_SEL							+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_LOG_CLAMPED, opcode 130 (0x82).

Scalar Base-2 IEEE Log

Instructions **LOG_IEEE**

Description Scalar base-2 IEEE log.

```
If (src0 == 1.0f) {
    dst = 0.0f;
}
Else {
    dst = ApproximateLog2(src0);
}
```

Microcode

C	DC	DR	DST_GPR				BS	ALU_INST					OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM		S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_LOG_IEEE, opcode 131 (0x83).

Scalar Logical Shift Left

Instructions **LSHL_INT**

Description Scalar logical shift left. Zero is shifted into the vacated locations. `src1` is interpreted as an unsigned integer. If `src1` is > 31, the result is 0.

```
dst = src0 << src1
```

Microcode

C	DC	DR	DST_GPR						BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM				S1N	S1C	S1R	SRC1_SEL				S0N	S0C	S0R	SRC0_SEL							+0

Format `ALU_WORD0` (page 10-23) and `ALU_WORD1_OP2` (page 10-26).

Instruction Field `ALU_INST == OP2_INST_LSHL_INT`, opcode 23 (0x17).

Scalar Logical Shift Right

Instructions **LSHR_INT**

Description Scalar logical shift right. Zero is shifted into the vacated locations. The five lsb of `src1` are interpreted as an unsigned integer.

```
dst = src0 >> (src1 & 0x1F)
```

Microcode

C	DC	DR	DST_GPR							BS	ALU_INST											OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM							S1N	S1C	S1R	SRC1_SEL							S0N	S0C	S0R	SRC0_SEL						+0

Format `ALU_WORD0` (page 10-23) and `ALU_WORD1_OP2` (page 10-26).

Instruction Field `ALU_INST == OP2_INST_LSHR_INT`, opcode 22 (0x16).

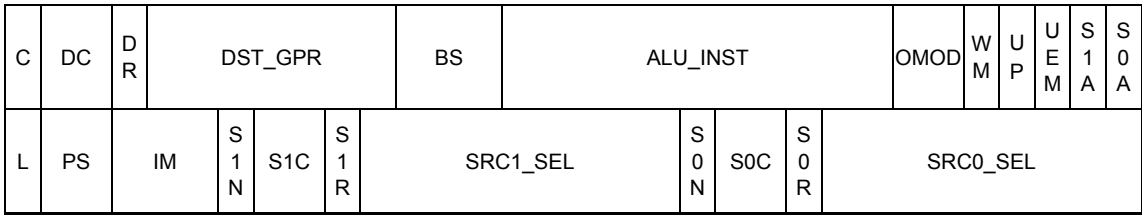
Floating-Point Maximum

Instructions **MAX**

Description Floating-point maximum.

```
If (src0 >= src1) {
    dst = src0;
}
Else {
    dst = src1;
}
```

Microcode



Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_MAX, opcode 3 (0x3).

Double-Precision Floating-Point Maximum

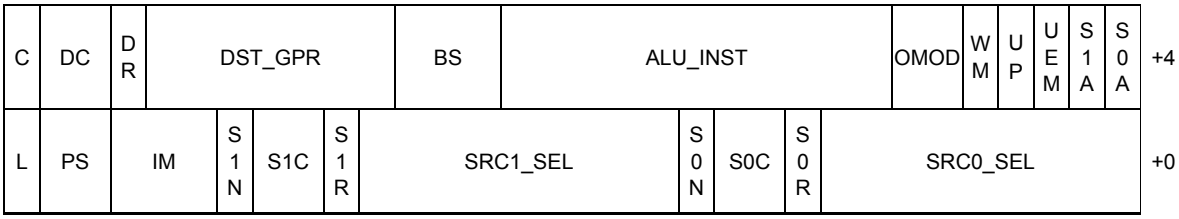
Instructions **MAX_64**

Description The instruction specifies which one of two data elements in a four-element vector is operated on (the two dwords of a double precision floating point number), and the result can be stored in the wz or yx elements of the destination GPR.

```
if (src0 > src1)
  dst = src0;
else
  dst = src1;

max(-0,+0)=max(+0,-0)=+0
```

Microcode



Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_MAX_64, opcode 189 (0xBD).

Floating-Point Maximum, DirectX 10

Instructions **MAX_DX10**

Description Floating-point maximum. This instruction uses the DirectX 10 method of handling of NaNs.

```
If (src0 >= src1) {
    dst = src0;
}
Else {
    dst = src1;
}
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST						OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL							+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_MAX_DX10, opcode 5 (0x5).

Integer Maximum

Instructions **MAX_INT**

Description Integer maximum, based on signed integer operands.

```
If (src0 >= src1) {
    dst = src0;
}
Else {
    dst = src1;
}
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_MAX_INT, opcode 54 (0x36).

Unsigned Integer Maximum

Instructions **MAX_UINT**

Description Integer maximum, based on unsigned integer operands.

```
If (src0 >= src1) {
    dst = src0;
}
Else {
    dst = src1;
}
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST						OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL							+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_MAX_UINT; opcode 56 (0x38).

Four-Channel Maximum

Instructions **MAX4**

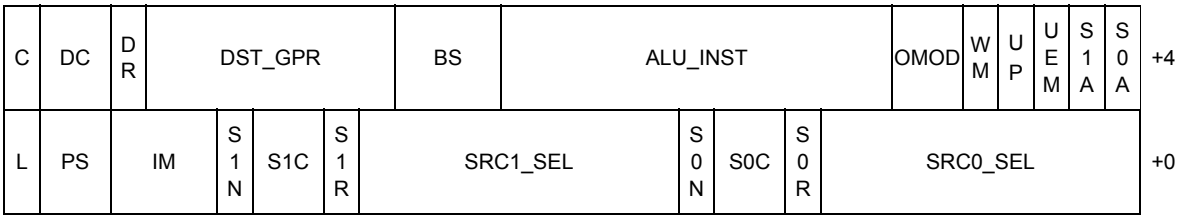
Description Four-channel maximum. The result is replicated in all four vector channels. This reduction instruction must be executed on all four channels of a single vector. Reduction operations compute only one output, so the values in the output modifier (OMOD) and output clamp (CLAMP) fields must be the same for all four instructions.

Only the PV.X register channel holds the result of this operation, and the processor selects this swizzle code in the bypass operation.

This instruction is not available in the ALU.Trans unit.

```
dst = max(srcA.W, srcA.Z, srcA,Y, srcA.X);
```

Microcode



Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_MAX4, opcode 193 (0xC1).

Masked Count Bits Set 32 High

Instructions **MBCNT_32HI_INT**

Description ANDs the high 32 bits of the mask based on the thread position (0-63) in wavefront before performing BCNT.

```
Masked_src0 = ((1 < thread_position) -1) >> 32 & src0
count = 0;
    for (i = 0 to 31) {
        count = count + Masked_src0[i];
}
dst = count;
```

See MBCNT_32LO_ACCUM_PREV_INT for intended usage.

Microcode

C	DC	DR	DST_GPR						BS	ALU_INST								OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM				S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL							+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_MBCNT_32HI_INT, opcode 179 (0xB3).

Masked Count Bits Set 32 Low

Instructions **MBCNT_32LO_ACCUM_PREV_INT**

Description ANDs the low 32 bits of the mask based on thread position (0-63) in a wavefront before executing BCNT. Adds this to the previous channel's MBCNT_32HI_INT result. This only works with MBCNT_32HI_INT in the previous channel.

Not legal if used in channel w. The previous channel z is channel w's BCNT_INT result. The previous channel y is channel z's BCNT_INT result. The previous channel x is channel y's BCNT_INT result.

```
Masked _src0 = (1 << thread_position) - 1 & src0;
count = 0;
    for (i = 0 to 31) {
        count = count + Masked_src0[i];
}
dst = count + prev_slot_mbcnt_32hi_int_dst;
```

Intended usage for compaction:

SQ constant {w,z,y,x} = {active_mask_high, active_mask_low, active_mask_high, active_mask_low}

w channel - bcnt_int
z channel - bcnt_accum_prev_int
y channel - mbcnt_32_hi_int
x channel - mbcnt_32lo_accum_prev_int

The z channel dst is the number of threads in a wave.

The x channel dst is the position of this thread in the wave.

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST						OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL				S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_MBCNT_32LO_ACCUM_PREV_INT, opcode 183 (0xB7).

Floating-Point Minimum

Instructions **MIN**

Description Floating-point minimum.

```
If (src0 < src1) {
    dst = src0;
}
Else {
    dst = src1;
}
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST						OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL				S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_MIN, opcode 4 (0x4).

Double-Precision Floating-Point Minimum

Instructions **MIN_64**

Description The instruction specifies which one of two data elements in a four-element vector is operated on (the two dwords of a double precision floating point number), and the result can be stored in the wz or yx elements of the destination GPR.

```
if (src0 < src1)
  dst = src0;
else
  dst = src1;

min(-0,+0)=min(+0,-0)=-0
```

Microcode

C	DC	DR	DST_GPR			BS	ALU_INST				OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM		S1N	S1C	S1R	SRC1_SEL		S0N	S0C	S0R	SRC0_SEL					+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_MIN_64, opcode 188 (0xBC).

Floating-Point Minimum, DirectX 10

Instructions **MIN_DX10**

Description Floating-point minimum. This instruction uses the DirectX 10 method of handling of NaNs.

```
If (src0 < src1) {
    dst = src0;
}
Else {
    dst = src1;
}
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST						OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL							+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_MIN_DX10, opcode 6 (0x6).

Signed Integer Minimum

Instructions **`MIN_INT`**

Description Integer minimum, based on signed integer operands.

```
If (src0 < src1) {
    dst = src0;
}
Else {
    dst = src1;

}
```

Microcode

C	DC	DR	DST_GPR	BS	ALU_INST	OMOD	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1C	S1R	SRC1_SEL	S0N	S0C	S0R	SRC0_SEL	+0

Format `ALU_WORD0` (page 10-23) and `ALU_WORD1_OP2` (page 10-26).

Instruction Field `ALU_INST == OP2_INST_MIN_INT`, opcode 55 (0x37).

Unsigned Integer Minimum

Instructions **MIN_UINT**

Description Integer minimum, based on unsigned integer operands.

```
If (src0 < src1) {
    dst = src0;
}
Else {
    dst = src1;
}
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST						OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL							+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_MIN_UINT, opcode 57 (0x39).

Copy To GPR

Instructions **MOV**

Description Copy a single operand from a GPR, constant, or previous result to a GPR.

MOV can be used as an alternative to the NOP instruction. Unlike NOP, which does not preserve the current PV or PS register value in the slot in which it executes, a MOV can be made to preserve PV and PS register values if the it is performed with a disabled write mask.

```
dst = src0
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST								OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL							+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_MOV, opcode 25 (0x19).

Copy Signed Integer To Integer in AR and GPR

Instructions **MOVA_INT**

Description Clamp the signed integer to the range [-256, +255], and copy the result to the address register (AR) and to a GPR.

```
dst = Undefined;
dstI = src0;
If (dstI < -256) {
    dstI = 0x100; //-256
}
If (dstI > 0xFF) {
    dstI = 0x100 //-256
}
Export(dstI); // signed 9-bit integer
```

Microcode

C	DC	DR	DST_GPR				BS	ALU_INST				OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL				+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_MOVA_INT, opcode 204 (0xCC).

Floating-Point Multiply

Instructions **MUL**

Description Floating-point multiply. Zero times anything equals zero.

```
dst = src0 * src1;
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_MUL, opcode 1 (0x1).

Floating-Point Multiply, 64-Bit

Instructions **MUL_64**

Description Floating-point 64-bit multiply. Multiplies a double-precision value in src0.YX by a double-precision value in src1.YX, and places the lower 64 bits of the result in dst.YX.

```
dst = src0 * src1;
```

Table 9.7 Result of MUL_64 Instruction

	src1										
src0	**-inf**	**-F[1]**	**-1.0**	**-denorm**	**-0**	**+0**	**+denorm**	**+1.0**	**+F[1]**	**+inf**	**NaN[2]**
-inf	+inf	+inf	+inf	NaN64	NaN64	NaN64	NaN64	-inf	-inf	-inf	src1 (NaN64)
-F	+inf	+F	-src0	+0	+0	-0	-0	src0	-F	-inf	src1 (NaN64)
-1.0	+inf	-src1	+1.0	+0	+0	-0	-0	-1.0	-src1	-inf	src1 (NaN64)
-denorm	NaN64	+0	+0	+0	+0	-0	-0	-0	-0	NaN64	src1 (NaN64)
-0	NaN64	+0	+0	+0	+0	-0	-0	-0	-0	NaN64	src1 (NaN64)
+0	NaN64	-0	-0	-0	-0	+0	+0	+0	+0	NaN64	src1 (NaN64)
+denorm	NaN64	-0	-0	-0	-0	+0	+0	+0	+0	NaN64	src1 (NaN64)
+1.0	-inf	src1	-1.0	-0	-0	+0	+0	+1.0	src1	+inf	src1 (NaN64)
+F	-inf	-F	-src0	-0	-0	+0	+0	src0	+F	+inf	src1 (NaN64)
+inf	-inf	-inf	-inf	NaN64	NaN64	NaN64	NaN64	+inf	+inf	+inf	src1 (NaN64)
NaN	src0 (NaN64)	src0 (NaN64)	src0 (NaN64)	src0 (NaN64)	src0 (NaN64)	src0 (NaN64)	src0 (NaN64)	src0 (NaN64)	src0 (NaN64)	src0 (NaN64)	src0 (NaN64)

1. F is a finite floating-point value.
2. NaN64 = 0xFFF8000000000000. An NaN64 is a propagated NaN value from the input listed.

```
(A * B) == (B * A)
```

Coissue The MUL_64 instruction is a four-slot instruction. Therefore, a single MUL_64 instruction can be issued in slots 0, 1, 2, and 3. Slot 4 can contain any other valid instruction.

Microcode

C	DC	DR	DST_GPR				BS	ALU_INST					OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL					+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Floating-Point Multiply, 64-Bit (Cont.)

Instruction Field ALU_INST == OP2_INST_MUL_64,
for scalar operations used on vector or transcendental unit: opcode 27 (0x1B)
for vector operations on vector units only: opcode 202 (0xCA).

Example The following example coissues one MUL_64 instruction in slots 0, 1, 2, and 3:

```
Input data:

Input data  3.0 (0x4008000000000000)
Input data  6.0 (0x4018000000000000)

mov ra.h, l(0x40080000)   //high dword (Input 1)
mov rb.l, l(0x00000000)   //low  dword

mov rc.h, l(0x40180000)   //high dword (Input 2)
mov rd.l, l(0x00000000)   //low  dword

Issue instruction:

MUL_64  re.x ra.h rc.h; //can be any vector element
MUL_64  rf.y ra.h rc.h; //can be any vector element
MUL_64  rg.z ra.h rc.h; //can be any vector element
MUL_64  rh.w rb.l rd.l; //can be any vector element

Result:

3.0 * 6.0 = 18.0 (0x4032000000000000)

re.x = 0x00000000    (LSB of Input 1 and Input 2 mul64 result)
rf.y = 0x40320000    (MSB of Input 1 and Input 2 mul64 result)
rg.z = 0x00000000    (LSB of Input 1 and Input 2 mul64 result)
rh.w = 0x40320000    (MSB of Input 1 and Input 2 mul64 result)

The hardware puts the result in two different slot pairs, as shown above.
```

Input Modifiers Input modifiers (Section 4.7.2, on page 4-10) can be applied to the source operands during the destination X channel (slot 0), Y channel (slot 1), or Z channel (slot 2). These slots contain the sign bits of the sources.

Output Modifiers Output modifiers (Section 4.9.1, on page 4-25) can be applied to the destination during the destination X channel (slot 0) or Z channel (slot 2).

Floating-Point Multiply, IEEE

Instructions **`MUL_IEEE`**

Description Floating-point multiply. Uses IEEE rules for zero times anything.

```
dst = src0 * src1;
```

Microcode

C	DC	DR	DST_GPR	BS	ALU_INST	OMOD	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1C	S1R	SRC1_SEL	S0N	S0C	S0R	SRC0_SEL		+0

Format `ALU_WORD0` (page 10-23) and `ALU_WORD1_OP2` (page 10-26).

Instruction Field `ALU_INST == OP2_INST_MUL_IEEE`, opcode 2 (0x2).

Dependent Multiply with IEEE Rules

Instructions **MUL_IEEE_PREV**

Description Multiply src1 by previous channel's MUL result. The previous channel opcode must be MUL.

This instruction uses DX10 rules for 0*anything.

The output modifier and clamping on the w/z channel is not applied in the MUL_PREV path to the dependent channel (y/x).

Use in w or z channels is not legal.

The previous channel y is the w channel's MUL result.
The previous channel x is the z channel's MUL result.

```
Result = Arg2 * prev_slot_result
```

Example: Dependent mul, result in y

```
    w z y  x
   mul * mul_prev *
```

Example: 2 separate dependent muls, results in y and x

```
    w z y  x
   mul mul mul_prev mul_prev
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST						OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL				S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_MUL_IEEE_PREV, opcode 210 (0xD2).

Scalar Multiply Emulating LIT Operation

Instructions **MUL_LIT**

Description Scalar multiply with result replicated in all four vector channels. It is used primarily when emulating a LIT operation. Zero times anything is zero.

A LIT operation takes an input vector containing information about shininess and normals to the light, and it computes the diffuse and specular light components using Blinn's lighting equation, which is implemented as follows.

```
t1.y = max (src.x, 0)
t1.x_w -= 1
t1.z = log_clamp( src.y)
t1.w = mul_lit( src.z, t1.z, src.x)
t1.z = exp(t1.z)
dst = t1
```

The pseudocode for the MUL_LIT instruction is:

```
If ((src1 == -MAX_FLOAT) ||
    (src1 == -INFINITY) ||
    (src1 is NaN) ||
    (src2 <= 0.0f) ||
    (src2 is NaN)) {
    dst = -MAX_FLOAT;
}
Else {
    dst = src0 * src1;
}
```

Microcode

C	DC	DR	DST_GPR		BS	ALU_INST	SN	S2C	SR	SRC2_SEL		+4
L	PS	IM	S1N	S1C	S1R	SRC1_SEL	S0N	S0C	S0R	SRC0_SEL		+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP3 (page 10-32).

Instruction Field ALU_INST == OP3_INST_MUL_LIT, opcode 31 (0x1F).

Dependent Multiply

Instructions **MUL_PREV**

Description Multiply src1 by previous channel's MUL result. The previous channel opcode must be MUL.

The output modifier and clamping on the w/z channel is not applied in the MUL_PREV path to the dependent channel (y/x).

Use in w or z channels is not legal.

The previous channel y is the w channel's MUL result.
The previous channel x is the z channel's MUL result.

```
Result = Arg2 * prev_slot_result
```

Example: Dependent mul, result in y

```
    w  z y  x
    mul * mul_prev *
```

Example: 2 separate dependent muls, results in y and x

```
     w z y  x
    mul mul mul_prev mul_prev
```

Microcode

C	DC	DR	DST_GPR				BS	ALU_INST						OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_MUL_PREV, opcode 209 (0xD1).

24-Bit Unsigned Integer Multiply (Low-Order)

Instructions **MUL_UINT24**

Description Src 0 and 1 treated as 24-bit unsigned integers. Bits [31:24] are ignored. The result represents the low-order 32 bits of the 48-bit multiply result, `mul_result[31:0]`.

```
dst = src0[23:0] * src1[23:0] // low order bits
```

Microcode

C	DC	DR	DST_GPR							BS	ALU_INST						OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM				S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL						+0

Format `ALU_WORD0` (page 10-23) and `ALU_WORD1_OP2` (page 10-26).

Instruction Field `ALU_INST == OP2_INST_MUL_UINT24`, opcode 181 (0xB5).

Floating-Point Multiply-Add

Instructions **MULADD**

Description Floating-point multiply-add (MAD). Gives same results as ADD after MUL.

```
dst = src0 * src1 + src2;
```

Microcode

C	DC	DR	DST_GPR		BS	ALU_INST	SN	S2C	SR	SRC2_SEL		+4
L	PS	IM	S1N	S1C	S1R	SRC1_SEL	S0N	S0C	S0R	SRC0_SEL		+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP3 (page 10-32).

Instruction Field ALU_INST == OP3_INST_MULADD, opcode 20 (0x14).

Floating-Point Multiply-Add, Divide by 2

Instructions **MULADD_D2**

Description Floating-point multiply-add (MAD), followed by divide by 2.

```
dst = (src0 * src1 + src2) *.5;
```

Microcode

C	DE	DR	DST_GPR			BS	ALU_INST (**11000**)	S2N	S2E	S2R	SRC2_SEL		+4
L	PS	IM		S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL		+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP3 (page 10-32).

Instruction Field ALU_INST == OP3_INST_MULADD_D2, opcode 23 (0x17).

IEEE Floating-Point Multiply-Add

Instructions **MULADD_IEEE**

Description Floating-point multiply-add (MAD). Uses IEEE rules for zero times anything. Gives same results as ADD after MUL_IEEE. Uses IEEE rules for zero times anything.

```
dst = src0 * src1 + src2;
```

Microcode

C	DE	DR	DST_GPR			BS	ALU_INST (**11000**)	S2N	S2E	S2R	SRC2_SEL	+4
L	PS	IM		S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP3 (page 10-32).

Instruction Field ALU_INST == OP3_INST_MULADD_IEEE, opcode 24 (0x18).

Dependent Multiply Add With IEEE Rules

Instructions **MULADD_IEEE_PREV**

Description Multiply src1 by the previous slot's MUL result. Previous slot opcode must be MUL.

This version uses DX10 (IEEE) rules for 0*anything.

Output modifier and clamping on w/z channel is NOT applied in the MUL_PREV mul_prev path to the dependent channel (y/x).

Do not use in w or z channels.

Channel y previous is w channel's MUL result.

Channel x previous is z channel's MUL result.

Result = Arg2 * prev_slot_result

Example 1: Dependent mul, result in y:

```
wzy  x
mul*mul_prev*
```

Example 2: Two separate dependent MULs, results in y and x:

```
w zy  x
mulmulmul_prevmul_prev
```

Microcode

C	DC	DR	DST_GPR				BS	ALU_INST					OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL					+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_MULADD_IEEE_PREV, opcode 213 (0xD5).

Floating-Point Multiply-Add, Multiply by 2

Instructions **MULADD_M2**

Description Floating-point multiply-add (MAD), followed by multiply by 2.

```
dst = (src0 * src1 + src2) * 2;
```

Microcode

C	DE	DR	DST_GPR			BS	ALU_INST (**11000**)	S2N	S2E	S2R	SRC2_SEL		+4
L	PS	IM		S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL		+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP3 (page 10-32).

Instruction Field ALU_INST == OP3_INST_MULADD_M2, opcode 21 (0x15).

Floating-Point Multiply-Add, Multiply by 4

Instructions **MULADD_M4**

Description Floating-point multiply-add (MAD), followed by multiply by 4.

```
dst = (src0 * src1 + src2) * 4;
```

Microcode

C	DE	DR	DST_GPR			BS	ALU_INST (**11000**)	S2N	S2E	S2R	SRC2_SEL		+4
L	PS	IM		S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL		+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP3 (page 10-32).

Instruction Field ALU_INST == OP3_INST_MULADD_M4, opcode 22 (0x16).

Dependent Multiply-Add

Instructions **MULADD_PREV**

Description The IEEE version uses DX10 rules for 0*anything.

Multiply src0 and src1, then add the result to previous channel's result. The previous channel opcode must result in a 32-bit, single-precision floating-point.

The output modifier and clamping on the w/z channel is not allowed (results are undefined).

Use in w or z channels is not legal.

Previous channel y is the w channel's FP32 result.
Previous channel x is the z channel's FP32 result.

```
dst = (src0 * src1) + prev_slot_result
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM				S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL					+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_MULADD_PREV, opcode 212 (0xD4).

24-Bit Unsigned Integer Multiply-Add

Instructions **MULADD_UINT24**

Description Src0 and src1 treated as 24-bit unsigned integers. Src2 is treated as a 32-bit signed or unsigned integer. Bits [31:24] are ignored. The result represents the low-order 32 bits of the multiply-add result.

```
dst = src0[23:0] * src1[23:0] + src[31:0] // low order bits
```

Microcode

C	DC	DR	DST_GPR			BS	ALU_INST	SN	S2C	SR	SRC2_SEL		+4
L	PS	IM		S1N	S1C	S1R	SRC1_SEL	S0N	S0C	S0R	SRC0_SEL		+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP3 (page 10-32).

Instruction Field ALU_INST == OP3_INST_MULADD_UINT24, opcode 16 (0x10).

Signed Scalar Multiply, High-Order 32 Bits

Instructions **MULHI_INT**

Description Scalar multiplication. The arguments are interpreted as signed integers. The result represents the high-order 32 bits of the multiply result.

```
dst = src0 * src1 // high-order bits
```

Microcode

C	DC	DR	DST_GPR							BS		ALU_INST										OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM						S1N	S1C	S1R	SRC1_SEL						S0N	S0C	S0R	SRC0_SEL							+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_MULHI_INT, opcode 144 (0x90).

Unsigned Scalar Multiply, High-Order 32 Bits

Instructions **MULHI_UINT**

Description Scalar multiplication. The arguments are interpreted as unsigned integers. The result represents the high-order 32 bits of the multiply result.

```
dst = src0 * src1 // high-order bits
```

Microcode

C	DC	DR	DST_GPR	BS	ALU_INST	OMOD	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1C	S1R	SRC1_SEL	S0N	S0C	S0R	SRC0_SEL	+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_MULHI_UINT, opcode 146 (0x92).

24-Bit Unsigned Integer Multiply (High-Order)

Instructions **`MULHI_UINT24`**

Description Src0 and src1 are treated as 24-bit unsigned integers. Bits [31:24] are ignored. The result represents the high-order 16 bits of the 48-bit multiply result, `{16'b0, mul_result[47:32]}`.
`dst = src0[23:0] * src1 [23:0] // high order bits`

Microcode

C	DC	DR	DST_GPR	BS	ALU_INST	OMOD	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1C	S1R	SRC1_SEL	S0N	S0C	S0R	SRC0_SEL	+0

Format `ALU_WORD0` (page 10-23) and `ALU_WORD1_OP2` (page 10-26).

Instruction Field `ALU_INST == OP2_INST_MULHI_UINT24`, opcode 178 (0xB2).

Signed Scalar Multiply, Low-Order 32-Bits

Instructions **MULLO_INT**

Description Scalar multiplication. The arguments are interpreted as signed integers. The result represents the low-order 32 bits of the multiply result.

```
dst = src0 * src1 // low-order bits
```

Microcode

C	DC	D R	DST_GPR							BS	ALU_INST										OMOD	W M	U P	U E M	S 1 A	S 0 A	+4
L	PS	IM							S 1 N	S1C	S 1 R	SRC1_SEL							S 0 N	S0C	S 0 R	SRC0_SEL					+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_MULLO_INT, opcode 143 (0x8F).

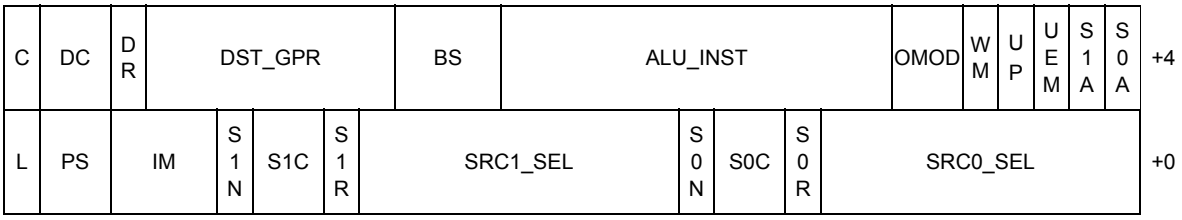
Unsigned Scalar Multiply, Low-Order 32-Bits

Instructions **MULLO_UINT**

Description Scalar multiplication. The arguments are interpreted as unsigned integers. The result represents the low-order 32 bits of the multiply result.

```
dst = src0 * src1 // low-order bits
```

Microcode



Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_MULLO_UINT, opcode 145 (0x91).

No Operation

Instructions **NOP**

Description No operation. The instruction slot is not used. NOP instructions perform no writes to GPRs, and they invalidate the PV and PS register values.

After all instructions in an instruction group are processed, any ALU.[X,Y,Z,W] or ALU.Trans operation that is unspecified implicitly executes a NOP instruction, thus invalidating the values in the corresponding channels of the PV and PS registers.

dst is undefined.

See the CF version of NOP on page 9-40.

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_NOP, opcode 26 (0x1A).

Bit-Wise NOT

Instructions **`NOT_INT`**

Description Logical bit-wise NOT.
`dst = ~src0`

Microcode

C	DC	DR	DST_GPR	BS	ALU_INST	OMOD	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1C	S1R	SRC1_SEL	S0N	S0C	S0R	SRC0_SEL	+0

Format `ALU_WORD0` (page 10-23) and `ALU_WORD1_OP2` (page 10-26).

Instruction Field `ALU_INST == OP2_INST_NOT_INT`, opcode 51 (0x33).

Four-Bit Signed Integer to 32-Bit Float

Instructions **OFFSET_TO_FLT**

Description Four-bit signed integer to 32-bit float conversion for interpolation in the kernel.

src0	dst
1000	-0.5f
1001	-0.4375f
1010	-0.375f
1011	-0.3125f
1100	-0.25f
1101	-0.1875f
1110	-0.125f
1111	-0.0625f
0000	0.0f
0001	0.0625f
0010	0.125f
0011	0.1875f
0100	0.25f
0101	0.3125f
0110	0.375f
0111	0.4375f

Microcode

C	DC	DR	DST_GPR			BS	ALU_INST					OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM		S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL					+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_OFFSET_TO_FLT, opcode 180 (0xB4).

Logical Bit-Wise OR

Instructions **OR_INT**

Description Logical bit-wise OR.
`dst = src0 | src1`

Microcode

C	DC	DR	DST_GPR							BS		ALU_INST										OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM						S1N	S1C	S1R	SRC1_SEL						S0N	S0C	S0R	SRC0_SEL							+0

Format `ALU_WORD0` (page 10-23) and `ALU_WORD1_OP2` (page 10-26).

Instruction Field `ALU_INST == OP2_INST_OR_INT`, opcode 49 (0x31).

Predicate Counter Clear

Instructions **PRED_SET_CLR**

Description Predicate counter clear. Updates predicate register.

```
dst = +MAX_FLOAT;
predicate_result = skip;
```

Microcode

C	DC	DR	DST_GPR							BS	ALU_INST						OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL									+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_PRED_SET_CLR, opcode 38 (0x26).

Predicate Counter Invert

Instructions **PRED_SET_INV**

Description Predicate counter invert. Updates predicate register.

```
If (src0 == 1.0f) {
    dst = 0.0f;
    predicate_result = execute;
}
Else {
    If (src0 == 0.0f) {
        dst = 1.0f;
    }
    Else {
        dst = src0;
    }
    predicate_result = skip;
}
```

Microcode

C	DC	DR	DST_GPR	BS	ALU_INST	OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM	S1N	S1C	S1R	SRC1_SEL	S0N	S0C	S0R	SRC0_SEL		+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_PRED_SET_INV, opcode 36 (0x24).

Predicate Counter Pop

Instructions **PRED_SET_POP**

Description Pop predicate counter. This updates the predicate register.

```
If (src0 <= src1) {
    dst = 0.0f;
    predicate_result = execute;
}
Else {
    dst = src0 - src1;
    predicate_result = skip;
}
```

Microcode

C	DC	DR	DST_GPR						BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM				S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_PRED_SET_POP, opcode 37 (0x25).

Predicate Counter Restore

Instructions **PRED_SET_RESTORE**

Description Predicate counter restore. Updates predicate register.

```
If (src0 == 0.0f) {
    dst = 0.0f;
    predicate_result = execute;
}
Else {
    dst = src0;
    predicate_result = skip;
}
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_PRED_SET_RESTORE, opcode 39 (0x27).

Floating-Point Predicate Set If Equal

Instructions **PRED_SETE**

Description Floating-point predicate set if equal. Updates predicate register.

```
If (src0 == src1) {
    dst = 0.0f;
    predicate_result = execute;
} Else {
    dst = 1.0f;
    predicate_result = skip;
}
```

Microcode

C	DC	DR	DST_GPR						BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C		S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_PRED_SETE, opcode 32 (0x20).

Floating-Point Predicate Set If Equal, 64-Bit

Instructions **PRED_SETE_64**

Description Floating-point 64-bit predicate set if equal. Updates the predicate register. Compares two double-precision floating-point numbers in src0.YX and src1.YX, or src0.WZ and src1.WZ, and returns 0x0 if src0==src1 or 0xFFFFFFFF; otherwise, it returns the unsigned integer result in dst.YX or dst.WZ.

The instruction can also establish a predicate result (execute or skip) for subsequent predicated instruction execution. This additional control allows a compiler to support one-instruction issue for if-elseif operations, or an integer result for nested flow-control, by using single-precision operations to manipulate a predicate counter.

```
if (src0 == src1)
{
    dst = 0x0;
    predicate_result = execute;
}
else
{
    dst = 0xFFFFFFFF;
    predicate_result = skip;
}
```

Table 9.8 Result of PRED_SETE_64 Instruction

	src1								
src0	**-inf**	**-F[1]**	**-denorm[2]**	**-0**	**+0**	**+denorm[2]**	**+F[1]**	**+inf**	**NaN**
-inf	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
-F[1]	FALSE	TRUE or FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
-denorm[2]	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE
-0	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE
+0	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE
+denorm[2]	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE
+F[1]	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE or FALSE	FALSE	FALSE
+inf	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE
NaN	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

1. F is a finite floating-point value.
2. Denorms are treated arithmetically and obey rules of appropriate zero.

Coissue PRED_SETE_64 is a two-slot instruction. The following coissues are possible:

- A single PRED_SETE_64 instruction in slots 0 and 1, and any valid instructions in slots 2, 3, and 4, except other predicate-set instructions.
- A single PRED_SETE_64 instruction in slots 2 and 3, and any valid instructions in slots 0, 1, and 4, except other predicate-set instructions.
- Two PRED_SETE_64 instructions in slots 0, 1, 2, and 3,and any valid instruction in slot 4, except other predicate-set instructions.

Floating-Point Predicate Set If Equal, 64-Bit (Cont.)

Microcode

C	DC	DR	DST_GPR			BS	ALU_INST				OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM	S1N	S1C	S1R	SRC1_SEL		S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_PRED_SETE_64, opcode 200 (0xC8).

Example The following examples issue a single PRED_SETE_64 instruction in two slots.

```
Input data:

Input data 6.0 (0x4018000000000000)
Input data 3.0 (0x4008000000000000)

mov ra.h, l(0x40180000) //high dword (Input 1)
mov rb.l, l(0x00000000) //low dword

mov rc.h, l(0x40080000) //high dword (Input 2)
mov rd.l, l(0x00000000) //low dword

Issue a single PRED_SETE_64 instruction in slots 3 and 2:

PRED_SETE_64 re.x ra.h ra.h //can be any vector element
PRED_SETE_64 rf.y rb.l rb.l //can be any vector element

Result:

PRED_SETE_64 (0x4018000000000000,0x4018000000000000) =
PRED_SETE_64 (6.0,6.0) => result = 0x0, predicate_result = execute

re.x = 0x0
rf.y = 0x0

predicate = execute

Or, issue a single PRED_SETE_64 instruction in slots 1 and 0:

PRED_SETE_64 re.z rc.h ra.h //can be any vector element
PRED_SETE_64 rf.w rd.l rb.l //can be any vector element

Result:

PRED_SETE_64 (0x4008000000000000,0x4018000000000000) =
PRED_SETE_64 (3.0,6.0) => result = 0xFFFFFFFF, predicate_result = skip

re.z = 0xFFFFFFFF
rf.w = 0xFFFFFFFF

predicate = skip
```

Input Modifiers Input modifiers (Section 4.7.2, on page 4-10) can be applied to the source operands during the destination X channel (slot 0) and Z channel (slot 2). These slots contain the sign bits of the sources.

Output Modifiers The instruction does not take output modifiers.

Integer Predicate Set If Equal

Instructions **PREDE_INT**

Description Integer predicate set if equal. Updates predicate register.

```
If (src0 == src1) {
    dst = 0.0f;
    SetPredicateKillReg(Execute);
}
Else {
    dst = 1.0f;
    SetPredicateKillReg (Skip);
}
```

Microcode

C	DC	DR	DST_GPR							BS	ALU_INST									OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM						S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL							+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_PREDE_INT, opcode 66 (0x42).

Floating-Point Predicate Counter Increment If Equal

Instructions **PRED_SETE_PUSH**

Description Floating-point predicate counter increment if equal. Updates predicate register.

```
If ( (src1 == 0.0f) && (src0 == 0.0f) ) {
    dst = 0.0f;
    predicate_result = execute;
}
Else {
    dst = src0 + 1.0f;
    predicate_result = skip;
}
```

Microcode

C	DC	DR	DST_GPR						BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM				S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_PRED_SETE_PUSH, opcode 40 (0x28).

Integer Predicate Counter Increment If Equal

Instructions **PRED_SETE_PUSH_INT**

Description Integer predicate counter increment if equal. Updates predicate register.

```
If ( (src1 == 0x0) && (src0 == 0.0f) ) {
    dst = 0.0f;
    predicate_result = execute;
}
Else {
    dst = src0 + 1.0f;
    predicate_result = skip;
}
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM				S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL					+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_PRED_SETE_PUSH_INT, opcode 74 (0x4A).

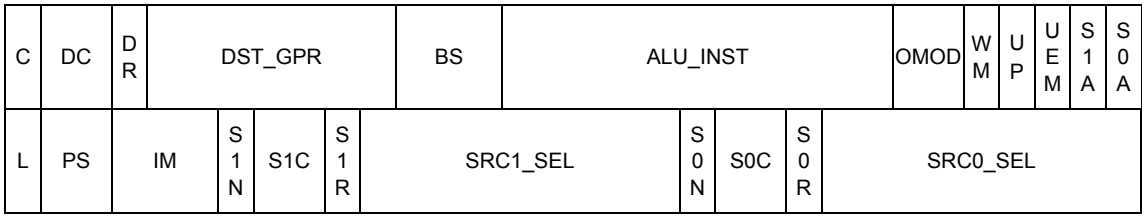
Floating-Point Predicate Set If Greater Than Or Equal

Instructions **PRED_SETGE**

Description Floating-point predicate set if greater than or equal. Updates predicate register.

```
If (src0 >= src1) {
    dst = 0.0f;
    predicate_result = execute;
} Else {
    dst = 1.0f;
    predicate_result = skip;
}
```

Microcode



Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_PRED_SETGE, opcode 34 (0x22).

Floating-Point Predicate Set If Greater Than Or Equal, 64-Bit

Instructions **PRED_SETGE_64**

Description Floating-point 64-bit predicate set if greater than or equal. Updates the predicate register. Compares two double-precision floating-point numbers in `src0.YX` and `src1.YX`, or `src0.WZ` and `src1.WZ`, and returns 0x0 if `src0>=src1` or 0xFFFFFFFF; otherwise, it returns the unsigned integer result in `dst.YX` or `dst.WZ`.

