User Guide

AMD

R600-Family Instruction Set Architecture

December 2008

Revision 0.35

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

Preface

About This Document

This document describes the instruction set architecture (ISA) native to the R600 family of processors. It defines the instructions and formats accessible to programmers and compilers.

The document serves two purposes.

- It specifies the microcode (including the format of each type of microcode instruction) and the relevant program state (including how the program state interacts with the microcode). Some microcode fields are mutually dependent; not all possible settings for all fields are legal. This document specifies the valid combinations.
- It provides the programming guidelines for compiler writers to maximize processor performance.

For an understanding of the software environment in which the R600 family of processors operate, see the *ATI CTM Guide, Technical Reference Manual*, which describes the interface by which a host controls an R600-family processor. In this document, the term “R600” refers the entire family of R600 processors.

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 R600 family of processors’ hardware and programming environment (Chapter 1). Chapter 2 describes the organization of an R600-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 the vertex-fetch clauses. Chapter 6 describes the texture-fetch clauses. Chapter 7 describes instruction details, first by broad categories, and following this, in alphabetic order by mnemonic. Finally, Chapter 8 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 R600-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.
- *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

Contact Information

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

For questions concerning AMD Stream products, please email: streamcomputing@amd.com.

For questions about developing with AMD Stream, please email: streamdeveloper@amd.com.

You can learn more about AMD Stream at: http://ati.amd.com/technology/streamcomputing.

We also have a growing community of AMD Stream users! Come visit us at the AMD Stream Developer Forum to find out what applications other users are trying on their AMD Stream products!

http://forums.amd.com/devforum/categories.cfm?catid=328

Contents

Figures

Tables

Chapter 1 Introduction

The R600-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 R600-family processor.

Figure 1.1 shows a block diagram of the R600-family processors.

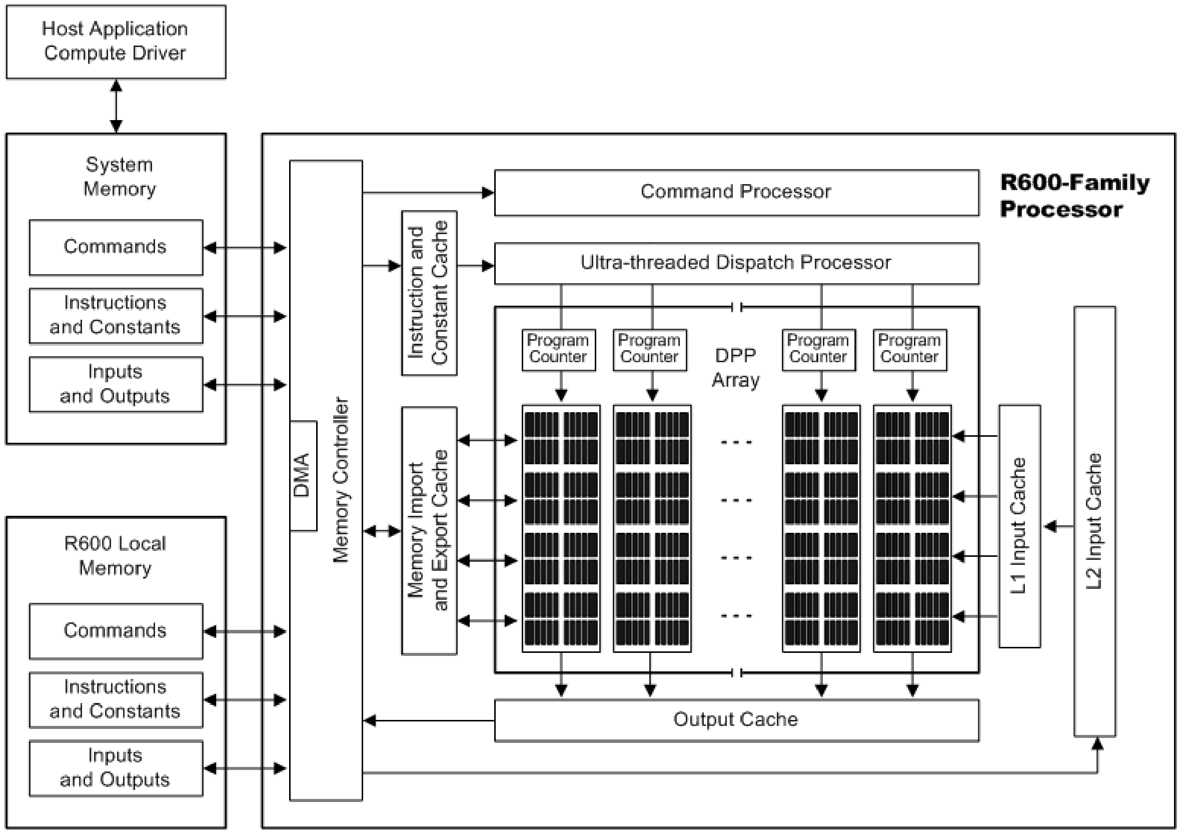


Figure 1.1 R600-Family Block Diagram

It includes a data-parallel processor (DPP) array, a command processor, a memory controller, and other logic (not shown). The R600 command processor reads commands that the host has written to memory-mapped R600 registers in the system-memory address space. The command processor sends hardware-generated interrupts to the host when the command is completed. The R600 memory controller has direct access to all of R600 local 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 R600 local memory directly, but it can command the R600 to copy programs and data between system memory and R600 memory. A complete application for the R600 includes two parts:

- a program running on the host processor, and
- programs, called *kernels*, running on the R600 processor.

The R600 programs are controlled by host commands, which

- set R600-internal base-address and other configuration registers,
- specify the data domain on which the R600 is to operate,
- invalidate and flush caches on the R600, and
- cause the R600 to begin execution of a program.

The R600 driver program runs on the host.

The DPP array is the heart of the R600 processor. The array is organized as a set of SIMD pipelines, each independent from the others, that operate in parallel on streams of floating-point or integer data. The SIMD pipelines can process data or, through the memory controller, transfer data to, or from, memory. Computation in a SIMD pipeline can be made conditional. Outputs written to memory can also be made conditional. R600 software stores data to memory by first allocating space in a memory buffer, then exporting data from GPRs to that buffer. The R600 export facility is also used to import (read) data from memory.