The instruction can also establish a predicate result (execute or skip) for subsequent predicated instruction execution. This additional control allows a compiler to support one-instruction issue for if/elseif operations or an integer result for nested flow-control by using single-precision operations to manipulate a predicate counter.

```
if (src0>=src1)
{
    result = 0x0;
    predicate_result = execute;
}
else
{
    result = 0xFFFFFFFF;
    predicate_result = skip;
}
```

Table 9.9 Result of PRED_SETGE_64 Instruction

	src1								
src0	**-inf**	**-F[1]**	**-denorm[2]**	**-0**	**+0**	**+denorm[2]**	**+F[1]**	**+inf**	**NaN**
-inf	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
-F[1]	TRUE	TRUE or FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
-denorm[2]	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE
-0	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE
+0	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE
+denorm[2]	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE
+F[1]	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE or FALSE	FALSE	FALSE
+inf	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE
NaN	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

1. F is a finite floating-point value.
2. Denorms are treated arithmetically and obey rules of appropriate zero.

Coissue `PRED_SETGE_64` is a two-slot instruction. The following coissues are possible:

- A single `PRED_SETGE_64` instruction in slots 0 and 1, and any valid instructions in slots 2, 3, and 4, except other predicate-set instructions.
- A single `PRED_SETGE_64` instruction in slots 2 and 3, and any valid instructions in slots 0, 1, and 4, except other predicate-set instructions.
- Two `PRED_SETGE_64` instructions in slots 0, 1, 2, and 3,and any valid instruction in slot 4, except other predicate-set instructions.

Floating-Point Predicate Set If Greater Than Or Equal, 64-Bit (Cont.)

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL				S0N	S0C	S0R	SRC0_SEL							+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_PRED_SETGE_64, opcode 201 (0xC9).

Example The following examples issue a single PRED_SETGE_64 instruction in two slots:

```
Input data:

Input data => 0x4018000000000000 (6.0)
Input data => 0x4008000000000000 (3.0)

mov ra.h, l(0x40180000) //high dword (Input 1)
mov rb.l, l(0x00000000) //low dword

mov rc.h, l(0x40080000) //high dword (Input 2)
mov rd.l, l(0x00000000) //low dword

Issue a single PRED_SETGE_64 instruction in slots 3 and 2:

PRED_SETGE_64 re.x ra.h ra.h //can be any vector element
PRED_SETGE_64 rf.y rb.l rb.l //can be any vector element

Result:

pred_setge64(0x4018000000000000,0x4018000000000000) =
pred_setge64(6.0,6.0) => result = 0x0, predicate_result = execute

re.x = 0x0
rf.y = 0x0

predicate = execute
```

Floating-Point Predicate Set If Greater Than Or Equal, 64-Bit (Cont.)

Or, issue a single PRED_SETGE_64 instruction in slots 3 and 2.

```
PRED_SETGE_64 re.x ra.h rc.h //can be any vector element
PRED_SETGE_64 rf.y rb.l rd.l //can be any vector element
```

Result:

```
pred_setge64(0x4018000000000000,0x4008000000000000) =
pred_setge64(6.0,3.0) => result = 0x0, predicate_result = execute

re.x = 0x0
rf.y = 0x0

predicate = execute
```

Or, issue a single PRED_SETGE_64 instruction in slots 1 and 0:

```
PRED_SETGE_64 re.z rc.h ra.h //can be any vector element
PRED_SETGE_64 rf.w rd.l rb.l //can be any vector element
```

Result:

```
pred_setge64(0x4008000000000000,0x4018000000000000) =
pred_setge64(3.0,6.0) => result = 0xFFFFFFFF, predicate_result = skip

re.z = 0xFFFFFFFF
rf.w = 0xFFFFFFFF

predicate = skip
```

Input Modifiers Input modifiers (Section 4.7.2, on page 4-10) can be applied to the source operands during the destination X channel (slot 0) and Z channel (slot 2). These slots contain the sign bits of the sources.

Output Modifiers The instruction does not take output modifiers.

Integer Predicate Set If Greater Than Or Equal

Instructions **PRED_SETGE_INT**

Description Integer predicate set if greater than or equal. Updates predicate register.

```
If (src0 >= src1) {
    dst = 0.0f;
    SetPredicateKillReg (Execute);
}
Else {
    dst = 1.0f;
    SetPredicateKillReg (Skip);
}
```

Microcode

C	DC	DR	DST_GPR						BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM				S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_PRED_SETGE_INT, opcode 68 (0x44).

Predicate Counter Increment If Greater Than Or Equal

Instructions **PRED_SETGE_PUSH**

Description Predicate counter increment if greater than or equal. Updates predicate register.

```
If ( (src1 >= 0.0f) && (src0 == 0.0f) ) {
    dst = 0.0f;
    predicate_result = execute;
}
Else {
    dst = src0 + 1.0f;
    predicate_result = skip;
}
```

Microcode

C	DC	DR	DST_GPR			BS	ALU_INST					OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL				+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_PRED_SETGE_PUSH, opcode 42 (0x2A).

Integer Predicate Counter Increment If Greater Than Or Equal

Instructions **PRED_SETGE_PUSH_INT**

Description Integer predicate counter increment if greater than or equal. Updates predicate register.

```
If ( (src1 >= 0x0) && (src0 == 0.0f) ) {
    dst = 0.0f;
    predicate_result = execute;
}
Else {
    dst = src0 + 1.0f;
    predicate_result = skip;
}
```

Microcode

<table>
<tr><td>C</td><td>DC</td><td>DR</td><td colspan="3">DST_GPR</td><td>BS</td><td colspan="3">ALU_INST</td><td>OMOD</td><td>WM</td><td>UP</td><td>UEM</td><td>S1A</td><td>S0A</td><td>+4</td></tr>
<tr><td>L</td><td>PS</td><td colspan="2">IM</td><td>S1N</td><td>S1C</td><td>S1R</td><td>SRC1_SEL</td><td>S0N</td><td>S0C</td><td>S0R</td><td colspan="6">SRC0_SEL</td><td>+0</td></tr>
</table>

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_PRED_SETGE_PUSH_INT, opcode 76 (0x4C).

Unsigned Integer Predicate Set If Greater Than Or Equal

Instructions **PRED_SETGE_UINT**

Description Unsigned integer predicate set if greater than or equal. Updates predicate register.

```
If (src0 >= src1) {
    dst = 0.0f;
    SetPredicateKillReg (Execute);
}
Else {
    dst = 1.0f;
    SetPredicateKillReg (Skip);
}
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM				S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL					+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_PRED_SETGE_UINT, opcode 31 (0x1F).

Floating-Point Predicate Set If Greater Than

Instructions **PRED_SETGT**

Description Floating-point predicate set if greater than. Updates predicate register.

```
If (src0 > src1) {
    dst = 0.0f;
    predicate_result = execute;
}
Else {
    dst = 1.0f;
    predicate_result = skip;
}
```

Microcode

C	DC	DR	DST_GPR						BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM				S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_PRED_SETGT, opcode 33 (0x21).

Floating-Point Predicate Set If Greater Than, 64-Bit

Instructions **PRED_SETGT_64**

Description Floating-point 64-bit predicate set if greater than. Updates the predicate register. Compares two double-precision floating-point numbers in `src0.YX` and `src1.YX`, or `src0.WZ` and `src1.WZ`, and returns 0x0 if src0>src1 or 0xFFFFFFFF; otherwise, it returns the unsigned integer result in `dst.YX` or `dst.WZ`.

The instruction can also optionally establish a predicate result (execute or skip) for subsequent predicated instruction execution. This additional control allows a compiler to support one-instruction issue for if/elseif operations, or an integer result for nested flow-control, by using single-precision operations to manipulate a predicate counter.

```
if (src0>src1)
{
    result = 0x0;
    predicate_result = execute;
}
else
{
    result = 0xFFFFFFFF;
    predicate_result = skip;
}
```

Table 9.10 Result of PRED_SETGT_64 Instruction

	src1								
src0	**-inf**	**-F[1]**	**-denorm[2]**	**-0**	**+0**	**+denorm[2]**	**+F[1]**	**+inf**	**NaN**
-inf	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
-F[1]	TRUE	TRUE or FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
-denorm[2]	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
-0	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
+0	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
+denorm[2]	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
+F[1]	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE or FALSE	FALSE	FALSE
+inf	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE
NaN	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

1. F is a finite floating-point value.
2. Denorms are treated arithmetically and obey rules of appropriate zero.

Coissue `PRED_SETGT_64` is a two-slot instruction. The following coissues are possible:

- A single `PRED_SETGT_64` instruction in slots 0 and 1, and any valid instructions in slots 2, 3, and 4, except other predicate-set instructions.
- A single `PRED_SETGT_64` instruction in slots 2 and 3, and any valid instructions in slots 0, 1, and 4, except other predicate-set instructions.
- Two `PRED_SETGT_64` instructions in slots 0, 1, 2, and 3,and any valid instruction in slot 4, except other predicate-set instructions.

Floating-Point Predicate Set If Greater Than, 64-Bit (Cont.)

Microcode

C	DC	DR	DST_GPR		BS	ALU_INST		OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM	S1N	S1C	S1R	SRC1_SEL	S0N	S0C	S0R	SRC0_SEL				+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_PRED_SETGT_64, opcode 199 (0xC7).

Example The following examples issue a single PRED_SETGT_64 instruction in two slots:

Input data:

```
Input data 6.0 (0x4018000000000000)
Input data 3.0 (0x4008000000000000)

mov ra.h, l(0x40180000) //high dword (Input 1)
mov rb.l, l(0x00000000) //low dword

mov rc.h, l(0x40080000) //high dword (Input 2)
mov rd.l, l(0x00000000) // low dword
```

Issue a single PRED_SETGT_64 instruction in slots 3 and 2:

```
PRED_SETGT_64 re.x ra.h rc.h //can be any vector element
PRED_SETGT_64 rf.y rb.l rd.l //can be any vector element
```

Result:

```
pred_setgt64(0x4018000000000000,0x4008000000000000) =
pred_setgt64(6.0,3.0) => result = 0x0, predicate_result = execute

re.x = 0x0
rf.y = 0x0

predicate = execute
```

Or, issue a single PRED_SETGT_64 instruction in slots 1 and 0:

```
PRED_SETGT_64 re.z rc.h ra.h //can be any vector element
PRED_SETGT_64 rf.w rd.l rb.l //can be any vector element
```

Result:

```
pred_setgt64(0x4008000000000000,0x4018000000000000) =
pred_setgt64(3.0,6.0) => result = 0xFFFFFFFF, predicate_result = skip

re.z = 0xFFFFFFFF
rf.w = 0xFFFFFFFF

predicate = skip
```

Input Modifiers Input modifiers (Section 4.7.2, on page 4-10) can be applied to the source operands during the destination X channel (slot 0) and Z channel (slot 2). These slots contain the sign bits of the sources.

Output Modifiers The instruction does not take output modifiers.

Integer Predicate Set If Greater Than

Instructions **PRED_SETGT_INT**

Description Integer predicate set if greater than. Updates predicate register.

```
If (src0 > src1) {
    dst = 0.0f;
    SetPredicateKillReg (Execute);
}
Else {
    dst = 1.0f;
    SetPredicateKillReg (Skip);
}
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM				S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL					+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_PRED_SETGT_INT, opcode 67 (0x43).

Predicate Counter Increment If Greater Than

Instructions **PRED_SETGT_PUSH**

Description Predicate counter increment if greater than. Updates predicate register.

```
If ( (src1 > 0.0f) && (src0 == 0.0f) ) {
    dst = 0.0f;
    predicate_result = execute;
}
Else {
    dst = src0.W + 1.0f;
    predicate_result = skip;
}
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL				S0N	S0C	S0R	SRC0_SEL							+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_PRED_SETGT_PUSH, opcode 41 (0x29).

Integer Predicate Counter Increment If Greater Than

Instructions **PRED_SETGT_PUSH_INT**

Description Integer predicate counter increment if greater than. Updates predicate register.

```
If ( (src1 > 0x0) && (src0 == 0.0f) ) {
    dst = 0.0f;
    predicate_result = execute;
}
Else {
    dst = src0 + 1.0f;
    predicate_result = skip;
}
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM				S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL					+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_PRED_SETGT_PUSH_INT, opcode 75 (0x4B).

Unsigned Integer Predicate Set If Greater Than

Instructions **PRED_SETGT_UINT**

Description Unsigned integer predicate set if greater than. Updates predicate register.

```
If (src0 > src1) {
    dst = 0.0f;
    SetPredicateKillReg (Execute);
}
Else {
    dst = 1.0f;
    SetPredicateKillReg (Skip);
}
```

Microcode

C	DC	DR	DST_GPR						BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM				S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_PRED_SETGT_UINT, opcode 30 (0x1E).

Predicate Counter Increment If Less Than Or Equal

Instructions **PRED_SETLE_PUSH_INT**

Description Predicate counter increment if less than or equal. Updates predicate register.

```
If ( (src1 <= 0x0) && (src0 == 0.0f) ) {
    dst = 0.0f;
    predicate_result = execute;
}
Else {
    dst = src0 + 1.0f;
    predicate_result = skip;
}
```

Microcode

C	DC	DR	DST_GPR							BS	ALU_INST											OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM							S1N	S1C	S1R	SRC1_SEL							S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_PRED_SETLE_PUSH_INT, opcode 79 (0x4F).

Predicate Counter Increment If Less Than

Instructions **PRED_SETLT_PUSH_INT**

Description Predicate counter increment if less than. Updates predicate register.

```
If ( (src1 < 0x0) && (src0 == 0.0f) ) {
    dst = 0.0f;
    predicate_result = execute;
}
Else {
    dst = src0 + 1.0f;
    predicate_result = skip;
}
```

Microcode

C	DC	DR	DST_GPR						BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM				S1N	S1C	S1R	SRC1_SEL				S0N	S0C	S0R	SRC0_SEL							+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_PRED_SETLT_PUSH_INT, opcode 78 (0x4E).

Floating-Point Predicate Set If Not Equal

Instructions **PRED_SETNE**

Description Floating-point predicate set if not equal. Updates predicate register.

```
If (src0 != src1) {
    dst = 0.0f;
    predicate_result = execute;
}
Else {
    dst = 1.0f;
    predicate_result = skip;
}
```

Microcode

C	DC	DR	DST_GPR				BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL							+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_PRED_SETNE, opcode 35 (0x23).

Scalar Predicate Set If Not Equal

Instructions **PRED_SETNE_INT**

Description Scalar predicate set if not equal. Updates predicate register.

```
If (src0 != src1) {
    dst = 0.0f;
    SetPredicateKillReg (Execute);
}
Else {
    dst = 1.0f;
    SetPredicateKillReg (Skip);
}
```

Microcode

C	DC	DR	DST_GPR						BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM				S1N	S1C	S1R	SRC1_SEL				S0N	S0C	S0R	SRC0_SEL							+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_PRED_SETNE_INT, opcode 69 (0x45).

Predicate Counter Increment If Not Equal

Instructions **PRED_SETNE_PUSH**

Description Predicate counter increment if not equal. Updates predicate register.

```
If ( (src1 != 0.0f) && (src0 == 0.0f) ) {
    dst = 0.0f;
    predicate_result = execute;
}
Else {
    dst = src0 + 1.0f;
    predicate_result = skip;
}
```

Microcode

C	DC	DR	DST_GPR						BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM					S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL					+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_PRED_SETNE_PUSH, opcode 43 (0x2B).

Predicate Counter Increment If Not Equal

Instructions **PRED_SETNE_PUSH_INT**

Description Predicate counter increment if not equal. Updates predicate register.

```
If ( (src1 != 0x0) && (src0 == 0.0f) ) {
    dst = 0.0f;
    predicate_result = execute;
}
Else {
    dst = src0 + 1.0f;
    predicate_result = skip;
}
```

Microcode

C	DC	DR	DST_GPR						BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM				S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_PRED_SETNE_PUSH_INT, opcode 77 (0x4D).

Double Reciprocal

Instructions **RECIP_64**

Description src0_d is composed of high-order double bits on src0, and low-order double bits on src1.

The result is a high-order dword of recip64; the low-order bits are assumed to be 0.

```
If (src0_d == 1.0f) {
  Result = 1.0F;
}
Else {
  Result = ApproximateRecip(src0_d);
}
```

Microcode

C	DC	DR	DST_GPR				BS	ALU_INST						OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL				S0N	S0C	S0R	SRC0_SEL					+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_RECIP_64, opcode 149 (0x95).

Scalar Reciprocal, Clamp to Maximum

Instructions **RECIP_CLAMPED**

Description Scalar reciprocal.

```
If (src0 == 1.0f) {
    dst = 1.0f;
}
Else {
    dst = RECIP_IEEE(src0);
}
// clamp dst
If (dst == -INFINITY) {
    dst = -MAX_FLOAT;
}
If (dst == +INFINITY) {
    dst = +MAX_FLOAT;
}
```

Microcode

C	DC	DR	DST_GPR			BS	ALU_INST				OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM		S1N	S1C	S1R	SRC1_SEL		S0N	S0C	S0R	SRC0_SEL					+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_RECIP_CLAMPED, opcode 132 (0x84).

Double Reciprocal Clamped

Instructions **RECIP_CLAMPED_64**

Description Src0_d is composed of high-order double bits on src0, and low-order double bits on src1.

The result is a high-order dword of recip64_clamped; low-order bits are assumed to be 0.

```
If (src0_d == 1.0F) {
  Result = 1.0F;
}
Else {
  Result = RECIP_IEEE(src0_d);
}
// clamp result
if (Result == -INFINITY) {
  Result = -MAX_FLOAT;
}
if (Result == +INFINITY) {
  Result = +MAX_FLOAT;
}
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_RECIP_CLAMPED_64, opcode 150 (0x96).

Scalar Reciprocal, Clamp to Zero

Instructions **RECIP_FF**

Description Scalar reciprocal.

```
If (src0 == 1.0f) {
    dst = 1.0f;
}
Else {
    dst = RECIP_IEEE(src0);
}
// clamp dst
if (dst == -INFINITY) {
    dst = -ZERO;
}
if (dst == +INFINITY) {
    dst = +ZERO;
}
```

Microcode

C	DC	DR	DST_GPR			BS	ALU_INST			OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM		S1N	S1C	S1R	SRC1_SEL		S0N	S0C	S0R	SRC0_SEL				+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_RECIP_FF, opcode 133 (0x85).

Scalar Reciprocal, IEEE Approximation

Instructions **RECIP_IEEE**

Description Scalar reciprocal.

```
If (src0 == 1.0f) {
    dst = 1.0f;
}
Else {
    dst = ApproximateRecip(src0);
}
```

Microcode

C	DC	DR	DST_GPR			BS	ALU_INST					OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM		S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL					+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_RECIP_IEEE, opcode 134 (0x86).

Signed Integer Scalar Reciprocal

Instructions **RECIP_INT**

Description Scalar integer reciprocal. The source is a signed integer. The result is a fractional signed integer. The result for 0 is undefined.

```
dst = ApproximateRecip(src0);
```

Microcode

C	DC	DR	DST_GPR							BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM					S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_RECIP_INT, opcode 147 (0x93).

Unsigned Integer Scalar Reciprocal

Instructions **RECIP_UINT**

Description Scalar unsigned integer reciprocal. The source is an unsigned integer. The result is a fractional unsigned integer. The result for 0 is undefined.

```
dst = ApproximateRecip(src0);
```

Microcode

C	DC	DR	DST_GPR							BS	ALU_INST									OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM					S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL							+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_RECIP_UINT, opcode 148 (0x94).

Double Reciprocal Square Root

Instructions **RECIPSQRT_64**

Description Src0_d is composed of high-order double bits on src0, and low-order double bits on src1.

Result is high order dword of recipsqrt64, low order bits assumed to be 0.

```
If (src0_d == 1.0f) {
  Result = 1.0F;
}
Else {
  Result = ApproximateRecip(src0_d);
}
```

Microcode

C	DC	DR	DST_GPR						BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM				S1N	S1C	S1R	SRC1_SEL				S0N	S0C	S0R	SRC0_SEL							+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_RECIPSQRT_64, opcode 151 (0x97).

Scalar Reciprocal Square Root, Clamp to Maximum

Instructions **RECIPSQRT_CLAMPED**

Description Scalar reciprocal square root.

```
If (src0 == 1.0f) {
    dst = 1.0f;
}
Else {
    dst = RECIPSQRT_IEEE(src0);
}
// clamp dst
if (dst == -INFINITY) {
    dst = -MAX_FLOAT;
}
if (dst == +INFINITY) {
    dst = +MAX_FLOAT;
}
```

Microcode

C	DC	DR	DST_GPR			BS	ALU_INST					OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM		S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL					+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_RECIPSQRT_CLAMPED, opcode 135 (0x87).

Double Reciprocal Square Root Clamped

Instructions **RECIPSQRT_CLAMPED_64**

Description Src0_d is composed of high-order double bits on src0, and low-order double bits on src1.

The result is a high-order dword of `recipsqrt64_clamped`; the low-order bits are assumed to be 0.

```
If (src0_d == 1.0F) {
  Result = 1.0F;
}
Else {
  Result = RECIPSQRT_IEEE(src0_d);
}
// clamp result
if (Result == -INFINITY) {
  Result = -MAX_FLOAT;
}
if (Result == +INFINITY) {
  Result = +MAX_FLOAT;
}
```

Microcode

C	DC	DR	DST_GPR				BS	ALU_INST					OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL					+0

Format `ALU_WORD0` (page 10-23) and `ALU_WORD1_OP2` (page 10-26).

Instruction Field `ALU_INST == OP2_INST_RECIPSQRT_CLAMPED_64`, opcode 152 (0x98).

Scalar Reciprocal Square Root, Clamp to Zero

Instructions **RECIPSQRT_FF**

Description Scalar reciprocal square root.

```
If (src0 == 1.0f) {
    dst = 1.0f;
}
Else {
    dst = RECIPSQRT_IEEE(src0);
}
// clamp dst
if (dst == -INFINITY) {
    dst = -ZERO;
}
if (dst == +INFINITY) {
    dst = +ZERO;
}
```

Microcode

C	DC	DR	DST_GPR			BS	ALU_INST				OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM		S1N	S1C	S1R	SRC1_SEL		S0N	S0C	S0R	SRC0_SEL					+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_RECIPSQRT_FF, opcode 136 (0x88).

Scalar Reciprocal Square Root, IEEE Approximation

Instructions **RECIPSQRT_IEEE**

Description Scalar reciprocal square root.

```
If (src0 == 1.0f) {
    dst = 1.0f;
}
Else {
    dst = ApproximateRecipSqrt(src0);
}
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST						OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL							+0

Format `ALU_WORD0` (page 10-23) and `ALU_WORD1_OP2` (page 10-26).

Instruction Field `ALU_INST == OP2_INST_RECIPSQRT_IEEE`, opcode 137 (0x89).

Floating-Point Round To Nearest Even Integer

Instructions **RNDNE**

Description Floating-point round to nearest even integer.

```
dst = FLOOR(src0 + 0.5f);
If ( (FLOOR(src0)) == Even) && (FRACT(src0 == 0.5f)){
    dst -= 1.0f
}
```

Microcode

C	DC	DR	DST_GPR			BS	ALU_INST					OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL				+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_RNDNE, opcode 19 (0x13).

Sum of Absolute Differences With Accumulation Into MSB

Instructions **SAD_ACCUM_HI_UINT**

Description Executes 4x4 SAD with src0 and src1; then, accumulates the results into the msbs of src2. The overflow is lost.

```
dst = (|src0[31:24] - src1[31:24]| +
|src0[23:16] - src1[23:16]| +
|src0[15:8] - src1[15:8]| +
|src0[7:0] - src1[7:0]|) << 16
+src2
```

Microcode

C	DC	DR	DST_GPR		BS	ALU_INST	SN	S2C	SR	SRC2_SEL		+4
L	PS	IM	S1N	S1C	S1R	SRC1_SEL	S0N	S0C	S0R	SRC0_SEL		+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP3 (page 10-32).

Instruction Field ALU_INST == OP3_INST_SAD_ACCUM_HI_UINT, opcode 15 (0xF).

Sum of Absolute Differences With Accumulation From Previous Channel

Instructions **SAD_ACCUM_PREV_UINT**

Description Executes a 4x4 SAD with src0 and src1; then, accumulates the result int the lsbs of the SAD result from the previous channel. Valid instructions for the previous channel are sad_accum_uint, sad_accum_hi_uint, and sad_accum_prev_uint. Overflow into previous SAD bit 16 is allowed.

Using this in channel w is not legal.

Previous channel z is w channel's sad_accum_*_uint result.
Previous channel y is z channel's sad_accum_*_uint result.
Previous channel x is y channel's sad_accum_*_uint result.

```
dst = |src0[31:24] - src1[31:24]| +
|src0[23:16] - src1[23:16]| +
|src0[15:8] - src1[15:8]| +
|src0[7:0] - src1[7:0]|
+prev_slot_sad_accum_*_uint_result
```

Microcode

C	DC	DR	DST_GPR			BS	ALU_INST			OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM		S1N	S1C	S1R	SRC1_SEL		S0N	S0C	S0R	SRC0_SEL				+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_SAD_ACCUM_PREV_UINT, opcode 207 (0xCF).

Sum of Absolute Differences With Accumulation Into LSB

Instructions **SAD_ACCUM_UINT**

Description Executes a 4x4 SAD with src0 and src1; then, accumulate the results into the lsbs of src2. Overflow into src2[16] is allowed.

```
dst = |src0[31:24] - src1[31:24]| +
|src0[23:16] - src1[23:16]| +
|src0[15:8] - src1[15:8]| +
|src0[7:0] - src1[7:0]|
+src2
```

Microcode

C	DC	DR	DST_GPR		BS	ALU_INST	SN	S2C	SR	SRC2_SEL		+4
L	PS	IM	S1N	S1C	S1R	SRC1_SEL	S0N	S0C	S0R	SRC0_SEL		+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP3 (page 10-32).

Instruction Field ALU_INST == OP3_INST_SAD_ACCUM_UINT, opcode 14 (0xE).

Move Index From GPR To Index Register 0

Instructions **SET_CF_IDX0**

Description Moves the index value from the GPR into index register 0. It retrieves the first active pixel from the sequencer's AR register. The index is used for texture resource, sampler, and constant buffer indexing. The value is clamped to [0,255]. Do not combine this instruction with waterfalling, MOVA_INT, or LDS instructions. It is not legal to set both IDX0 and IDX1 in the same VLIW instruction.

Microcode

C	DC	D R	DST_GPR							BS	ALU_INST											OMOD	W M	U P	U E M	S 1 A	S 0 A	+4
L	PS	IM							S 1 N	S1C	S 1 R	SRC1_SEL							S 0 N	S0C	S 0 R	SRC0_SEL						+0

Format `ALU_WORD0` (page 10-23) and `ALU_WORD1_OP2` (page 10-26).

Instruction Field `ALU_INST == OP2_INST_SET_CF_IDX0`, opcode 88 (0x58).

Move Index From GPR To Index Register 1

Instructions **SET_CF_IDX1**

Description Moves the index value from the GPR into index register 1. It retrieves the first active pixel from the sequencer's AR register. The index is used for texture resource, sampler, and constant buffer indexing. The value is clamped to [0,255]. Do not combine this instruction with waterfalling, MOVA_INT, or LDS instructions. It is not legal to set both IDX0 and IDX1 in the same VLIW instruction.

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_SET_CF_IDX1, opcode 89 (0x59).

Set Local/Global Mode and LDS Size

Instructions **SET_LDS_SIZE**

Description Sets both the local/global mode, and the LDS size:

SRC0 = size, SRC1 = global mode

Default mode is local. The (LDS size – 1) is the highest offset that can be accessed. Writes outside this range are discarded; reads return 0.

Microcode

C	DC	DR	DST_GPR							BS	ALU_INST									OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM					S1N	S1C	S1R	SRC1_SEL							S0N	S0C	S0R	SRC0_SEL						+0

Format `ALU_WORD0` (page 10-23) and `ALU_WORD1_OP2` (page 10-26).

Instruction Field `ALU_INST == OP2_INST_SET_LDS_SIZE`, opcode 90 (0x5A).

Override Rounding and Denorm Modes

Instructions **SET_MODE**

Description Overrides the rounding and denorm modes until the end of clause.

SRC0_SEL, bits 3:0 for single-precision. Bits 1:0 are for round mode; bits 3:2 are for denormal mode: bit 3 = output denorm control; bit 2 = input denorm control. SRC1_SEL, bits 3:0 for double-precision. Bits 1:0 are for round mode; bits 3:2 are for denormal mode: bit 3 = output denorm control; bit 2 = input denorm control.

Round modes are:

- 0 = Round to nearest even
- 3 = Round toward 0 (truncate)
- 1 = Round toward +infinity
- 2 = Round toward -infinity

Denormal handling

0 = flush denorms to 0.
1 = allow denorms.

Microcode

C	DC	DR	DST_GPR			BS	ALU_INST				OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM		S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL				+0

Format `ALU_WORD0` (page 10-23) and `ALU_WORD1_OP2` (page 10-26).

Instruction Field `ALU_INST == OP2_INST_SET_MODE`, opcode 87 (0x57).

Floating-Point Set If Equal

Instructions **SETE**

Description Floating-point set if equal.

```
If (src0 = src1) {
    dst = 1.0f;
}
Else {
    dst = 0.0f;
}
```

Microcode

C	DC	DR	DST_GPR			BS	ALU_INST					OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM		S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL					+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_SETE, opcode 8 (0x8).

Double-Precision Floating-Point If Greater Than Or Equal

Instructions **SETE_64**

Description Double precision floating-point set if greater than or equal.

Similar to existing PRED_SETGE_64, but does not set the predicate register, and results are swapped.

Operation:

```
if (src0>=src1)
  result = 0xFFFFFFFFFFFFFFFF;
else
  result = 0x0;
```

Microcode

C	DC	DR	DST_GPR						BS	ALU_INST								OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM				S1N	S1C	S1R	SRC1_SEL							S0N	S0C	S0R	SRC0_SEL					+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_SETE_64, opcode 184 (0xB8).

Floating-Point Set If Equal DirectX 10

Instructions **SETE_DX10**

Description Floating-point set if equal, based on floating-point source operands. The result, however, is an integer.

```
If (src0 == src1) {
    dst = 0xFFFFFFFF;
}
Else {
    dst = 0x0;
}
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_SETE_DX10, opcode 12 (0xC).

Integer Set If Equal

Instructions **SETE_INT**

Description Integer set if equal, based on signed or unsigned integer source operands.

```
If (src0 = src1) {
    dst = 0xFFFFFFFF;
}
Else {
    dst = 0x0;
}
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST						OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL							+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_SETE_INT, opcode 58 (0x3A).

Floating-Point Set If Greater Than Or Equal

Instructions **SETGE**

Description Floating-point set if greater than or equal.

```
If (src0 >= src1) {
    dst = 1.0f;
}
Else {
    dst = 0.0f;

}
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST						OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM				S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL				+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_SETGE, opcode 10 (0xA).

Double-Precision Floating-Point Set If Greater Than Or Equal

Instructions **SETGE_64**

Description Similar to existing PRED_SETGE_64, but does not set the predicate register, and results are swapped.

```
if (src0>=src1)
  result = 0xFFFFFFFFFFFFFFFF;
else
  result = 0x0;
```

Microcode

C	DC	DR	DST_GPR							BS	ALU_INST								OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM				S1N	S1C	S1R	SRC1_SEL							S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_SETGE_64, opcode 187 (0xBB).

Floating-Point Set If Greater Than Or Equal, DirectX 10

Instructions **SETGE_DX10**

Description Floating-point set if greater than or equal, based on floating-point source operands. The result, however, is an integer.

```
If (src0 >= src1) {
    dst = 0xFFFFFFFF;
}
Else {
    dst = 0x0;
}
```

Microcode

C	DC	DR	DST_GPR			BS	ALU_INST				OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM		S1N	S1C	S1R	SRC1_SEL		S0N	S0C	S0R	SRC0_SEL					+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_SETGE_DX10, opcode 14 (0xE).

Signed Integer Set If Greater Than Or Equal

Instructions **SETGE_INT**

Description Integer set if greater than or equal, based on signed integer source operands.

```
If (src0 >= src1) {
    dst = 0xFFFFFFFF;
}
Else {
    dst = 0x0;
}
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST						OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL							+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_SETGE_INT; opcode 60 (0x3C).

Unsigned Integer Set If Greater Than Or Equal

Instructions **SETGE_UINT**

Description Integer set if greater than or equal, based on unsigned integer source operands.

```
If (src0 >= src1) {
    dst = 0xFFFFFFFF;
}
Else {
    dst = 0x0;
}
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_SETGE_UINT, opcode 63 (0x3F).

Floating-Point Set If Greater Than

Instructions **SETGT**

Description Floating-point set if greater than.

```
If (src0 > src1) {
    dst = 1.0f;
}
Else {
    dst = 0.0f;
}
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST						OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL							+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_SETGT, opcode 9 (0x9).

Double-Precision Floating-Point Set If Greater Than

Instructions **SETGT_64**

Description Similar to existing PRED_SETGT_64, but does not set the predicate register, and results are swapped.

```
if (src0>src1)
  result = 0xFFFFFFFFFFFFFFFF;
else
  result = 0x0;
```

Microcode

C	DC	DR	DST_GPR	BS	ALU_INST	OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM	S1N	S1C	S1R	SRC1_SEL	S0N	S0C	S0R	SRC0_SEL		+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_SETGT_64, opcode 186 (0xBA).

Floating-Point Set If Greater Than, DirectX 10

Instructions **SETGT_DX10**

Description Floating-point set if greater than, based on floating-point source operands. The result, however, is an integer.

```
If (src0 > src1) {
    dst = 0xFFFFFFFF;
}
Else {
    dst = 0x0;
}
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_SETGT_DX10, opcode 13 (0xD).

Signed Integer Set If Greater Than

Instructions **SETGT_INT**

Description Integer set if greater than, based on signed integer source operands.

```
If (src0 > src1) {
    dst = 0xFFFFFFFF;
}
Else {
    dst = 0x0;
}
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM				S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL					+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_SETGT_INT, opcode 59 (0x3B).

Unsigned Integer Set If Greater Than

Instructions **SETGT_UINT**

Description Integer set if greater than, based on unsigned integer source operands.

```
If (src0 > src1) {
    dst = 0xFFFFFFFF;
}
Else {
    dst = 0x0;
}
```

Microcode

C	DC	DR	DST_GPR							BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM				S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL							+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_SETGT_UINT, opcode 62 (0x3E).

Floating-Point Set If Not Equal

Instructions **SETNE**

Description Floating-point set if not equal.

```
If (src0 != src1) {
    dst = 1.0f;
}
Else {
    dst = 0.0f;
}
```

Microcode

C	DC	DR	DST_GPR			BS	ALU_INST				OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM		S1N	S1C	S1R	SRC1_SEL		S0N	S0C	S0R	SRC0_SEL					+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_SETNE, opcode 11 (0xB).

Double-Precision Floating-Point Set If Not Equal

Instructions **SETNE_64**

Description Similar to existing PRED_SETE_64, but does not set the predicate register; results are swapped, and condition is != rather than ==.

```
if (src0!=src1)
  result = 0xFFFFFFFFFFFFFFFF;
else
  result = 0x0;
```

Microcode

C	DC	DR	DST_GPR							BS	ALU_INST									OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM					S1N	S1C	S1R	SRC1_SEL						S0N	S0C	S0R	SRC0_SEL							+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_SETNE_64, opcode 185 (0xB9).

Floating-Point Set If Not Equal, DirectX 10

Instructions **SETNE_DX10**

Description Floating-point set if not equal, based on floating-point source operands. The result, however, is an integer.

```
If (src0 != src1) {
    dst = 0xFFFFFFFF;
}
Else {
    dst = 0x0;
}
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_SETNE_DX10, opcode 15 (0xF).

Integer Set If Not Equal

Instructions **SETNE_INT**

Description Integer set if not equal, based on signed or unsigned integer source operands.

```
If (src0 != src1) {
    dst = 0xFFFFFFFF;
}
Else {
    dst = 0x0;
}
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST						OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL							+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_SETNE_INT; opcode 61 (0x3D).

Scalar Sine

Instructions **SIN**

Description Scalar sine. Input must be normalized from radians by dividing by 2*PI. The valid input domain is [-256, +256], which corresponds to an un-normalized input domain [-512*PI, +512*PI]. Out-of-range input results in float 0.

```
dst = ApproximateSin(src0);
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_SIN, opcode 141 (0x8D).

Double Square Root

Instructions **SQRT_64**

Description Src0_d is composed of high-order double bits on src0, and low-order double bits on src1. The result is a high-order dword of sqrt64; low-order bits assumed to be 0.

```
If (src0_d == 1.0f) {
  Result = 1.0f;
}
Else {
  Result = ApproximateSqrt(src0_d);
}
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL				S0N	S0C	S0R	SRC0_SEL							+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_SQRT_64, opcode 153 (0x99).

Scalar Square Root, IEEE Approximation

Instructions **SQRT_IEEE**

Description Scalar square root. Useful for normal compression.

```
If (src0 == 1.0f) {
    dst = 1.0f;
}
Else {
    dst = ApproximateSqrt(src0);
}
```

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST							OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL						+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_SQRT_IEEE, opcode 138 (0x8A).

Store Flags

Instructions **STORE_FLAGS**

Description Writes a working copy of the exception flags into a GPR if `gprwr` is enabled. Flags are inclusive of the current VLIW.

Available only in the w channel.
`dst = exception flags`

Microcode

C	DC	DR	DST_GPR					BS	ALU_INST					OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL						+0

Format `ALU_WORD0` (page 10-23) and `ALU_WORD1_OP2` (page 10-26).

Instruction Field `ALU_INST == OP2_INST_STORE_FLAGS`, opcode 218 (0xDA).

Integer Subtract

Instructions **`SUB_INT`**

Description Integer subtract, based on signed or unsigned integer source operands.

```
dst = src1 - src0;
```

Microcode

C	DC	DR	DST_GPR	BS	ALU_INST	OMOD	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1C	S1R	SRC1_SEL	S0N	S0C	S0R	SRC0_SEL	+0

Format `ALU_WORD0` (page 10-23) and `ALU_WORD1_OP2` (page 10-26).

Instruction Field `ALU_INST == OP2_INST_SUB_INT`, opcode 53 (0x35).

Output Borrow Bit of Unsigned Integer Subtract

Instructions **SUBB_UINT**

Description Output borrow bit of an unsigned integer subtract. Used with SUB_INT to achieve DX11 USUBB opcode.

```
If (src1 > src0) {
    dst = 00000001;
}
Else {
    dst = 0;
}
```

Microcode

C	DC	DR	DST_GPR							BS	ALU_INST										OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM				S1N	S1C	S1R	SRC1_SEL						S0N	S0C	S0R	SRC0_SEL								+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_SUBB_UINT, opcode 83 (0x53).

Floating-Point Truncate

Instructions **TRUNC**

Description Floating-point integer part of source operand.

```
dst = trunc(src0);
```

Microcode

C	DC	DR	DST_GPR							BS		ALU_INST										OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM					S1N	S1C	S1R	SRC1_SEL					S0N	S0C	S0R	SRC0_SEL										+0

Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_TRUNC, opcode 17 (0x11).

Byte # Float

Instructions **UBYTE0_FLT**
UBYTE1_FLT
UBYTE2_FLT
UBYTE3_FLT

Description Byte # float, where # is 0, 1, 2, or 3. Perform an unsigned integer-to-float conversion on the specified byte of src0.