Host commands request a SIMD 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 SIMD pipeline loads instructions and data from memory, begins execution, and continues until the end of the kernel. As kernels are running, the R600 hardware automatically fetches instructions and data from memory into on-chip caches; R600 software plays no role in this. R600 software also can load data from off-chip memory into on-chip GPRs and caches.

Conceptually, each SIMD 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 R600 instructions from the hardware-maintained program state in the pipelines.

R600 programs do not support exceptions, interrupts, errors, or any other events that can interrupt its pipeline operation. In particular, it does not support IEEE floating-point exceptions. The 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 R600 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.

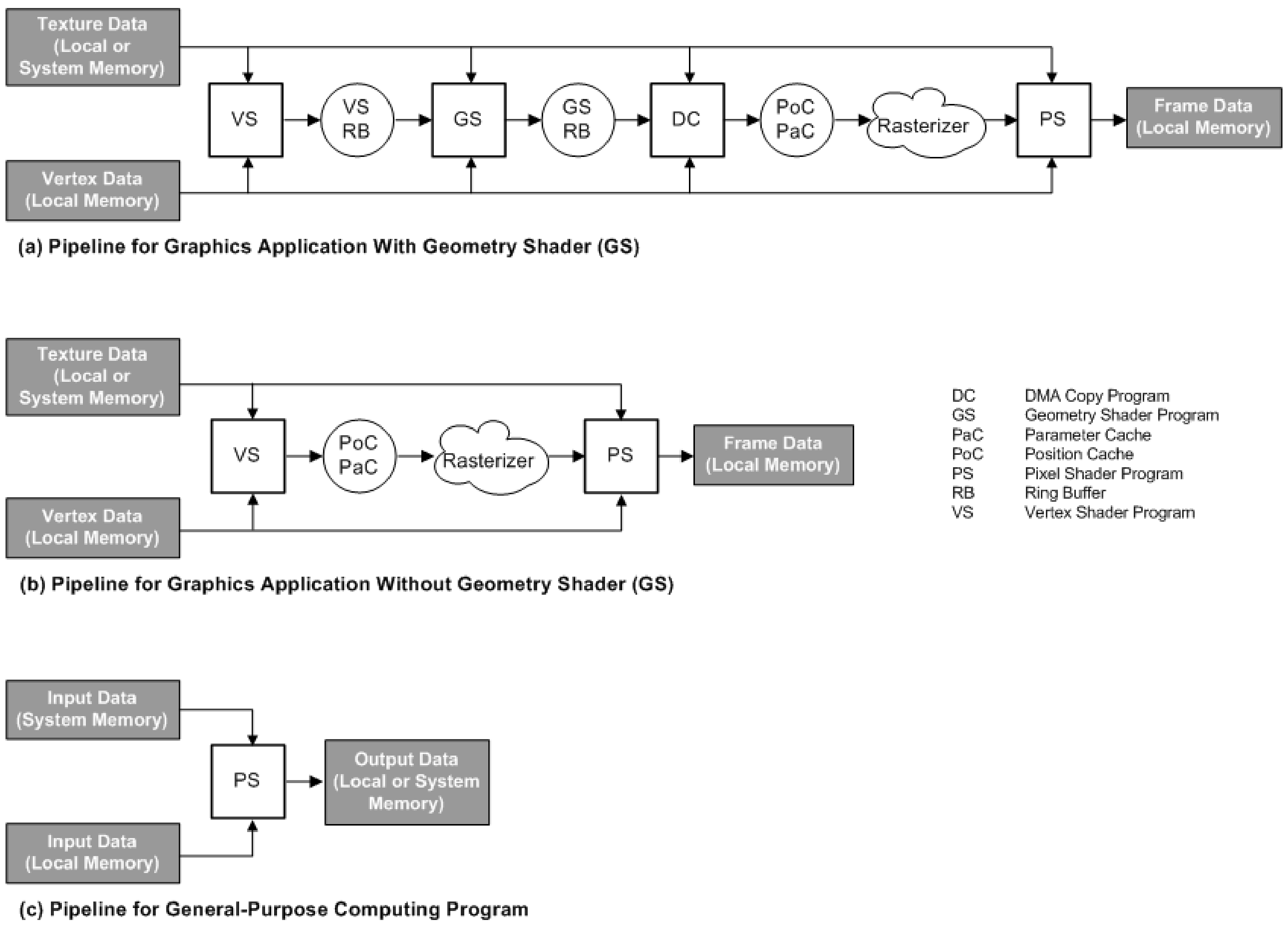

Texture Data (Local or System Memory)
Vertex Data (Local Memory)
VS
VS RB
GS
GS RB
DC
PoC PaC
Rasterizer
PS
Frame Data (Local Memory)
(a) Pipeline for Graphics Application With Geometry Shader (GS)
Texture Data (Local or System Memory)
Vertex Data (Local Memory)
VS
PoC PaC
Rasterizer
PS
Frame Data (Local Memory)
(b) Pipeline for Graphics Application Without Geometry Shader (GS)
DC DMA Copy Program
GS Geometry Shader Program
PaC Parameter Cache
PoC Position Cache
PS Pixel Shader Program
RB Ring Buffer
VS Vertex Shader Program
Input Data (System Memory)
Input Data (Local Memory)
PS
Output Data (Local or System Memory)
(c) Pipeline for General-Purpose Computing Program

Figure 1.2 Programmer's View of R600 Dataflow

The dataflow sequence starts by reading 2D vertices, 2D textures, or other 2D data from local R600 memory or system memory; it ends by writing 2D pixels or other 2D data results to local R600 memory. The R600 processor hides memory latency by keeping track of potentially hundreds of threads in different stages of execution, and by overlapping compute operations with memory-access operations.

Chapter 2 Program Organization and State

R600 programs consist of control-flow (CF), ALU, texture-fetch, and vertex-fetch instructions, which are described in this manual. ALU instructions can have up to three source operands and one destination operand. The instructions operate on 32-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 vertex-fetch and texture-fetch instructions for data loads, whereas general-computing applications typically use texture-fetch instructions for data loads.

2.1 Program Types

The following program types are commonly run on the R600 (see Figure 1.2, on page 1-3,):

- *Vertex Shader (VS)*—Reads vertices, processes them. Depending on whether a geometry shader (GS) is active, it outputs the results to either a VS ring buffer, or the parameter cache and position buffer. It does not introduce new primitives. When a GS is active, a vertex shader is a type of *Export Shader (ES)*. 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 one 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)*—Ttransfers 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:
  - reads data from the position buffer, parameter cache, and vertex geometry translator (VGT),
  - processes individual pixel quads (four pixel-data elements arranged in a 2-by-2 array), and

– writes output to up to eight local-memory buffers, called multiple render targets (MRTs), which can include one or more frame buffers.

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 R600 to run in one of two configurations—with or without a geometry shader program and a DMA copy program. Figure 1.2, on page 1-3, illustrates the processing order. Each type of flow is described in the following subsections.

2.1.2 Geometry Program Absent

Table 2.1 shows the order in which programs run when a geometry program is absent.

Table 2.1 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 sending a pointer to a buffer in local memory containing up to 64 vertex indices.
2. The R600 hardware groups the vectors for these vertices in its input buffers (remote memory).
3. When all vertices are ready to be processed, the R600 allocates GPRs and thread 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 VS program allocates space in the position buffer and exports the vertex positions (XYZW).
7. The VS program allocates parameter-cache and position-buffer space and exports parameters and positions for each vertex.
8. The VS program exits, and the R600 deallocates its GPR space.
9. When the VS program completes, the pixel shader (PS) program begins.

10. The R600 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 R600 deallocates its GPR space.

2.1.3 Geometry Shader Present

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

Table 2.2 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 R600 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.

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

8. The DC program exits, and the R600 deallocates the GPR space.
9. The PS program runs.
10. The R600 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.
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.

2.2 Instruction Terminology

Table 2.3 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 7. The register types are described in “Registers,” on page iv.

Table 2.3 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, “Texture-Fetch Microcode Formats,” page 6-1, Section 5.2, “Vertex-Fetch Microcode Formats,” page 5-2, and Chapter 8, “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 import or export data, and instructions that initiate the execution of ALU, texture-fetch, or vertex-fetch clauses. • ALU instructions (64 bits). • Texture-fetch instructions (128 bits). • Vertex-fetch 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 7, “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.

Table 2.3 Basic Instruction-Related Terms (Cont.)

Term	Size (bits)	Description
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 unlimited	A set of instructions of the same type. The types of clauses are: • ALU clauses (which contain ALU instruction groups). • Texture-fetch clauses. • Vertex-fetch clauses. Clauses are initiated by control flow (CF) instructions and are described in Section 2.3, “Control Flow and Clauses,” page 2-5, and Section 3.3, “Clause-Initiation Instructions,” page 3-5.
Allocate	n/a	Reserves storage space for data in an output buffer (a “scratch buffer,” “ring buffer,” “stream buffer,” or “reduction buffer”) or for data in an input buffer (a “scratch buffer” or “ring buffer”) prior to exporting (writing or 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* (write or read) operation can be performed.
Export	n/a	To do any of the following: • Write data from GPRs to an output buffer (a “scratch buffer,” “frame buffer,” “ring buffer,” “stream buffer,” or “reduction 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. The term *export* is a partial misnomer because it performs both input and output functions. Prior to exporting, an “allocate” operation must be performed to reserve space in the associated buffer.
Fetch	n/a	Load data, using a vertex-fetch or texture-fetch 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 set of x,y (2D) coordinates.
Quad	n/a	Four (x,y) data elements arranged in a 2-by-2 array.
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, texture-fetch, or vertex-fetch instructions.
  - Allocate space in an input or output buffer.
  - Export data to, or import data from, a buffer.
  - Control branching, looping, and stack operations.
- *Clause*—A homogeneous group of instructions; each clause comprises ALU, texture-fetch, or vertex-fetch instructions exclusively. A control flow instruction that initiates an ALU, texture-fetch, or vertex-fetch clause does so by referring to an appropriate clause.

Table 2.4 provdes a typical program flow example.

Table 2.4 Flow of a Typical Program

	Microcode Formats	
Function	**Control Flow (CF) Code**	**Clause Code**
Start loop.	`CF_DWORD[0,1]`	
Initiate texture-fetch clause.	`CF_DWORD[0,1]`	
Texture-fetch or vertex-fetch 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`	

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.
- indicate when a clause has completed.
- are required for buffer allocation in, and writing to, a program block's output buffer.

Some program types (VS, GS, DC, PS) 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). Texture-fetch clauses contain instructions for performing texture and constant-fetch reads from memory. Vertex-fetch clauses are devoted to obtaining vertex data from memory. Systems lacking a vertex cache can perform vertex-fetch operations in a texture clause instead.

A predicate is a bit that is set or cleared as the result of evaluating some condition, and is subsequently 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

There are four types of instructions:

- control flow instructions
- three clause types: control flow (CF), ALU, texture fetch, and vertex fetch.

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

A CF program has no maximum size; however, each clause has a maximum size. When a program is organized in memory, the instructions must be ordered as follows:

- All CF instructions.
- All ALU clauses.
- All texture-fetch and vertex-fetch clauses.

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

2.5 Program State

Table 2.5 through Table 2.8 summarize a programmer’s view of the R600 program state that is accessible by a single thread in an R600 program. The tables do not include:

- states that are maintained exclusively by R600 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 threads.

The column headings in Table 2.5 through Table 2.8 have the following meanings:

- *Access by R600 Software*—Readable (R), writable (W), or both (R/W) by software executing on the R600 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 R600 configuration registers, that accessible only to host software.
- *Number per Thread*—The maximum number of such state objects available to each thread. In some cases, the maximum number is shared by all executing threads.
- *Width*—The width, in bits, of the state object.

Table 2.5 Control-Flow State

State	Access by R600 S/W	Access by Host S/W	# per Thread	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 threads.

Table 2.6 ALU State

State	Access by R600 S/W	Access by Host S/W	# per Thread	Width (bits)	Description
General-Purpose Registers (GPRs)	R/W	No	127 minus 2 times Clause-Temporary GPRs	128 (4 x 32 bit)	Each thread 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 thread reduces the total number of GPRs available to the thread, as described immediately above.
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.
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.
Local Data Share (LDS)	R/W Read: up to 16 kB; Write: 16 to 256 bytes	No			Per-SIMD on-chip shared memory. Enables sharing of data between elements of a SIMD using an owner's write, shared-read model.
Global Data Share (GDS)	R/W Read: up to 16 kB Write: 4 to 256 bytes	No			Per-system on-chip shared memory. Enables sharing of data between all elements of all SIMDs using an owner's write, shared-read model.

Table 2.6 ALU State (Cont.)