For byte 0: `dst = uint2flt (src0 & 0xFF)`

For byte 1: `dst = uint2flt ((src0 >> 8) & 0xFF)`

For byte 2: `dst = uint2flt ((src0 >> 16) & 0xFF)`

For byte 3: `dst = uint2flt ((src0 >> 24) & 0xFF`

Microcode

C	DC	DR	DST_GPR				BS	ALU_INST						OMOD	WM	UP	UEM	S1A	S0A	+4
L	PS	IM			S1N	S1C	S1R	SRC1_SEL			S0N	S0C	S0R	SRC0_SEL						+0

Format `ALU_WORD0` (page 10-23) and `ALU_WORD1_OP2` (page 10-26).

Instruction Field `ALU_INST == OP2_INST_UBYTE0_FLT`, opcode 164 (0xA4).

`ALU_INST == OP2_INST_UBYTE1_FLT`, opcode 165 (0xA5).

`ALU_INST == OP2_INST_UBYTE2_FLT`, opcode 166 (0xA6).

`ALU_INST == OP2_INST_UBYTE3_FLT`, opcode 167 (0xA7).

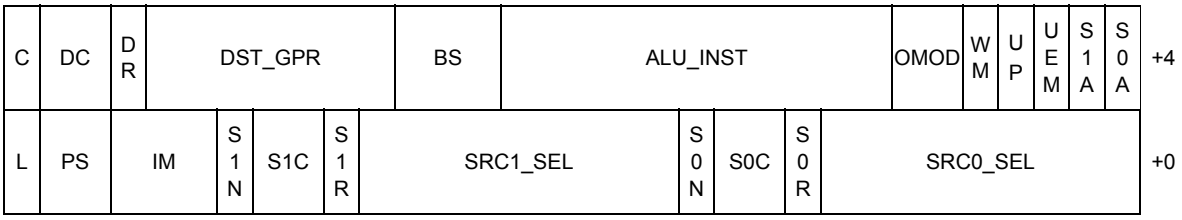
Unsigned Integer To Floating-point

Instructions **UINT_TO_FLT**

Description Unsigned integer to floating-point. The source is interpreted as an unsigned integer value, and it is converted to a floating-point result.

```
dst = (float) src0
```

Microcode



Format ALU_WORD0 (page 10-23) and ALU_WORD1_OP2 (page 10-26).

Instruction Field ALU_INST == OP2_INST_UINT_TO_FLT, opcode 156 (0xC9).

Logical Bit-Wise XOR

Instructions **XOR_INT**

Description Logical bit-wise XOR.
`dst = src0 ^ src1`

Microcode

C	DC	DR	DST_GPR	BS	ALU_INST	OMOD	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1C	S1R	SRC1_SEL	S0N	S0C	S0R	SRC0_SEL	+0

Format `ALU_WORD0` (page 10-23) and `ALU_WORD1_OP2` (page 10-26).

Instruction Field `ALU_INST == OP2_INST_XOR_INT`, opcode 50 (0x32).

9.3 Instructions for Fetches Through a Vertex Cache Clause

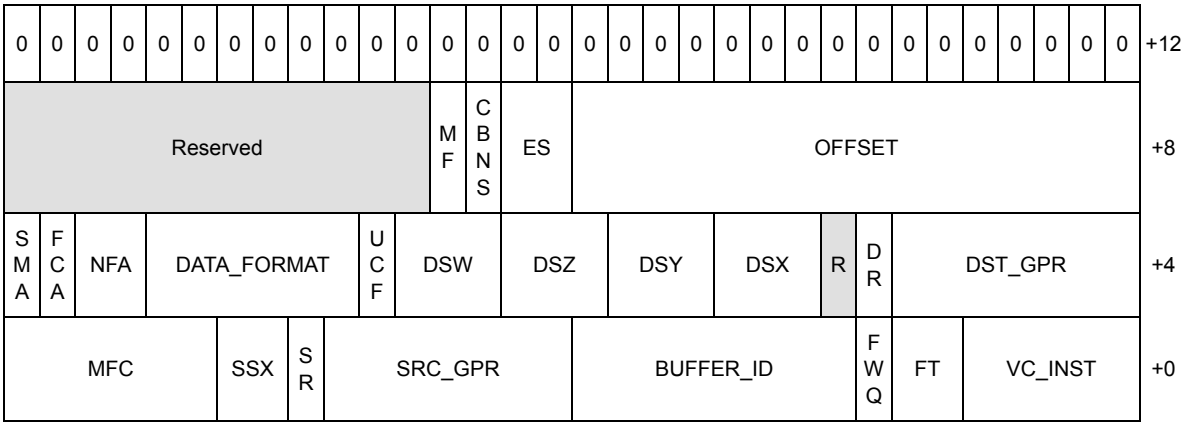
All of the instructions in this section have a mnemonic that begins with VC_INST_ in the VC_INST field of their microcode formats.

Fetch Through a Vertex Cache Clause

Instructions **FETCH**

Description Fetch through a vertex cache clause (X = unsigned integer index). These fetches specify the destination GPR directly.

Microcode



Format VTX_WORD0 (page 10-45), VTX_WORD1_GPR (page 10-47), and VTX_WORD2 (page 10-52).

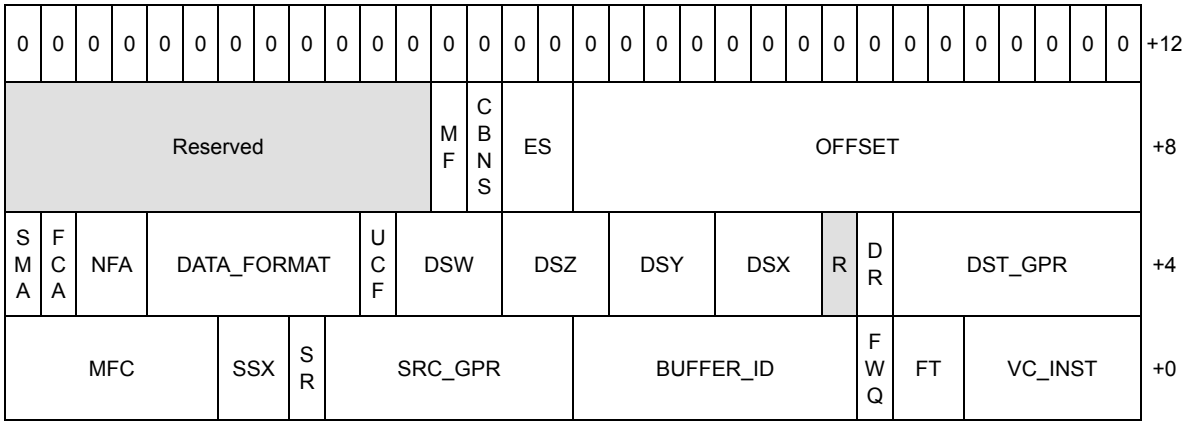
Instruction Field VC_INST == VC_INST_FETCH, opcode 0 (0x0).

Return Number of Elements in a Buffer

Instructions **GET_BUFFER_RESINFO**

Description Returns the number of elements in a buffer. This is a vertex fetch instruction and uses a vertex constant. It can be used only by a texture cache, not a vertex cache.

Microcode



Format VTX_WORD0 (page 10-45), VTX_WORD1_GPR (page 10-47), and VTX_WORD2 (page 10-52).

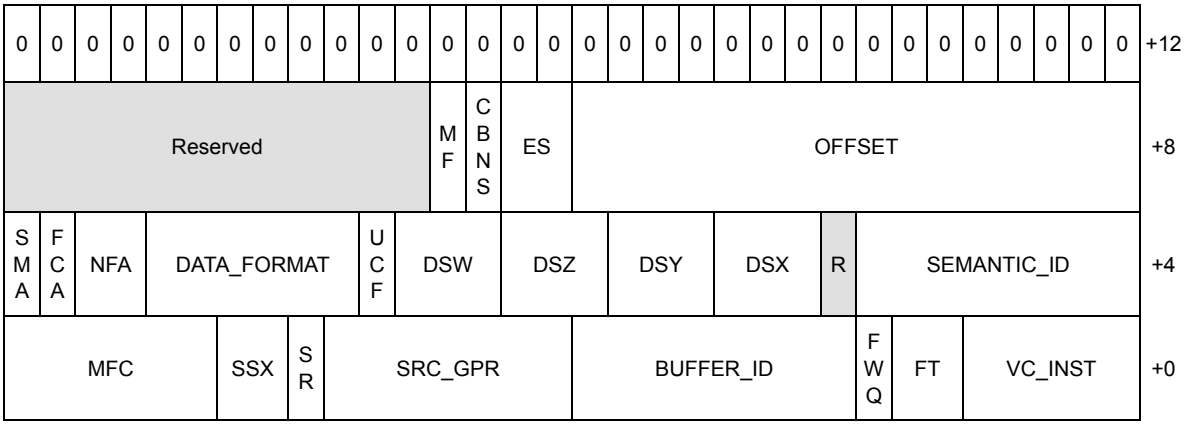
Instruction Field VC_INST == VC_INST_GET_BUFFER_RESINFO, opcode 14 (0xE).

Semantic Fetch Through a Vertex Cache Clause

Instructions **SEMANTIC**

Description Semantic fetch through a vertex cache clause. These fetches specify the 8-bit semantic ID that is looked up in a table to determine the GPR to which the data is written.

Microcode



Format VTX_WORD0 (page 10-45), VTX_WORD1_SEM (page 10-50), and VTX_WORD2 (page 10-52).

Instruction Field VC_INST == VC_INST_SEMANTIC, opcode 1 (0x1).

9.4 Instructions for a Fetch Through a Texture Cache Clause

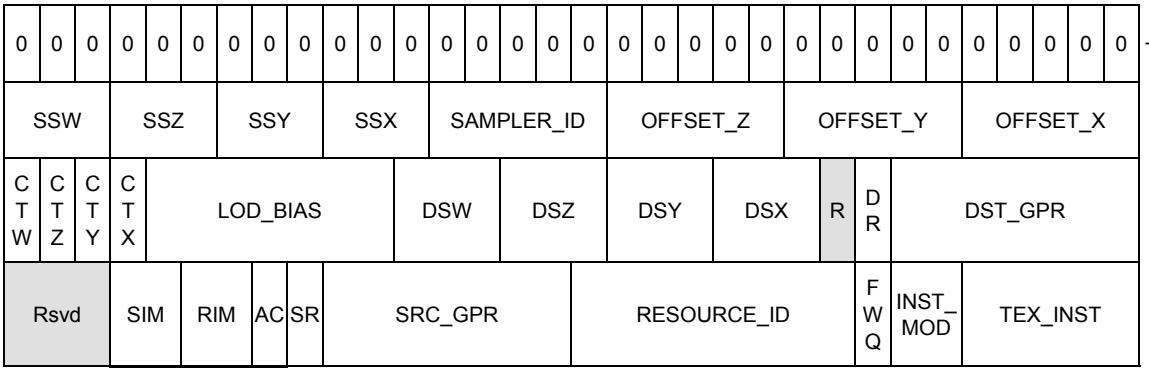
All of the instructions in this section have a mnemonic that begins with TEX_INST_ in the TEX_INST field of their microcode formats.

Fetch Four Texels (In A 2x23 Pattern)

Instructions **GATHER4**

Description Fetches unfiltered texels from a bilinear sample and packs them into xyzw.

Microcode



Format TEX_WORD0 (page 10-54), TEX_WORD1 (page 10-57), and TEX_WORD2 (page 10-58).

Instruction Field TEX_INST == TEX_INST_GATHER4, opcode 21 (0x15).

Gather4 With Depth Comparison

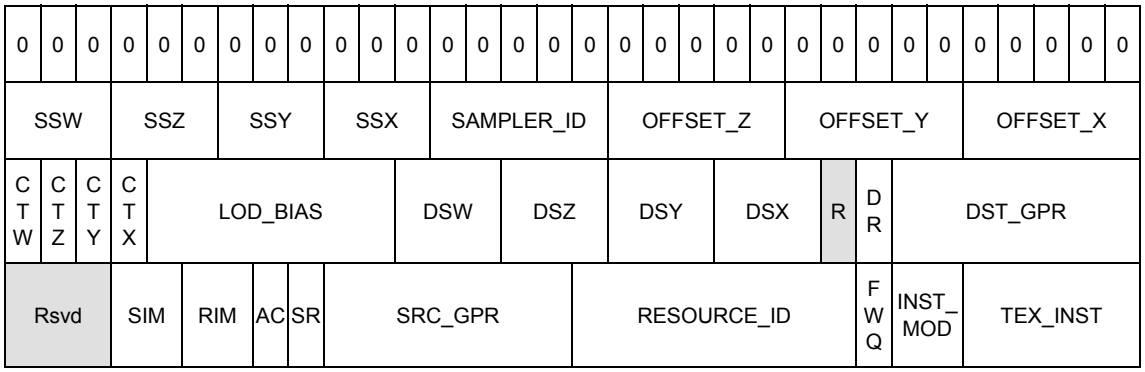
Instructions **GATHER4_C**

Description Fetches unfiltered texels from a bilinear sample, and performs a depth comparison similar to SAMPLE_C on each texel; then packs them into x, y, z, and w. Performs a depth comparison similar to SAMPLE_C.

This instruction compares the reference value in src0.W with the sampled value from memory. The reference value is converted to the source format before the compare. NANs are honored in the comparisons for formats supporting them, otherwise, they are converted to 0 or +/-MAX. A passing compare puts a 1.0 in the src0.X element. A failing compare puts a 0.0 in the src0.X element.

The reference Z value is specified in the w channel.

Microcode



Format TEX_WORD0 (page 10-54), TEX_WORD1 (page 10-57), and TEX_WORD2 (page 10-58).

Instruction Field TEX_INST == TEX_INST_GATHER4_C, opcode 29 (0x1D).

Gather4 With Depth Comparison and GPR Coordinate Offsets

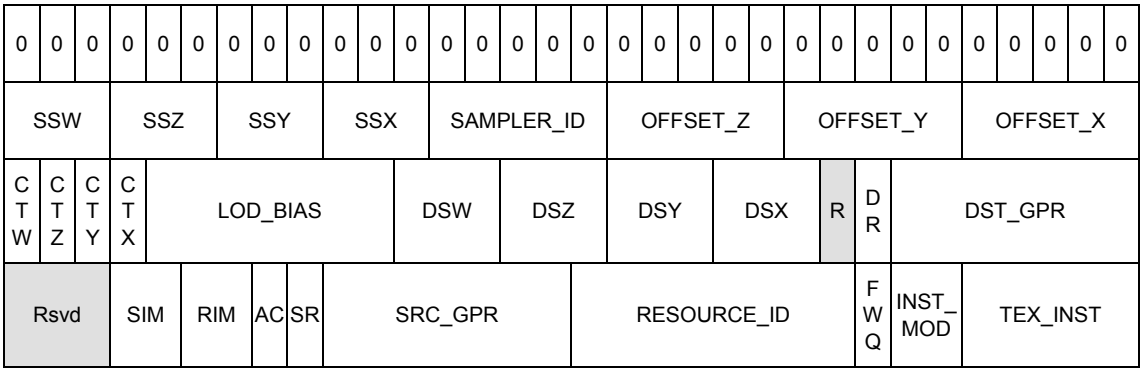
Instructions **GATHER4_C_O**

Description Fetches unfiltered texels from a bilinear sample using texel offsets from a previous SET_TEXTURE_OFFSETS instruction; then performs a depth comparison similar to SAMPLE_C on each texel, packing the results into x, y, z, and w. OFFSET_X, OFFSET_Y, and OFFSET_Z in the microcode are ignored.

This instruction compares the reference value in src0.W with the sampled value from memory. The reference value is converted to the source format before the compare. NANs are honored in the comparisons for formats supporting them, otherwise, they are converted to 0 or +/-MAX. A passing compare puts a 1.0 in the src0.X element. A failing compare puts a 0.0 in the src0.X element.

Uses previously set texture offsets, and reference Z value in the W channel.

Microcode



Format TEX_WORD0 (page 10-54), TEX_WORD1 (page 10-57), and TEX_WORD2 (page 10-58).

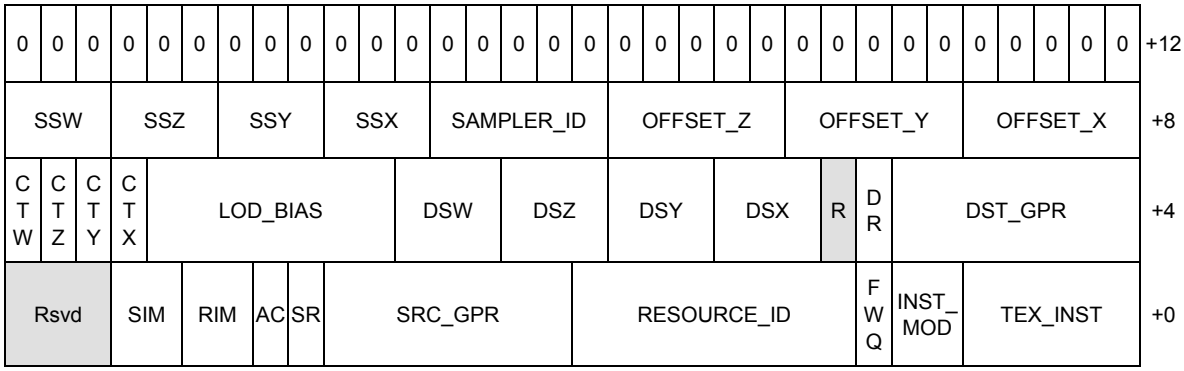
Instruction Field TEX_INST == TEX_INST_GATHER4_C_O, opcode 31 (0x1F).

Gather4 with GPR Coordinate Offsets

Instructions **GATHER4_O**

Description Fetches unfiltered texels from a bilinear sample and packs them into xyzw, using texel offsets from a previous SET_TEXTURE_OFFSETS instruction. OFFSET_X, OFFSET_Y, and OFFSET_Z in the microcode are ignored.

Microcode



Format TEX_WORD0 (page 10-54), TEX_WORD1 (page 10-57), and TEX_WORD2 (page 10-58).

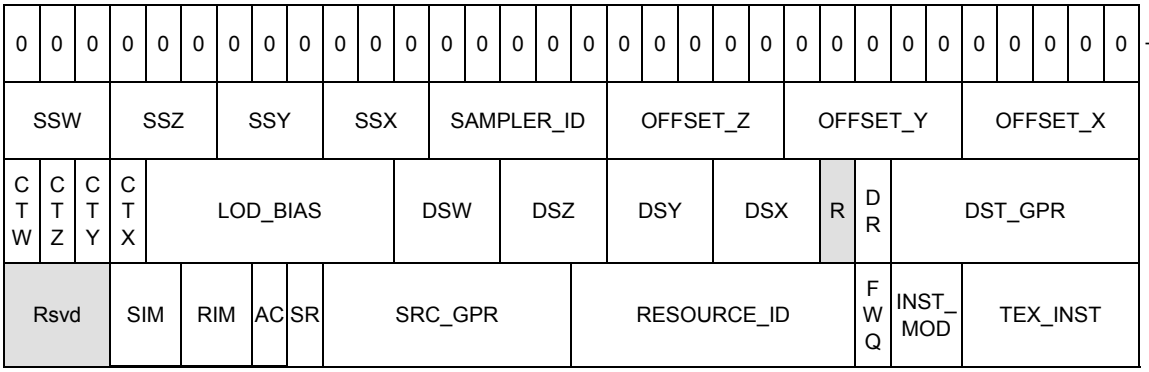
Instruction Field TEX_INST == TEX_INST_GATHER4_O, opcode 23 (0x61).

Get Slopes Relative To Horizontal

Instructions **GET_GRADIENTS_H**

Description Retrieve slopes relative to horizontal: X = dx/dh, Y = dy/dh, Z = dz/dh, W = dw/dh.

Microcode



Format TEX_WORD0 (page 10-54), TEX_WORD1 (page 10-57), and TEX_WORD2 (page 10-58).

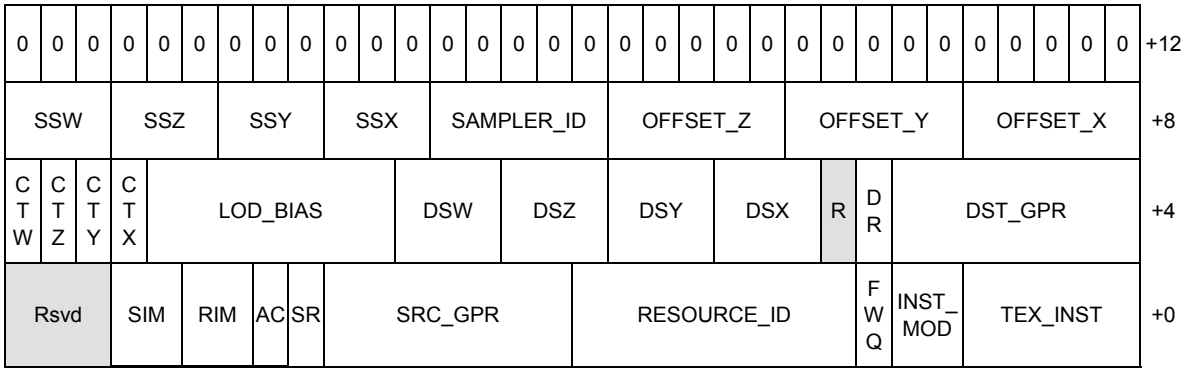
Instruction Field TEX_INST == TEX_INST_GET_GRADIENTS_H, opcode 7 (0x7).

Get Slopes Relative To Vertical

Instructions **GET_GRADIENTS_V**

Description Retrieve slopes relative to vertical: X = dx/dv, Y = dy/dv, Z = dz/dv, W = dw/dv.

Microcode



Format `TEX_WORD0` (page 10-54), `TEX_WORD1` (page 10-57), and `TEX_WORD2` (page 10-58).

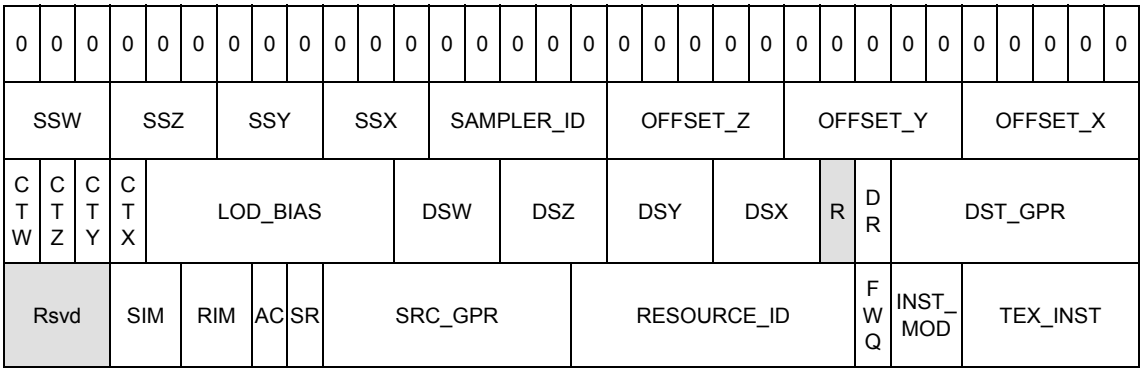
Instruction Field `TEX_INST == TEX_INST_GET_GRADIENTS_V`, opcode 8 (0x8).

Get Computed Level of Detail For Pixels

Instructions **GET_LOD**

Description Computed level of detail (LOD) for all pixels in quad. This instruction returns the clamped LOD into the X component of the dst GPR; the non-clamped is placed into the Y component of the dst GPR.

Microcode



Format TEX_WORD0 (page 10-54), TEX_WORD1 (page 10-57), and TEX_WORD2 (page 10-58).

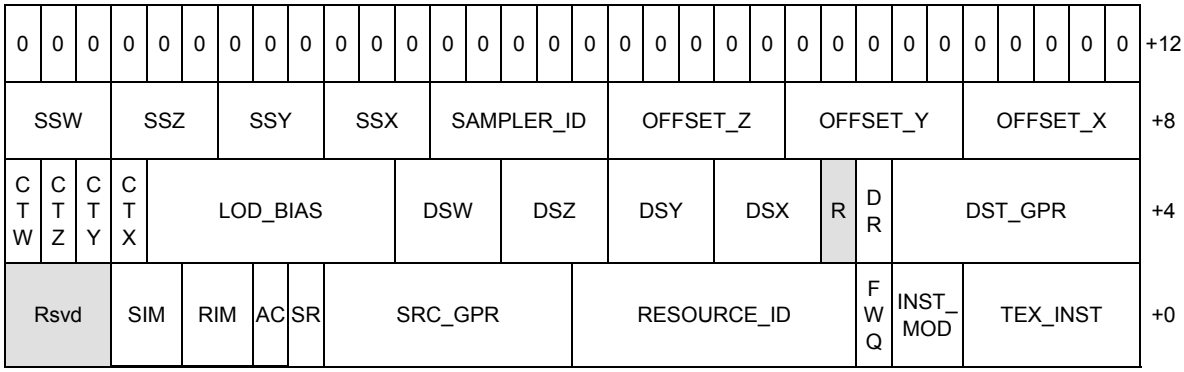
Instruction Field TEX_INST == TEX_INST_GET_COMP_TEX_LOD, opcode 6 (0x6).

Get Number of Samples

Instructions **GET_NUMBER_OF_SAMPLES**

Description Gets and returns the number of samples.

Microcode



Format TEX_WORD0 (page 10-54), TEX_WORD1 (page 10-57), and TEX_WORD2 (page 10-58).

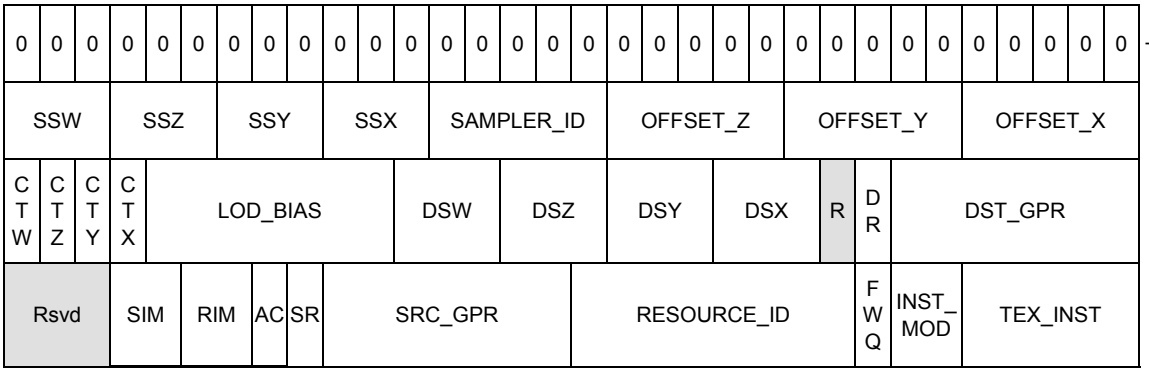
Instruction Field TEX_INST == TEX_INST_GET_NUMBER_OF_SAMPLES, opcode 5 (0x5).

Get Texture Resolution

Instructions **GET_TEXTURE_RESINFO**

Description Retrieve width, height, depth, and number of mipmap levels.

Microcode



Format TEX_WORD0 (page 10-54), TEX_WORD1 (page 10-57), and TEX_WORD2 (page 10-58).

Instruction Field TEX_INST == TEX_INST_GET_TEXTURE_RESINFO, opcode 4 (0x4).

Keep Gradients

Instructions **KEEP_GRADIENTS**

Description Compute the gradients from coordinates and store them. Using the provided address, the calculates the derivatives as they would be calculated in the sample instruction. It stores the derivatives for use by subsequent instructions that receive derivatives as extra parameters (for example: SAMPLE_D). Keep_Gradients is not meant to return data; however, unless masked or set to output constant 1 or 0 by the DS* fields, it returns the border color as determined by the sampler and resource. This instruction is equivalent to:

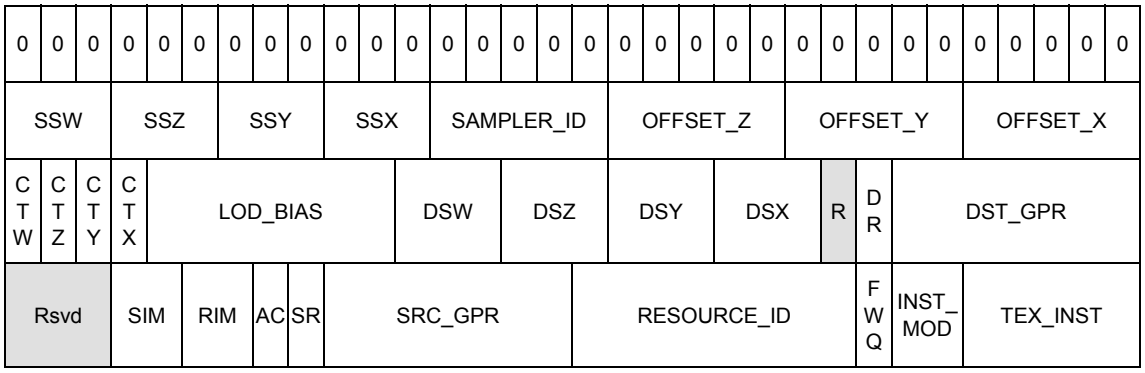
GetGradientsH

GetGradientsV

SetGradientsH

SetGradientsV

Microcode



Format `TEX_WORD0` (page 10-54), `TEX_WORD1` (page 10-57), and `TEX_WORD2` (page 10-58).

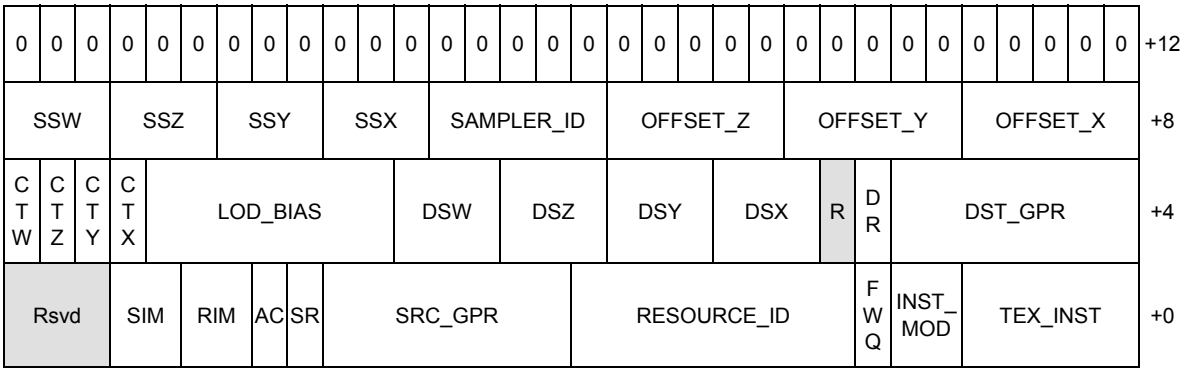
Instruction Field `TEX_INST == TEX_INST_KEEP_GRADIENTS`, opcode 10 (0xA).

Load Texture Elements

Instructions **LD**

Description Using an address of unsigned integers X, Y, Z, and W (where W is interpreted by the texture unit as LOD or LOD_BIAS), this instruction fetches data from a buffer or texel without filtering. The source data can come from any resource type other than a cubemap.

Microcode



Format TEX_WORD0 (page 10-54), TEX_WORD1 (page 10-57), and TEX_WORD2 (page 10-58).

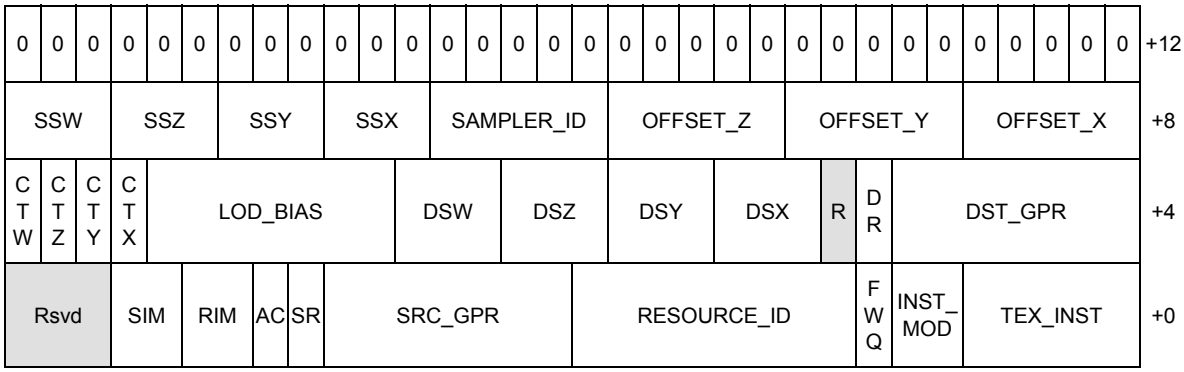
Instruction Field TEX_INST == TEX_INST_LD, opcode 3 (0x3).

Sample Texture

Instructions **SAMPLE**

Description Fetch a texture sample and do arithmetic on it. The RESOURCE_ID field specifies the texture sample. The SAMPLER_ID field specifies the arithmetic. The horizontal and vertical gradients for the source address are calculated by the hardware.

Microcode



Format TEX_WORD0 (page 10-54), TEX_WORD1 (page 10-57), and TEX_WORD2 (page 10-58).

Instruction Field TEX_INST == TEX_INST_SAMPLE, opcode 16 (0x10).

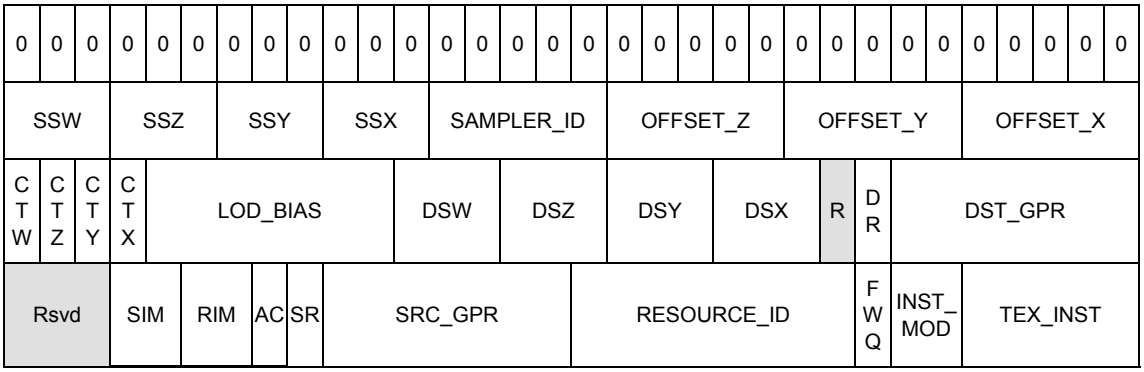
Sample Texture with Comparison

Instructions **SAMPLE_C**

Description Fetch a texture sample and process it. The RESOURCE_ID field specifies the texture sample. The SAMPLER_ID field specifies the arithmetic. The horizontal and vertical gradients for the source address are calculated by the hardware.

This instruction compares the reference value in src0.W with the sampled value from memory. The reference value is converted to the source format before the compare. NANs are honored in the comparisons for formats supporting them, otherwise, they are converted to 0 or +/-MAX. A passing compare puts a 1.0 in the src0.X element. A failing compare puts a 0.0 in the src0.X element.

Microcode



Format TEX_WORD0 (page 10-54), TEX_WORD1 (page 10-57), and TEX_WORD2 (page 10-58).

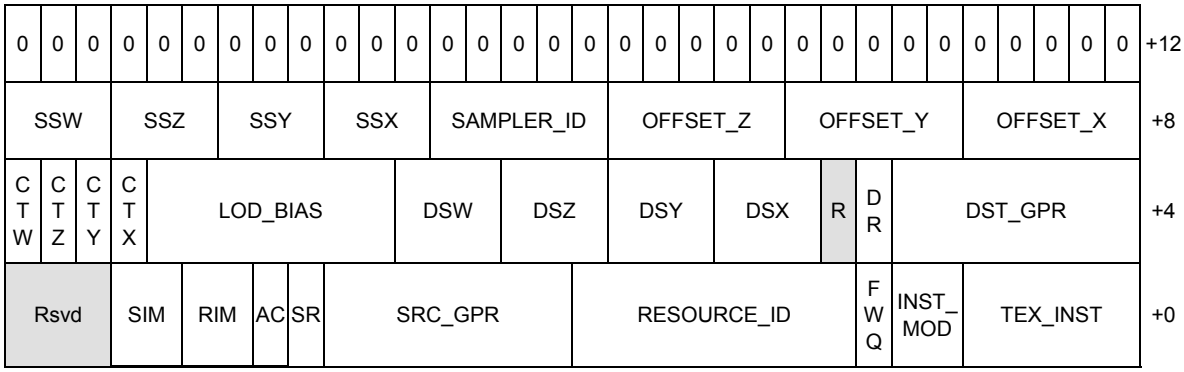
Instruction Field TEX_INST == TEX_INST_SAMPLE_C, opcode 24 (0x18).

Sample Texture with Comparison and Gradient

Instructions **SAMPLE_C_G**

Description This instruction behaves exactly like the SAMPLE_C instruction, except that instead of using the hardware-calculated horizontal and vertical gradients for the source address, the gradients are provided by software in the most recently executed set gradients H and set gradients V.

Microcode



Format TEX_WORD0 (page 10-54), TEX_WORD1 (page 10-57), and TEX_WORD2 (page 10-58).

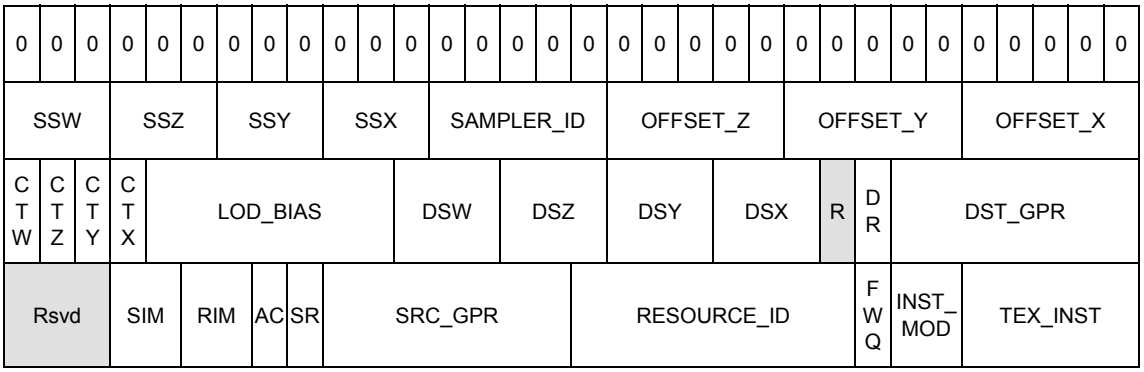
Instruction Field TEX_INST == TEX_INST_SAMPLE_C_G, opcode 28 (0x1C).

Sample Texture with Comparison, Gradient, and LOD Bias

Instructions **SAMPLE_C_G_LB**

Description This instruction behaves exactly like the SAMPLE_C_G instruction, except that a constant bias value, placed in the instruction's LOD_BIAS field by the compiler, is added to the computed LOD for the source address.

Microcode



Format TEX_WORD0 (page 10-54), TEX_WORD1 (page 10-57), and TEX_WORD2 (page 10-58).