State	Access by R600 S/W	Access by Host S/W	# per Thread	Width (bits)	Description
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.
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 2.7 Vertex-Fetch State

State	Access by R600 S/W	Access by Host S/W	# per Thread	Width (bits)	Description
Vertex-Fetch Constants	R	W	128	84	These describe the buffer format, etc.

Table 2.8 Texture-Fetch and Constant-Fetch State

State	Access by R600 S/W	Access by Host S/W	# per Thread	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 texture-fetch 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 texture-fetch 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 texture-fetch instructions.

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 R600 hardware maintains a single, multi-entry stack for saving and restoring active mask counters, returning addresses for subroutines.

CF instructions can:

- Execute an ALU, texture-fetch, or vertex-fetch 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).
- 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 a vertex-fetch clause. The address field in a VTX or VTX_TC control-flow instruction is unused; the address of the vertex-fetch 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.

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.

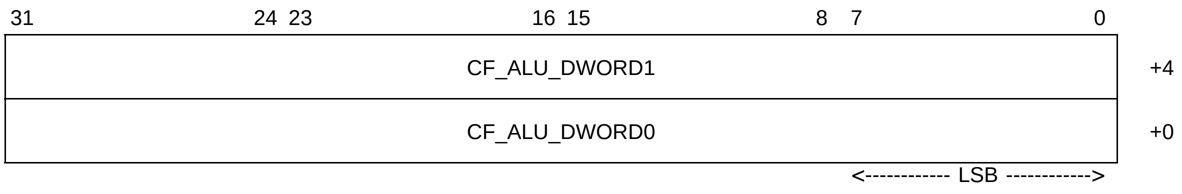
3.1 CF Microcode Encoding

The microcode formats and all of their fields are described in Chapter 8, "Microcode Formats.". An overview of the encoding is given below. The following instruction-related terms are used throughout the remainder of this document:

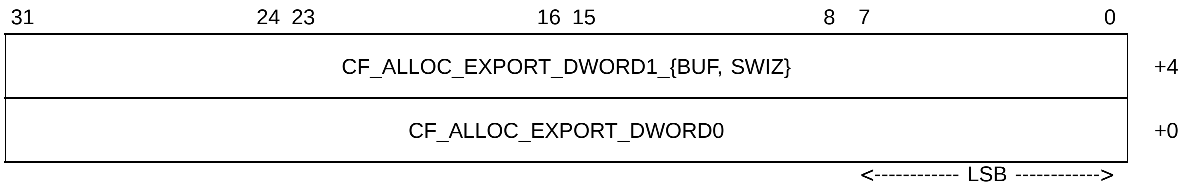
- *Microcode Format*—An encoding format whose fields specify instructions and associated parameters. Microcode formats are used in sets of two or four 32-bit doublewords (dwords). For example, the two mnemonics, CF_DWORD[0,1] indicate a microcode-format pair, CF_DWORD0 and CF_DWORD1, described in Section 8.1, "Control Flow (CF) Instructions," page 8-2.
- *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 8, "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.

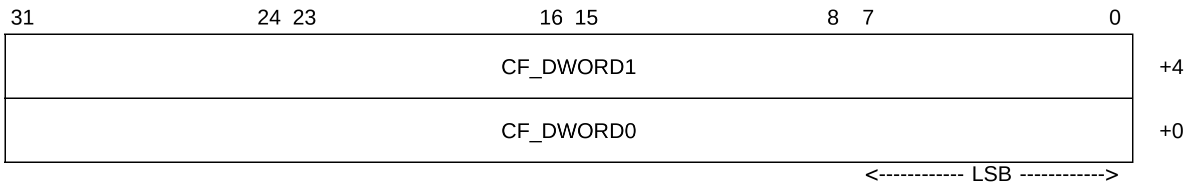
- CF microcode instructions that initiate ALU clauses use the following memory layout.



- CF microcode instructions that 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 use the following memory layout.


31
24 23
16 15
8 7
0
CF_ALLOC_EXPORT_DWORD1_{BUF, SWIZ}
+4
CF_ALLOC_EXPORT_DWORD0
+0
<------------ LSB ------------>

- 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 8.1, “Control Flow (CF) Instructions,” page 8-2.

Table 3.1 CF Microcode Field Summary

CF Microcode Field	CF Instruction Type					
	ALU[1]	Texture Fetch[2]	Vertex Fetch[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	Yes	Yes	Yes		No	No
CALL_COUNT		No	No		Note[10]	No
KCACHE_BANK[0,1]	Yes					
KCACHE_ADDR[0,1]	Yes					
KCACHE_MODE[0,1]	Yes					
USES_WATERFALL	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 texture-fetch instructions contain the string CF_INST_TEX.
3. CF vertex-fetch instructions contain the string CF_INST_VTX_.
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. CALL_COUNT is only used for CALL instructions.
11. INDEX_GPR is used if the TYPE field indicates an indexed read or write.
12. COMP_MASK is used if the TYPE field indicates a write operation; reads are never masked.

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 texture-fetch 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. Any clause type can be executed by any thread type.

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)
Texture Fetch	TEX[3]	Number of instructions	[1, 8]	Double quadword (128-bit)
Vertex Fetch	VTX*[4]	Number of instructions	[1, 8]	Double quadword (128-bit)

1. These instructions use the CF_ALU_DWORD[0,1] microcode formats, described in Section 8.1 on page 8-2.
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 8.1 on page 8-2.
4. These instructions use the CF_DWORD[0,1] microcode formats, described in Section 8.1 on page 8-2.

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 8.1 on page 8-2. The ALU instructions within an ALU clause are described in Chapter 4, "ALU Clauses," and Section 7.2, "ALU Instructions," page 7-41.

The USES_WATERFALL bit in an ALU* control-flow instruction is used to mark clauses that can use constant waterfalling. This bit allows the processor to take scheduling restrictions into account. This bit must be set for clauses containing an instruction that writes to the address register (AR), which include all MOVA* instructions. Setting this option on a clause that does not use the AR register results in decreased performance. The contents of the AR register are not valid past the end of the clause; the register must be written in every clause before it is read.

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-Fetch Clause Initiation and Execution

The VTX and VTX_TC control-flow instructions initiate a vertex-fetch clause, starting at the double-quadword-aligned (128-bit) offset in the ADDR field and containing COUNT + 1 instructions. The VTX_TC instruction issues the vertex fetch through the texture cache (TC) and is useful for systems that lack a vertex cache (VC).

The VTX and VTX_TC control-flow instructions are encoded using the CF_DWORD[0,1] microcode formats, which are described in Section 8.1 on page 8-2. The vertex-fetch instructions within a vertex-fetch clause are described in Chapter 5, "Vertex-Fetch Clauses," and Section 7.3, "Vertex-Fetch Instructions," page 7-181.

3.3.3 Texture-Fetch Clause Initiation and Execution

The TEX control-flow instruction initiates a texture-fetch or constant-fetch clause, starting at the double-quadword-aligned (128-bit) offset in the ADDR field and

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

containing COUNT + 1 instructions. There is only one instruction for texture fetch, and there are no special fields in the instruction for texture clause execution.

The TEX control-flow instruction is encoded using the CF_DWORD[0,1] microcode formats, which are described in Section 8.1 on page 8-2. The texture-fetch instructions within a texture-fetch clause are described in Chapter 6, “Texture-Fetch Clauses,” and Section 7.4, “Texture-Fetch Instructions,” page 7-183.

3.4 Import and Export Instructions

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

Importing and 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. The remaining instructions, MEM*, are used for memory operations toall buffer types.

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 thread 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.
	23:16	CF_PIXEL_MRT[7,0]_FOG	Frame Buffer multiple render target (MRT), with 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 Reads and Writes

All imports from, and exports to, memory use one of the following instructions:

- MEM_SCRATCH — Scratch buffer (read and write).
- MEM_REDUCTION — Reduction buffer (read and write).
- MEM_STREAM[0,3] — Stream buffer (write-only), for DirectX10 compliance, used by VS output for up to four streams.
- MEM_RING — Ring buffer (write-only), used for DC and GS output.
- MEM_EXPORT — Scatter reads and 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 imports or exports per program type (four scratch buffers in total). There is only one reduction buffer available; any program type can use it, but only one program can use it at a time. 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.
- IMPORT_READ — Read from buffer (scratch and reduction buffers only).
- IMPORT_READ_IND — Read from buffer using offset supplied by INDEX_GPR (scratch and reduction only).

The RW_GPR and RW_REL fields indicate the GPR address (first_gpr) to read the first value from, or 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 outputs, in doublewords, being written. The BURST_COUNT and ELEM_SIZE fields store the actual number minus one. ELEM_SIZE must be three (representing four doublewords) for scratch and reduction 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 read or write 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 or 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 and reduction 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 and writes 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 R670 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 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.

3.5 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 with it that 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 is 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 3-15. 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, but 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 the number of thread groups (simultaneously executing threads on a SIMD pipeline) per program type that are supported by the target 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 Size 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.

Table 3.7 Branch-Loop Instructions (Cont.)

Instruction	Condition Test Computed	Push	Pop	Jump	Description
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.
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					POP, POP

Table 3.7 Branch-Loop Instructions (Cont.)

Instruction	Condition Test Computed	Push	Pop	Jump	Description
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; POP.

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 vector 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.

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.

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.

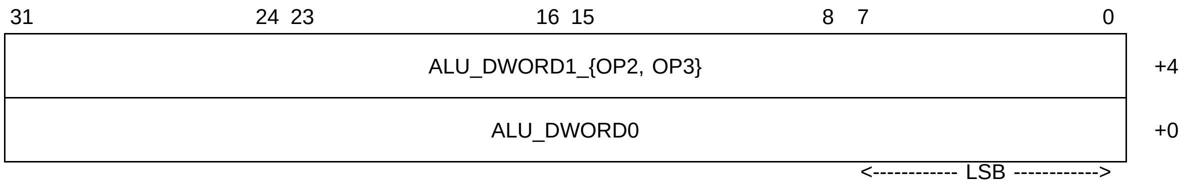


Figure 4.1 ALU Microcode Format Pair

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.

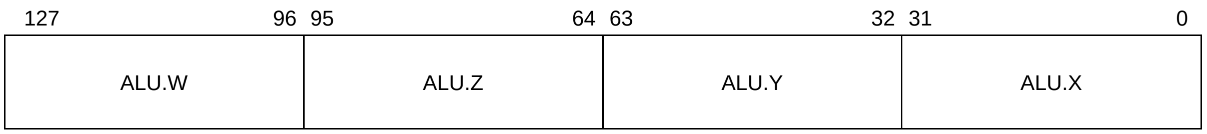


Figure 4.2 Organization of ALU Vector Elements in GPRs

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, R600 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 2.4 on page 2-6. 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 R600 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
         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.

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.

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 8-18) and ALU_DWORD1_OP3 (page 8-23). 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 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. A clause using AR must be initiated by a CF instruction with the USES_WATERFALL bit set.

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.

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.

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 for texture- and vertex-fetch instructions and 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 texture- and vertex-fetch instructions), 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 can reference up to two constants: one inline constant (literal), and one constant read from the on-chip constant file. All ALU constants are four 32-bit values. Also, the constants 0, 1, and 0.5 can be swapped for any GPR element as a swizzle option.

The ALU constants are available in one of two modes: DX9 (constant file) or DX10 (constant cache). In DX9 mode, the PS and VS registers each have 256 ALU constants available (both `set_constant` and `def_constant` versions). In DX10 mode, there is a constant cache with 256 ALU constants for the PS, shared by the VS and ES, and for the GS programs. In this mode, the constant-file is not available.

Each program can use up to 256 ALU constants from the constant file. The processor actually stores twice this number for each program: one set for `set_constant` constants and one for `def_constant`. There is a 256-bit mask that each program must initialize. This mask determines, for each constant, whether to use the `set_constant` or `def_constant` for this shader.

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 (in-line) 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 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.

4.6.4.4 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 the four elements 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. The processor sends the four elements at different times, so that it can optimize for receiving the X element three cycles before the W element. The GPR indices loaded by a MOVA instruction only persist for one clause; at the end of the clause they are invalidated.

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.