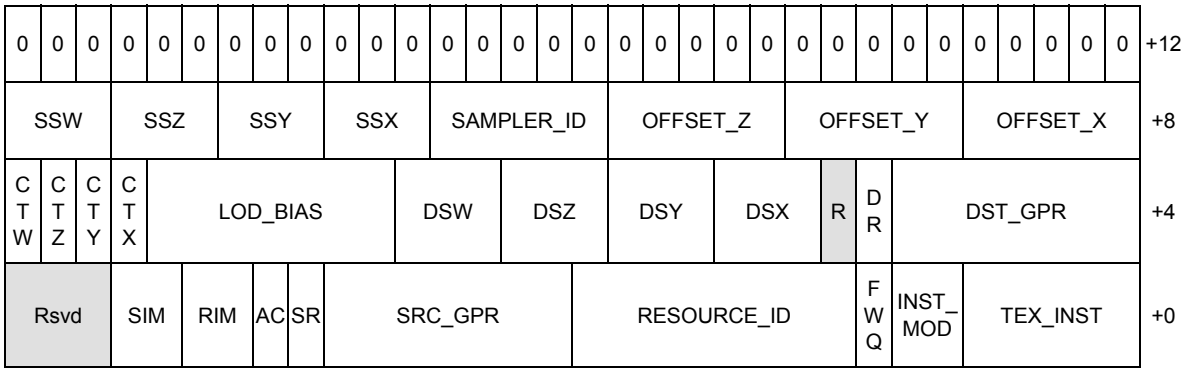
Instruction Field TEX_INST == TEX_INST_SAMPLE_C_G_LB, opcode 30 (0x1E).

Sample Texture with LOD

Instructions **SAMPLE_C_L**

Description This instruction behaves exactly like the SAMPLE_C instruction, except that the hardware-computed mipmap level of detail (LOD) is replaced with the LOD determined by the texture coordinate in src0.W.

Microcode



Format TEX_WORD0 (page 10-54), TEX_WORD1 (page 10-57), and TEX_WORD2 (page 10-58).

Instruction Field TEX_INST == TEX_INST_SAMPLE_C_L, opcode 25 (0x19).

Sample Texture with LOD Bias

Instructions **SAMPLE_C_LB**

Description This instruction behaves exactly like the SAMPLE_C instruction, except that a constant bias value, placed in the instruction's LOD_BIAS field by the compiler, is added to the computed LOD for the source address.

Microcode

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	+12
SSW			SSZ			SSY			SSX			SAMPLER_ID					OFFSET_Z					OFFSET_Y					OFFSET_X					+8
CTW	CTZ	CTY	CTX	LOD_BIAS								DSW			DSZ		DSY			DSX		R	DR	DST_GPR								+4
Rsvd			SIM			RIM		AC	SR	SRC_GPR										RESOURCE_ID				FWQ	INST_MOD	TEX_INST					+0

Format TEX_WORD0 (page 10-54), TEX_WORD1 (page 10-57), and TEX_WORD2 (page 10-58).

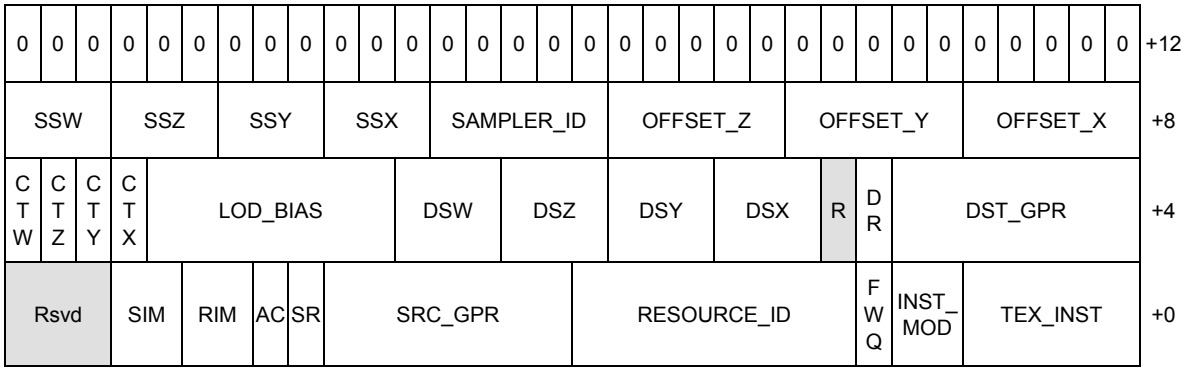
Instruction Field TEX_INST == TEX_INST_SAMPLE_C_LB, opcode 26 (0x1A).

Sample Texture with LOD Zero

Instructions **SAMPLE_C_LZ**

Description This instruction behaves exactly like the SAMPLE_C instruction, except that the mipmap level of detail (LOD) and fraction are forced to zero before level-clamping.

Microcode



Format TEX_WORD0 (page 10-54), TEX_WORD1 (page 10-57), and TEX_WORD2 (page 10-58).

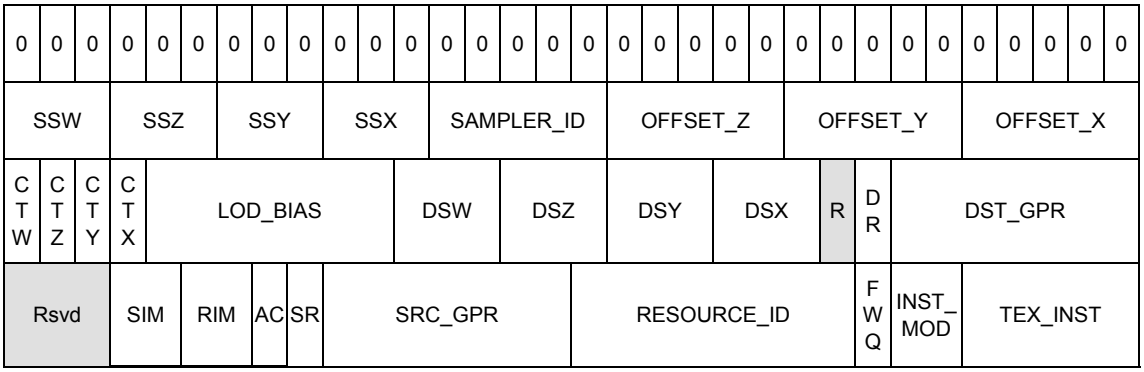
Instruction Field TEX_INST == TEX_INST_SAMPLE_C_LZ, opcode 27 (0x1B).

Sample Texture with Gradient

Instructions **SAMPLE_G**

Description This instruction behaves exactly like the SAMPLE instruction, except that instead of using the hardware-calculated horizontal and vertical gradients for the source address, the gradients are provided by software in the last-executed SET_GRADIENTS_H and SET_GRADIENTS_V instructions.

Microcode



Format TEX_WORD0 (page 10-54), TEX_WORD1 (page 10-57), and TEX_WORD2 (page 10-58).

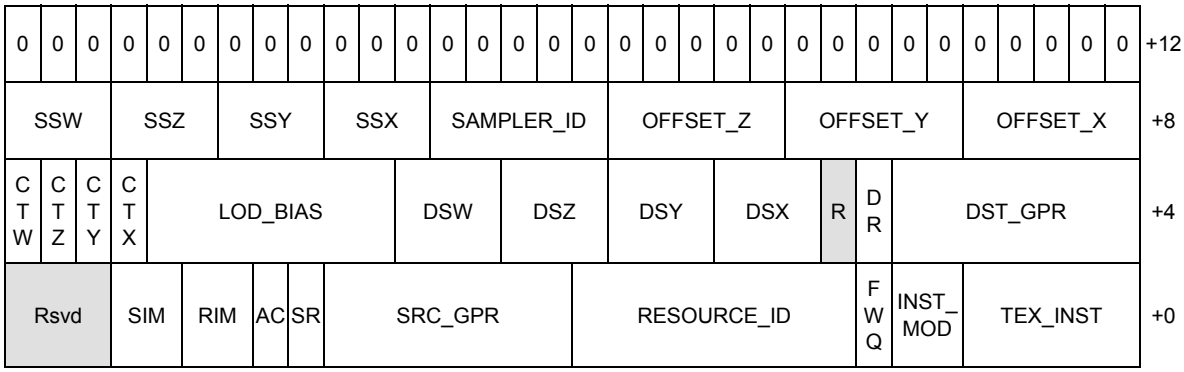
Instruction Field TEX_INST == TEX_INST_SAMPLE_G; opcode 20 (0x14).

Sample Texture with Gradient and LOD Bias

Instructions **SAMPLE_G_LB**

Description This instruction behaves exactly like the SAMPLE_G instruction, except that a constant bias value, placed in the instruction's LOD_BIAS field by the compiler, is added to the computed LOD for the source address.

Microcode



Format TEX_WORD0 (page 10-54), TEX_WORD1 (page 10-57), and TEX_WORD2 (page 10-58).

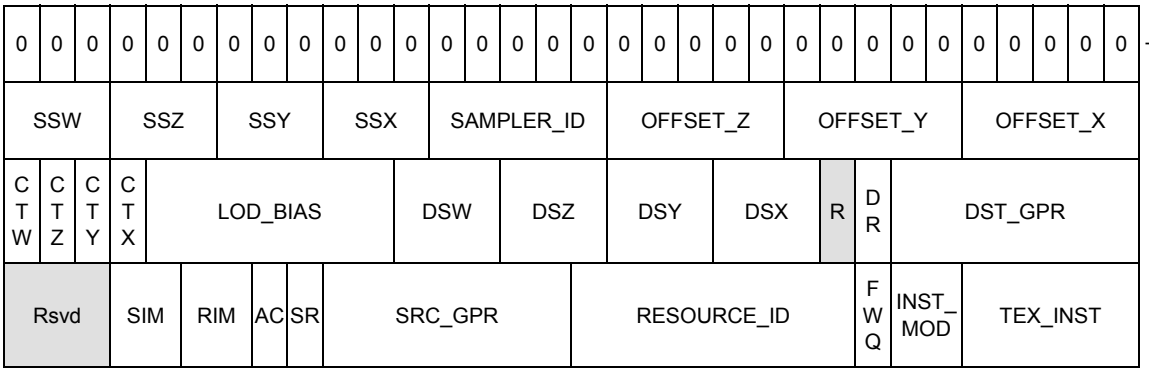
Instruction Field TEX_INST == TEX_INST_SAMPLE_G_LB, opcode 22 (0x16).

Sample Texture with LOD

Instructions **SAMPLE_L**

Description This instruction behaves exactly like the SAMPLE instruction, except that the hardware-computed mipmap level of detail (LOD) is replaced with the LOD determined by the texture coordinate in src0.W.

Microcode



Format TEX_WORD0 (page 10-54), TEX_WORD1 (page 10-57), and TEX_WORD2 (page 10-58).

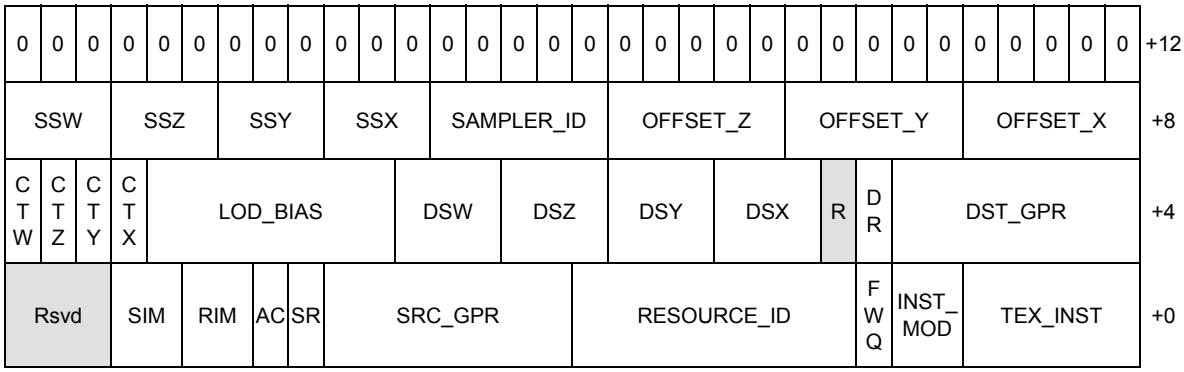
Instruction Field TEX_INST == TEX_INST_SAMPLE_L, opcode 17 (0x11).

Sample Texture with LOD Bias

Instructions **SAMPLE_LB**

Description This instruction behaves exactly like the SAMPLE instruction, except that a constant bias value, placed in the instruction's LOD_BIAS field by the compiler, is added to the computed LOD for the source address.

Microcode



Format TEX_WORD0 (page 10-54), TEX_WORD1 (page 10-57), and TEX_WORD2 (page 10-58).

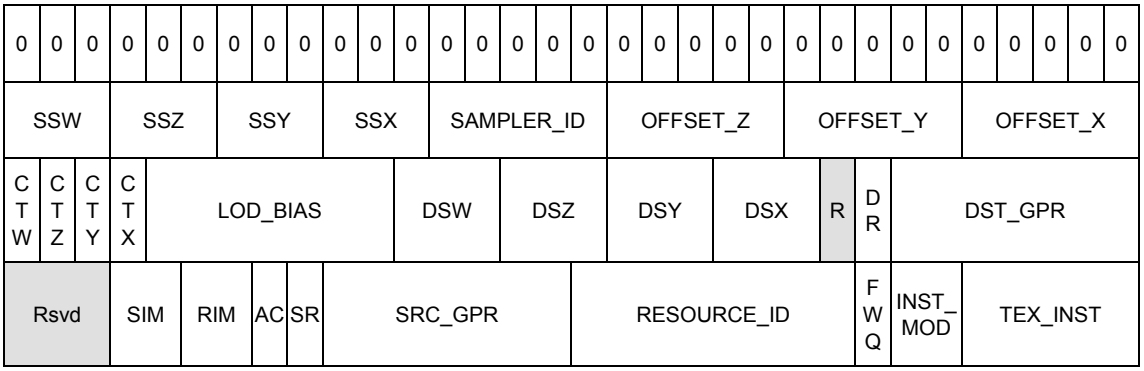
Instruction Field TEX_INST == TEX_INST_SAMPLE_LB, opcode 18 (0x12).

Sample Texture with LOD Zero

Instructions **SAMPLE_LZ**

Description This instruction behaves exactly like the SAMPLE instruction, except that the mipmap level of detail (LOD) and fraction are forced to zero before level-clamping.

Microcode



Format TEX_WORD0 (page 10-54), TEX_WORD1 (page 10-57), and TEX_WORD2 (page 10-58).

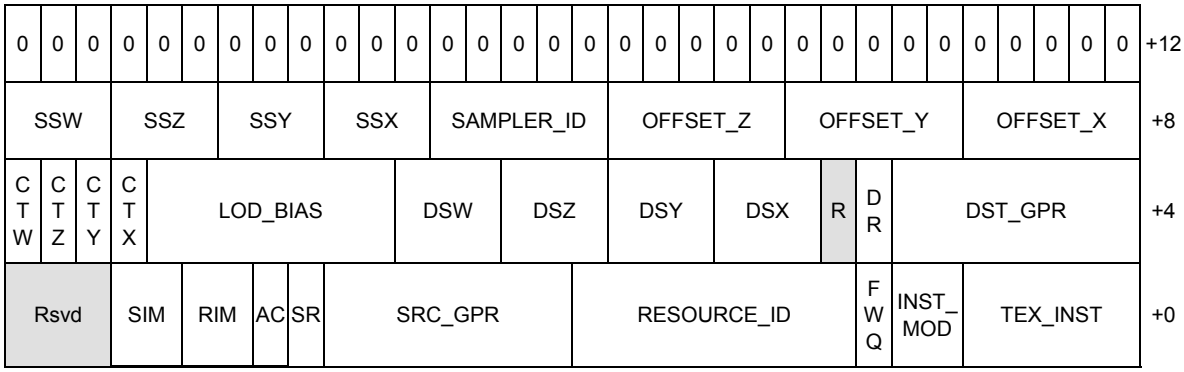
Instruction Field TEX_INST == TEX_INST_SAMPLE_LZ, opcode 19 (0x13).

Set Horizontal Gradients

Instructions **SET_GRADIENTS_H**

Description Set horizontal gradients specified by X, Y, Z coordinates.

Microcode



Format `TEX_WORD0` (page 10-54), `TEX_WORD1` (page 10-57), and `TEX_WORD2` (page 10-58).

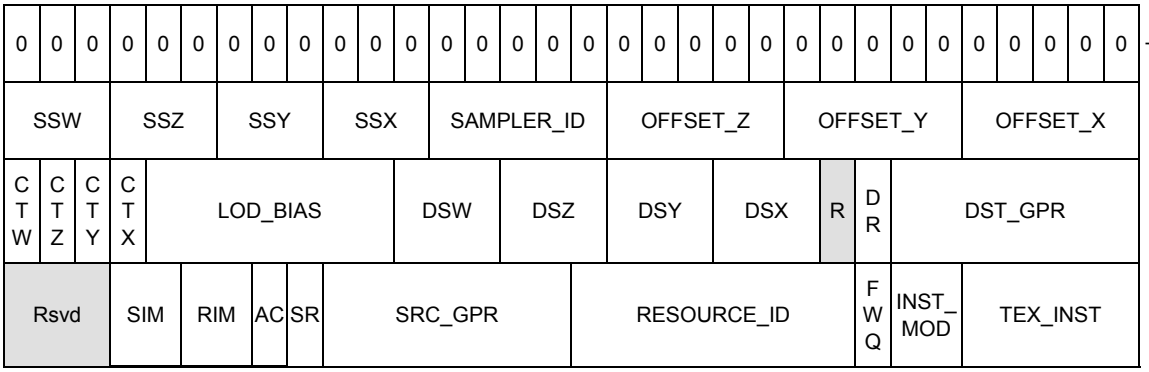
Instruction Field `TEX_INST == TEX_INST_SET_GRADIENTS_H`, opcode 11 (0xB).

Set Vertical Gradients

Instructions **SET_GRADIENTS_V**

Description Set vertical gradients specified by X, Y, Z coordinates.

Microcode



Format TEX_WORD0 (page 10-54), TEX_WORD1 (page 10-57), and TEX_WORD2 (page 10-58).

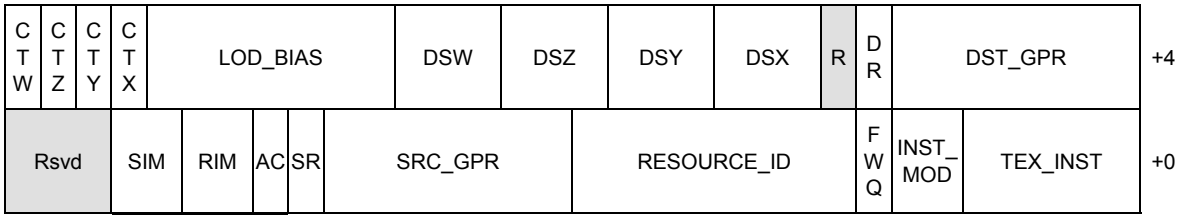
Instruction Field TEX_INST == TEX_INST_SET_GRADIENTS_V, opcode 12 (0xC).

Set Texture Offsets

Instructions **SET_TEXTURE_OFFSETS**

Description Sets texture offsets from a GPR for use with `gather4_o` and `gather4_c_o`.

Microcode



Format `TEX_WORD0` (page 10-54) and `TEX_WORD1` (page 10-57).

Instruction Field `TEX_INST == TEX_INST_SET_TEXTURE_OFFSETS`, opcode 9 (0x9).

9.5 Memory Read Instructions

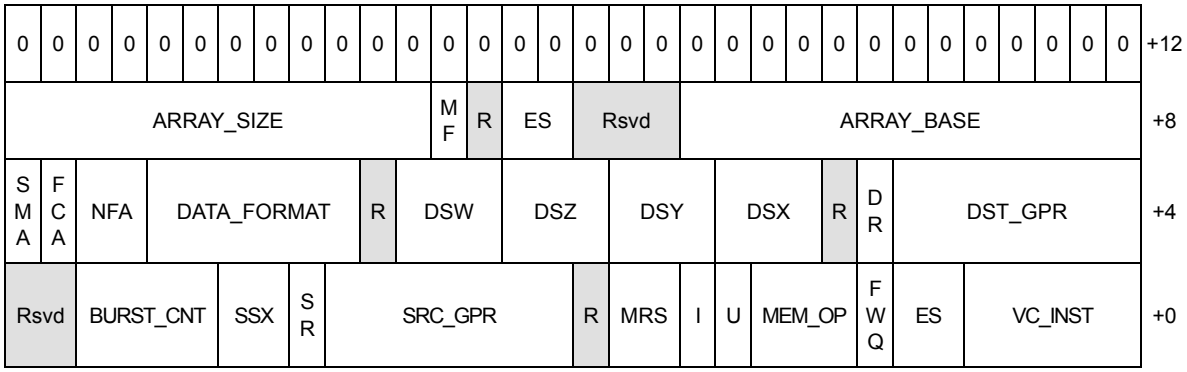
All of the instructions in this section have a mnemonic that begins with MEM_OP_RD in the MEM_OP field of their microcode formats.

Read Scatter Buffer

Instructions **MEM_RD_SCATTER**

Description Read the scatter buffer.

Microcode



Format MEM_RD_WORD0 (page 10-59), MEM_RD_WORD1 (page 10-61), and MEM_RD_WORD2 (page 10-63).

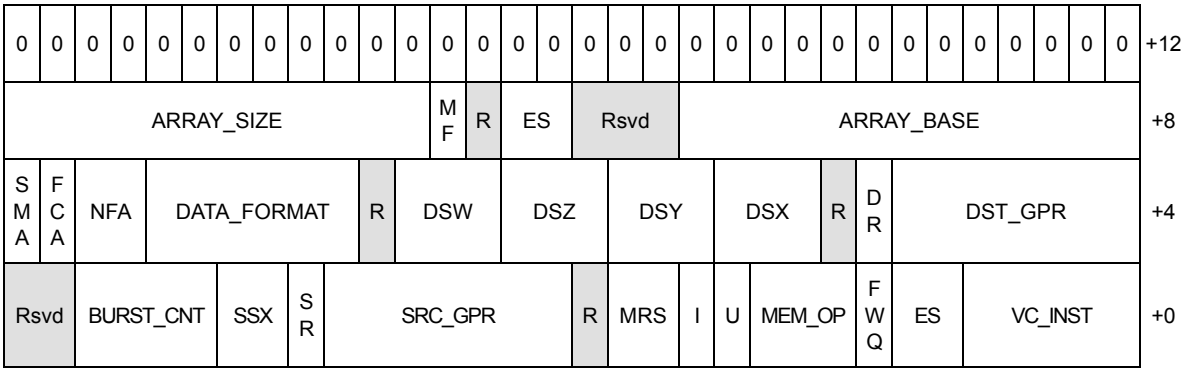
Instruction Field VC_INST == VC_INST_MEM, opcode 2 (0x2).
MEM_OP == MEM_RD_SCATTER, opcode 2 (0x2).

Read Scratch Buffer

Instructions **MEM_RD_SCRATCH**

Description Read the scratch (temporary) buffer.

Microcode



Format `MEM_RD_WORD0` (page 10-59), `MEM_RD_WORD1` (page 10-61), and `MEM_RD_WORD2` (page 10-63).

Instruction Field `VC_INST == VC_INST_MEM`, opcode 2 (0x2).
`MEM_OP == MEM_RD_SCRATCH`, opcode 0 (0x0)

9.6 Data Share Read/Write Instructions

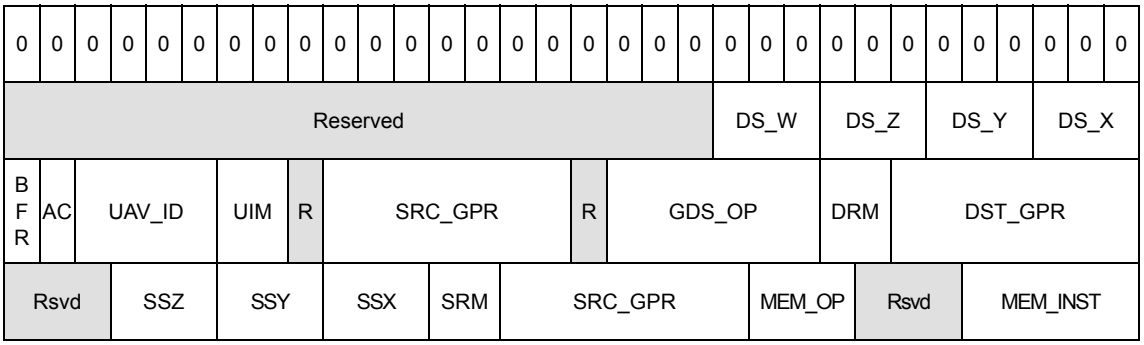
All of the instructions in this section have a mnemonic that begins with MEM_OP_ in the MEM_OP field of their microcode formats.

Global Data Share Write

Instructions **MEM_GDS**

Description Global data sharing read or write. Use only in a CF_INST_GDS clause.

Microcode



Format MEM_GDS_WORD0 (page 10-64), MEM_GDS_WORD1 (page 10-65), and MEM_GDS_WORD2 (page 10-68).

Instruction Field MEM_INST == MEM_INST_MEM, opcode 2 (0x2).

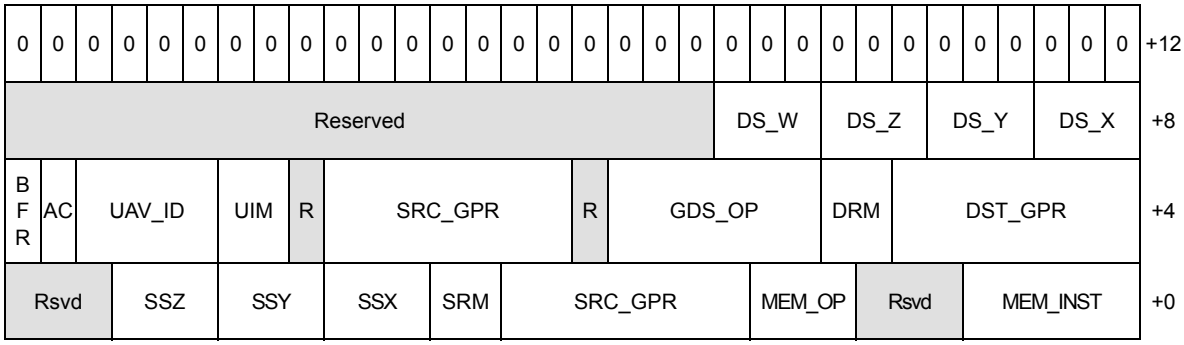
MEM_OP == MEM_GDS, opcode 4 (0x4).

Tesselation Buffer Write

Instructions **MEM_TF_WRITE**

Description Writes to a tesselation buffer. Use only in a CF_INST_GDS clause.

Microcode



Format MEM_GDS_WORD0 (page 10-64), MEM_GDS_WORD1 (page 10-65) and MEM_GDS_WORD2 (page 10-68).

Instruction Field MEM_INST == MEM_INST_MEM, opcode 2 (0x2).

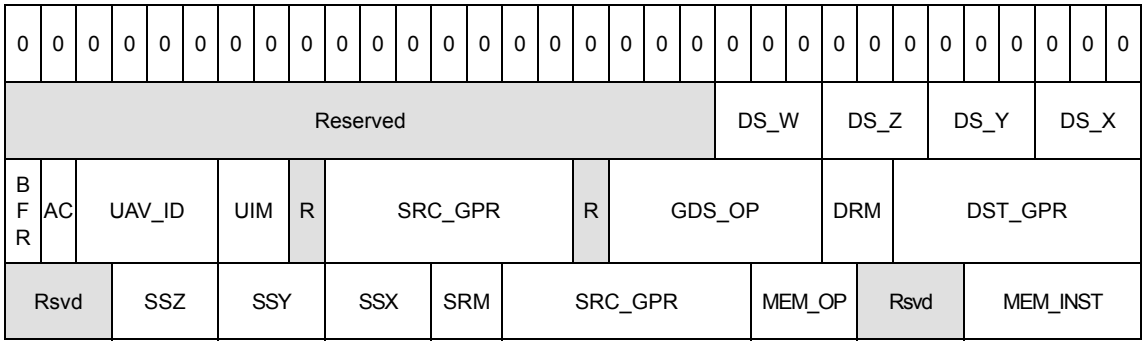
MEM_OP == MEM_TF_WRITE, opcode 5 (0x5).

Global Data Share Write

Instructions **GLOBAL_DS_WRITE**

Description Data sharing between SIMDs.

Microcode



Format MEM_GDS_WORD0 (page 10-64), MEM_GDS_WORD1 (page 10-65), and MEM_GDS_WORD2 (page 10-68).

Instruction Field MEM_INST == MEM_INST_MEM, opcode 2 (0x2).

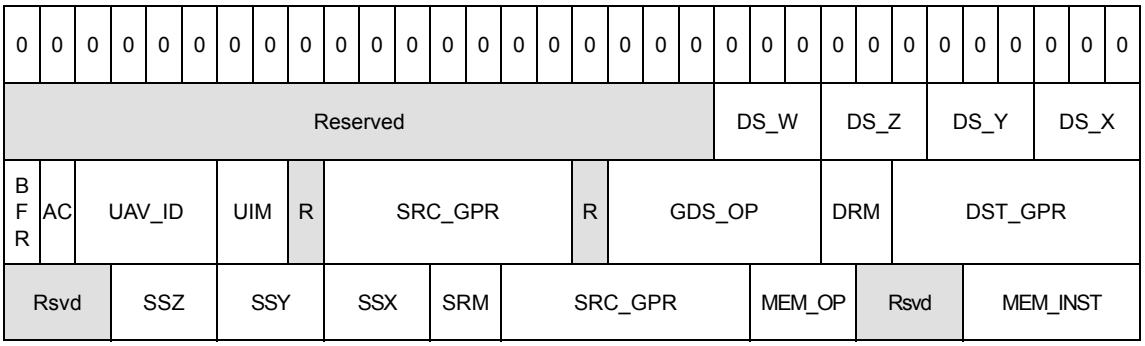
MEM_OP == MEM_GDS, opcode 4 (0x4)

Global Data Share Read

Instructions **GLOBAL_DS_READ**

Description Data sharing between SIMDs.

Microcode



Format MEM_GDS_WORD0 (page 10-64), MEM_GDS_WORD1 (page 10-65), and MEM_GDS_WORD2 (page 10-68).

Instruction Field MEM_INST == MEM_INST_MEM, opcode 2 (0x2).

MEM_OP == MEM_GDS, opcode 4 (0x4)

9.7 Local Data Share (LDS) Instructions

LDS Indexed Operation

Instructions **LDS_IDX_OP**

Description Issues an LDS indexed read/write/atomic operation. Supported only in the x element. Indexed operations are responsible for the address calculation of store, indexed read, and atomic operations.

Microcode

O3	DC	O2	O0	LDS_OP	BS	ALU_INST	O1	S2C	S2R	SRC2_SEL		+4
LAST	PRED_SEL	INDEX_MODE	O5	S1C	S1R	SRC1_SEL	O4	S0C	S0R	SRC0_SEL		+0

Format ALU_WORD0_LDS_IDX_OP (page 10-36) and ALU_WORD1_LDS_IDX_OP (page 10-39).

Instruction Field ALU_INST == OP3_INST_LDS_IDX_OP, opcode 17 (0x11).

where:

BS = BANK_SWIZZLE
DC = DST_CHAN
IM = INDEX_MODE
L = LAST
O# = IDX_OFFSET_#
PS = PRED_SEL
S0C = SRC0_CHAN
S0R = SRC0_REL
S1C = SRC1_CHAN
S1R = SRC1_REL
S2C = SRC2_CHAN
S2R = SRC2_REL

If the LDS_IDX_OP instruction is placed in the ALU_INST field, the LDS_OP field can take any of the following LDS instructions (Table 9.11).

Table 9.11 LDS Instructions for the LDS_OP Field

LDS Instruction	Description (C-Function Equivalent)	OP
LDS_ADD	LDS(dst) += src	1A1D
LDS_ADD_RET	LDS(dst) += src	1A1D
LDS_AND	LDS(dst) &= src	1A1D
LDS_AND_RET	LDS(dst) &= src	1A1D
LDS_ATOMIC_ORDERED_ ALLOC_RET	This instructions is for global data share (GDS) only.[1]	1A
LDS_BYTE_READ_RET	OQA = SignExtend(LDS(dst)[7:0])[2]	1A
LDS_BYTE_WRITE	LDS(dst) = src[7:0]	1A1D
LDS_CMP_STORE	LDS(dst) = (LDS(dst) == cmp) ? src : LDS(dst)	1A2D
LDS_CMP_STORE_SPF	LDS(dst) = (LDS(dst) == cmp) ? src : LDS(dst)	1A2D
LDS_CMP_XCHG_RET	(LDS(dst) == cmp) ? LDS(dst) = src : LDS(dst) = LDS(dst)	1A2D
LDS_CMP_XCHG_SPF_RET	(LDS(dst) == cmp) ? LDS(dst) = src : LDS(dst) = LDS(dst) [3]	1A2D
LDS_DEC	LDS(dst) = ((LDS(dst) == 0) || (LDS(dst) > src)) ? src : LDS(dst) - 1	1A1D
LDS_DEC_RET	LDS(dst) (LDS(dst) == 0) || (LDS(dst) > src) ? src : LDS(dst) - 1	1A1D
LDS_INC	(LDS(dst) >= src ? LDS(dst) = 0) : LDS(dst) ++	1A1D
LDS_INC_RET	(LDS(dst) >= src) ? LDS(dst) = 0 : LDS(dst) ++	1A1D
LDS_MAX_INT	LDS(dst) = max (LDS(dst) , src)	1A1D
LDS_MAX_INT_RET	LDS(dst) = max (LDS(dst) , src)	1A1D
LDS_MAX_UINT	LDS(dst) =max (LDS(dst) , src)	1A1D
LDS_MAX_UINT_RET	LDS(dst) = max (LDS(dst) , src)	1A1D
LDS_MIN_INT	LDS(dst) = min (LDS(dst) , src)	1A1D
LDS_MIN_INT_RET	LDS(dst) = min (LDS(dst) , src)	1A1D
LDS_MIN_UINT	LDS(dst) = min (LDS(dst) , src)	1A1D
LDS_MIN_UINT_RET	LDS(dst) = min (LDS(dst) , src)	1A1D
LDS_MSKOR	LDS(dst) = ((LDS(dst) &~msk) | src)	1A1D
LDS_MSKOR_RET	LDS(dst) = (LDS(dst) & ~msk) | src	1A2D
LDS_OR	LDS(dst) |= src	1A1D
LDS_OR_RET	LDS(dst) |= src	1A1D
LDS_READ_REL_RET	tmp = dst + LDS_IDX_OFFSET; OQA = LDS(dst), OQB = LDS(tmp) [4]	1A
LDS_READ_RET	OQA =LDS(dst) [5]	1A
LDS_READ2_RET	OQA = LDS(dst0), OQB = LDS(dst1) [6]	2A
LDS_READWRITE_RET	OQA = LDS(dst0), LDS(dst1 = data) [7]	2A1D
LDS_RSUB	LDS(dst) = src - LDS(dst)	1A1D
LDS_RSUB_RET	LDS(dst) = src - LDS(dst)	1A1D
LDS_SHORT_READ_RET	OQA = SignExtend(LDS(dst) [15:0]}[8]	1A
LDS_SHORT_WRITE	LDS(dst) = src[15:0]	1A1D
LDS_SUB	LDS(dst) = LDS(dst) - src	1A1D
LDS_SUB_RET	LDS(dst) = LDS(dst) - src	1A1D

Table 9.11 LDS Instructions for the LDS_OP Field

LDS Instruction	Description (C-Function Equivalent)	OP
LDS_UBYTE_READ_RET	OQA = {24'h0, LDS(dst)[7:0]}[9]	1A
LDS_USHORT_READ_RET	OQA = {16'h0, LDS(dst) [15:0]}[10]	1A
LDS_WRITE	LDS(dst) = src	1A1D
LDS_WRITE_REL	LDS(dst) = src0, LDS(tmp) = src1	1A2D
LDS_WRITE2	LDS(dst) = src0 , LDS(tmp) = src1	1A2D
LDS_XCHG_REL_RET	LDS(dst) = src0, LDS(dst + idx_offset) = src1)	1A2D
LDS_XCHG_RET	LDS(dst) = src	1A1D
LDS_XCHG2_RET	LDS(dst) = src0, LDS(dst + idx_offset*64) = src1	1A2D
LDS_XOR	LDS(dst) ^= src	1A1D
LDS_XOR_RET	LDS(dst) ^= src	1A1D

1. It is assumed that this description will be explained in the to-be-completed chapter on LDS.
2. It is assumed that this description will be explained in the to-be-completed chapter on LDS.
3. It is assumed that this description will be explained in the to-be-completed chapter on LDS.
4. It is assumed that this description will be explained in the to-be-completed chapter on LDS.
5. It is assumed that this description will be explained in the to-be-completed chapter on LDS.
6. It is assumed that this description will be explained in the to-be-completed chapter on LDS.
7. It is assumed that this description will be explained in the to-be-completed chapter on LDS.
8. It is assumed that this description will be explained in the to-be-completed chapter on LDS.
9. It is assumed that this description will be explained in the to-be-completed chapter on LDS.
10. It is assumed that this description will be explained in the to-be-completed chapter on LDS.

Chapter 10
Microcode Formats

This section specifies the microcode formats. The definitions can be used to simplify compilation by providing standard templates and enumeration names for the various instruction formats. Table 10.1 summarizes the microcode formats and their widths. The sections that follow provide details.

Table 10.1 Summary of Microcode Formats

Microcode Formats	Reference	Width (bits)	Function
Control Flow (CF) Instructions			
CF_WORD0 CF_GWS_WORD0 CF_WORD1	page 10-3 page 10-4 page 10-5	64	Implements general control-flow instructions.
CF_ALU_WORD0 CF_ALU_WORD1	page 10-8 page 10-9	64	Initiates ALU clauses.
CF_ALU_WORD0_EXT CF_ALU_WORD1_EXT	page 10-11 page 10-13		Extends the 64-bit dword to two 64-bit dwords.
CF_ALLOC_EXPORT_WORD0 CF_ALLOC_EXPORT_WORD0_RAT CF_ALLOC_EXPORT_WORD1_BUF CF_ALLOC_EXPORT_WORD1_SWIZ	page 10-14 page 10-16 page 10-19 page 10-21	64	Initiates and implements allocation and export instructions.
ALU Clause Instructions			
ALU_WORD0 ALU_WORD1_OP2 ALU_WORD1_OP3	page 10-23 page 10-26 page 10-32	64	Implements ALU instructions.
LDS Clause Instructions			
ALU_WORD0_LDS_IDX_OP ALU_WORD1_LDS_IDX_OP ALU_WORD1_LDS_DIRECT_LITERAL_LO ALU_WORD1_LDS_DIRECT_LITERAL_HI	page 10-36 page 10-39 page 10-43 page 10-44	64	Transfers data between LDS buffers and GPRs.
Instructions for a Fetch Through a Vertex Cache Clause			
VTX_WORD0 VTX_WORD1_GPR VTX_WORD1_SEM VTX_WORD2	page 10-45 page 10-47 page 10-50 page 10-52	96, padded to 128	Implements instructions for fetches through a vertex cache clause.
Instructions for a Fetch Through a Texture Cache Clause			
TEX_WORD0 TEX_WORD1 TEX_WORD2	page 10-54 page 10-57 page 10-58	96, padded to 128	Implements instructions for a fetch through a texture cache clause.

Table 10.1 Summary of Microcode Formats (Cont.)