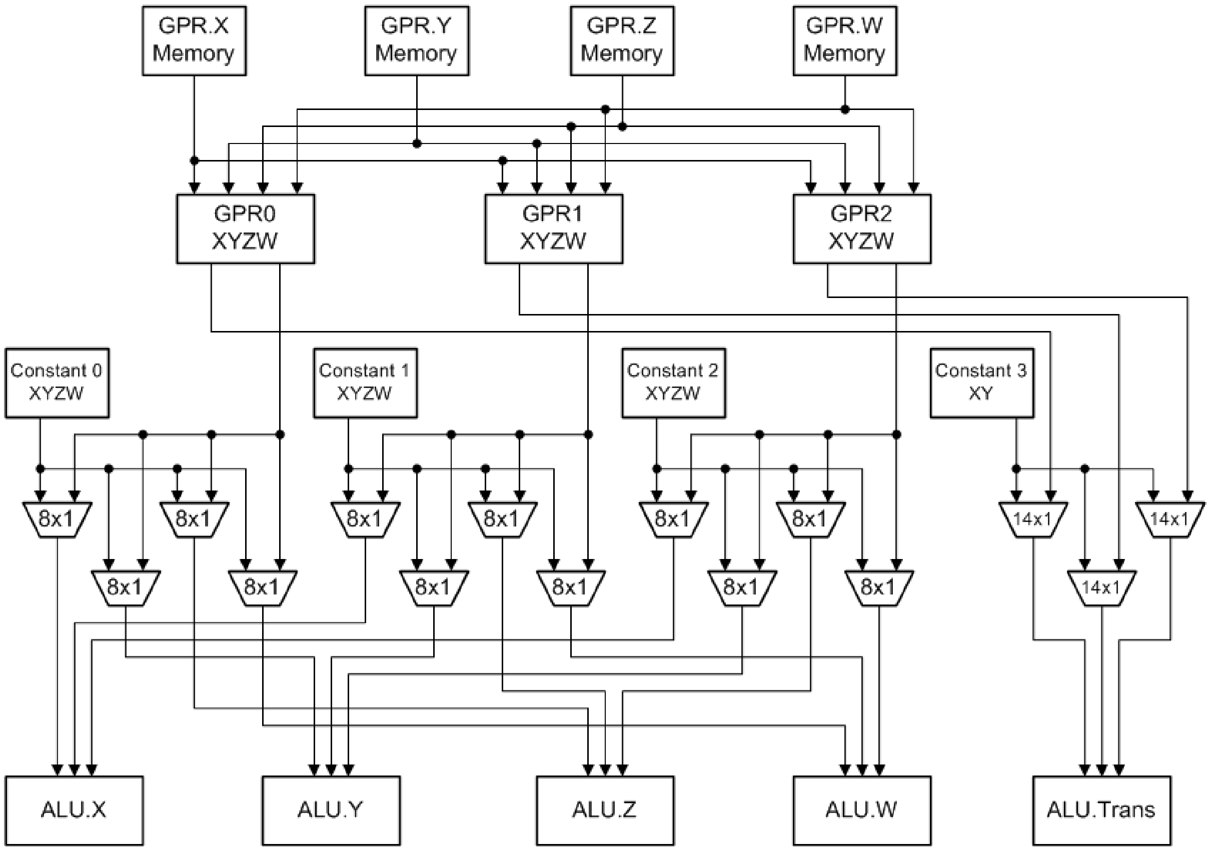


Figure 4.3 ALU Data Flow

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. They can be from four different addresses, and can all come from the same element. For example, an instruction group can access C0.X, C1.X, C2.X, and C3.X. 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