Microcode Formats	Reference	Width (bits)	Function
Memory Read Instructions			
MEM_RD_WORD0 MEM_RD_WORD1 MEM_RD_WORD2	page 10-59 page 10-61 page 10-63	96, padded to 128	Implements memory read instructions.
Global Data-Share Read/Write Instructions			
MEM_GDS_WORD0 MEM_GDS_WORD1 MEM_GDS_WORD2	page 10-64 page 10-65 page 10-68	96, padded to 128	Implements global data share read and write instructions.

The field-definition tables that accompany the descriptions in the sections below use the following notation.

- *int(2)* — A two-bit field that specifies an integer value.
- *enum(7)* — A seven-bit field that specifies an enumerated set of values (in this case, a set of up to 2^7 values). The number of valid values can be less than the maximum.
- *VALID_PIXEL_MODE (VPM)* — Refers to the VALID_PIXEL_MODE field that is indicated in the accompanying format diagram by the abbreviated symbol VPM.

Unless otherwise stated, all fields are readable and writable (the CF_INST fields of the CF_ALLOC_EXPORT_WORD1_BUF or the CF_ALLOC_EXPORT_WORD1_SWIZ formats are the only exceptions). The default value of all fields is zero. Any bitfield not identified is assumed to be reserved.

10.1 Control Flow (CF) Instructions

Control flow (CF) instructions include:

- General control flow instructions (conditional jumps, loops, subroutines).
- Export instructions.
- Clause-initiation instructions for ALU, fetch through a texture cache clause, fetch through a vertex cache clause, global data share, local data share, and memory read clauses.

All CF microcode formats are 64 bits wide.

Control Flow Doubleword 0

Instructions **CF_WORD0**

Description This is the low-order (least-significant) doubleword in the 64-bit microcode-format pair formed by CF_WORD[0,1]. This format pair is the default format for CF instructions.

Opcode

Field Name	*Bits*	*Format*
ADDR	[23:0]	int(24)

- For clause instructions, bits [26:3] (the offset times 8, producing a quad-word-aligned value) of the beginning of the clause in memory to execute.
- For control flow instructions: bits [34:3] (the offset times 8 producing a quadword-aligned value) of the control flow address to jump to (instructions that can jump).

Offsets are relative to the byte address specified in the host-written PGM_START_* register. Texture and Vertex clauses must start on 16-byte aligned addresses.

JUMPTABLE_SEL	[26:24]	enum(3)

(JTS) Selects the source of the offset used for CF_INST_JUMPTABLE instructions. This has no effect on other instructions.

0 CF_JUMPTABLE_SEL_CONST_A: use element A of jumptable constant selected by CF_CONST.

1 CF_JUMPTABLE_SEL_CONST_B: use element B of jumptable constant selected by CF_CONST.

2 CF_JUMPTABLE_SEL_CONST_C: use element C of jumptable constant selected by CF_CONST.

3 CF_JUMPTABLE_SEL_CONST_D: use element D of jumptable constant selected by CF_CONST.

4 CF_JUMPTABLE_SEL_INDEX_0: use index0.

5 CF_JUMPTABLE_SEL_INDEX_1: use index1.

Reserved	[31:27]	Reserved

Related CF_WORD1

Control Flow Global Wave Sync Doubleword 0

Instructions **CF_GWS_WORD0**

Description This is the control flow instruction word 0 used by global wave sync instructions.

Opcode

Field Name	Bits	Format
VALUE	[9:0]	int(10)
	Counter load value for Barrier and Init opcodes.	
Reserved	[15:10]	Reserved.
RESOURCE	[20:16]	int(5)
	Index of resource. 0-15 in on- shader-engine (SE) chips, 0-31 in two-SE chips.	
Reserved	[24:21]	Reserved.
SIGN	25	int(1)
(S)	When set, resource is treated as signed -512..511 (as opposed to unsigned 0 .. 1023).	
VAL_INDEX_MODE	[27:26]	enum(2)
(VIM)	Override counter load value from instruction using index0/index1 registers. 0 GWS_INDEX_NONE: use source from instruction. 1 GWS_INDEX_0: use index0 as the value. 2 GWS_INDEX_1: use index1 as the value. 3 GWS_INDEX_MIX: use a combination of index0 and index1 as the value.	
RSRC_INDEX_MODE	[29:28]	enum(2)
(RIM)	Overrides the resource index from the instruction using index0/index1 registers. 0 CF_INDEX_NONE: do not index the constant buffer. 1 CF_INDEX_0: add index0 to the constant (CB#/T#/S#/UAV#) number. 2 CF_INDEX_1: add index1 to the constant (CB#/T#/S#/UAV#) number. 3 CF_INVALID: invalid.	
GWS_OPCODE	[31:30]	enum(2)
	Specifies the atomic operation to execute on a resource. 0 GWS_SEMA_V: semaphore V(). 1 GWS_SEMA_P: semaphore P(). 2 GWS_BARRIER: wavefront barrier. 3 GWS_INIT: resource initialization.	

Related CF_WORD1

Control Flow Doubleword 1

Instructions **CF_WORD1**

Description This is the high-order (most-significant) doubleword in the 64-bit microcode-format pair formed by CF_WORD[0,1]. This format pair is the default format for CF instructions.

Opcode

Field Name	Bits	Format
POP_COUNT (PC)	[2:0]	int(3)
	Specifies the number of entries to pop from the stack, in the range [0, 7]. Only used by certain CF instructions that pop the branch-loop stack. Can be zero to indicate non-pop operation.	
CF_CONST	[7:3]	int(5)
	Specifies the CF constant to use for flow control statements. For LOOP/ENDLOOP, this specifies the integer constant to use for the loop counter, loop index initializer, and increment. For instructions using the COND field, this specifies the index of the boolean constant. See Section 3.7.3, on page 3-18 and Section 3.7.4, on page 3-19.	
COND	[9:8]	enum(2)
	Specifies how to evaluate the condition test for each pixel. Not used by all instructions. Can reference CF_CONST. 0 CF_COND_ACTIVE: condition test passes for active pixels. (Non-branch-loop instructions can use only this setting CF_INST[29:23] below, values 4 through 20 and 24.) 1 CF_COND_FALSE: condition test fails for all pixels. 2 CF_COND_BOOL: condition test passes iff pixel is active and boolean referenced by CF_CONST is true. 3 CF_COND_NOT_BOOL: condition test passes iff pixel is active and boolean referenced by CF_CONST is false.	
COUNT	[15:10]	int(6)
	Number of instructions to execute in the clause, minus one (clause instructions only). This is interpreted as the number of instruction slots in the range [1,16]. For a CALL instruction, this specifies the amount to increment the call nesting counter when executing; the CALL is skipped if the current nesting depth + CALL_COUNT > 32. This field is interpreted in the range [0,31]. For EMIT, CUT, EMIT-CUT, bits [10] are the stream ID.	
Reserved	[19:16]	Reserved.
VALIX_PIXEL_MODE (VPM)	20	int(1)
	0 Execute the instructions in this clause as if invalid pixels are active. 1 Execute the instructions in this clause as if invalid pixels were inactive. This is the antonym of WHOLE_QUAD_MODE. Caution: VALID_PIXEL_MODE is not the default mode; this bit is cleared by default. Make the default for this bit to 0. Set this bit only in the PS stage.	
END_OF_PROGRAM (EOP)	21	int(1)
	0 This instruction is not the last instruction of the CF program. 1 This instruction is the last instruction of the CF program. Execution ends after this instruction is issued.	

Control Flow Doubleword 1 (Cont.)

CF_INST	[29:22]	enum(8)

Type of instruction to evaluate in CF. CF_INST must be set to one of the following values.

0 CF_INST_NOP: perform no operation.

1 CF_INST_TC: execute fetch clause through the texture cache. CF_COND=ACTIVE is required.

2 CF_INST_VC: execute fetch through a vertex cache clause (if it exists). CF_COND=ACTIVE is required.

3 CF_INST_GDS: execute a global data share clause. (GDS, tesselation factor [TF].) CF_COND=ACTIVE is required.

4 CF_INST_LOOP_START: execute DirectX9 loop start instruction (push onto stack if loop body executes).

5 CF_INST_LOOP_END: execute DirectX9 loop end instruction (pop stack if loop is finished).

6 CF_INST_LOOP_START_DX10: execute DirectX10 loop start instruction (push onto stack if loop body executes).

7 CF_INST_LOOP_START_NO_AL: same as LOOP_START but does not push the loop index (aL) onto the stack or update aL.

8 CF_INST_LOOP_CONTINUE: execute continue statement (jump to end of loop if all pixels ready to continue).

9 CF_INST_LOOP_BREAK: execute a break statement (pop stack if all pixels ready to break).

10 CF_INST_JUMP: execute jump statement (can be conditional).

11 CF_INST_PUSH: push current per-pixel active state onto the stack OR jump and pop if no items are active.

12 Reserved.

13 CF_INST_ELSE: execute else statement (can be conditional) OR jump if no items are active.

14 CF_INST_POP: pop current per-pixel state from the stack. Jump if no pixels are enabled prior to pop.

17:15 Reserved.

18 CF_INST_CALL: execute subroutine call instruction (push onto stack).

19 CF_INST_CALL_FS: call fetch kernel. The address to call is stored in a state register in SQ.

20 CF_INST_RETURN: execute subroutine return instruction (pop address stack). Pair only with CF_INST_CALL.

21 CF_INST_EMIT_VERTEX: signal that GS has finished exporting a vertex to memory. CF_COND=ACTIVE is required.

22 CF_INST_EMIT_CUT_VERTEX: emit a vertex and an end of primitive strip marker. The next emitted vertex starts a new primitive strip. CF_COND=ACTIVE is required.

23 CF_INST_CUT_VERTEX: emit an end of primitive strip marker. The next emitted vertex starts a new primitive strip. CF_COND=ACTIVE is required.

24 CF_INST_KILL: kill pixels that pass the condition test (can be conditional). Jump if all pixels are killed. CF_COND=ACTIVE is required.

Control Flow Doubleword 1 (Cont.)

		25 Reserved. 26 CF_INST_WAIT_ACK: wait for write ACKs or fetch-read-ACKs to return before proceeding. Wait if the number of outstanding ACKs is greater than the value in the ADDR field. When using a non-zero value, note that TC_ACK requests can return out-of-order with respect to VC_ACK requests. For optimal performance, never set BARRIER_BEFORE for this instruction. 27 CF_INST_TC_ACK: execute a fetch through a texture cache clause or execute a constant fetch clause, with ACK. CF_COND=ACTIVE is required. All previous TC/VC/GDS requests must have completed if this instruction is issued without BARRIER_BEFORE being set. 28 CF_INST_VC_ACK: execute a fetch through a vertex cache clause through the vertex cache (if it exists; otherwise, use texture cache), with ACK. CF_COND=ACTIVE is required. All previous TC/VC/GDS requests must have completed if this instruction is issued without BARRIER_BEFORE being set. 29 CF_INST_JUMPTABLE: execute a jump through a jump table. This instruction is followed by a series of up to 256 jump instructions forming the jump table. The index into the table comes from either a loop-constant or a GPR through the index registers. The instruction after the last jump table entry must be indicated by the ADDR field. If no pixels are enabled after the condition test, execution continues at this address. 30 CF_INST_GLOBAL_WAVE_SYNC: synchronize waves across the chip, including multiple SIMDs and multiple shader engines. 31 CF_INST_HALT: halt this thread execution. The only way to restart execution is to write this instruction using WAVE0_CF-INST[01] once the hardware is idle.
	WHOLE_QUAD_MODE (WQM)	30 int(1) Active pixels: 0 Do not execute this instruction as if all pixels are active and valid. 1 Execute this instruction as if all pixels are active and valid. This is the antonym of the VALID_PIXEL_MODE field. Set only one of these bits (WHOLE_QUAD_MODE or VALID_PIXEL_MODE) at a time; they are mutually exclusive.
	BARRIER (B)	31 int(1) Synchronization barrier: 0 This instruction can run in parallel with prior CF instructions. 1 All prior CF instructions must complete before this instruction executes.
Related	CF_WORD0	

Control Flow ALU Doubleword 0

Instructions **CF_ALU_WORD0**

Description This is the low-order (least-significant) doubleword in the 64-bit microcode-format pair formed by CF_ALU_WORD[0,1]. The instructions specified with this format are used to initiate ALU clauses. The ALU instructions that execute within an ALU clause are described in Section 10.2, on page 10-22.

Opcode

Field Name	Bits	Format
ADDR	[21:0]	int(22)
	Bits [24:3] of the byte offset (producing a quadword-aligned value) of the clause to execute. The offset is relative to the byte address specified by PGM_START_* register.	
KCACHE_BANK0 (KB0)	[25:22]	int(4)
	Bank (constant buffer number) for first set of locked cache lines.	
KCACHE_BANK1 (KB1)	[29:26]	int(4)
	Bank (constant buffer number) for second set of locked cache lines.	
KCACHE_MODE0 (KM0)	[31:30]	enum(2)
	Mode for first set of locked cache lines. 0 CF_KCACHE_NOP: do not lock any cache lines. 1 CF_KCACHE_LOCK_1: lock cache line KCACHE_BANK, ADDR. 2 CF_KCACHE_LOCK_2: lock cache lines KCACHE_BANK, ADDR and KCACHE_BANK, ADDR+1. 3 CF_KCACHE_LOCK_LOOP_INDEX: lock cache lines KCACHE_BANK, LOOP/16+ADDR and KCACHE_BANK, LOOP/16+ADDR+1, where LOOP is the current loop index (aL).	

Related CF_ALU_WORD1

Control Flow ALU Doubleword 1

Instructions **CF_ALU_WORD1**

Description This is the high-order (most-significant) doubleword in the 64-bit microcode-format pair formed by CF_ALU_WORD[0,1]. The instructions specified with this format are used to initiate ALU clauses. The instructions that execute within an ALU clause are described in Section 10.2, on page 10-22.

Opcode

Field Name	*Bits*	*Format*
KCACHE_MODE1 (KM1)	[1:0]	enum(2)
	Mode for second set of locked cache lines: 0 CF_KCACHE_NOP: do not lock any cache lines. 1 CF_KCACHE_LOCK_1: lock cache line KCACHE_BANK, ADDR. 2 CF_KCACHE_LOCK_2: lock cache lines KCACHE_BANK, ADDR+1. 3 CF_KCACHE_LOCK_LOOP_INDEX: lock cache lines KCACHE_BANK, LOOP/16+ADDR and KCACHE_BANK[0.1], LOOP/16+ADDR+1, where LOOP is current loop index (aL).	
KCACHE_ADDR0	[9:2]	int(8)
	Constant buffer address for first set of locked cache lines. In units of cache lines, where a line holds sixteen 128-bit constants (byte addr[15:8]).	
KCACHE_ADDR1	[17:10]	int(8)
	Constant buffer address for second set of locked cache lines.	
COUNT	[24:18]	int(7)
	Number of 64-bit instruction slots in the range [1,128] to execute in the clause, minus one.	
ALT_CONST (AC)	25	int(1)
	0 This ALU clause does not use constants from an alternate thread type. 1 This ALU clause uses constants from an alternate thread type: PS->VS, VS->GS, GS->VS, ES->GS. Note that ES and VS share constants. Has no effect on HS, LS, and CS.	

Control Flow ALU Doubleword 1 (Cont.)

	CF_INST	[29:26]	enum(4)

Type of ALU instruction to evaluate in CF. For this encoding, CF_INST must be one of the following values.

8 CF_INST_ALU: each PRED_SET* instruction updates the active state but does not update the stack.

9 CF_INST_ALU_PUSH_BEFORE: execute CF_PUSH, then CF_INST_ALU.

10 CF_INST_ALU_POP_AFTER: execute CF_INST_ALU, then CF_INST_POP.

11 CF_INST_ALU_POP2_AFTER: execute CF_INST_ALU_POP2, then CF_INST_POP twice.

12 CF_INST_ALU_EXTENDED: ALU clause instruction extension for indexed constant buffers and four constant buffers per clause. This is the first half of the ALU instruction pair. Defines constant buffer 2 and 3, and index-select for all four constant buffers.

13 CF_INST_ALU_CONTINUE: each PRED_SET* causes a continue operation on the masked pixels. This is equivalent to the following instruction sequence: CF_INST_PUSH, CF_INST_ALU, CF_INST_ELSE, CF_INST_CONTINUE, CF_POP. This disables the pixels for the rest of this loop iterations, but enables them again for the following loop.

14 CF_INST_ALU_BREAK: each PRED_SET* causes a break operation on the masked pixels. This is equivalent to the following instruction sequence: CF_INST_PUSH, CF_INST_ALU, CF_INST_ELSE, CF_INST_CONTINUE, CF_POP. This disables the pixels for the rest of this loop iteration, as well as for all subsequent loops.

15 CF_INST_ALU_ELSE_AFTER: execute CF_INST_ALU, then CF_INST_ELSE.

	WHOLE_QUAD_MODE (WQM)	30	int(1)

0 Do not execute this clause as if all pixels are active and valid.

1 Execute this clause as if all pixels are active and valid.

This is the antonym of the VALID_PIXEL_MODE field. Set only one of these bits (WHOLE_QUAD_MODE or VALID_PIXEL_MODE) at a time; they are mutually exclusive. Set this only in the PS stage.

	BARRIER (B)	31	int(1)

Synchronization barrier.

0 This instruction can run in parallel with prior instructions.

1 All prior instructions must complete before this instruction executes.

Related CF_ALU_WORD0

Control Flow ALU Doubleword 0 Extended

Instructions **CF_ALU_WORD0_EXT**

Description This extends the low-order (least-significant) doubleword in the 64-bit microcode-format pair formed by CF_ALU_WORD0, so the clause consists of four dwords: EXT1, EXT0, CF_ALU_WORD1, and CF_ALU_WORD0. The ALU instructions that execute within an ALU clause are described in Section 10.2, on page 10-22.

Opcode

Field Name	Bits	Format
Reserved	[3:0]	Reserved.
KCACHE_BANK_INDEX_MODE0 (KBIM0)	[5:4]	enum(2) Bank relative offset select. Add the indicated offset to the constant buffer bank number in KCACHE_BANK0. Indexed locks of banks 14 and 15 are ignored. 0 CF_INDEX_NONE: do not index the constant buffer. 1 CF_INDEX_0: add index0 to the constant (CB#/T#/S#/UAV#) number. 2 CF_INDEX_1: add index1 to the constant (CB#/T#/S#/UAV#) number. 3 CF_INVALID: invalid.
KCACHE_BANK_INDEX_MODE1 (KBIM1)	[7:6]	enum(2) Bank relative offset select. Add the indicated offset to the constant buffer bank number in KCACHE_BANK1. Indexed locks of banks 14 and 15 are ignored. 0 CF_INDEX_NONE: do not index the constant buffer. 1 CF_INDEX_0: add index0 to the constant (CB#/T#/S#/UAV#) number. 2 CF_INDEX_1: add index1 to the constant (CB#/T#/S#/UAV#) number. 3 CF_INVALID: invalid.
KCACHE_BANK_INDEX_MODE2 (KBIM2)	[9:8]	enum(2) Bank relative offset select. Add the indicated offset to the constant buffer bank number in KCACHE_BANK2. Indexed locks of banks 14 and 15 are ignored. 0 CF_INDEX_NONE: do not index the constant buffer. 1 CF_INDEX_0: add index0 to the constant (CB#/T#/S#/UAV#) number. 2 CF_INDEX_1: add index1 to the constant (CB#/T#/S#/UAV#) number. 3 CF_INVALID: invalid.
KCACHE_BANK_INDEX_MODE3 (KBIM3)	[11:10]	enum(2) Bank relative offset select. Add the indicated offset to the constant buffer bank number in KCACHE_BANK3. Indexed locks of banks 14 and 15 are ignored. 0 CF_INDEX_NONE: do not index the constant buffer. 1 CF_INDEX_0: add index0 to the constant (CB#/T#/S#/UAV#) number. 2 CF_INDEX_1: add index1 to the constant (CB#/T#/S#/UAV#) number. 3 CF_INVALID: invalid.
Reserved	[21:12]	Reserved.
KCACHE_BANK2 (KB2)	[25:22]	int(4) Bank (constant buffer number) for third set of locked cache lines.
KCACHE_BANK3 (KB3)	[29:26]	int(4) Bank (constant buffer number) for fourth set of locked cache lines.

Control Flow ALU Doubleword 0 Extended (Cont.)

	KCACHE_MODE2 (KM2)	[31:30]	enum(2) Mode for third set of locked cache lines. 0 CF_KCACHE_NOP: do not lock any cache lines. 1 CF_KCACHE_LOCK_1: lock cache lines [bank][addr]. 2 CF_KCACHE_LOCK_2: lock cache lines [bank][addr] and [bank][addr+1]. 3 CF_KCACHE_LOCK_LOOP_INDEX: lock cache lines [bank][loop/16+addr] and [bank][loop/16+addr+1], where loop is current loop index.
Related	CF_ALU_WORD1_EXT		

Control Flow ALU Doubleword 1 Extended

Instructions **CF_ALU_WORD1_EXT**

Description This extends the high-order (most-significant) doubleword in the 64-bit microcode-format pair formed by CF_ALU_WORD1, so the clause consists of four dwords: EXT1, EXT0, CF_ALU_WORD1, and CF_ALU_WORD0. The ALU instructions that execute within an ALU clause are described in Section 10.2, on page 10-22.

Opcode

Field Name	Bits	Format
KCACHE_MODE3	[1:0]	enum(2)
(KM3)	Mode for fourth set of locked cache lines. 0 CF_KCACHE_NOP: do not lock any cache lines. 1 CF_KCACHE_LOCK_1: lock cache lines [bank][addr]. 2 CF_KCACHE_LOCK_2: lock cache lines [bank][addr] and [bank][addr+1]. 3 CF_KCACHE_LOCK_LOOP_INDEX: lock cache lines [bank][loop/16+addr] and [bank][loop/16+addr+1], where loop is current loop index.	
KCACHE_ADDR2	[9:2]	int(8)
	Bank (constant buffer number) for third set of locked cache lines.	
KCACHE_ADDR3	[17:10]	int(8)
	Bank (constant buffer number) for fourth set of locked cache lines.	
Reserved	[25:18]	Reserved.
CF_INST	[29:26]	enum(4)
	Type of ALU instruction to evaluate in CF. Must be CF_INST_ALU_EXTENDED. 8 CF_INST_ALU: each PRED_SET updates the active state but does not update the stack. 9 CF_INST_ALU_PUSH_BEFORE: execute CF_PUSH, then CF_INST_ALU. 10 CF_INST_ALU_POP_AFTER: execute CF_INST_ALU, then CF_INST_POP. 11 CF_INST_ALU_POP2_AFTER: execute CF_INST_ALU, then CF_INST_POP twice. 12 CF_INST_ALU_EXTENDED: ALU clause instruction extension for indexed constant buffers and four constant buffers per clause. This CF is the first half of the ALU instruction pair. Defines constant buffers 2 and 3, and index-select for all four constant buffers. 13 CF_INST_ALU_CONTINUE: each PRED_SET causes a continue operation on the masked pixels. Equivalent to CF_INST_PUSH, CF_INST_ALU, CF_INST_ELSE, CF_INST_CONTINUE, CF_POP. 14 CF_INST_ALU_BREAK: each PRED_SET causes a break operation on the masked pixels. Equivalent to CF_INST_PUSH, CF_INST_ALU, CF_INST_ELSE, CF_INST_CONTINUE, CF_POP. 15 CF_INST_ALU_ELSE_AFTER: execute CF_INST_ALU, then CF_INST_ELSE.	
Reserved	30	Reserved.
BARRIER	31	int(1)
	0 This instruction/clause can run in parallel with prior instructions. 1 All prior CF instructions/clauses must complete before this instruction/clause executes.	

Related CF_ALU_WORD0_EXT

Control Flow Allocate, Import, or Export Doubleword 0

Instructions **CF_ALLOC_EXPORT_WORD0**

Description This is the low-order (least-significant) doubleword in the 64-bit microcode-format pair formed by CF_ALLOC_EXPORT_WORD0 and CF_ALLOC_EXPORT_WORD1_{BUF, SWIZ}. It is used to reserve storage space in an input or output buffer, write data from GPRs into an output buffer, or read data from an input buffer into GPRs. Each instruction using this format pair can use either the BUF or the SWIZ version of the second doubleword—all instructions have both BUF and SWIZ versions. The instructions specified with this format pair are used to initiate allocation, import, or export clauses.

Opcode

Field Name	*Bits*	*Format*
ARRAY_BASE	[12:0]	int(13)

- For scratch or reduction input or output, this is the base address of the array in multiples of four doublewords [0,32764].
- For stream or ring output, this is the base address of the array in multiples of one doubleword [0,8191].
- For pixel or Z output, this is the index of the first export (framebuffer 0..7; computed Z: 61).
- For parameter output, this is the parameter index of the first export [0,31].
- For position output, this is the position index of the first export [60,63].

TYPE [14:13] enum(2)

Type of allocation, import, or export. In the types below, the first value (PIXEL, POS, PARAM) is used with the CF_INST_EXPORT* instruction; the second value (WRITE, WRITE_IND, WRITE_ACK, and WRITE_IND_ACK) is used with the CF_INST_MEM* instruction:

0 EXPORT_PIXEL: write pixel. Available only for Pixel Shader (PS).
EXPORT_WRITE: write to memory buffer.

1 EXPORT_POS: write position. Available only to Vertex Shader (VS).
EXPORT_WRITE_IND: write to memory buffer, use offset in INDEX_GPR.

2 EXPORT_PARAM: write parameter cache. Available only to Vertex Shader (VS).
EXPORT_WRITE_ACK: write to memory buffer, request and ACK when write is in memory. For unordered access views (UAVs), ACK guarantees that the return value has been written to memory.

3 Unused for SX exports.
EXPORT_WRITE_IND_ACK: write to memory buffer with offset in INDEX_GPR ; get an ACK when done. For unordered access views (UAVs), ACK guarantees that the return value has been written to memory.

RW_GPR [21:15] int(7)

GPR register from which to read data.

RW_REL (RR) 22 enum(1)

Indicates whether GPR is an absolute address, or relative to the loop index (aL).

0 Absolute: no relative addressing.

1 Relative: add current loop index (aL) value to this address.

INDEX_GPR [29:23] int(7)

For any indexed import or export, this GPR contains an index that is used to determine the address of the first export. The index is multiplied by (ELEM_SIZE + 1). Only the X element is used (other elements ignored, no swizzle allowed).

Control Flow Allocate, Import, or Export Doubleword 0 (Cont.)

	ELEM_SIZE (ES)	[31:30]	int(2)
		Number of doublewords per array element, minus one. This field is interpreted as a value in [1,2,4] (3 is not supported). The value from INDEX_GPR and the loop index (aL) counter are multiplied by this factor, if applicable. Also, BURST_COUNT is multiplied by this factor for CF_INST_MEM*. This field is ignored for CF_INST_EXPORT*. Normally, ELEMSIZE = four doublewords for scratch buffers, one doubleword for other buffer types.	
Related	CF_ALLOC_EXPORT_WORD1_BUF		
	CF_ALLOC_EXPORT_WORD1_SWIZ		

Control Flow Allocate, Import, or Export Doubleword 0 Unordered Access View (UAV)

Instructions **CF_ALLOC_EXPORT_WORD0_RAT**

Description This is the least-significant doubleword in the 64-bit microcode format pair formed by CF_ALLOC_EXPORT_WORD0_RAT and CF_ALLOC_EXPORT_WORD1_BUF. It describes a write to a unordered access view (UAV) buffer. These exports allow simple writes to the UAV buffer, or atomic reduction operations that combine data exported from GPRs with data already in the buffer.

Opcode

Field Name	*Bits*	*Format*
RAT_ID	[3:0]	int(4)
	Unordered access view (UAV) ID.	
RAT_INST	[9:4]	enum(6)

UAV instruction.

0 EXPORT_RAT_INST_NOP: no operation.

1 EXPORT_RAT_INST_STORE_TYPED: destination = source. Replace with format conversion (any resource format allowed). This is the only cached UAV opcode that can write more than one dword (up to four).

2 EXPORT_RAT_INST_STORE_RAW: destination = source (no flushing of denorms). This is the only permitted opcode for CACHELESS UAVs. It can write up to two dwords from the x and y elements for CACHELESS targets, but only one dword for cached targets.

3 EXPORT_RAT_INST_STORE_RAW_FDENORM: destination = source. Flush denorms to zero.

4 EXPORT_RAT_INST_CMPXCHG_INT: dst = (cmp == dst) ? src:dst. Simple bitwise compare.

5 EXPORT_RAT_INST_CMPXCHG_FLT: dst = (cmp == dst) ? src:dst. Floating point compare with denorms.

6 EXPORT_RAT_INST_CMPXCHG_FDENORM: dst = (cmp == dst) ? src:dst. Flush denorms to zero, float compare.

7 EXPORT_RAT_INST_ADD: dest = src + dst. Non-saturating integer add.

8 EXPORT_RAT_INST_SUB: dst = dst - src. Non-saturating integer sub.

9 EXPORT_RAT_INST_RSUB: dst = src - dst. Non-saturating reverse subtract.

10 EXPORT_RAT_INST_MIN_INT: dst = (src < dst) ? src:dst. Signed.

11 EXPORT_RAT_INST_MIN_UINT: dst = (src < dst) ? src:dst. Unsigned.

12 EXPORT_RAT_INST_MAX_INT: dst = (src > dst) ? src : dst. Signed.

13 EXPORT_RAT_INST_MAX_UINT: dst = (src > dst) ? src : dst. Unsigned.

14 EXPORT_RAT_INST_AND: dst = dst & src. Bitwise.

15 EXPORT_RAT_INST_OR: dst = dst | src. Bitwise.

16 EXPORT_RAT_INST_XOR: dst = dst ^ src. Bitwise.

17 EXPORT_RAT_INST_MSKOR: dst = (dst & ~mask) | src.

18 EXPORT_RAT_INST_INC_UINT: dst = (dst >= src) ? 0 : dst+1.

19 EXPORT_RAT_INST_DEC_UINT: dst = ((dst==0 | (dst > src)) ? src : dst-1.

31:20 Reserved.

32 EXPORT_RAT_INST_NOP_RTN: Internal use by SX only (flush+ack with no opcode). Return dword.

33 Reserved.

34 EXPORT_RAT_INST_XCHG_RTN: dst = src (no flushing of denorms). Return dword.

35 EXPORT_RAT_INST_XCHG_FDENORM_RTN: dst = src (flush denorms to zero). Return float.

36 EXPORT_RAT_INST_CMPXCHG_INT_RTN: dst = (cmp == dst) ? src : dst. simple bitwise compare. Return dword.

Control Flow Allocate, Import, or Export Doubleword 0 Unordered Access View (UAV) (Cont.)

	37	EXPORT_RAT_INST_CMPXCHG_FLT_RTN: dst = (cmp == dst) ? src : dst. Floating point compare with denorms. Return float.
	38	EXPORT_RAT_INST_CMPXCHG_FDENORM_RTN: dst = (cmp == dst) ? src : dst. Flush denorms to zero, float compare. Return float.
	39	EXPORT_RAT_INST_ADD_RTN: dst = src + dst. Non-saturating integer add. Return dword.
	40	EXPORT_RAT_INST_SUB_RTN: dst = dst - src. Non-saturating integer sub. Return dword.
	41	EXPORT_RAT_INST_RSUB_RTN: dst = src - dst. Non-saturating reverse subtract. Return dword.
	42	EXPORT_RAT_INST_MIN_INT_RTN: dst = (src < dst) ? src : dst. signed. Return dword.
	43	XPORT_RAT_INST_MIN_UINT_RTN: dst = (src < dst) ? src : dst. Unsigned. Return dword.
	44	EXPORT_RAT_INST_MAX_INT_RTN: dst = (src > dst) ? src : dst. Signed. Return dword.
	45	EXPORT_RAT_INST_MAX_UINT_RTN: dst = (src > dst) ? src : dst. Unsigned return dword.
	46	EXPORT_RAT_INST_AND_RTN: dst = dst & src. Bitwise. Return dword.
	47	EXPORT_RAT_INST_OR_RTN: dst = dst | src. Bitwise. Return dword.
	48	EXPORT_RAT_INST_XOR_RTN: dst = dst ^ src. Bitwise. Return dword.
	49	EXPORT_RAT_INST_MSKOR_RTN: dst = (dst & ~mask) | src. Return dword.
	50	EXPORT_RAT_INST_INC_UINT_RTN: dst = (dst >= src) ? 0 : dst+1. Return uint.
	51	EXPORT_RAT_INST_DEC_UINT_RTN: dst = ((dst==0 | (dst > src)) ? src : dst-1. Return uint.
Reserved	10	Reserved.
RAT_INDEX_MODE (RIM)	[12:11]	enum(2)
	UAV index select: non-indexed, add idx0 or idx1 to RAT_ID.	
	0	CF_INDEX_NONE: Do not index the constant buffer.
	1	CF_INDEX_0: Add index0 to the constant (CB#/T#/S#/UAV#) number.
	2	CF_INDEX_1: Add index1 to the constant (CB#/T#/S#/UAV#) number.
	3	CF_INVALID: Invalid.
TYPE	[14:13]	enum(2)
	Type of allocation/export.	
	0	EXPORT_PIXEL: Write the pixel. EXPORT_WRITE: Write to the memory buffer.
	1	EXPORT_POS: Write the position. EXPORT_WRITE_IND: write to memory buffer, use offset in INDEX_GPR.
	2	EXPORT_PARAM: write parameter cache. EXPORT_WRITE_ACK: write to memory buffer, request an ACK when write is committed to memory. For UAV, ACK guarantees return value has been written to memory.
	3	Unused for SX exports.EXPORT_WRITE_IND_ACK: write to memory buffer with offset in INDEX_GPR, get an ACK when done. For UAV, ACK guarantees return value has been written to memory.
RW_GPR	[21:15]	int(7)
	GPR register from which to read data. Depending on the RAT_INST opcode, this GPR contains either: • up to four dwords of data to write, or • a dword of source data in the X element, a dword return address in the Y element, and a dword of compare data in the W element.	

Control Flow Allocate, Import, or Export Doubleword 0 Unordered Access View (UAV) (Cont.)

RW_REL (RR)	22	enum(1)
	Indicates whether GPR is an absolute address, or relative to the loop index (aL). 0 Absolute: no relative addressing. 1 Relative: add current loop index (aL) value to this address.	
INDEX_GPR	[29:23]	int(7)
	Select the GPR that holds the buffer address coordinates. The index is multiplied by (ELEM_SIZE + 1). The X, Y, and Z components contain the address within the 1-d surface, 2-d surface, 3-d surface, or it is 2-d slice address, depending on the format of the UAV surface.	
ELEM_SIZE (ES)	[31:30]	int(2)
	Number of doublewords per array element, minus one. This field is interpreted as a value in [1,2,4] (3 is not supported). The value from INDEX_GPR and the loop index (aL) counter are multiplied by this factor, if applicable. Also, BURST_COUNT is multiplied by this factor for CF_INST_MEM*. This field is ignored for CF_INST_EXPORT*. Normally, ELEMSIZE = four doublewords for scratch buffers, one doubleword for other buffer types.	

Related

CF_ALLOC_EXPORT_WORD1_BUF

CF_ALLOC_EXPORT_WORD1_SWIZ

Control Flow Allocate, Import, or Export Doubleword 1 Buffer

Instructions **CF_ALLOC_EXPORT_WORD1_BUF**

Description Word 1 of the control flow instruction. This subencoding is used by allocations/exports for all input/outputs to scratch, ring, and stream buffers.

Opcode

Field Name	Bits	Format
ARRAY_SIZE	[11:0]	
	Array size (elem-size units). MEM_WR_SCRATCH: Represents values [1,4096] when ELEMSIZE = 0, [4,16384] when ELEMSIZE = 3. MEM_WR_SCATTER: unused (no effect). RAT_CACHELESS export, array_size[7:0] carries the dword stride for burst exports. Stride is 1..256 dwords.	
COMP_MASK	[15:12]	int(4)
	XYZW component mask (X is the LSB). Write the component iff the corresponding bit is 1. User must enable all components that contain address/data for the operation. For UAV store-raw, set to 0x1; for other UAVs, set to 0xF.	
BURST_COUNT	[19:16]	int(4)
	Number of MRTs, positions, parameters, or logical export values to allocate and/or export, minus one. This field is interpreted as a value in [1..16].	
VALID_PIXEL_MODE (VPM)	20	int(1)
	If set, do not export data for invalid pixels. Caution: this is not the 'default' mode; set this bit to 0 by default. Note that Pix/Pos/PC exports use the valid mask and active mask, and mem-exports use the active mask only. Set this only in the PS stage.	
END_OF_PROGRAM (EOP)	21	int(1)
	If set, this instruction is the last instruction of the CF program. Execution ends after this instruction is issued.	
CF_INST	[29:22]	enum(8)
	Type of instruction to execute in CF. The value must be one of the allocation/export instructions listed below. 64 CF_INST_MEM_STREAM0_BUF0: perform a memory write on the stream 0, buffer 0. 65 CF_INST_MEM_STREAM0_BUF1: perform a memory write on the stream 0, buffer 1. 66 CF_INST_MEM_STREAM0_BUF2: perform a memory write on the stream 0, buffer 2. 67 CF_INST_MEM_STREAM0_BUF3: perform a memory write on the stream 0, buffer 3. 68 CF_INST_MEM_STREAM1_BUF0: perform a memory write on the stream 1, buffer 0. 69 CF_INST_MEM_STREAM1_BUF1: perform a memory write on the stream 1, buffer 1. 70 CF_INST_MEM_STREAM1_BUF2: perform a memory write on the stream 1, buffer 2. 71 CF_INST_MEM_STREAM1_BUF3: perform a memory write on the stream 1, buffer 3. 72 CF_INST_MEM_STREAM2_BUF0: perform a memory write on the stream 2, buffer 0.	

Control Flow Allocate, Import, or Export Doubleword 1 Buffer (Cont.)

		73	CF_INST_MEM_STREAM2_BUF1: perform a memory write on the stream 2, buffer 1.
		74	CF_INST_MEM_STREAM2_BUF2: perform a memory write on the stream 2, buffer 2.
		75	CF_INST_MEM_STREAM2_BUF3: perform a memory write on the stream 2, buffer 3.
		76	CF_INST_MEM_STREAM3_BUF0: perform a memory write on the stream 3, buffer 0.
		77	CF_INST_MEM_STREAM3_BUF1: perform a memory write on the stream 3, buffer 1.
		78	CF_INST_MEM_STREAM3_BUF2: perform a memory write on the stream 3, buffer 2.
		79	CF_INST_MEM_STREAM3_BUF3: perform a memory write on the stream 3, buffer 3.
		80	CF_INST_MEM_WR_SCRATCH: perform a memory write on the scratch buffer.
		82	CF_INST_MEM_RING: perform a memory write on the ring buffer.
		83	Reserved.
		84	Reserved.
		85	CF_INST_MEM_EXPORT: perform a memory write on the shared buffer.
		86	CF_INST_MEM_RAT: export to a Random Access Target - full functionality (via CB).
		87	CF_INST_MEM_RAT_CACHELESS: export to a Random Access Target without caching - reduced functionality (via DB).
		88	CF_INST_MEM_RING1: write to ring 1 (currently only applies to GSVS ring).
		89	CF_INST_MEM_RING2: write to ring 2 (currently only applies to GSVS ring).
		90	CF_INST_MEM_RING3: write to ring 3 (currently only applies to GSVS ring).
		91	CF_INST_MEM_EXPORT_COMBINED: Memory export (scatter), for single dword exports only. Combined Address and Data in one export (data = x, data = y, address = w). Must be non-indexed-write, and no burst-writes.
		92	CF_INST_MEM_RAT_COMBINED_CACHELESS: export to a Random Access Target - reduced functionality (via DB). Combined Address and Data in one export (data = x, data = y; address = w). Must be non-indexed-write, and no burst-writes.
MARK (M)	30	int(1)	
	Mark memory write to be acknowledged with the next write-ack. Only applies to memory writes (scratch, scatter, etc.), not pixel/position/parameter.		
BARRIER (B)	31	int(1)	
	Synchronization barrier.		
	0	This instruction can run in parallel with prior instructions.	
	1	All prior instructions must complete before this instruction executes.	

Related CF_ALLOC_EXPORT_WORD0, CF_ALLOC_EXPORT_WORD0_RAT
CF_ALLOC_EXPORT_WORD1_SWIZ

Control Flow Allocate, Import, or Export Doubleword 1 Swizzle

Instructions **CF_ALLOC_EXPORT_WORD1_SWIZ**

Description Word 1 of the control flow instruction. This subencoding is used by allocations/exports for PIXEL, POS, and PARAM.

Opcode

Field Name	*Bits*	*Format*
SEL_X	[2:0]	enum(3)
SEL_Y	[5:3]	enum(3)
SEL_Z	[8:6]	enum(3)
SEL_W	[11:9]	enum(3)
	Specifies the source for each element of the import or export. 0 SEL_X: use X element. 1 SEL_Y: use Y element. 2 SEL_Z: use Z element. 3 SEL_W: use W element. 4 SEL_0: use constant 0.0. 5 SEL_1: use constant 1.0. 6 Reserved. 7 SEL_MASK: mask this element.	
Reserved	[15:12]	Reserved.
BURST_COUNT	[19:16]	int(4)
	Number of MRTs, positions, parameters, or logical export values to allocate and/or export, minus one. This field is interpreted as a value in [1..16].	
VALID_PIXEL_MODE (VPM)	20	int(1)
	If set, do not export data for invalid pixels. Caution: this is not the 'default' mode; set this bit to 0 by default. Note that Pix/Pos/PC exports use the valid mask and active mask, and mem-exports use the active mask only. Set this only in the PS stage.	
END_OF_PROGRAM (EOP)	21	int(1)
	If set, this instruction is the last instruction of the CF program. Execution ends after this instruction is issued.	
CF_INST	[29:22]	enum(8)
	Type of instruction to execute in CF. The value must be one of the allocation/export instructions listed below. All other values are reserved. 83 CF_INST_EXPORT: export only (not last). Used for PIXEL, POS, PARAM exports. 84 CF_INST_EXPORT_DONE: export only (last export). Used for PIXEL, POS, PARAM exports.	
MARK (M)	30	int(1)
	Mark memory write to be acknowledged with the next write-ack. Only applies to memory writes (scratch, scatter, etc.), not pixel/position/parameter.	

Control Flow Allocate, Import, or Export Doubleword 1 Swizzle (Cont.)

	BARRIER (B)	31	int(1)
		Synchronization barrier.	
		0	This instruction can run in parallel with prior instructions.
		1	All prior instructions must complete before this instruction executes.
Related	CF_ALLOC_EXPORT_WORD0, CF_ALLOC_EXPORT_WORD0_RAT CF_ALLOC_EXPORT_WORD1_BUF		

10.2 ALU Instructions

ALU clauses are initiated using the CF_ALU_WORD[0,1] format pair, described in Section 10.1, on page 10-2. After the clause is initiated, the instructions below can be issued. ALU instructions are used to build ALU instruction groups, as described in Section 4.3, on page 4-3. All ALU microcode formats are 64 bits wide.

ALU Doubleword 0

Instructions **ALU_WORD0**

Description This is the low-order (least-significant) doubleword in the 64-bit microcode-format pair formed by ALU_WORD0 and ALU_WORD1_{OP2, OP3}. Each instruction using this format pair has either an OP2 or an OP3 version (not both). Source for operands src0, src1.

Opcode

Field Name	*Bits*	*Format*
SRC0_SEL	[8:0]	enum(9)
SRC1_SEL	[21:13]	enum(9)

Location or value of this source operand.

[127:0] Value in GPR[127:0].
[159:128] Kcache constants in bank 0.
[191:160] Kcache constants in bank 1.
[255:192] inline constant values.
[287:256] Kcache constants in bank 2.
[319:288] Kcache constants in bank 3.

219 ALU_SRC_LDS_OQ_A: Use contents of LDS Output Queue A and leave it on the queue.
220 ALU_SRC_LDS_OQ_B: Use contents of LDS Output Queue B and leave it on the queue.
221 ALU_SRC_LDS_OQ_A_POP: Use contents of LDS Output Queue A, and pop both the A and B queues at the end of the instruction group(xyzwt).
222 ALU_SRC_LDS_OQ_B_POP: Use contents of LDS Output Queue B, and pop both the A and B queues at the end of the instruction group(xyzwt).
223 ALU_SRC_LDS_DIRECT_A: Direct read of LDS on the A cycle. Address is defined in literal constant-0 (xy).
224 ALU_SRC_LDS_DIRECT_B: Direct read of LDS on the B cycle. Address is defined in literal constant-0 (xy).
227 ALU_SRC_TIME_HI: Upper 32 bits of 64-bit clock counter.
228 ALU_SRC_TIME_LO: Lower 32 bits of 64-bit clock counter.
229 ALU_SRC_MASK_HI: Upper 32bits of active mask.
230 ALU_SRC_MASK_LO: Lower 32bits of active mask.
231 ALU_SRC_HW_WAVE_ID: Hardware wave ID (int).
232 ALU_SRC_SIMD_ID: Simd id (int).
233 ALU_SRC_SE_ID: Shader engine ID (int).
234 ALU_SRC_HW_THREADGRP_ID: Hardware thread group ID (int) within simd. CS and HS only.
235 ALU_SRC_WAVE_ID_IN_GRP: Wave id within thread group (int). CS and HS only.
236 ALU_SRC_NUM_THREADGRP_WAVES: Number of waves in thread group (int). CS and HS only, must barrier before using.
237 ALU_SRC_HW_ALU_ODD: Is this clause executing on the even(0) or odd(1) path (int)
238 ALU_SRC_LOOP_IDX: Current value of the loop index (int)
240 ALU_SRC_PARAM_BASE_ADDR: Parameter cache base (LDS_ALLOC_PS), (int).

ALU Doubleword 0 (Cont.)

	241	ALU_SRC_NEW_PRIM_MASK: Bit mask. One bit per quad; if set, it indicates that this quad starts a new primitive. Mask omits bit for first quad, since it always begins a new primitive. For example, in a vectorsize 64 system, this mask is {[15:1],1'b1}.242 ALU_SRC_PRIM_MASK_HI: Upper 32 bits of 64-bit expansion of NEW_PRIM_MASK. Used for general parameter interp. See SQ-arch spec for details.
	243	ALU_SRC_PRIM_MASK_LO: Lower 32 bits of 64-bit expansion of NEW_PRIM_MASK. Used for general parameter interp. See SQ-arch spec for details.
	244	ALU_SRC_1_DBL_L: special constant 1.0 double-float, LSW.
	245	ALU_SRC_1_DBL_M: special constant 1.0 double-float, MSW.
	246	ALU_SRC_0_5_DBL_L: special constant 0.5 double-float, LSW.
	247	ALU_SRC_0_5_DBL_M: special constant 0.5 double-float, MSW.
	248	ALU_SRC_0: the constant 0.0.
	249	ALU_SRC_1: the constant 1.0 float.
	250	ALU_SRC_1_INT: the constant 1 integer.
	251	ALU_SRC_M_1_INT: the constant -1 integer.
	252	ALU_SRC_0_5: the constant 0.5 float.
	253	ALU_SRC_LITERAL: literal constant.
	254	ALU_SRC_PV: the previous ALU.[X,Y,Z,W] (vector) result.
	255	ALU_SRC_PS: the previous ALU.Trans (scalar) result.
SRC0_REL (S0R) SRC1_REL (S1R)	9 22	enum(1) enum(1)
	0	Absolute: no relative addressing.
	1	Relative: add index from INDEX_MODE to this address.
SRC0_CHAN (S0C) SRC1_CHAN (S1C)	[11:10] [24:23]	enum(2) enum(2)
	Source element to use for this operand.	
	0	CHAN_X: Use X element.
	1	CHAN_Y: Use Y element.
	2	CHAN_Z: Use Z element.
	3	CHAN_W: Use W element.
SRC0_NEG (S0N) SRC1_NEG (S1N)	12 25	int(1) int(1)
	Negation.	
	0	Do not negate input for this operand.
	1	Negate input for this operand. Use only for floating-point inputs.
INDEX_MODE (IM)	[28:26]	enum(3)
	Relative addressing mode, using the address register (AR) or the loop index (aL), for operands that have the SRC_REL or DST_REL bit set.	
	0	INDEX_AR_X - For constants/gpr: add AR.X.
	4	INDEX_LOOP - Add loop index (aL).
	5	INDEX_GLOBAL - Treat GPR address as absolute, not thread-relative.
	6	INDEX_GLOBAL_AR_X- Treat GPR address as absolute, and add GPR-index (AR.X).

ALU Doubleword 0 (Cont.)

	PRED_SEL (PS)	[30:29]	enum(2)
		Predicate to apply to this instruction. 0 PRED_SEL_OFF: execute all pixels. 1 Reserved. 2 PRED_SEL_ZERO: execute if predicate = 0. 3 PRED_SEL_ONE: execute if predicate = 1.	
	LAST (L)	31	int(1)
		Last instruction in an instruction group. 0 This is not the last instruction (64-bit word) in the current instruction group. 1 This is the last instruction (64-bit word) in the current instruction group.	
Related	ALU_WORD1_OP2 ALU_WORD1_OP3		

ALU Doubleword 1 Zero to Two Source Operands

Instructions **ALU_WORD1_OP2**

Description This is the high-order (most-significant) doubleword in the 64-bit microcode-format pair formed by ALU_WORD0 and ALU_WORD1_{OP2, OP3}. Each instruction using this format pair has either an OP2 or an OP3 version (not both). The OP2 version specifies ALU instructions that take zero to two source operands, plus a destination operand.

Opcode

Field Name	*Bits*	*Format*
SRC0_ABS (S0A)	0	int(1)
SRC1_ABS (S1A)	1	int(1)

Absolute value.

0 Use the actual value of the input for this operand.

1 Use the absolute value of the input for this operand. Use only for floating-point inputs. This function is performed before negation.

UPDATE_EXEC_MASK (UEM)	2	int(1)

Update active mask.

0 Do not update the active mask after executing this instruction.

1 Update the active mask after executing this instruction, based on the current predicate.

UPDATE_PRED (UP)	3	int(1)

Update predicate.

0 Do not update the stored predicate.

1 Update the stored predicate based on the predicate operation computed here.

WRITE_MASK (WM)	4	int(1)

Write result to destination vector element.

0 Do not write this scalar result to the destination GPR vector element.

1 Write this scalar result to the destination GPR vector element.

OMOD	[6:5]	enum(2)

Output modifier.

0 ALU_OMOD_OFF: identity. This value must be used for operations that produce an integer result.

1 ALU_OMOD_M2: multiply by 2.0.

2 ALU_OMOD_M4: multiply by 4.0.

3 ALU_OMOD_D2: divide by 2.0.

ALU Doubleword 1 Zero to Two Source Operands (Cont.)

ALU_INST [17:7] enum(11)

Instruction. The top three bits of this field must be zero. See Chapter 7 for descriptions of each instruction. **Opcodes 0 to 95 can be used on either the vector or transcendental unit. Opcodes 129 to 159 are for transcendental units only. Opcodes 160 to 255 are for vector units only.**

0 to 95 are for vector or transcendental units.

0 OP2_INST_ADD
1 OP2_INST_MUL
2 OP2_INST_MUL_IEEE
3 OP2_INST_MAX
4 OP2_INST_MIN
5 OP2_INST_MAX_DX10
6 OP2_INST_MIN_DX10
7 Reserved.
8 OP2_INST_SETE
9 OP2_INST_SETGT
10 OP2_INST_SETGE
11 OP2_INST_SETNE
12 OP2_INST_SETE_DX10
13 OP2_INST_SETGT_DX10
14 OP2_INST_SETGE_DX10
15 OP2_INST_SETNE_DX10
16 OP2_INST_FRACT
17 OP2_INST_TRUNC
18 OP2_INST_CEIL
19 OP2_INST_RNDNE
20 OP2_INST_FLOOR
21 OP2_INST_ASHR_INT
22 OP2_INST_LSHR_INT
23 OP2_INST_LSHL_INT
24 Reserved
25 OP2_INST_MOV
26 OP2_INST_NOP
27 OP2_INST_MUL_64
28 OP2_INST_FLT64_TO_FLT32
29 OP2_INST_FLT32_TO_FLT64
30 OP2_INST_PRED_SETGT_UINT
31 OP2_INST_PRED_SETGE_UINT
32 OP2_INST_PRED_SETE
33 OP2_INST_PRED_SETGT
34 OP2_INST_PRED_SETGE
35 OP2_INST_PRED_SETNE
36 OP2_INST_PRED_SET_INV
37 OP2_INST_PRED_SET_POP
38 OP2_INST_PRED_SET_CLR
39 OP2_INST_PRED_SET_RESTORE
40 OP2_INST_PRED_SETE_PUSH
41 OP2_INST_PRED_SETGT_PUSH
42 OP2_INST_PRED_SETGE_PUSH

ALU Doubleword 1 Zero to Two Source Operands (Cont.)

ALU_INST	[17:8]	enum(10)
	43	OP2_INST_PRED_SETNE_PUSH
	44	OP2_INST_KILLE
	45	OP2_INST_KILLGT
	46	OP2_INST_KILLGE
	47	OP2_INST_KILLNE
	48	OP2_INST_AND_INT
	49	OP2_INST_OR_INT
	50	OP2_INST_XOR_INT
	51	OP2_INST_NOT_INT
	52	OP2_INST_ADD_INT
	53	OP2_INST_SUB_INT
	54	OP2_INST_MAX_INT
	55	OP2_INST_MIN_INT
	56	OP2_INST_MAX_UINT
	57	OP2_INST_MIN_UINT
	58	OP2_INST_SETE_INT
	59	OP2_INST_SETGT_INT
	60	OP2_INST_SETGE_INT
	61	OP2_INST_SETNE_INT
	62	OP2_INST_SETGT_UINT
	63	OP2_INST_SETGE_UINT
	64	OP2_INST_KILLGT_UINT
	65	OP2_INST_KILLGE_UINT
	66	OP2_INST_PREDE_INT
	67	OP2_INST_PRED_SETGT_INT
	68	OP2_INST_PRED_SETGE_INT
	69	OP2_INST_PRED_SETNE_INT
	70	OP2_INST_KILLE_INT
	71	OP2_INST_KILLGT_INT
	72	OP2_INST_KILLGE_INT
	73	OP2_INST_KILLNE_INT
	74	OP2_INST_PRED_SETE_PUSH_INT
	75	OP2_INST_PRED_SETGT_PUSH_INT
	76	OP2_INST_PRED_SETGE_PUSH_INT
	77	OP2_INST_PRED_SETNE_PUSH_INT
	78	OP2_INST_PRED_SETLT_PUSH_INT
	79	OP2_INST_PRED_SETLE_PUSH_INT
	80	OP2_INST_FLT_TO_INT
	81	OP2_INST_BFREV_INT
	82	OP2_INST_ADDC_UINT
	83	OP2_INST_SUBB_UINT
	84	OP2_INST_GROUP_BARRIER
	85	OP2_INST_GROUP_SEQ_BEGIN
	86	OP2_INST_GROUP_SEQ_END
	87	OP2_INST_SET_MODE
	88	OP2_INST_SET_CF_IDX0
	89	OP2_INST_SET_CF_IDX1
	90	OP2_INST_SET_LDS_SIZE
	128:91	*reserved*

ALU Doubleword 1 Zero to Two Source Operands (Cont.)

129 to 159 are for transcendental units only.

129 OP2_INST_EXP_IEEE
130 OP2_INST_LOG_CLAMPED
131 OP2_INST_LOG_IEEE
132 OP2_INST_RECIP_CLAMPED
133 OP2_INST_RECIP_FF
134 OP2_INST_RECIP_IEEE
135 OP2_INST_RECIPSQRT_CLAMPED
136 OP2_INST_RECIPSQRT_FF
137 OP2_INST_RECIPSQRT_IEEE
138 OP2_INST_SQRT_IEEE
141 OP2_INST_SIN
142 OP2_INST_COS
143 OP2_INST_MULLO_INT
144 OP2_INST_MULHI_INT
145 OP2_INST_MULLO_UINT
146 OP2_INST_MULHI_UINT
147 OP2_INST_RECIP_INT
148 OP2_INST_RECIP_UINT
149 OP2_INST_RECIP_64
150 OP2_INST_RECIP_CLAMPED_64
151 OP2_INST_RECIPSQRT_64
152 OP2_INST_RECIPSQRT_CLAMPED_64
153 OP2_INST_SQRT_64
154 OP2_INST_FLT_TO_UINT
155 OP2_INST_INT_TO_FLT
156 OP2_INST_UINT_TO_FLT

160 to 255 are for vector units only.

160 OP2_INST_BFM_INT
162 OP2_INST_FLT32_TO_FLT16
163 OP2_INST_FLT16_TO_FLT32
164 OP2_INST_UBYTE0_FLT
165 OP2_INST_UBYTE1_FLT
166 OP2_INST_UBYTE2_FLT
167 OP2_INST_UBYTE3_FLT
170 OP2_INST_BCNT_INT
171 OP2_INST_FFBH_UINT
172 OP2_INST_FFBL_INT
173 OP2_INST_FFBH_INT
174 OP2_INST_FLT_TO_UINT4
175 OP2_INST_DOT_IEEE
176 OP2_INST_FLT_TO_INT_RPI
177 OP2_INST_FLT_TO_INT_FLOOR
178 OP2_INST_MULHI_UINT24
179 OP2_INST_MBCNT_32HI_INT
180 OP2_INST_OFFSET_TO_FLT
181 OP2_INST_MUL_UINT24
182 OP2_INST_BCNT_ACCUM_PREV_INT
183 OP2_INST_MBCNT_32LO_ACCUM_PREV_INT
184 OP2_INST_SETE_64
185 OP2_INST_SETNE_64

ALU Doubleword 1 Zero to Two Source Operands (Cont.)

		186 OP2_INST_SETGT_64 187 OP2_INST_SETGE_64 188 OP2_INST_MIN_64 189 OP2_INST_MAX_64 190 OP2_INST_DOT4 191 OP2_INST_DOT4_IEEE 192 OP2_INST_CUBE 193 OP2_INST_MAX4 196 OP2_INST_FREXP_64 197 OP2_INST_LDEXP_64 198 OP2_INST_FRACT_64 199 OP2_INST_PRED_SETGT_64 200 OP2_INST_PRED_SETE_64 201 OP2_INST_PRED_SETGE_64 202 OP2_INST_MUL_64 203 OP2_INST_ADD_64 204 OP2_INST_MOVA_INT 205 OP2_INST_FLT64_TO_FLT32 206 OP2_INST_FLT32_TO_FLT64 207 OP2_INST_SAD_ACCUM_PREV_UINT 208 OP2_INST_DOT 209 OP2_INST_MUL_PREV 210 OP2_INST_MUL_IEEE_PREV 211 OP2_INST_ADD_PREV 212 OP2_INST_MULADD_PREV 213 OP2_INST_MULADD_IEEE_PREV 214 OP2_INST_INTERP_XY 215 OP2_INST_INTERP_ZW 216 OP2_INST_INTERP_X 217 OP2_INST_INTERP_Z 218 OP2_INST_STORE_FLAGS 219 OP2_INST_LOAD_STORE_FLAGS 220 OP2_INST_LDS_1A: DO NOT USE. Use OP3_LDS_IDX_OP instead. This is for hardware SP bus only 221 OP2_INST_LDS_1A1D: DO NOT USE. Use OP3_LDS_IDX_OP instead. This is for hardware SP bus only 223 OP2_INST_LDS_2A: DO NOT USE. Use OP3_LDS_IDX_OP instead. This is for hardware SP bus only 224 OP2_INST_INTERP_LOAD_P0 225 OP2_INST_INTERP_LOAD_P10 226 OP2_INST_INTERP_LOAD_P20
BANK_SWIZZLE	[20:18]	enum(3)
(BS)	Specifies how to load operands into the SP.	0 ALU_VEC_012, SQ_ALU_SCL_210 1 ALU_VEC_021, SQ_ALU_SCL_122 2 ALU_VEC_120, SQ_ALU_SCL_212 3 ALU_VEC_102, SQ_ALU_SCL_221 4 ALU_VEC_201 5 ALU_VEC_210 6-8 Reserved.

ALU Doubleword 1 Zero to Two Source Operands (Cont.)

DST_GPR	[27:21]	enum(7)
	Destination address to which result is written. Always a GPR address.	
DST_REL (DR)	28	enum(1)
	Specifies whether to use absolute or relative addressing. 0 ABSOLUTE: no relative addressing. 1 RELATIVE: add index from INDEX_MODE to this address.	
DST_CHAN (DC)	[30:29]	enum(2)
	Specifies to which element of DST_GPR the result is written. 0 CHAN_X: write to X element of destination. 1 CHAN_Y: write to Y element of destination. 2 CHAN_Z: write to Z element of destination. 3 CHAN_W: write to W element of destination.	
CLAMP (C)	31	int(1)
	If set, clamp the result to [0.0, 1.0]. Not mathematically defined for opcodes that produce integer results.	

Related ALU_WORD0
ALU_WORD1_OP3

ALU Doubleword 1 Three Source Operands

Instructions **ALU_WORD1_OP3**

Description This is the high-order (most-significant) doubleword in the 64-bit microcode-format pair formed by ALU_WORD0 and ALU_WORD1_{OP2, OP3}. Each instruction using this format pair has either an OP2 or an OP3 version (not both). The OP3 version specifies ALU instructions that take three source operands, plus a destination operand.

Opcode

Field Name	*Bits*	*Format*
SRC2_SEL	[8:0]	enum(9)

Location or value of this source operand.

[127:0] Value in GPR[127:0].
[159:128] Kcache constants in bank 0.
[191:160] Kcache constants in bank 1.
[255:192] inline constant values.
[287:256] Kcache constants in bank 2.
[319:288] Kcache constants in bank 3.
Other special values are shown below.

219 ALU_SRC_LDS_OQ_A: Use contents of LDS output queue A, and leave it on the queue.
220 ALU_SRC_LDS_OQ_B: Use contents of LDS output queue B, and leave it on the queue.
221 ALU_SRC_LDS_OQ_A_POP: Use contents of LDS output queue A, and pop both the A and B queues at the end of the instruction group (xyzwt).
222 ALU_SRC_LDS_OQ_B_POP: Use contents of LDS output queue B, and pop both the A and B queues at the end of the instruction group (xyzwt).
223 ALU_SRC_LDS_DIRECT_A: Direct read of LDS on the A cycle. Address is defined in literal constant-0 (xy).
224 ALU_SRC_LDS_DIRECT_B: Direct read of LDS on the B cycle. Address is defined in literal constant-0 (xy).
227 ALU_SRC_TIME_HI: Upper 32 bits of 64-bit clock counter.
228 ALU_SRC_TIME_LO: Lower 32 bits of 64-bit clock counter.
229 ALU_SRC_MASK_HI: Upper 32bits of active mask.
230 ALU_SRC_MASK_LO: Lower 32bits of active mask.
231 ALU_SRC_HW_WAVE_ID: Hardware wave ID (int).
232 ALU_SRC_SIMD_ID: Simd id (int).
233 ALU_SRC_SE_ID: Shader engine ID (int).
234 ALU_SRC_HW_THREADGRP_ID: Hardware thread group ID (int) within simd. CS and HS only.
235 ALU_SRC_WAVE_ID_IN_GRP: Wave id within thread group (int). CS and HS only.
236 ALU_SRC_NUM_THREADGRP_WAVES: Number of waves in thread group (int). CS and HS only; must barrier before using.
237 ALU_SRC_HW_ALU_ODD: This clause executes on the even (0) or odd (1) path (int).
238 ALU_SRC_LOOP_IDX: Current value of the loop index (int)
240 ALU_SRC_PARAM_BASE_ADDR: Parameter cache base (LDS_ALLOC_PS), (int).

ALU Doubleword 1 Three Source Operands (Cont.)

	241	ALU_SRC_NEW_PRIM_MASK: Bit mask. 1 bit per quad, '1' indicates that this quad starts a new primitive. The mask omits bit for first quad because it always begins a new primitive. For example, in a vectorsize 64 system, this mask is {[15:1],1'b1}.
	242	ALU_SRC_PRIM_MASK_HI: Upper 32 bits of 64-bit expansion of NEW_PRIM_MASK. Used for general parameter interpolation.
	243	ALU_SRC_PRIM_MASK_LO: Lower 32 bits of 64-bit expansion of NEW_PRIM_MASK. Used for general parameter interpolation.
	244	ALU_SRC_1_DBL_L: special constant 1.0 double-float, LSW.
	245	ALU_SRC_1_DBL_M: special constant 1.0 double-float, MSW.
	246	ALU_SRC_0_5_DBL_L: special constant 0.5 double-float, LSW.
	247	ALU_SRC_0_5_DBL_M: special constant 0.5 double-float, MSW.
	248	ALU_SRC_0: the constant 0.0.
	249	ALU_SRC_1: the constant 1.0 float.
	250	ALU_SRC_1_INT: the constant 1 integer.
	251	ALU_SRC_M_1_INT: the constant -1 integer.
	252	ALU_SRC_0_5: the constant 0.5 float.
	253	ALU_SRC_LITERAL: literal constant.
	254	ALU_SRC_PV: previous ALU.[X,Y,Z,W] result.
	255	ALU_SRC_PS: previous ALU.Trans result.
SRC2_REL (SR)	9	enum(1)
	Addressing mode for this source operand.	
	0	Absolute: no relative addressing.
	1	Relative: add index from INDEX_MODE to this address. See ALU_WORD0, on page 10-23, for the specification of INDEX_MODE.
SRC2_CHAN (S2C)	[11:10]	enum(2)
	Source element to use for this operand.	
	0	CHAN_X: Use X element.
	1	CHAN_Y: Use Y element.
	2	CHAN_Z: Use Z element.
	3	CHAN_W: Use W element.
SRC2_NEG (SN)	12	int(1)
	Negation.	
	0	Do not negate input for this operand.
	1	Negate input for this operand. Use only for floating-point inputs.

ALU Doubleword 1 Three Source Operands (Cont.)

ALU_INST	[17:13]	enum(5)
	Instruction. Gaps in opcode values are not marked in the list below. See Chapter 9 for descriptions of each instruction. Note: opcode values do not begin at zero.	
	Opcodes 4..17 are vector-only.	
	Opcodes 20..31 can be used in either the vector or trans unit.	
	Opcode 31 is trans-only.	
	4 OP3_INST_BFE_UINT	
	5 OP3_INST_BFE_INT	
	6 OP3_INST_BFI_INT	
	7 OP3_INST_FMA	
	9 OP3_INST_CNDNE_64	
	10 OP3_INST_FMA_64	
	11 OP3_INST_LERP_UINT	
	12 OP3_INST_BIT_ALIGN_INT	
	13 OP3_INST_BYTE_ALIGN_INT	
	14 OP3_INST_SAD_ACCUM_UINT	
	15 OP3_INST_SAD_ACCUM_HI_UINT	
	16 OP3_INST_MULADD_UINT24	
	17 OP3_INST_LDS_IDX_OP: This opcodes implies ALU_WORD*_LDS_IDX_OP encoding.	
	20 OP3_INST_MULADD	
	21 OP3_INST_MULADD_M2	
	22 OP3_INST_MULADD_M4	
	23 OP3_INST_MULADD_D2	
	24 OP3_INST_MULADD_IEEE	
	25 OP3_INST_CNDE	
	26 OP3_INST_CNDGT	
	27 OP3_INST_CNDGE	
	28 OP3_INST_CNDE_INT	
	29 OP3_INST_CNDGT_INT	
	30 OP3_INST_CNDGE_INT	
	31 OP3_INST_MUL_LIT	
BANK_SWIZZLE (BS)	[20:18]	enum(3)
	Specifies how to load operands into the SP.	
	0 ALU_VEC_012, SQ_ALU_SCL_210	
	1 ALU_VEC_021, SQ_ALU_SCL_122	
	2 ALU_VEC_120, SQ_ALU_SCL_212	
	3 ALU_VEC_102, SQ_ALU_SCL_221	
	4 ALU_VEC_201	
	5 ALU_VEC_210	
	6-8 Reserved.	
DST_GPR	[27:21]	enum(7)
	Destination address to which result is written. Always a GPR address.	
DST_REL (DR)	28	enum(1)
	Specifies whether to use absolute or relative addressing.	
	0 ABSOLUTE: no relative addressing.	
	1 RELATIVE: add index from INDEX_MODE to this address.	

ALU Doubleword 1 Three Source Operands (Cont.)

	DST_CHAN (DC)	[30:29] enum(2) Specifies to which element of DST_GPR the result is written. 0 CHAN_X: write to X element of destination. 1 CHAN_Y: write to Y element of destination. 2 CHAN_Z: write to Z element of destination. 3 CHAN_W: write to W element of destination.
	CLAMP	31 int(1) If set, clamp the result to [0.0, 1.0]. Not mathematically defined for opcodes that produce integer results.
Related	ALU_WORD0 ALU_WORD1_OP2	

ALU Doubleword 0 for LDS IDX

Instructions **ALU_WORD0_LDS_IDX_OP**

Description This is the least-significant doubleword in the 64-bit microcode-format pair formed by ALU_WORD0_LDS_IDX_OP and ALU_WORD1_LDS_IDX_OP. These ALU opcodes move data between GPRs and the local data store (LDS). Indexed operations take the LDS address from a GPR and either read, write, or perform an atomic arithmetic operation on data in the LDS with GPR data, then write back the result to the LDS.

Opcode

Field Name	*Bits*	*Format*
SRC0_SEL	[8:0]	enum(9)
SRC1_SEL	[21:13]	enum(9)

[127:0] Value in GPR[127:0].
[159:128] Kcache constants in bank 0.
[191:160] Kcache constants in bank 1.
[255:192] inline constant values.
[287:256] Kcache constants in bank 2.
[319:288] Kcache constants in bank 3.

219 ALU_SRC_LDS_OQ_A: Use contents of LDS output queue A, and leave it on the queue.
220 ALU_SRC_LDS_OQ_B: Use contents of LDS output queue B, and leave it on the queue.
221 ALU_SRC_LDS_OQ_A_POP: Use contents of LDS output queue A, and pop both the A and B queues at the end of the instruction group (xyzwt).
222 ALU_SRC_LDS_OQ_B_POP: Use contents of LDS output queue B, and pop both the A and B queues at the end of the instruction group (xyzwt).
223 ALU_SRC_LDS_DIRECT_A: Direct read of LDS on the A cycle. Address is defined in literal constant-0 (xy).
224 ALU_SRC_LDS_DIRECT_B: Direct read of LDS on the B cycle. Address is defined in literal constant-0 (xy).
227 ALU_SRC_TIME_HI: Upper 32 bits of 64-bit clock counter.
228 ALU_SRC_TIME_LO: Lower 32 bits of 64-bit clock counter.
229 ALU_SRC_MASK_HI: Upper 32bits of active mask.
230 ALU_SRC_MASK_LO: Lower 32bits of active mask.
231 ALU_SRC_HW_WAVE_ID: Hardware wave ID (int)
232 ALU_SRC_SIMD_ID: SIMD id (int).
233 ALU_SRC_SE_ID: Shader engine ID (int).
234 ALU_SRC_HW_THREADGRP_ID: Hardware thread group ID (int) within SIMD. CS and HS only.
235 ALU_SRC_WAVE_ID_IN_GRP: Wave id within thread group (int). CS and HS only.
236 ALU_SRC_NUM_THREADGRP_WAVES: Number of waves in thread group (int). CS and HS only, must barrier before using.
237 ALU_SRC_HW_ALU_ODD: Is this clause executing on the even(0) or odd(1) path (int).
238 ALU_SRC_LOOP_IDX: Current value of the loop index (int).
240 ALU_SRC_PARAM_BASE_ADDR: Parameter cache base (LDS_ALLOC_PS) (int).
241 ALU_SRC_NEW_PRIM_MASK: Bit mask. One bit per quad. Set indicates that this quad starts a new primitive. Mask omits bit for first quad because it always begins a new primitive. For example, in a vectorsize 64 system, this mask is {[15:1],1'b1}.

ALU Doubleword 0 for LDS IDX

	242 ALU_SRC_PRIM_MASK_HI: Upper 32 bits of 64-bit expansion of NEW_PRIM_MASK. Used for general parameter interpolation. 243 ALU_SRC_PRIM_MASK_LO: Lower 32 bits of 64-bit expansion of NEW_PRIM_MASK. Used for general parameter interpolation. 244 ALU_SRC_1_DBL_L: special constant 1.0 double-float, LSW. 245 ALU_SRC_1_DBL_M: special constant 1.0 double-float, MSW. 246 ALU_SRC_0_5_DBL_L: special constant 0.5 double-float, LSW. 247 ALU_SRC_0_5_DBL_M: special constant 0.5 double-float, MSW. 248 ALU_SRC_0: special constant 0.0. 249 ALU_SRC_1: special constant 1.0 float. 250 ALU_SRC_1_INT: special constant 1 integer. 251 ALU_SRC_M_1_INT: special constant -1 integer. 252 ALU_SRC_0_5: special constant 0.5 float. 253 ALU_SRC_LITERAL: literal constant. 254 ALU_SRC_PV: previous vector result. 255 ALU_SRC_PS: previous scalar result.
SRC0_REL SRC1_REL	9 enum(1) 22 enum(1) Relative addressing. 0 No relative addressing used. 1 Add index from INDEX_MODE to this address.
SRC0_CHAN SRC1_CHAN	[11:10] enum(2) [24:23] enum(2) Specifies which element of the source to use for this operand. 0 CHAN_X: Use X element. 1 CHAN_Y: Use Y element. 2 CHAN_Z: Use Z element. 3 CHAN_W: Use W element.
IDX_OFFSET_4	12 int(1) Index offset bit 4.
IDX_OFFSET_5	25 int(1) Index offset bit 5.
INDEX_MODE	[28:26] enum(3) Specifies the relative addressing mode to use for operands that have the REL bit set. 0 INDEX_AR_X: constant/GPR: add AR.X. 4 INDEX_LOOP:add current loop index value. 5 INDEX_GLOBAL:treat GPR address as absolute, not thread-relative. 6 INDEX_GLOBAL_AR_X:treat GPR address as absolute, and add GPR-index (AR.X).
PRED_SEL	[30:29] enum(2) Predicate to apply to this instruction. 0 PRED_SEL_OFF:execute all pixels. 1 Reserved. 2 PRED_SEL_ZERO:execute when predicate = 0. 3 PRED_SEL_ONE:execute when predicate = 1.

ALU Doubleword 0 for LDS IDX

	LAST	31	int(1)
		When set, indicates this is the last 64-bit word for this instruction.	
Related	ALU_WORD0_LDS_IDX_OP		

ALU Doubleword 1 for LDS IDX

Instructions **ALU_WORD1_LDS_IDX_OP**

Description This is the most-significant doubleword in the 64-bit microcode-format pair formed by ALU_WORD0_LDS_IDX_OP and ALU_WORD1_LDS_IDX_OP.. These ALU opcodes move data between GPRs and the local data store (LDS). Indexed operations take the LDS address from a GPR and either read, write, or perform an atomic arithmetic operation on data in the LDS with GPR data, then write back the result to the LDS.

Opcode

Field Name	*Bits*	*Format*
SRC2_SEL	[8:0]	enum(9)

[127:0] Value in GPR[127:0].
[159:128] Kcache constants in bank 0.
[191:160] Kcache constants in bank 1.
[255:192] inline constant values.
[287:256] Kcache constants in bank 2.
[319:288] Kcache constants in bank 3.

219 ALU_SRC_LDS_OQ_A: Use contents of LDS output queue A, and leave it on the queue.
220 ALU_SRC_LDS_OQ_B: Use contents of LDS output queue B, and leave it on the queue.
221 ALU_SRC_LDS_OQ_A_POP: Use contents of LDS output queue A, and pop both the A and B queues at the end of the instruction group (xyzwt).
222 ALU_SRC_LDS_OQ_B_POP: Use contents of LDS output queue B, and pop both the A and B queues at the end of the instruction group (xyzwt).
223 ALU_SRC_LDS_DIRECT_A: Direct read of LDS on the A cycle. Address is defined in literal constant-0 (xy).
224 ALU_SRC_LDS_DIRECT_B: Direct read of LDS on the B cycle. Address is defined in literal constant-0 (xy).
227 ALU_SRC_TIME_HI: Upper 32 bits of 64-bit clock counter.
228 ALU_SRC_TIME_LO: Lower 32 bits of 64-bit clock counter.
229 ALU_SRC_MASK_HI: Upper 32bits of active mask.
230 ALU_SRC_MASK_LO: Lower 32bits of active mask.
231 ALU_SRC_HW_WAVE_ID: Hardware wave ID (int)
232 ALU_SRC_SIMD_ID: SIMD id (int).
233 ALU_SRC_SE_ID: Shader engine ID (int).
234 ALU_SRC_HW_THREADGRP_ID: Hardware thread group ID (int) within SIMD. CS and HS only.
235 ALU_SRC_WAVE_ID_IN_GRP: Wave id within thread group (int). CS and HS only.
236 ALU_SRC_NUM_THREADGRP_WAVES: Number of waves in thread group (int). CS and HS only, must barrier before using.
237 ALU_SRC_HW_ALU_ODD: Is this clause executing on the even(0) or odd(1) path (int).
238 ALU_SRC_LOOP_IDX: Current value of the loop index (int).
240 ALU_SRC_PARAM_BASE_ADDR: Parameter cache base (LDS_ALLOC_PS) (int).

ALU Doubleword 1 for LDS IDX

		241 ALU_SRC_NEW_PRIM_MASK: Bit mask. One bit per quad. Set indicates that this quad starts a new primitive. Mask omits bit for first quad because it always begins a new primitive. For example, in a vectorsize 64 system, this mask is {[15:1],1'b1}. 242 ALU_SRC_PRIM_MASK_HI: Upper 32 bits of 64-bit expansion of NEW_PRIM_MASK. Used for general parameter interpolation. 243 ALU_SRC_PRIM_MASK_LO: Lower 32 bits of 64-bit expansion of NEW_PRIM_MASK. Used for general parameter interpolation. 244 ALU_SRC_1_DBL_L: special constant 1.0 double-float, LSW. 245 ALU_SRC_1_DBL_M: special constant 1.0 double-float, MSW. 246 ALU_SRC_0_5_DBL_L: special constant 0.5 double-float, LSW. 247 ALU_SRC_0_5_DBL_M: special constant 0.5 double-float, MSW. 248 ALU_SRC_0: special constant 0.0. 249 ALU_SRC_1: special constant 1.0 float. 250 ALU_SRC_1_INT: special constant 1 integer. 251 ALU_SRC_M_1_INT: special constant -1 integer. 252 ALU_SRC_0_5: special constant 0.5 float. 253 ALU_SRC_LITERAL: literal constant. 254 ALU_SRC_PV: previous vector result. 255 ALU_SRC_PS: previous scalar result.
SRC2_REL	9	enum(1) Relative addressing. 0 No relative addressing used. 1 Add index from INDEX_MODE to this address.
SRC2_CHAN	[11:10]	enum(2) Specifies which element of the source to use for this operand. 0 CHAN_X: Use X element. 1 CHAN_Y: Use Y element. 2 CHAN_Z: Use Z element. 3 CHAN_W: Use W element.
IDX_OFFSET_1	12	int(1) Index offset bit 4.
ALU_INST	[17:13]	enum(5) The only legal value for this field is: 17 OP3_INST_LDS_IDX_OP: This opcodes implies ALU_WORD*_LDS_IDX_OP encoding.
BANK_SWIZZLE	[20:18]	enum(3) Specifies how to load operands into the SP. 0 ALU_VEC_012, ALU_SCL_210 1 ALU_VEC_021, ALU_SCL_122 2 ALU_VEC_120, ALU_SCL_212 3 ALU_VEC_102, ALU_SCL_221 4 ALU_VEC_201 5 ALU_VEC_210

ALU Doubleword 1 for LDS IDX

LDS_OP [26:21] enum(6)

Local data share atomic opcode.