BANK_SWIZZLE	src0	src1	src2
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*.

```
procedure reserve_cfile($sel, $elem)
begin
   $resmatch := undef;
   $resempty := undef;
   for $res in {3, 2, 1, 0}
      if !defined(HW_CFILE$res_ADDR)
         $resempty := $res;
      elsif HW_CFILE$res_ADDR == $sel and HW_CFILE$res_ELEM ==
$elem
          $resmatch := $res;
    if defined($resmatch)
        // Read for this scalar element already reserved, nothing to
do here.
    elsif defined($resempty)
        HW_CFILE$resempty_ADDR := $sel;
        HW_CFILE$resempty_ELEM := $elem;
    else
        assert "All cfile read ports are used, cannot reference
C$sel,
            vector element $elem.";
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 Ecevution 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 7.2, "ALU Instructions," page 7-41, 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	
ADD_64	Floating-point 64-bit add.
ADD_INT	Integer add based on signed or unsigned integer elements.
AND_INT	Logical bit-wise AND.
CMOVE_INT	Integer conditional move equal based on integer (either signed or unsigned).
CMOVGE_INT	Integer conditional move greater than or equal based on signed integer values.
CMOVGT_INT	Integer conditional move greater than based on signed integer values.
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.
MAX_INT	Integer maximum based on signed integer elements.
MAX_UINT	Integer maximum based on unsigned integer elements.

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

Mnemonic	Description
MIN_INT	Integer minimum based on signed integer elements.
MIN_UINT	Integer minimum based on signed unsigned integer elements.
MOV	Single-operand move.
MUL_64	Floating-point multiply, 64-bit.
MULADD_64	Floating-point multiply-add, 64-bit.
NOP	No operation.
NOT_INT	Logical bit-wise NOT.
OR_INT	Logical bit-wise OR.
PRED_SETE_64	Floating-point predicate set if equal, 64-bit.
PRED_SETE_INT	Integer predicate set equal. Update predicate register.
PRED_SETE_PUSH_INT	Integer predicate counter increment equal. Update predicate register.
PRED_SETGE_64	Floating-point predicate set if greater than or equal, 64-bit.
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_SETGT_64	Floating-point predicate set, if greater than, 64-bit.
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_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.
SUB_INT	Integer subtract based on signed or unsigned integer elements.
XOR_INT	Logical bit-wise XOR.
Floating-Point Operations	
ADD	Floating-point add.
CEIL	Floating-point ceiling function.
CMOVE	Floating-point conditional move equal.
CMOVGE	Floating-point conditional move greater than equal.
CMOVGT	Floating-point conditional move greater than.
FLOOR	Floating-point floor function.
FRACT	Floating-point fractional part of src1.

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

Mnemonic	Description
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.
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.

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

Mnemonic	Description
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	
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.
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.
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.

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

Mnemonic	Description
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	
ASHR_INT	Scalar arithmetic shift right. The sign bit is shifted into the vacated locations. src1 is interpreted as an unsigned integer. If src1 is > 31, the result is either 0x0 or -0x1, depending on the sign of src0.
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.
INT_TO_FLT	The input is interpreted as a signed integer value and converted to a floating-point value.
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.
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.
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.

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

Mnemonic	Description
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_EXECUTE_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_EXECUTE_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_EXECUTE_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. The write 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.

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_EXECUTE_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_EXECUTE_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.

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 7.2, “ALU Instructions,” page 7-41. 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.

Chapter 5
Vertex-Fetch Clauses

Software initiates a vertex-fetch clause with the VTX or VTX_TC control-flow instructions, both of which use the CF_DWORD[0,1] microcode formats. Vertex-fetch instructions within the clause use the VTX_DWORD0, VTX_DWORD1_{SEM, GPR}, and VTX_DWORD2 microcode formats, with a fourth (high-order) doubleword of zeros.

5.1 Clause Construction

A vertex-fetch clause consists of instructions that fetch vertices from the vertex buffer based on a GPR address. A vertex-fetch clause can be at most eight instructions long. Vertex fetches using a semantic table use the VTX_DWORD1_SEM microcode format to specify the nine-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_DWORD1_GPR microcode format, which specifies the destination GPR directly.

Each vertex-fetch instruction within the vertex-fetch clause has a BUFFER_ID field that specifies the buffer containing the vertex-fetch constants, 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_GPL + 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 vertex-fetch instructions cannot be predicated; predicated vertex fetches must be done at the CF level by making the vertex-fetch clause instruction conditional. All vertex instructions in the clause are executed with the conditional constraint specified by the CF instruction.

5.2 Vertex-Fetch Microcode Formats

Vertex-fetch microcode formats 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.

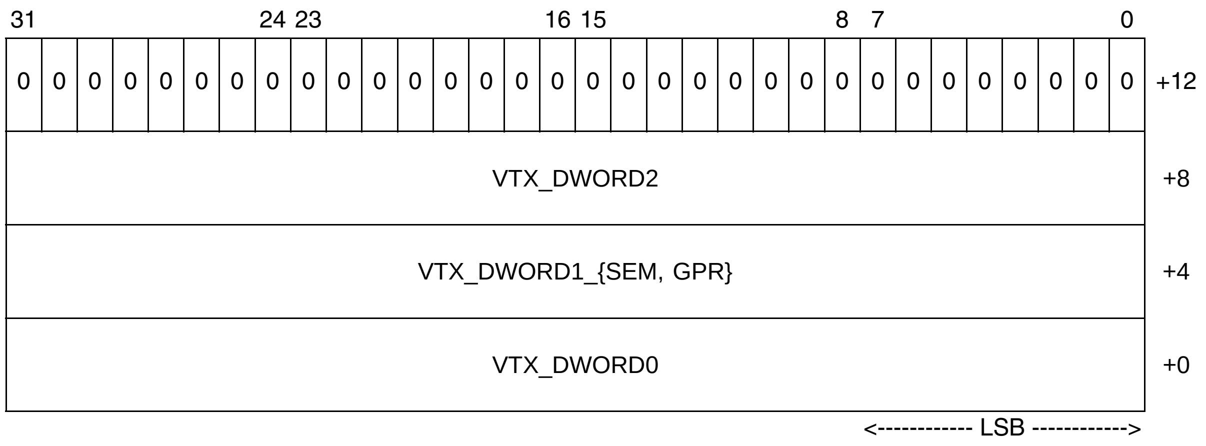


Figure 5.1 Vertex-Fetch Microcode-Format 4-Tuple

Chapter 6
Texture-Fetch Clauses

Software initiates a texture-fetch clause with the TEX control-flow instruction, which uses the CF_DWORD[0 1] microcode formats. Texture-fetch instructions within the clause use the TEX_DWORD[0,1,2] microcode formats, with a fourth (high-order) doubleword of zeros.

A texture-fetch clause consists of instructions that lookup texture elements, called *texels*, based on a GPR address. Texture instructions are used for both texture-fetch 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 texture fetches 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 Texture-Fetch Microcode Formats