0 DS_INST_ADD: OP(dst,src, ...) dst=src0_sel, src=src1_sel. 1A1D ADD(dst,src) : DS(dst) += src. dst is src0_sel, src is src1_sel.

1 DS_INST_SUB: 1A1D SUB(dst,src) : DS(dst) = DS(dst) - src.

2 DS_INST_RSUB: 1A1D RSUB(dst,src): DS(dst) = src - DS(dst).

3 DS_INST_INC: 1A1D INC(dst) : (DS(dst)>=src) ? DS(dst) = 0 : DS(dst)++.

4 DS_INST_DEC: 1A1D DEC(dst) : DS(dst) = ((DS(dst)==0) || (DS(dst)>src)) ? src : DS(dst)-1.

5 DS_INST_MIN_INT: 1A1D MIN(dst,src) : DS(dst) = min (DS(dst),src).

6 DS_INST_MAX_INT: 1A1D MAX(dst,src) : DS(dst) = max(DS(dst),src).

7 DS_INST_MIN_UINT: 1A1D MIN(dst,src) : DS(dst) = min (DS(dst),src).

8 DS_INST_MAX_UINT: 1A1D MAX(dst,src) : DS(dst) = max(DS(dst),src).

9 DS_INST_AND: 1A1D AND(dst,src) : DS(dst) &= src.

10 DS_INST_OR: 1A1D OR(dst,src) : DS(dst) |= src

11 DS_INST_XOR: 1A1D XOR(dst,src) : DS(dst) ^= src

12 DS_INST_MSKOR: 1A2D MKSOR(dst,mask,src) : DS(dst) = ((DS(dst) & ~msk) | src).

13 DS_INST_WRITE: 1A1D WRITE(dst,src) : DS(dst) = src.

14 DS_INST_WRITE_REL: 1A2D WRITEREL(dst,src0,src1) : tmp = dst + DS_idx_offset (offset in dwords). DS(dst) = src0, DS(tmp) = src1.

15 DS_INST_WRITE2: 1A2D WRITE2(dst,src0,src1) : tmp = dst+(DS_idx_offset * 64). DS(dst) = src0, DS(tmp) = src1.

16 DS_INST_CMP_STORE: 1A2D CMP_STORE(dst, cmp, src) : DS(dst) = (DS(dst) == cmp) ? src : DS(dst)

17 DS_INST_CMP_STORE_SPF: 1A2D CMP_STORE_SPF(dst, cmp, src) : DS(dst) = (DS(dst) == cmp) ? src : DS(dst)

18 DS_INST_BYTE_WRITE: 1A1D BYTEWRITE (dst, src) : DS(dst) = src[7:0]

19 DS_INST_SHORT_WRITE: 1A1D SHORTWRITE(dst, src) : DS(dst) = src[15:0]

20-31 Reserved.

32 DS_INST_ADD_RET: 1A1D ADD(dst,src) : OQA=DS(dst), DS(dst) += src. dst is src0_sel, src is src1_sel.

33 DS_INST_SUB_RET: 1A1D SUB(dst,src) : OQA=DS(dst), DS(dst) = DS(dst) - src.

34 DS_INST_RSUB_RET: 1A1D RSUB(dst,src) : OQA=DS(dst), DS(dst) = src - DS(dst).

35 DS_INST_INC_RET: 1A1D INC(dst) : OQA=DS(dst), (DS(dst)>=src) ? DS(dst) = 0 : DS(dst)++.

36 DS_INST_DEC_RET: 1A1D DEC(dst) : OQA=DS(dst), DS(dst) = ((DS(dst)==0) || (DS(dst)>src)) ? src : DS(dst)-1.

37 DS_INST_MIN_INT_RET: 1A1D MIN(dst,src) : OQA=DS(dst), DS(dst) = min (DS(dst),src).

38 DS_INST_MAX_INT_RET: 1A1D MAX(dst,src) : OQA=DS(dst), DS(dst) = max(DS(dst),src).

39 DS_INST_MIN_UINT_RET: 1A1D MIN(dst,src) : OQA=DS(dst), DS(dst) = min (DS(dst),src).

40 DS_INST_MAX_UINT_RET: 1A1D MAX(dst,src) : OQA=DS(dst), DS(dst) = max(DS(dst),src)

41 DS_INST_AND_RET: 1A1D AND(dst,src) : OQA=DS(dst), DS(dst) &= src

42 DS_INST_OR_RET: 1A1D OR(dst,src) : OQA=DS(dst), DS(dst) |= src

43 DS_INST_XOR_RET: 1A1D XOR(dst,src) : OQA=DS(dst), DS(dst) ^= src

ALU Doubleword 1 for LDS IDX

	44 DS_INST_MSKOR_RET: 1A2D MSKOR(dst,msk,src) : OQA=DS(dst), DS(dst) = ((DS(dst) & ~msk) | src). 45 DS_INST_XCHG_RET: 1A1D Exchange(dst,src) : OQA=DS(dst), DS(dst) = src. 46 DS_INST_XCHG_REL_RET: 1A2D ExchangeRel(dst,src0,src1) : tmp = dst + DS_idx_offset. OQA=DS(dst), OQB=DS(tmp); DS(dst)=src0, DS(tmp)=src1. 47 DS_INST_XCHG2_RET: 1A2D Exchange2(dst,src0,src1) : tmp = dst + DS_idx_offset*64. OQA=DS(dst), OQB=DS(tmp); DS(dst)=src0, DS(tmp)=src1. 48 DS_INST_CMP_XCHG_RET: 1A2D CompareExchange(dst,cmp,src) : OQA=DS(dst); (DS(dst)==cmp) ? DS(dst)=src : DS(dst)=DS(dst). 49 DS_INST_CMP_XCHG_SPF_RET: 1A2D CompareExchangeSPF(dst,cmp,src) : OQA=DS(dst); (DS(dst)==cmp) ? DS(dst)=src : DS(dst)=DS(dst). 50 DS_INST_READ_RET: 1A READ(dst) : OQA = DS(dst). 51 DS_INST_READ_REL_RET: 1A READ_REL(dst) : tmp=dst+sq_DS_idx_offset; OQA=DS(dst), OQB=DS(tmp). 52 DS_INST_READ2_RET: 2A READ2(dst0,dst1) : OQA=DS(dst0), OQB=DS(dst1). 53 DS_INST_READWRITE_RET: 2A1D READWRITE(dst0,dst1,data) : OQA=DS(dst0), DS(dst1)=data. 54 DS_INST_BYTE_READ_RET: 1A BYTEREAD(dst) : OQA=SignExtend(DS(dst)[7:0]). 55 DS_INST_UBYTE_READ_RET: 1A UBYTEREAD(dst) : OQA={24'h0, DS(dst)[7:0]}. 56 DS_INST_SHORT_READ_RET: 1A SHORTREAD(dst) : OQA=SignExtend(DS(dst)[15:0]} 57 DS_INST_USHORT_READ_RET: 1A USHORTREAD(dst) : OQA={16'h0, DS(dst)[15:0]} 62:58 Reserved. 63 DS_INST_ATOMIC_ORDERED_ALLOC_RET: 1A GDS-only (intercepted by ordered alloc unit). This adds the 7 lsb of 1a to a hidden ordered append count in wave order and returns the pre-op value to the specified destination register. This opcode can only be used by GDS and with broadcast first set.
IDX_OFFSET_0	27 int(1) Index offset bit 0. Dword offset, except for LDS_OP value 15, which is a 64-dword offset.
IDX_OFFSET_2	28 int(1) Index offset bit 2.
DST_CHAN	[30:29] enum(2) Specifies to which DST_GPR element results are written. 0 CHAN_X: write to X element of destination. 1 CHAN_Y: write to Y element of destination. 2 CHAN_Z: write to Z element of destination. 3 CHAN_W: write to W element of destination.
INDEX_OFFSET_3	31 int(1) Index offset bit 3.

Related ALU_WORD1_LDS_iDX_OP

Literal Doubleword0 Constant Contents for Direct LDS Reads

Instructions **ALU_WORD1_LDS_DIRECT_LITERAL_LO**

Description When an ALU instruction includes a direct-read of LDS, the instruction must be followed by a 64-bit literal constant formed by ALU_WORD1_LDS_DIRECT_LITERAL_LO and ALU_WORD1_LDS_DIRECT_LITERAL_HI. This defines the address from which to read. An LDS direct read occurs when one of the source selects to an ALU operation is ALU_SRC_LDS_DIRECT_A or ALU_SRC_LDS_DIRECT_B.

Opcode

Field Name	*Bits*	*Format*
OFFSET_A	[12:0]	int(13)
	Dword offset for LDS direct read.	
STRIDE_A	[19:13]	int(7)
	Dword stride. Stride must not cause bank conflict in LDS RAM.	
RESERVED	[21:20]	Reserved.
	Bank (constant buffer number) for second set of locked cache lines.	
THREAD_REL_A	22	int(1)
	Bank (constant buffer number) for second set of locked cache lines.	
RESERVED	[31:23]	Reserved.

Related CF_ALU_WORD1

Literal Doubleword1 Constant Contents for Direct LDS Reads

Instructions **ALU_WORD1_LDS_DIRECT_LITERAL_HI**

Description When an ALU instruction includes a direct-read of LDS, the instruction must be followed by a 64-bit literal constant formed by ALU_WORD1_LDS_DIRECT_LITERAL_LO and ALU_WORD1_LDS_DIRECT_LITERAL_HI. This defines the address from which to read. An LDS direct read occurs when one of the source selects to an ALU operation is ALU_SRC_LDS_DIRECT_A or ALU_SRC_LDS_DIRECT_B.

Opcode

Field Name	*Bits*	*Format*
OFFSET_B	[12:0]	int(13)
	Dword offset for LDS direct read.	
STRIDE_B	[19:13]	int(7)
	Dword stride. Stride must not cause bank conflict in LDS RAM.	
	[21:20]	Reserved.
	Bank (constant buffer number) for second set of locked cache lines.	
THREAD_REL_B	22	int(1)
	Bank (constant buffer number) for second set of locked cache lines.	
	[30:23]	Reserved.
DIRECT_READ_32	31	int(1)
	0	Read 16 dwords for A and B on each of four cycles.
	1	Read 32 dwords for A in one cycle, then 32 dwords for B in the next cycle, then repeat.

Related CF_ALU_WORD1

10.3 Instructions for Fetches Through a Vertex Cache Clause

Fetches through vertex cache clauses are specified in the CF_WORD0 and CF_WORD1 formats, described in Section 10.1, on page 10-2. After the clause is specified, the instructions below can be issued. Graphics programs typically use these instructions to load vertex data from off-chip memory into GPRs. General-computing programs typically do not use these instructions; instead, they use instructions for a fetch through a texture cache clause to load all data.

All microcode formats for fetches through a vertex cache clause are 64 bits wide.

Fetch Through a Vertex Cache Clause Doubleword 0

Instructions **VTX_WORD0**

Description This is the low-order (least-significant) doubleword in the 128-bit 4-tuple formed by VTX_WORD0, VTX_WORD1_{SEM, GPR}, VTX_WORD2, plus a doubleword filled with zeros, as described in Chapter 5. Each instruction using this format 4-tuple has either an SEM or an GPR version (not both) for its second doubleword. The instructions are specified in the VTX_WORD0 doubleword.

Opcode

Field Name	*Bits*	*Format*
VC_INST	[4:0]	enum(5)
	Instruction. 0 VC_INST_FETCH: fetch through a vertex cache clause (X = uint32 index). Use VTX_WORD1_GPR (page 10-47). Not for use with MEM_RD_WORD* or MEM_GDS_WORD* encodings. 1 VC_INST_SEMANTIC: semantic fetch through a vertex cache clause. Use VTX_WORD1_SEM (page 10-50). Not for use with MEM_RD_WORD* or MEM_GDS_WORD* encodings. 14 VC_INST_GET_BUFFER_RESINFO: returns the number of elements in a buffer. This is a fetch through a vertex cache clause and uses a vertex constant; it can be serviced only by TC, not by VC. All other values are reserved.	
FETCH_TYPE (FT)	[6:5]	enum(2)
	Specifies which index offset to send to the vertex cache. 0 VTX_FETCH_VERTEX_DATA 1 VTX_FETCH_INSTANCE_DATA 2 VTX_FETCH_NO_INDEX_OFFSET	
FETCH_WHOLE_QUAD (FWQ)	7	int(1)
	0 Texture instruction can ignore inactive pixels. 1 Texture instruction must fetch data for all pixels in any quad which as at least one pixel is both active and valid. The result can be used as source coordinate of a dependent read. Set this only in PS stage.	
BUFFER_ID	[15:8]	int(8)
	Constant ID to use for this fetch through a vertex cache clause (indicates the buffer address, size, and format).	
SRC_GPR	[22:16]	int(7)
	Source GPR address to get fetch address from.	
SRC_REL (SR)	23	enum(1)
	Specifies whether source address is absolute or relative to an index. 0 Absolute: no relative addressing. 1 Relative: add current loop index (aL) value to this address.	
SRC_SEL_X (SSX)	[25:24]	enum(2)
	Specifies which element of SRC to use for the fetch address. 0 SEL_X: use X element. 1 SEL_Y: use Y element. 2 SEL_Z: use Z element. 3 SEL_W: use W element.	

Fetch Through a Vertex Cache Clause Doubleword 0

	MEGA_FETCH_COUNT (MFC)	[31:26]	int(6) For a mega-fetch, specifies the number of bytes to fetch at once. For mini-fetch, number of bytes to fetch if the processor converts this instruction into a mega-fetch. This value's range is [1,64].
Related	VTX_WORD1_GPR VTX_WORD1_SEM VTX_WORD2		

Fetch Through a Vertex Cache Clause Doubleword 1 GPR

Instructions **VTX_WORD1_GPR**

Description This doubleword is part of the 128-bit 4-tuple formed by VTX_WORD0, VTX_WORD1_{SEM, GPR}, VTX_WORD2, plus a doubleword filled with zeros (DWORD3), as described in Chapter 5. Each instruction using this format 4-tuple has either a SEM or GPR format (not both) for its second doubleword. The instructions are specified in the VTX_WORD0 doubleword. This GPR format is used by FETCH instructions that specify a destination GPR directly. See the next format for the semantic-table option.

Opcode

Field Name	*Bits*	*Format*
DST_GPR	[6:0]	int(7)
	Destination GPR address to which result is written.	
DST_REL (DR)	7	enum(1)
	Specifies whether destination address is absolute or relative to an index. 0 Absolute: no relative addressing. 1 Relative: add current loop index (aL) value to this address.	
Reserved	8	Reserved. Set to 0.
DST_SEL_X (DSX)	[11:9]	enum(3)
DST_SEL_Y (DSY)	[14:12]	enum(3)
DST_SEL_Z (DSZ)	[17:15]	enum(3)
DST_SEL_W (DSW)	[20:18]	enum(3)
	Specifies which element of the result to write to DST.XYZW. Can be used to mask elements when writing to the destination GPR. 0 SEL_X: use X element. 1 SEL_Y: use Y element. 2 SEL_Z: use Z element. 3 SEL_W: use W element. 4 SEL_0: use constant 0.0. 5 SEL_1: use constant 1.0. 6 Reserved. 7 SEL_MASK: mask this element.	
USE_CONST_FIELDS (UCF)	21	int(1)
	0 Use format given in this instruction. 1 Use format given in the fetch constant instead of in this instruction.	

Fetch Through a Vertex Cache Clause Doubleword 1 GPR (Cont.)

DATA_FORMAT	[27:22]	int(6)

Specifies vertex data format (ignored if USE_CONST_FIELDS is set).

Note that in the following list, numbers 3, 18, 20, 21, 23, 24, 44, 45, 46, and 54 through 62 are for vertex fetches; all others are for texture fetches.

0	FMT_INVALID	32	FMT_16_16_16_16_FLOAT
1	FMT_8	33	FMT_RESERVED_33
2	FMT_4_4	34	FMT_32_32_32_32
3	FMT_3_3_2	35	FMT_32_32_32_32_FLOAT
4	FMT_RESERVED_4	36	FMT_RESERVED_36
5	FMT_16	37	FMT_1
6	FMT_16_FLOAT	38	FMT_1_REVERSED
7	FMT_8_8	39	FMT_GB_GR
8	FMT_5_6_5	40	FMT_BG_RG
9	FMT_6_5_5	41	FMT_32_AS_8
10	FMT_1_5_5_5	42	FMT_32_AS_8_8
11	FMT_4_4_4_4	43	FMT_5_9_9_9_SHAREDEXP
12	FMT_5_5_5_1	44	FMT_8_8_8
13	FMT_32	45	FMT_16_16_16
14	FMT_32_FLOAT	46	FMT_16_16_16_FLOAT
15	FMT_16_16	47	FMT_32_32_32
16	FMT_16_16_FLOAT	48	FMT_32_32_32_FLOAT
17	FMT_8_24	49	FMT_BC1
18	FMT_8_24_FLOAT	50	FMT_BC2
19	FMT_24_8	51	FMT_BC3
20	FMT_24_8_FLOAT	52	FMT_BC4
21	FMT_10_11_11	53	FMT_BC5
22	FMT_10_11_11_FLOAT	54	FMT_APC0
23	FMT_11_11_10	55	FMT_APC1
24	FMT_11_11_10_FLOAT	56	FMT_APC2
25	FMT_2_10_10_10	57	FMT_APC3
26	FMT_8_8_8_8	58	FMT_APC4
27	FMT_10_10_10_2	59	FMT_APC5
28	FMT_X24_8_32_FLOAT	60	FMT_APC6
29	FMT_32_32	61	FMT_APC7
30	FMT_32_32_FLOAT	62	FMT_CTX1
31	FMT_16_16_16_16	63	FMT_RESERVED_63

NUM_FORMAT_ALL (NFA)	[29:28]	enum(2)

Format of returning data (N is the number of bits derived from DATA_FORMAT and gamma) (ignored if USE_CONST_FIELDS is set).

0 NUM_FORMAT_NORM: repeating fraction number (0.N) with range [0,1] if unsigned, or [-1, 1] if signed.

1 NUM_FORMAT_INT: integer number (N.0) with range [0, 2^N] if unsigned, or [-2^M, 2^M] if signed (M = N - 1).

2 NUM_FORMAT_SCALED: integer number stored as a S23E8 floating-point representation (1 == 0x3F800000).

FORMAT_COMP_ALL (FCA)	30	enum(1)

Specifies sign of source elements (ignored if USE_CONST_FIELDS = 1).

0 FORMAT_COMP_UNSIGNED

1 FORMAT_COMP_SIGNED

Fetch Through a Vertex Cache Clause Doubleword 1 GPR (Cont.)

	SRF_MODE_ALL (SMA)	31	enum(1)
		Mapping to use when converting from signed repeating fraction (SRF) to float (ignored if USE_CONST_FIELDS is set).	
		0	SRF_MODE_ZERO_CLAMP_MINUS_ONE: data represents numbers in the range [-1.0, 1.0] in increments of 1/(2^numBits-1-1). For example, 4 bit numbers use increments of 1/7. The -1 has two encodings.
		1	SRF_MODE_NO_ZERO: OpenGL format lacking representation for zero. Data represents numbers in the range [-1.0, 1.0] with no representation of zero and only one representation of -1. Increments in 2/(2^numBits-1-1). For example, 4 bit numbers use increments of 2/15.
Related	VTX_WORD0		
	VTX_WORD1_SEM		
	VTX_WORD2		

Fetch Through a Vertex Cache Clause Doubleword 1 Semantic-Table Specification

Instructions **VTX_WORD1_SEM**

Description This doubleword is part of the 128-bit 4-tuple formed by VTX_WORD0, VTX_WORD1_{SEM, GPR}, VTX_WORD2, plus a doubleword filled with zeros, as described in Chapter 5. Each instruction using this format 4-tuple has either a SEM or GPR format (not both) for its second doubleword. The instructions are specified in the VTX_WORD0 doubleword. This SEM format is used by SEMANTIC instructions that specify a destination using a semantic table.

Opcode

Field Name	*Bits*	*Format*
SEMANTIC_ID	[7:0]	int(8)
	Specifies an eight-bit semantic ID used to look up the destination GPR in the semantic table. The semantic table is written by the host and maintained by hardware.	
Reserved	8	Reserved. Set to 0.
DST_SEL_X (DSX) DST_SEL_Y (DSY) DST_SEL_Z (DSZ) DST_SEL_W (DSW)	[11:9] [14:12] [17:15] [20:18]	enum(3) enum(3) enum(3) enum(3)
	Specifies which element of the result to write to DST.XYZW. Can be used to mask elements when writing to the destination GPR. 0 SEL_X: use X element. 1 SEL_Y: use Y element. 2 SEL_Z: use Z element. 3 SEL_W: use W element. 4 SEL_0: use constant 0.0. 5 SEL_1: use constant 1.0. 6 Reserved. 7 SEL_MASK: mask this element.	
USE_CONST_FIELDS (UCF)	21	int(1)
	0 Use format given in this instruction. 1 Use format given in the fetch constant instead of in this instruction.	
DATA_FORMAT	[27:22]	int(6)
	Specifies vertex data format (ignored if USE_CONST_FIELDS is set). See list for DATA_FORMAT [27:22] in VTX_WORD1_GPR, page 10-47.	
NUM_FORMAT_ALL (NFA)	[29:28]	enum(2)
	Format of returning data (N is the number of bits derived from DATA_FORMAT and gamma) (ignored if USE_CONST_FIELDS is set). 0 NUM_FORMAT_NORM: repeating fraction number (0.N) with range [0,1] if unsigned, or [-1, 1] if signed. 1 NUM_FORMAT_INT: integer number (N.0) with range [0, 2^N] if unsigned, or [-2^M, 2^M] if signed (M = N - 1). 2 NUM_FORMAT_SCALED: integer number stored as a S23E8 floating-point representation (1 == 0x3F800000).	
FORMAT_COMP_ALL (FCA)	30	enum(1)
	Specifies sign of source elements (ignored if USE_CONST_FIELDS = 1). 0 FORMAT_COMP_UNSIGNED 1 FORMAT_COMP_SIGNED	

Fetch Through a Vertex Cache Clause Doubleword 1 Semantic-Table Specification (Cont.)

	SRF_MODE_ALL (SMA)	31	enum(1)
		Mapping to use when converting from signed repeating fraction (SRF) to float (ignored if USE_CONST_FIELDS is set).	
		0	SRF_MODE_ZERO_CLAMP_MINUS_ONE: data represents numbers in the range [-1.0, 1.0] in increments of $1/(2^{\text{numBits-1}}-1)$. For example, 4 bit numbers use increments of 1/7. The -1 has two encodings.
		1	SRF_MODE_NO_ZERO: OpenGL format lacking representation for zero. Data represents numbers in the range [-1.0, 1.0] with no representation of zero and only one representation of -1. Increments in $2/(2^{\text{numBits-1}}-1)$. For example, 4 bit numbers use increments of 2/15.
Related	VTX_WORD0		
	VTX_WORD1		
	VTX_WORD1_GPR		
	VTX_WORD2		

Fetch Through a Vertex Cache Clause Doubleword 2

Instructions **VTX_WORD2**

Description This is the high-order (most-significant) doubleword in the 128-bit 4-tuple formed by VTX_WORD0, VTX_WORD1_{SEM, GPR}, VTX_WORD2, plus a doubleword filled with zeros, as described in Chapter 5.

Opcode

Field Name	*Bits*	*Format*
OFFSET	[15:0]	int(16)
	Offset to begin reading from. Byte-aligned.	
ENDIAN_SWAP (ES)	[17:16]	enum(2)
	Endian control (ignored if USE_CONST_FIELDS is set). 0 ENDIAN_NONE: no endian swap (XOR by 0). 1 ENDIAN_8IN16: 8-bit swap in 16 bit word (XOR by 1): AABBCCDD -> BBAADDCC. 2 ENDIAN_8IN32: 8-bit swap in a 32-bit word (XOR by 3): AABBCCDD -> DDCCBBAA.	
CONST_BUF_NO_STRIDE (CBNS)	18	int(1)
	0 Do not force stride to zero for constant buffer fetches that use absolute addresses. 1 Force stride to zero for constant buffer fetches that use absolute addresses.	
MEGA_FETCH (MF)	19	int(1)
	0 This instruction is a mini-fetch. 1 This instruction is a mega-fetch.	
ALT_CONST	20	int(1)
	0 This ALU clause does not use constants from an alternate thread. 1 This ALU clause uses constants from an alternate thread type: PS->VS, VS->GS, GS->VS, ES->GS. Note that ES and VS share constants.	
BUFFER_INDEX_MODE (BIM)	[22:21]	enum(2)
	Specifies whether to add index0 or index1 to the vertex buffer resource. ID#. 0 CF_INDEX_NONE: do not index the constant buffer. 1 CF_INDEX_0: add index0 to the constant (CB#/T#/S#/UAV#) number. 2 CF_INDEX_1: add index1 to the constant (CB#/T#/S#/UAV#) number. 3 CF_INVALID: invalid.	
Reserved	[31:21]	Reserved.

Related

VTX_WORD0
VTX_WORD1_GPR
VTX_WORD1_SEM

10.4 Instructions for Fetches Through a Texture Cache Clause

Fetches through a texture cache clause are initiated using the CF_WORD[0,1] formats, described in Section 10.1, on page 10-2. After the clause is initiated, the instructions below can be issued. Graphics programs typically use fetches through a texture cache clause to load texture data from memory into GPRs. General-computing programs typically use fetches through a texture cache clause as conventional data loads from memory into GPRs that are unrelated to textures.

All microcode formats for fetches through a texture cache clause are 96 bits wide, formed by three doublewords, and padded with zeros to 128 bits.

Fetch Through a Texture Cache Clause Doubleword 0

Instructions **TEX_WORD0**

Description This is the low-order (least-significant) doubleword in the 128-bit 4-tuple formed by TEX_WORD[0,1,2] plus a doubleword filled with zeros, as described in Chapter 6.

Opcode

Field Name	Bits	Format
TEX_INST	[4:0]	enum(5)

Instruction.

0 Reserved.
1 Reserved.
2 Reserved.
3 TEX_INST_LD: fetch data, address XYZL are uint32.
4 TEX_INST_GET_TEXTURE_RESINFO: retrieve width, height, depth, number of mipmap levels.
5 TEX_INST_GET_NUMBER_OF_SAMPLES: retrieve width, height, depth, number of samples of an MSAA surface.
6 TEX_INST_GET_COMP_TEX_LOD: X = clamped LOD; Y = non-clamped.
7 TEX_INST_GET_GRADIENTS_H: slopes relative to horizontal: X = dx/dh, Y = dy/dh, Z = dz/dh, W = dw/dh.
8 TEX_INST_GET_GRADIENTS_V: slopes relative to vertical: X = dx/dv, Y = dy/dv, Z = dz/dv, W = dw/dv.
9 TEX_INST_SET_TEXTURE_OFFSETS: sets texture offsets from a GPR for use with GATHER4_O and GATHER4_C_O.
10 TEX_INST_KEEP_GRADIENTS: Compute gradients from coordinates and store them.
11 TEX_INST_SET_GRADIENTS_H: XYZ set horizontal gradients.
12 TEX_INST_SET_GRADIENTS_V: XYZ set vertical gradients.
13 Reserved.
14 Reserved.
15 Reserved.
16 TEX_INST_SAMPLE
17 TEX_INST_SAMPLE_L
18 TEX_INST_SAMPLE_LB
19 TEX_INST_SAMPLE_LZ
20 TEX_INST_SAMPLE_G
21 TEX_INST_GATHER4: fetches unfiltered texels from a bilinear sample, packs into xyzw.
22 TEX_INST_SAMPLE_G_LB
23 TEX_INST_GATHER4_O
24 TEX_INST_SAMPLE_C
25 TEX_INST_SAMPLE_C_L
26 TEX_INST_SAMPLE_C_LB
27 TEX_INST_SAMPLE_C_LZ
28 TEX_INST_SAMPLE_C_G
29 TEX_INST_GATHER4_C
30 TEX_INST_SAMPLE_C_G_LB
31 TEX_INST_GATHER4_C_O

Fetch Through a Texture Cache Clause Doubleword 0 (Cont.)

INST_MOD	[6:5]	int(2)

Instruction modifier. Different meaning for different TEX_INSTs. Used for: LD, GetGradientsH/V, and Gather4.

Opcode	Instruction	Modifier Description
3	LD	Determines the type of load operation to be done. 0 ld (Normal load operation.) 1 ldfptr (Perform special load operation to retrieve fragment pointers for a MSAA surface.) 2-3 Reserved.
7	GetGradientsH	Determines the type of GetGradientsH operation. 0 Use coarse derivative calculation (all pixels in the quad use the same gradients). 1 Use fine derivative calculation (each pixel in the quad has a unique gradient). 2-3 Reserved.
8	GetGradientsV	Determines the type of GetGradientsV operation. 0 Use coarse derivative calculation (all pixels in the quad use the same gradients). 1 Use fine derivative calculation (each pixel in the quad has a unique gradient). 2-3 Reserved.
21	Gather4	Determines the element to be retrieved by the Gather4 operation. 0 Returns the X element. 1 Returns the Y element. 2 Returns the Z element. 3 Returns the W element.

FETCH_WHOLE_QUAD (FWQ)	7	int(1)

0 Texture instruction can ignore inactive pixels.

1 Texture instruction fetches data for all pixels in any quad which as at least one pixel both active and valid. Result can be used as source coordinate of a dependent read.

RESOURCE_ID	[15:8]	int(8)

Surface ID to read from (specifies the buffer address, size, and format). 160 available for GS and PS programs; 176 shared across FS and VS.

SRC_GPR	[22:16]	int(7)

Source GPR address to get the texture lookup address from.

SRC_REL (SR)	23	enum(1)

Indicate whether source address is absolute or relative to an index.

0 Absolute: no relative addressing.

1 Relative: add current loop index (aL) value to this address.

ALT_CONST (AC)	24	int(1)

0 This ALU clause does not use constants from an alternate thread.

1 This ALU clause uses constants from an alternate thread type: PS->VS, VS->GS, GS->VS, ES->GS. Note that ES and VS share constants. Has no effect on HS, LS, or CS.

Fetch Through a Texture Cache Clause Doubleword 0 (Cont.)

	RESOURCE_INDEX_MODE (RIM)	[26:25]	enum(2) Specifies whether to add index0 or index1 to the resource ID#. 0 CF_INDEX_NONE: do not index the constant buffer. 1 CF_INDEX_0: add index0 to the constant (CB#/T#/S#/UAV#) number. 2 CF_INDEX_1: add index1 to the constant (CB#/T#/S#/UAV#) number. 3 CF_INVALID: invalid.
	SAMPLER_INDEX_MODE (SIM)	[28:27]	enum(2) Specifies whether to add index0 or index1 to the sampler ID#. 0 CF_INDEX_NONE: do not index the constant buffer. 1 CF_INDEX_0: add index0 to the constant (CB#/T#/S#/UAV#) number. 2 CF_INDEX_1: add index1 to the constant (CB#/T#/S#/UAV#) number. 3 CF_INVALID: invalid.
	Reserved	[31:29]	Reserved.
Related	TEX_WORD1 TEX_WORD2		

Fetch Through a Texture Cache Clause Doubleword 1

Instructions **TEX_WORD1**

Description This is the middle doubleword in the 128-bit 4-tuple formed by TEX_WORD[0,1,2] plus a doubleword filled with zeros, as described in Chapter 6.

Opcode

Field Name	*Bits*	*Format*
DST_GPR	[6:0]	int(7)
	Destination GPR address to which result is written.	
DST_REL (DR)	7	enum(1)
	Specifies whether destination address is absolute or relative to an index. 0 Absolute: no relative addressing. 1 Relative: add current loop index (aL) value to this address.	
Reserved	8	Reserved.
DST_SEL_X (DSX)	[11:9]	enum(3)
DST_SEL_Y (DSY)	[14:12]	enum(3)
DST_SEL_Z (DSZ)	[17:15]	enum(3)
DST_SEL_W (DSW)	[20:18]	enum(3)
	Specifies which element of the result to write to DST.XYZW. Can be used to mask elements when writing to destination GPR. 0 SEL_X: use X element. 1 SEL_Y: use Y element. 2 SEL_Z: use Z element. 3 SEL_W: use W element. 4 SEL_0: use constant 0.0. 5 SEL_1: use constant 1.0. 6 Reserved. 7 SEL_MASK: mask this element.	
LOD_BIAS	[27:21]	int(7)
	Constant level-of-detail (LOD) bias to add to the computed bias for this lookup. Twos-complement S3.4 fixed-point value with range [-4, 4).	
COORD_TYPE_X (CTX)	28	enum(1)
COORD_TYPE_Y (CTY)	29	enum(1)
COORD_TYPE_Z (CTZ)	30	enum(1)
COORD_TYPE_W (CTW)	31	enum(1)
	Specifies the type of source element. 0 TEX_UNNORMALIZED: Element is in [0, dim); repeat and mirror modes unavailable. 1 TEX_NORMALIZED: Element is in [0,1]; repeat and mirror modes available.	

Related
TEX_WORD0
TEX_WORD2

Fetch Through a Texture Cache Clause Doubleword 2

Instructions **TEX_WORD2**

Description This is the high-order (most-significant) doubleword in the 128-bit 4-tuple formed by TEX_WORD[0,1,2] plus a doubleword filled with zeros, as described in Chapter 6.

Opcode

Field Name	*Bits*	*Format*
OFFSET_X	[4:0]	int(5)
	Value added to X element of texel address before sampling (in texel space). S3.1 fixed-point value ranging from [-8, 8).	
OFFSET_Y	[9:5]	int(5)
	Value added to Y element of texel address before sampling (in texel space). S3.1 fixed-point value ranging from [-8, 8).	
OFFSET_Z	[14:10]	int(5)
	Value added to Z element of texel address before sampling (in texel space). S3.1 fixed-point value ranging from [-8, 8).	
SAMPLER_ID	[19:15]	int(5)
	Sampler ID to use (specifies filter options, etc.). Value in the range [0, 17].	
SRC_SEL_X (SSX)	[22:20]	enum(3)
SRC_SEL_Y (SSY)	[25:23]	enum(3)
SRC_SEL_Z (SSZ)	[28:26]	enum(3)
SRC_SEL_W (SSW)	[31:29]	enum(3)
	Specifies the element source for SRC.XYZW. 0 SEL_X: use X element. 1 SEL_Y: use Y element. 2 SEL_Z: use Z element. 3 SEL_W: use W element. 4 SEL_0: use constant 0.0. 5 SEL_1: use constant 1.0.	

Related TEX_WORD0
TEX_WORD1

10.5 Memory Read Instructions

The following are instructions to read from these buffer types:

- scratch
- reduction
- global

Memory-Read Clause Instruction Doubleword 0

Instructions **MEM_RD_WORD0**

Description Memory read instruction doubleword 0.

Opcode

Field Name	Bits	Format
MEM_INST	[4:0]	enum(5)
	Opcode, borrowed from the instruction set for a fetch through a vertex cache clause. Must be MEM_INST_MEM. The only legal value is 2: MEM_INST_MEM: memory read/write. All other values are illegal. This opcode is exclusively for MEM_RD_WORD* and MEM_GDS_WORD* encodings.	
ELEM_SIZE	[6:5]	int(2)
	Number of dwords per element, minus one. This field is interpreted as a value: 1,2, or 4 (3 is illegal). The value from INDEX_GPR is multiplied by this factor, if applicable. Normally, ELEM_SIZE = four dwords for scratch, one dword for other types.	
FETCH_WHOLE_QUAD	7	int(1)
	0 Texture instruction can ignore inactive pixels. 1 Texture instruction must fetch data for all pixels in any quad that has at least one pixel valid. The result can be used as a source coordinate of a dependent read. Set this only in PS stage.	
MEM_OP	[10:8]	enum(3)
	Sub-opcode for scratch and scatter memory reads. The sub-opcode must match the CF_INST opcode used to issue the clause (see value descriptions below). 0 MEM_RD_SCRATCH: Scratch (temp) buffer read. Use only in CF_INST_VC/TC[_ACK] clauses. 2 MEM_RD_SCATTER: Scatter (mem-export) buffer read. Use only in CF_INST_VC/TC[_ACK] clauses. 4 Reserved. 5 Reserved. 6 Reserved. 7 Reserved.	
UNCACHED	11	int(1)
	Uncached (cache-bypass) read. When writing and reading in one kernel pass, this bit must be set.	
INDEXED	12	int(1)
	Indexed access (set) or not (cleared). Indexed includes source-GPR in address calculation.	
RESERVED	[15:13]	Reserved.
MEM_REQ_SIZE	[14:13]	int(2)
	Must be cleared for Evergreen family and later products.	
Reserved	15	Reserved.
SRC_GPR	[22:16]	int(7)
	Source GPR address from which to get fetch address.	

Memory-Read Clause Instruction Doubleword 0 (Cont.)

SRC_REL	23	enum(1)	none
	Indicate whether source address is absolute or relative to an index. 0 Absolute: no relative addressing. 1 Relative: add current loop index value to this address.		
SRC_SEL_X	[25:24]	enum(2)	none
	Indicate which component of src to use for the fetch address. 0 SEL_X: use X component 1 SEL_Y: use Y component 2 SEL_Z: use Z component 3 SEL_W: use W component		
BURST_CNT	[29:26]	int(4)	none
	Burst count 0 indicates one read, 15 indicates 16 reads. ARRAY_BASE and DST_GPR are incremented for each step in the burst.		
Reserved	[31:30]		Reserved.

Related MEM_RD_WORD1, MEM_RD_WORD2.

Memory-Read Instruction Doubleword 1

Instructions **MEM_RD_WORD1**

Description Memory read instruction doubleword 1.

Opcode

Field Name	*Bits*	*Format*
DST_GPR	[6:0]	int(7)
	Destination GPR address to which the result is written.	
DST_REL	7	enum(1)
	Indicate whether destination address is absolute or relative to an index. 0 Absolute: no relative addressing. 1 Relative: add current loop index value to this address.	
Reserved	8	Reserved.
DST_SEL_X	[11:9]	enum(3)
DST_SEL_Y	[14:12]	enum(3)
DST_SEL_Z	[17:15]	enum(3)
DST_SEL_W	[20:18]	enum(3)
	Indicate which component of the result to write to dst.XYZW. Can be used to mask out components when writing to destination GPR. 0 SEL_X: use X component. 1 SEL_Y: use Y component. 2 SEL_Z: use Z component. 3 SEL_W: use W component. 4 SEL_0: use constant 0.0. 5 SEL_1: use constant 1.0. 6 Reserved. 7 SEL_MASK: mask out this component.	
Reserved	21	Reserved.
DATA_FORMAT	[27:22]	int(6)
	Indicate vertex data format. See list for DATA_FORMAT [27:22] in VTX_WORD1_GPR, page 10-47.	
NUM_FORMAT_ALL	[29:28]	enum(2)
	Format of returning data (N is the number of bits derived from DATA_FORMAT and gamma). 0 NUM_FORMAT_NORM: repeating fraction number (0.N) with range [0, 1] if unsigned, or [-1, 1] if signed. 1 NUM_FORMAT_INT: integer number (N.0) with range [0, 2^N] if unsigned, or [-2^M, 2^M] if signed (M = N - 1). 2 NUM_FORMAT_SCALED: integer number stored as a S23E8 floating-point representation (1 == 0x3F800000).	
FORMAT_COMP_ALL	30	enum(1)
	Indicate if source components are signed. 0 FORMAT_COMP_UNSIGNED. 1 FORMAT_COMP_SIGNED.	

Memory-Read Instruction Doubleword 1 (Cont.)

SRF_MODE_ALL	31	enum(0)
	Mapping to use when converting from signed repeating fraction (SRF) to float.	
	0	SRF_MODE_ZERO_CLAMP_MINUS_ONE: data represents numbers in the range [-1.0, 1.0] in increments of $1/(2^{\wedge numBits-1}-1)$. For example, 4 bit numbers use increments of 1/7. The -1 has two encodings.
	1	SRF_MODE_NO_ZERO: OpenGL format lacking representation for zero. Data represents numbers in the range [-1.0, 1.0] with no representation of zero and only one representation of -1. Increments in $2/(2^{\wedge numBits-1}-1)$. For example, 4 bit numbers use increments of 2/15.

Related MEM_RD_WORD0, MEM_RD_WORD2.

Memory-Read Clause Instruction Doubleword 2

Instructions **MEM_RD_WORD2**

Description Memory read clause instruction doubleword 2.

Opcode

Field Name	*Bits*	*Format*	
ARRAY_BASE	[12:0]	int(13)	• For scratch or reduction input or output, this is the base address of the array in multiples of four doublewords [0,32764]. • For stream or ring output, this is the base address of the array in multiples of one doubleword [0,8191].
Reserved	[15:13]		Reserved.
ENDIAN_SWAP	[17:16]	enum(2)	Endian control (ignored if USE_CONST_FIELDS = 1). 0 ENDIAN_NONE: no endian swap (XOR by 0) 1 ENDIAN_8IN16: Eight-bit swap in 16-bit word (XOR by 1): AABBCCDD -> BBAADDCC 2 ENDIAN_8IN32: Eight-bit swap in 32-bit word (XOR by 3): AABBCCDD -> DDCCBBAA
Reserved	[19:18]		Reserved.
ARRAY_SIZE	[31:20]	int(12)	The array size is calculated in the following way: Four element sizes (ELEMSIZE) are available; these specify 1, 2, or 4 dwords. ELEMSIZE=0 represents one dword, with possible values up to 4096; ELEMSIZE=3 represents four dwords, with possible values up to 16,384. Used only for scratch reads (no effect on scatter). Also see the ARRAY_SIZE field in the CF_ALLOC_EXPORT_WORD1_BUF instruction, on page 10-19.

Related MEM_RD_WORD0, MEM_RD_WORD1.

10.6 Global Data Share Read/Write Instructions

The section describes instructions that transfer data between GPRs and global data share memory.

Memory: Global Data-Share Instruction Doubleword 0

Instructions **MEM_GDS_WORD0**

Description Global memory data share instruction word 0.

Opcode

Field Name	Bits	Format
MEM_INST	[4:0]	enum(5)
	The only legal value is 2: MEM_INST_MEM: memory read/write. All other values are illegal. Use only for MEM_RD_WORD* and MEM_GDS_WORD* encodings.	
Reserved	[7:5]	Reserved.
MEM_OP	[10:8]	enum(3)
	Sub-opcode for GDS read/writes or TF-buffer writes. The subopcode must match the CF_INST opcode used to issue the clause, as indicated below. 0 Reserved. 1 Reserved. 2 Reserved. 4 MEM_GDS: Global data sharing read or write. Use only in CF_INST_GDS clause. 5 MEM_TF_WRITE: Tesselation buffer write. Use only in CF_INST_GDS clause. 6 Reserved. 7 Reserved.	
SRC_GPR	[17:11]	int(7)
	Source GPR (supplies data to GDS or TF buffer). TF_write: X=(tf_idx + tf_base), Y=tf_lod, Z=unused.	
SRC_REL_MODE (SRM)	[19:18]	enum(2)
	Indicate whether source-GPR is absolute or relative to an index or global GPR. 0 REL_NONE: Normal mode - no offset applied to GPR address. 1 REL_LOOP: add current loop index value. 2 REL_GLOBAL: treat GPR address as absolute, not thread-relative.	
SRC_SEL_X	[22:20]	enum(3)
SRC_SEL_Y	[25:23]	enum(3)
SRC_SEL_Z	[28:26]	enum(3)
	Select source component from GPR.xzyw01. Set unused components to 0. 0 SEL_X: use X component. 1 SEL_Y: use Y component. 2 SEL_Z: use Z component. 3 SEL_W: use W component. 4 SEL_0: use constant 0.0. 5 SEL_1: use constant 1.0. 6 Reserved. 7 Reserved.	
Reserved	[31:29]	Reserved.

Related MEM_GDS_WORD1, MEM_GDS_WORD2.

Memory: Global Data-Share Instruction Doubleword 1

Instructions **MEM_GDS_WORD1**

Description Global memory data share instruction dword 1.

Opcode

Field Name	*Bits*	*Format*
DST_GPR	[6:0]	int(7)
	For GDS operations that return data, this specifies to which GPR data is returned. A return of one value is written the X element. If two values are returned, the results are written to the X and Y elements. This is ignored if no value is returned or if this is a TF_WRITE.	
DST_REL_MODE	[8:7]	enum(2)
(DRM)	Indicate whether the source GPR is absolute or relative to an index, or global GPR. This is ignored if there is no return value or if this is a tessellation factor write. 0 REL_NONE: Normal mode; no offset applied to GPR address. 1 REL_LOOP: add current loop index value. 2 REL_GLOBAL: treat GPR address as absolute, not thread-relative.	
GDS_OP	[14:9]	enum(6)
	Global data share operation. Ignored for tessellation factor write. 0 DS_INST_ADD: OP(dst,src, ...) dst=src0_sel, src=src1_sel. 1A1D ADD(dst,src) : DS(dst) += src. dst is src0_sel, src is src1_sel. 1 DS_INST_SUB: 1A1D SUB(dst,src) : DS(dst) = DS(dst) - src. 2 DS_INST_RSUB: 1A1D RSUB(dst,src): DS(dst) = src - DS(dst). 3 DS_INST_INC: 1A1D INC(dst) : (DS(dst)>=src) ? DS(dst) = 0 : DS(dst)++. 4 DS_INST_DEC: 1A1D DEC(dst) : DS(dst) = ((DS(dst)==0) || (DS(dst)>src)) ? src : DS(dst)-1. 5 DS_INST_MIN_INT: 1A1D MIN(dst,src) : DS(dst) = min (DS(dst),src). 6 DS_INST_MAX_INT: 1A1D MAX(dst,src) : DS(dst) = max(DS(dst),src). 7 DS_INST_MIN_UINT: 1A1D MIN(dst,src) : DS(dst) = min (DS(dst),src). 8 DS_INST_MAX_UINT: 1A1D MAX(dst,src) : DS(dst) = max(DS(dst),src) 9 DS_INST_AND: 1A1D AND(dst,src) : DS(dst) &= src. 10 DS_INST_OR: 1A1D OR(dst,src) : DS(dst) |= src. 11 DS_INST_XOR: 1A1D XOR(dst,src) : DS(dst) ^= src. 12 DS_INST_MSKOR: 1A2D MKSOR(dst,mask,src) : DS(dst) = ((DS(dst) & ~msk) | src). 13 DS_INST_WRITE: 1A1D WRITE(dst,src) : DS(dst) = src. 14 DS_INST_WRITE_REL: 1A2D WRITEREL(dst,src0,src1) : tmp = dst + sq_DS_idx_offset (offset in dwords). DS(dst) = src0, DS(tmp) = src1. 15 DS_INST_WRITE2: 1A2D WRITE2(dst,src0,src1) : tmp = dst+(sq_DS_idx_offset * 64). DS(dst) = src0, DS(tmp) = src1. 16 DS_INST_CMP_STORE: 1A2D CMP_STORE(dst, cmp, src) : DS(dst) = (DS(dst) == cmp) ? src : DS(dst). 17 DS_INST_CMP_STORE_SPF: 1A2D CMP_STORE_SPF(dst, cmp, src) : DS(dst) = (DS(dst) == cmp) ? src : DS(dst). 18 DS_INST_BYTE_WRITE: 1A1D BYTEWRITE (dst, src) : DS(dst) = src[7:0]. 19 DS_INST_SHORT_WRITE: 1A1D SHORTWRITE(dst, src) : DS(dst) = src[15:0] 32 DS_INST_ADD_RET: 1A1D ADD(dst,src) : OQA=DS(dst), DS(dst) += src. dst is src0_sel, src is src1_sel. 33 DS_INST_SUB_RET: 1A1D SUB(dst,src) : OQA=DS(dst), DS(dst) = DS(dst) - src.	

Memory: Global Data-Share Instruction Doubleword 1

		34	DS_INST_RSUB_RET: 1A1D RSUB(dst,src) : OQA=DS(dst), DS(dst) = src - DS(dst)
		35	DS_INST_INC_RET: 1A1D INC(dst) : OQA=DS(dst), (DS(dst)>=src) ? DS(dst) = 0 : DS(dst)++
		36	DS_INST_DEC_RET: 1A1D DEC(dst) : OQA=DS(dst), DS(dst) = ((DS(dst)==0) || (DS(dst)>src)) ? src : DS(dst)-1
		37	DS_INST_MIN_INT_RET: 1A1D MIN(dst,src) : OQA=DS(dst), DS(dst) = min (DS(dst),src)
		38	DS_INST_MAX_INT_RET: 1A1D MAX(dst,src) : OQA=DS(dst), DS(dst) = max(DS(dst),src)
		39	DS_INST_MIN_UINT_RET: 1A1D MIN(dst,src) : OQA=DS(dst), DS(dst) = min (DS(dst),src)
		40	DS_INST_MAX_UINT_RET: 1A1D MAX(dst,src) : OQA=DS(dst), DS(dst) = max(DS(dst),src)
		41	DS_INST_AND_RET: 1A1D AND(dst,src) : OQA=DS(dst), DS(dst) &= src
		42	DS_INST_OR_RET: 1A1D OR(dst,src) : OQA=DS(dst), DS(dst) |= src
		43	DS_INST_XOR_RET: 1A1D XOR(dst,src) : OQA=DS(dst), DS(dst) ^= src
		44	DS_INST_MSKOR_RET: 1A2D MSKOR(dst,msk,src) : OQA=DS(dst), DS(dst) = ((DS(dst) & ~msk) | src).
		45	DS_INST_XCHG_RET: 1A1D Exchange(dst,src) : OQA=DS(dst), DS(dst) = src
		46	DS_INST_XCHG_REL_RET: 1A2D ExchangeRel(dst,src0,src1) : tmp = dst + sq_DS_idx_offset. OQA=DS(dst), OQB=DS(tmp); DS(dst)=src0, DS(tmp)=src1
		47	DS_INST_XCHG2_RET: 1A2D Exchange2(dst,src0,src1) : tmp = dst + sq_DS_idx_offset*64. OQA=DS(dst), OQB=DS(tmp); DS(dst)=src0, DS(tmp)=src1
		48	DS_INST_CMP_XCHG_RET: 1A2D CompareExchange(dst,cmp,src) : OQA=DS(dst); (DS(dst)==cmp) ? DS(dst)=src : DS(dst)=DS(dst)
		49	DS_INST_CMP_XCHG_SPF_RET: 1A2D CompareExchangeSPF(dst,cmp,src) : OQA=DS(dst); (DS(dst)==cmp) ? DS(dst)=src : DS(dst)=DS(dst)
		50	DS_INST_READ_RET: 1A READ(dst) : OQA = DS(dst)
		51	DS_INST_READ_REL_RET: 1A READ_REL(dst) : tmp=dst+sq_DS_idx_offset; OQA=DS(dst), OQB=DS(tmp)
		52	DS_INST_READ2_RET: 2A READ2(dst0,dst1) : OQA=DS(dst0), OQB=DS(dst1)
		53	DS_INST_READWRITE_RET: 2A1D READWRITE(dst0,dst1,data) : OQA=DS(dst0), DS(dst1)=data
		54	DS_INST_BYTE_READ_RET: 1A BYTEREAD(dst) : OQA=SignExtend(DS(dst)[7:0])
		55	DS_INST_UBYTE_READ_RET: 1A UBYTEREAD(dst) : OQA={24'h0, DS(dst)[7:0]}
		56	DS_INST_SHORT_READ_RET: 1A SHORTREAD(dst) : OQA=SignExtend(DS(dst)[15:0]}
		57	DS_INST_USHORT_READ_RET: 1A USHORTREAD(dst) : OQA={16'h0, DS(dst)[15:0]}
		63	DS_INST_ATOMIC_ORDERED_ALLOC_RET: 1A GDS-only (intercepted by ordered alloc unit). This adds the 7 lsb of 1a to a hidden ordered append count in wave order and returns the pre-op value to the specified destination register. This opcode can only be used by GDS and with broadcast first set.
Reserved	15		Reserved.
SRC_GPR	[22:16]		int(7) Dword offset for GDS read or write. Ignored if for tessellation factor write.
Reserved	23		Reserved.

Memory: Global Data-Share Instruction Doubleword 1

UAV_INDEX_MODE	[25:24]	enum(2)
(UIM)	Indicate whether index0, index1, or nothing to the UAV_ID. 0 CF_INDEX_NONE: Do not index the constant buffer. 1 CF_INDEX_0: add index0 to the constant (CB#/T#/S#/UAV#) number. 2 CF_INDEX_1: add index1 to the constant (CB#/T#/S#/UAV#) number. 3 CF_INVALID: invalid.	
UAV_ID	[29:26]	
	Identifies append/consume count within group of a context. Do not use with TF.	
ALLOC_CONSUME	30	
(AC)	When set, accesses append/consume counter. Ignored for tessellation factor write and GDS with no return.	
BCAST_FIRST_REQ	31	
(BFR)	GDS processes and responds to the first active pixel only. Return data is broadcast to all pixels regardless of active status.	

Related MEM_GDS_WORD0, MEM_GDS_WORD2.

Memory: Data-Share Read Instruction Doubleword 0

Instructions **MEM_GDS_WORD2**

Description Global data share instruction doubleword 2.

DST_SEL_X	[2:0]	enum(3)
DST_SEL_Y	[5:3]	enum(3)
DST_SEL_Z	[8:6]	enum(3)
DST_SEL_W	[11:9]	enum(3)
	Select destination component from GPR.xzyw01.	
	0 SEL_X: use X component	
	1 SEL_Y: use Y component	
	2 SEL_Z: use Z component	
	3 SEL_W: use W component	
	4 SEL_0: use constant 0.0	
	5 SEL_1: use constant 1.0	
	6 Reserved.	
	7 SEL_MASK: mask out this component.	
Reserved	[31:12]	Reserved.

Related MEM_GDS_WORD0, MEM_GDS_WORD1.

Appendix A
Instruction Table

Instruction	Description	Page
MULADD_IEEE_PREV	Dependent Multiply Add With IEEE Rules	9-158
MULADD_M2	Floating-Point Multiply-Add, Multiply by 2	9-159
MULADD_M4	Floating-Point Multiply-Add, Multiply by 4	9-160
MULADD_PREV	Dependent Multiply-Add	9-161
MULADD_UINT24	24-Bit Unsigned Integer Multiply-Add	9-162
MULHI_INT	Signed Scalar Multiply, High-Order 32 Bits	9-163
MULHI_UINT	Unsigned Scalar Multiply, High-Order 32 Bits	9-164
MULHI_UINT24	24-Bit Unsigned Integer Multiply (High-Order)	9-165
MULLO_INT	Signed Scalar Multiply, Low-Order 32-Bits	9-166
MULLO_UINT	Unsigned Scalar Multiply, Low-Order 32-Bits	9-167
NOP	No Operation	9-168
NOT_INT	Bit-Wise NOT	9-169
OFFSET_TO_FLT	Four-Bit Signed Integer to 32-Bit Float	9-170
OR_INT	Logical Bit-Wise OR	9-171
PRED_SET_CLR	Predicate Counter Clear	9-172
PRED_SET_INV	Predicate Counter Invert	9-173
PRED_SET_POP	Predicate Counter Pop	9-174
PRED_SET_RESTORE	Predicate Counter Restore	9-175
PRED_SETE	Floating-Point Predicate Set If Equal	9-176
PRED_SETE_64	Floating-Point Predicate Set If Equal, 64-Bit	9-177
PREDE_INT	Integer Predicate Set If Equal	9-179
PRED_SETE_PUSH	Floating-Point Predicate Counter Increment If Equal	9-180
PRED_SETE_PUSH_INT	Integer Predicate Counter Increment If Equal	9-181
PRED_SETGE	Floating-Point Predicate Set If Greater Than Or Equal	9-182
PRED_SETGE_64	Floating-Point Predicate Set If Greater Than Or Equal, 64-Bit	9-183
PRED_SETGE_INT	Integer Predicate Set If Greater Than Or Equal	9-186
PRED_SETGE_PUSH	Predicate Counter Increment If Greater Than Or Equal	9-187
PRED_SETGE_PUSH_INT	Integer Predicate Counter Increment If Greater Than Or Equal	9-188
PRED_SETGE_UINT	Unsigned Integer Predicate Set If Greater Than Or Equal	9-189
PRED_SETGT	Floating-Point Predicate Set If Greater Than	9-190
PRED_SETGT_64	Floating-Point Predicate Set If Greater Than, 64-Bit	9-191
PRED_SETGT_INT	Integer Predicate Set If Greater Than	9-193

Instruction	Description	Page
GLOBAL_DS_READ	Global Data Share Read	9-285
Local Data Share (LDS) Instructions		
LDS_IDX_OP	LDS Indexed Operation	9-286

Glossary of Terms

Term	**Description**
*	Any number of alphanumeric characters in the name of a microcode format, microcode parameter, or instruction.
< >	Angle brackets denote streams.
[1,2)	A range that includes the left-most value (in this case, 1) but excludes the right-most value (in this case, 2).
[1,2]	A range that includes both the left-most and right-most values (in this case, 1 and 2).
{BUF, SWIZ}	One of the multiple options listed. In this case, the string *BUF* or the string *SWIZ*.
{x | y}	One of the multiple options listed. In this case, x or y.
0.0	A single-precision (32-bit) floating-point value.
0x	Indicates that the following is a hexadecimal number.
1011b	A binary value, in this example a 4-bit value.
29'b0	29 bits with the value 0.
7:4	A bit range, from bit 7 to 4, inclusive. The high-order bit is shown first.
ABI	Application Binary Interface.
absolute	A displacement that references the base of a code segment, rather than an instruction pointer. See relative.
active mask	A 1-bit-per-pixel mask that controls which pixels in a "quad" are really running. Some pixels might not be running if the current "primitive" does not cover the whole quad. A mask can be updated with a `PRED_SET*` ALU instruction, but updates do not take effect until the end of the ALU clause.
address stack	A stack that contains only addresses (no other state). Used for flow control. Popping the address stack overrides the instruction address field of a flow control instruction. The address stack is only modified if the flow control instruction decides to jump.
ACML	AMD Core Math Library. Includes implementations of the full BLAS and LAPACK routines, FFT, Math transcendental and Random Number Generator routines, stream processing backend for load balancing of computations between the CPU and GPU compute device.
aL (also AL)	Loop register. A three-component vector (x, y and z) used to count iterations of a loop.
allocate	To reserve storage space for data in an output buffer ("scratch buffer," "ring buffer," "stream buffer," or "reduction buffer") or for data in an input buffer ("scratch buffer" or "ring buffer") before exporting (writing) or importing (reading) data or addresses to, or from that buffer. Space is allocated only for data, not for addresses. After allocating space in a buffer, an "export" operation can be done.

Term	Description
ALU	Arithmetic Logic Unit. Responsible for arithmetic operations like addition, subtraction, multiplication, division, and bit manipulation on integer and floating point values. In stream computing, these are known as *stream cores*. ALU.[X,Y,Z,W] - an ALU that can perform four vector operations in which the four operands (integers or single-precision floating point values) do not have to be related. It performs “SIMD” operations. Thus, although the four operands need not be related, all four operations execute the same instruction. ALU.Trans - An ALU unit that can perform one ALU.Trans (transcendental, scalar) operation, or advanced integer operation, on one integer or single-precision floating-point value, and replicate the result. A single instruction can co-issue four ALU.Trans operations to an ALU.[X,Y,Z,W] unit and one (possibly complex) operation to an ALU.Trans unit, which can then replicate its result across all four component being operated on in the associated ALU.[X,Y,Z,W] unit.
AR	Address register.
aTid	Absolute thread id. It is the ordinal count of all threads being executed (in a draw call).
b	A bit, as in *1Mb* for one megabit, or *lsb* for least-significant bit.
B	A byte, as in *1MB* for one megabyte, or *LSB* for least-significant byte.
BLAS	Basic Linear Algebra Subroutines.
border color	Four 32-bit floating-point numbers (XYZW) specifying the border color.
branch granularity	The number of threads executed during a branch. For AMD GPUs, branch granularity is equal to wavefront granularity.
burst mode	The limited write combining ability. See write combining.
byte	Eight bits.
cache	A read-only or write-only on-chip or off-chip storage space.
CAL	Compute Abstraction Layer. A device-driver library that provides a forward-compatible interface to AMD Accelerated Parallel Processing compute devices. This lower-level API gives users direct control over the hardware: they can directly open devices, allocate memory resources, transfer data and initiate kernel execution. CAL also provides a JIT compiler for AMD IL.
CF	Control Flow.
cfile	Constant file or constant register.
channel	A component in a vector.
clamp	To hold within a stated range.
clause	A group of instructions that are of the same type (all stream core, all fetch, etc.) executed as a group. A clause is part of a CAL program written using the compute device ISA. Executed without pre-emption.
clause size	The total number of slots required for an stream core clause.
clause temporaries	Temporary values stored at GPR that do not need to be preserved past the end of a clause.
clear	To write a bit-value of 0. Compare “set”.

Term	Description
command	A value written by the host processor directly to the GPU compute device. The commands contain information that is not typically part of an application program, such as setting configuration registers, specifying the data domain on which to operate, and initiating the start of data processing.
command processor	A logic block in the R700 (HD4000-family of devices) that receives host commands, interprets them, and performs the operations they indicate.
component	(1) A 32-bit piece of data in a “vector”. (2) A 32-bit piece of data in an array. (3) One of four data items in a 4-component register.
compute device	A parallel processor capable of executing multiple threads of a kernel in order to process streams of data.
compute kernel	Similar to a pixel shader, but exposes data sharing and synchronization.
compute shader	Similar to a pixel shader, but exposes data sharing and synchronization.
compute unit pipeline	A hardware block consisting of five stream cores, one stream core instruction decoder and issuer, one stream core constant fetcher, and support logic. All parts of a compute unit pipeline receive the same instruction and operate on different data elements. Also known as “slice.”
constant buffer	Off-chip memory that contains constants. A constant buffer can hold up to 1024 four-component vectors. There are fifteen constant buffers, referenced as cb0 to cb14. An immediate constant buffer is similar to a constant buffer. However, an immediate constant buffer is defined within a kernel using special instructions. There are fifteen immediate constant buffers, referenced as icb0 to icb14.
constant cache	A constant cache is a hardware object (off-chip memory) used to hold data that remains unchanged for the duration of a kernel (constants). “Constant cache” is a general term used to describe constant registers, constant buffers or immediate constant buffers.
constant file	Same as constant register.
constant index register	Same as “AR” register.
constant registers	On-chip registers that contain constants. The registers are organized as four 32-bit component of a vector. There are 256 such registers, each one 128-bits wide.
constant waterfalling	Relative addressing of a constant file. See waterfalling.
context	A representation of the state of a device.
core clock	See engine clock. The clock at which the GPU compute device stream core runs.
CPU	Central Processing Unit. Also called host. Responsible for executing the operating system and the main part of the application. The CPU provides data and instructions to the GPU compute device.
CRs	Constant registers. There are 512 CRs, each one 128 bits wide, organized as four 32-bit values.
CS	Compute shader; commonly refered to as a compute kernel. A shader type, analogous to VS/PS/GS/ES.
CTM	Close-to-Metal. A thin, HW/SW interface layer. This was the predecessor of the AMD CAL.
DC	Data Copy Shader.

Term	**Description**
device	A *device* is an entire AMD Accelerated Parallel Processing compute device.
DMA	Direct-memory access. Also called DMA engine. Responsible for independently transferring data to, and from, the GPU compute device's local memory. This allows other computations to occur in parallel, increasing overall system performance.
double word	Dword. Two words, or four bytes, or 32 bits.
double quad word	Eight words, or 16 bytes, or 128 bits. Also called "octword."
domain of execution	A specified rectangular region of the output buffer to which threads are mapped.
DPP	Data-Parallel Processor.
dst.X	The X "slot" of an destination operand.
dword	Double word. Two words, or four bytes, or 32 bits.
element	A component in a vector.
engine clock	The clock driving the stream core and memory fetch units on the GPU compute device.
enum(7)	A seven-bit field that specifies an enumerated set of decimal values (in this case, a set of up to 27 values). The valid values can begin at a value greater than, or equal to, zero; and the number of valid values can be less than, or equal to, the maximum supported by the field.
event	A token sent through a pipeline that can be used to enforce synchronization, flush caches, and report status back to the host application.
export	To write data from GPRs to an output buffer (scratch, ring, stream, frame or global buffer, or to a register), or to read data from an input buffer (a "scratch buffer" or "ring buffer") to GPRs. The term "export" is a partial misnomer because it performs both input and output functions. Prior to exporting, an allocation operation must be performed to reserve space in the associated buffer.
FC	Flow control.
FFT	Fast Fourier Transform.
flag	A bit that is modified by a CF or stream core operation and that can affect subsequent operations.
FLOP	Floating Point Operation.
flush	To writeback and invalidate cache data.
FMA	Fused multiply add.
frame	A single two-dimensional screenful of data, or the storage space required for it.
frame buffer	Off-chip memory that stores a frame. Sometimes refers to the all of the GPU memory (excluding local memory and caches).
FS	Fetch subroutine. A global program for fetching vertex data. It can be called by a "vertex shader" (VS), and it runs in the same thread context as the vertex program, and thus is treated for execution purposes as part of the vertex program. The FS provides driver independence between the process of fetching data required by a VS, and the VS itself. This includes having a semantic connection between the outputs of the fetch process and the inputs of the VS.

Term	Description
function	A subprogram called by the main program or another function within an AMD IL stream. Functions are delineated by FUNC and ENDFUNC.
gather	Reading from arbitrary memory locations by a thread.
gather stream	Input streams are treated as a memory array, and data elements are addressed directly.
global buffer	GPU memory space containing the arbitrary address locations to which uncached kernel outputs are written. Can be read either cached or uncached. When read in uncached mode, it is known as mem-import. Allows applications the flexibility to read from and write to arbitrary locations in input buffers and output buffers, respectively.
global memory	Memory for reads/writes between threads. On ATI Radeon™ HD 5XXX series devices and later, atomic operations can be used to synchronize memory operations.
GPGPU	General-purpose compute device. A GPU compute device that performs general-purpose calculations.
GPR	General-purpose register. GPRs hold vectors of either four 32-bit IEEE floating-point, or four 8-, 16-, or 32-bit signed or unsigned integer or two 64-bit IEEE double precision data components (values). These registers can be indexed, and consist of an on-chip part and an off-chip part, called the “scratch buffer,” in memory.
GPU	Graphics Processing Unit. An integrated circuit that renders and displays graphical images on a monitor. Also called Graphics Hardware, Compute Device, and Data Parallel Processor.
GPU engine clock frequency	Also called 3D engine speed.
GPU compute device	A parallel processor capable of executing multiple threads of a kernel in order to process streams of data.
GS	Geometry Shader.
HAL	Hardware Abstraction Layer.
host	Also called CPU.
iff	If and only if.
IL	Intermediate Language. In this manual, the AMD version: AMD IL. A pseudo-assembly language that can be used to describe kernels for GPU compute devices. AMD IL is designed for efficient generalization of GPU compute device instructions so that programs can run on a variety of platforms without having to be rewritten for each platform.
in flight	A thread currently being processed.
instruction	A computing function specified by the *code* field of an IL_OpCode token. Compare “opcode”, “operation”, and “instruction packet”.
instruction packet	A group of tokens starting with an IL_OpCode token that represent a single AMD IL instruction.
int(2)	A 2-bit field that specifies an integer value.
ISA	Instruction Set Architecture. The complete specification of the interface between computer programs and the underlying computer hardware.
kcache	A memory area containing “waterfall” (off-chip) constants. The cache lines of these constants can be locked. The “constant registers” are the 256 on-chip constants.

Term	Description
kernel	A user-developed program that is run repeatedly on a stream of data. A parallel function that operates on every element of input streams. A device program is one type of kernel. Unless otherwise specified, an AMD Accelerated Parallel Processing compute device program is a kernel composed of a main program and zero or more functions. Also called Shader Program. This is not to be confused with an OS kernel, which controls hardware.
LAPACK	Linear Algebra Package.
LDS	Local Data Share. Part of local memory. These are read/write registers that support sharing between all threads in a group. Synchronization is required.
LERP	Linear Interpolation.
local memory fetch units	Dedicated hardware that a) processes fetch instructions, b) requests data from the memory controller, and c) loads registers with data returned from the cache. They are run at stream core or engine clock speeds. Formerly called texture units.
LOD	Level Of Detail.
loop index	A register initialized by software and incremented by hardware on each iteration of a loop.
lsb	Least-significant bit.
LSB	Least-significant byte.
MAD	Multiply-Add. A fused instruction that both multiplies and adds.
mask	(1) To prevent from being seen or acted upon. (2) A field of bits used for a control purpose.
MBZ	Must be zero.
mem-export	An AMD IL term random writes to the global buffer.
mem-import	Uncached reads from the global buffer.
memory clock	The clock driving the memory chips on the GPU compute device.
microcode format	An encoding format whose fields specify instructions and associated parameters. Microcode formats are used in sets of two or four. For example, the two mnemonics, `CF_DWORD[0,1]` indicate a microcode-format pair, `CF_DWORD0` and `CF_DWORD1`.
MIMD	Multiple Instruction Multiple Data. – Multiple SIMD units operating in parallel (Multi-Processor System) – Distributed or shared memory
MRT	Multiple Render Target. One of multiple areas of local GPU compute device memory, such as a “frame buffer”, to which a graphics pipeline writes data.
MSAA	Multi-Sample Anti-Aliasing.
msb	Most-significant bit.
MSB	Most-significant byte.
neighborhood	A group of four threads in the same wavefront that have consecutive thread IDs (Tid). The first Tid must be a multiple of four. For example, threads with Tid = 0, 1, 2, and 3 form a neighborhood, as do threads with Tid = 12, 13, 14, and 15.

Term	**Description**
normalized	A numeric value in the range [a, b] that has been converted to a range of 0.0 to 1.0 using the formula: normalized value = value/ (b–a+ 1)
oct word	Eight words, or 16 bytes, or 128 bits. Same as “double quad word”. Also referred to as octa word.
opcode	The numeric value of the *code* field of an “instruction”.
opcode token	A 32-bit value that describes the operation of an instruction.
operation	The function performed by an “instruction”.
PaC	Parameter Cache.
PCI Express	A high-speed computer expansion card interface used by modern graphics cards, GPU compute devices and other peripherals needing high data transfer rates. Unlike previous expansion interfaces, PCI Express is structured around point-to-point links. Also called PCIe.
PoC	Position Cache.
pop	Write “stack” entries to their associated hardware-maintained control-flow state. The POP_COUNT field of the CF_DWORD1 microcode format specifies the number of stack entries to pop for instructions that pop the stack. Compare “push.”
pre-emption	The act of temporarily interrupting a task being carried out on a computer system, without requiring its cooperation, with the intention of resuming the task at a later time.
processor	Unless otherwise stated, the AMD Accelerated Parallel Processing compute device.
program	Unless otherwise specified, a program is a set of instructions that can run on the AMD Accelerated Parallel Processing compute device. A device program is a type of kernel.
PS	Pixel Shader, aka pixel kernel.
push	Read hardware-maintained control-flow state and write their contents onto the stack. Compare pop.
PV	Previous vector register. It contains the previous four-component vector result from a ALU.[X,Y,Z,W] unit within a given clause.
quad	For a compute kernel, this consists of four consecutive work-items. For pixel and other shaders, this is a group of 2x2 threads in the NDRange. Always processed together.
rasterization	The process of mapping threads from the domain of execution to the SIMD engine. This term is a carryover from graphics, where it refers to the process of turning geometry, such as triangles, into pixels.
rasterization order	The order of the thread mapping generated by rasterization.
RAT	Random Access Target. Same as UAV. Allows, on DX11 hardware, writes to, and reads from, any arbitrary location in a buffer.
RB	Ring Buffer.
register	For a GPU, this is a 128-bit address mapped memory space consisting of four 32-bit components.
relative	Referencing with a displacement (also called offset) from an index register or the loop index, rather than from the base address of a program (the first control flow [CF] instruction).

Term	Description
render backend unit	The hardware units in a processing element responsible for writing the results of a kernel to output streams by writing the results to an output cache and transferring the cache data to memory.
resource	A block of memory used for input to, or output from, a kernel.
ring buffer	An on-chip buffer that indexes itself automatically in a circle.
Rsvd	Reserved.
sampler	A structure that contains information necessary to access data in a resource. Also called Fetch Unit.
SC	Shader Compiler.
scalar	A single data component, unlike a vector which contains a set of two or more data elements.
scatter	Writes (by uncached memory) to arbitrary locations.
scatter write	Kernel outputs to arbitrary address locations. Must be uncached. Must be made to a memory space known as the global buffer.
scratch buffer	A variable-sized space in off-chip-memory that stores some of the “GPRs”.
set	To write a bit-value of 1. Compare “clear”.
shader processor	Pre-OpenCL term that is now deprecated. Also called thread processor.
shader program	User developed program. Also called kernel.
SIMD	Pre-OpenCL term that is now deprecated. Single instruction multiple data unit. – Each SIMD receives independent stream core instructions. – Each SIMD applies the instructions to multiple data elements.
SIMD Engine	Pre-OpenCL term that is now deprecated. A collection of thread processors, each of which executes the same instruction each cycle.
SIMD pipeline	In OpenCL terminology: compute unit pipeline. Pre-OpenCL term that is now deprecated. A hardware block consisting of five stream cores, one stream core instruction decoder and issuer, one stream core constant fetcher, and support logic. All parts of a SIMD pipeline receive the same instruction and operate on different data elements. Also known as “slice.”
Simultaneous Instruction Issue	Input, output, fetch, stream core, and control flow per SIMD engine.
SKA	Stream KernelAnalyzer. A performance profiling tool for developing, debugging, and profiling stream kernels using high-level stream computing languages.
slot	A position, in an “instruction group,” for an “instruction” or an associated literal constant. An ALU instruction group consists of one to seven slots, each 64 bits wide. All ALU instructions occupy one slot, except double-precision floating-point instructions, which occupy either two or four slots. The size of an ALU clause is the total number of slots required for the clause.
SPU	Shader processing unit.
SR	Globally shared registers. These are read/write registers that support sharing between all wavefronts in a SIMD (not a thread group). The sharing is column sharing, so threads with the same thread ID within the wavefront can share data. All operations on SR are atomic.

Term	Description
src0, src1, etc.	In floating-point operation syntax, a 32-bit source operand. Src0_64 is a 64-bit source operand.
stage	A sampler and resource pair.
stream	A collection of data elements of the same type that can be operated on in parallel.
stream buffer	A variable-sized space in off-chip memory that stores an instruction stream. It is an output-only buffer, configured by the host processor. It does not store inputs from off-chip memory to the processor.
stream core	The fundamental, programmable computational units, responsible for performing integer, single, precision floating point, double precision floating point, and transcendental operations. They execute VLIW instructions for a particular thread. Each processing element handles a single instruction within the VLIW instruction.
stream operator	A node that can restructure data.
swizzling	To copy or move any component in a source vector to any element-position in a destination vector. Accessing elements in any combination.
thread	Pre-OpenCL term that is now deprecated. One invocation of a kernel corresponding to a single element in the domain of execution. An instance of execution of a shader program on an ALU. Each thread has its own data; multiple threads can share a single program counter.
thread group	Pre-OpenCL term that is now deprecated. It contains one or more thread blocks. Threads in the same thread-group but different thread-blocks might communicate to each through global per-SIMD shared memory. This is a concept mainly for global data share (GDS). A thread group can contain one or more wavefronts, the last of which can be a partial wavefront. All wavefronts in a thread group can run on only one SIMD engine; however, multiple thread groups can share a SIMD engine, if there are enough resources.
thread processor	Pre-OpenCL term that is now deprecated. The hardware units in a SIMD engine responsible for executing the threads of a kernel. It executes the same instruction per cycle. Each thread processor contains multiple stream cores. Also called shader processor.
thread-block	Pre-OpenCL term that is now deprecated. A group of threads which might communicate to each other through local per SIMD shared memory. It can contain one or more wavefronts (the last wavefront can be a partial wavefront). A thread-block (all its wavefronts) can only run on one SIMD engine. However, multiple thread blocks can share a SIMD engine, if there are enough resources to fit them in.
Tid	Thread id within a thread block. An integer number from 0 to Num_threads_per_block-1
token	A 32-bit value that represents an independent part of a stream or instruction.
UAV	Unordered Access View. Same as random access target (RAT). They allow compute shaders to store results in (or write results to) a buffer at any arbitrary location. On DX11 hardware, UAVs can be created from buffers and textures. On DX10 hardware, UAVs cannot be created from typed resources (textures).
uncached read/write unit	The hardware units in a GPU compute device responsible for handling uncached read or write requests from local memory on the GPU compute device.
vector	(1) A set of up to four related values of the same data type, each of which is an element. For example, a vector with four elements is known as a “4-vector” and a vector with three elements is known as a “3-vector”. (2) See “AR”. (3) See ALU.[X,Y,Z,W].

Term	Description
VLIW design	Very Long Instruction Word. – Co-issued up to 6 operations (5 stream cores + 1 FC); where FC = flow control. – 1.25 Machine Scalar operation per clock for each of 64 data elements – Independent scalar source and destination addressing
vTid	Thread ID within a thread group.
waterfall	To use the address register (AR) for indexing the GPRs. Waterfall behavior is determined by a “configuration registers.”
wavefront	Group of threads executed together on a single SIMD engine. Composed of quads. A full wavefront contains 64 threads; a wavefront with fewer than 64 threads is called a partial wavefront. Wavefronts that have fewer than a full set of threads are called partial wavefronts. For the HD4000-family of devices, there are 64. 32, 16 threads in a full wavefront. Threads within a wavefront execute in lockstep.
write combining	Combining several smaller writes to memory into a single larger write to minimize any overhead associated with write commands.

Index

B

C

D

L

M

N

O

P

Q

R

S

T

U

V

W