Texture-fetch microcode formats 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.

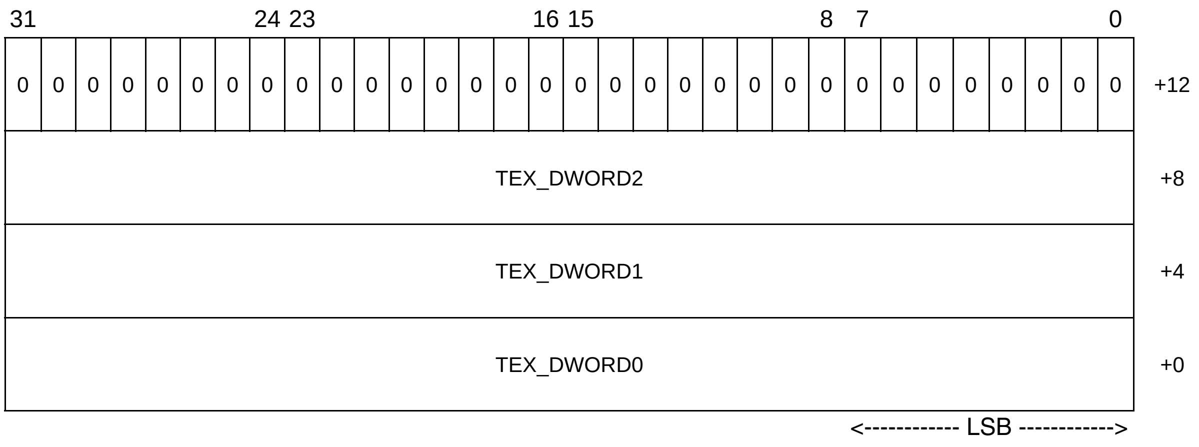


Figure 6.1 Texture-Fetch Microcode-Format 4-Tuple

6.2 Constant-Fetch Operations

The buffer ID space, specified in the `RESOURCE_ID` field of the `TEX_DWORD0` microcode format, is eight bits wide, allowing constant and texture fetch 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:

- Texture-fetch instructions 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.

Chapter 7 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 8, where the instructions are ordered by their microcode formats, rather than alphabetically by mnemonic. That chapter also defines the microcode field-name acronyms.

7.1 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 7.2, “ALU Instructions,” page 7-41.

Microcode

B	W Q M	CF_INST	U W	COUNT	KCACHE_ADDR1	KCACHE_ADDR0	K M 1	+4
K M 0		K B 1	K B 0	ADDR				+0

Format CF_ALU_DWORD0 (page 8-7) and CF_ALU_DWORD1 (page 8-8).

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 7.2, “ALU Instructions,” page 7-41.

Microcode

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

Format CF_ALU_DWORD0 (page 8-7) and CF_ALU_DWORD1 (page 8-8).

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 7.2, “ALU Instructions,” page 7-41.

Microcode

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

Format CF_ALU_DWORD0 (page 8-7) and CF_ALU_DWORD1 (page 8-8).

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 7.2, “ALU Instructions,” page 7-41.

Microcode

B	WQM	CF_INST	UW	COUNT	KCACHE_ADDR1	KCACHE_ADDR0	KM1	+4

KM0	KB1	KB0	ADDR	+0

Format CF_ALU_DWORD0 (page 8-7) and CF_ALU_DWORD1 (page 8-8).

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

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 7.2, “ALU Instructions,” page 7-41.

Microcode

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

Format CF_ALU_DWORD0 (page 8-7) and CF_ALU_DWORD1 (page 8-8).

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 7.2, "ALU Instructions," page 7-41.

Microcode

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

Format CF_ALU_DWORD0 (page 8-7) and CF_ALU_DWORD1 (page 8-8).

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 7.2, "ALU Instructions," page 7-41.

Microcode

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

Format CF_ALU_DWORD0 (page 8-7) and CF_ALU_DWORD1 (page 8-8).

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

B	WQM	CF_INST	VPM	EOP	Rsvd	CALL_COUNT	COUNT	COND	CF_CONST	PC	+4
ADDR											+0

Format	CF_DWORD0 (page 8-3) and CF_DWORD1 (page 8-4).
Instruction Field	CF_INST == CF_INST_CALL, opcode 13 (0xD).

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 vector 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

B	WQM	CF_INST	VPM	EOP	Rsvd	CALL_COUNT	COUNT	COND	CF_CONST	PC	+4
ADDR											+0

Format CF_DWORD0 (page 8-3) and CF_DWORD1 (page 8-4).

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

End Primitive Strip, Start New Primitve 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

B	WQM	CF_INST	VPM	EOP	Rsvd	CALL_COUNT	COUNT	COND	CF_CONST	PC	+4
ADDR											+0

Format CF_DWORD0 (page 8-3) and CF_DWORD1 (page 8-4).

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

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

B	WQM	CF_INST	VPM	EOP	Rsvd	CALL_COUNT	COUNT	COND	CF_CONST	PC	+4
ADDR											+0

Format `CF_DWORD0` (page 8-3) and `CF_DWORD1` (page 8-4).

Instruction Field `CF_INST == CF_INST_ELSE`, opcode 17 (0x11).

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

B	WQM	CF_INST	VPM	EOP	Rsvd	CALL_COUNT	COUNT	COND	CF_CONST	PC	+4
ADDR											+0

Format	`CF_DWORD0` (page 8-3) and `CF_DWORD1` (page 8-4).
Instruction Field	`CF_INST == CF_INST_EMIT_CUT_VERTEX`, opcode 19 (0x13).

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

B	WQM	CF_INST	VPM	EOP	Rsvd	CALL_COUNT	COUNT	COND	CF_CONST	PC	+4
ADDR											+0

Format `CF_DWORD0` (page 8-3) and `CF_DWORD1` (page 8-4).

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

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

Format CF_ALLOC_EXPORT_DWORD0 (page 8-10) and either CF_ALLOC_EXPORT_DWORD1_BUF (page 8-12) or CF_ALLOC_EXPORT_DWORD1_SWIZ (page 8-15).

Instruction Field CF_INST == CF_INST_EXPORT, opcode 39 (0x27).

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	WQM	CF_INST	VPM	EOP	BC	EL	Reserved	SEL_W	SEL_Z	SEL_Y	SEL_X	+4

ES	INDEX_GPR	RR	RW_GPR	TYPE	ARRAY_BASE	+0

Format CF_ALLOC_EXPORT_DWORD0 (page 8-10) and either CF_ALLOC_EXPORT_DWORD1_BUF (page 8-12) or CF_ALLOC_EXPORT_DWORD1_SWIZ (page 8-15).

Instruction Field CF_INST == CF_INST_EXPORT_DONE, opcode 40 (0x28).

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

B	WQM	CF_INST	VPM	EOP	Rsvd	CALL_COUNT	COUNT	COND	CF_CONST	PC	+4
ADDR											+0

Format `CF_DWORD0` (page 8-3) and `CF_DWORD1` (page 8-4).

Instruction Field `CF_INST == CF_INST_JUMP`, opcode 16 (0x10).

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

B	WQM	CF_INST	VPM	EOP	Rsvd	CALL_COUNT	COUNT	COND	CF_CONST	PC	+4
ADDR											+0

Format `CF_DWORD0` (page 8-3) and `CF_DWORD1` (page 8-4).

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

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	VPM	EOP	Rsvd	CALL_COUNT	COUNT	COND	CF_CONST	PC	+4
ADDR											+0

Format CF_DWORD0 (page 8-3) and CF_DWORD1 (page 8-4).

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

B	WQM	CF_INST	VPM	EOP	Rsvd	CALL_COUNT	COUNT	COND	CF_CONST	PC	+4
ADDR											+0

Format CF_DWORD0 (page 8-3) and CF_DWORD1 (page 8-4).

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

B	WQM	CF_INST	VPM	EOP	Rsvd	CALL_COUNT	COUNT	COND	CF_CONST	PC	+4
ADDR											+0

Format CF_DWORD0 (page 8-3) and CF_DWORD1 (page 8-4).

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

B	WQM	CF_INST	VPM	EOP	Rsvd	CALL_COUNT	COUNT	COND	CF_CONST	PC	+4
ADDR											+0

Format CF_DWORD0 (page 8-3) and CF_DWORD1 (page 8-4).

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

B	WQM	CF_INST	VPM	EOP	Rsvd	CALL_COUNT	COUNT	COND	CF_CONST	PC	+4
ADDR											+0

Format `CF_DWORD0` (page 8-3) and `CF_DWORD1` (page 8-4).

Instruction Field `CF_INST == CF_INST_LOOP_START_DX10`, opcode 4 (0x4).

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

B	WQM	CF_INST	VPM	EOP	Rsvd	CALL_COUNT	COUNT	COND	CF_CONST	PC	+4
ADDR											+0

Format CF_DWORD0 (page 8-3) and CF_DWORD1 (page 8-4).

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

Access Scatter Buffer

Instructions **MEM_EXPORT**

Description Used only by the RV670.

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.

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	WQM	CF_INST	VPM	EOP	BC	EL	COMP_MASK	ARRAY_SIZE	+4
ES		INDEX_GPR	RR	RW_GPR			TYPE	ARRAY_BASE	+0

Format CF_ALLOC_EXPORT_DWORD0 (page 8-10) and either CF_ALLOC_EXPORT_DWORD1_BUF (page 8-12) or CF_ALLOC_EXPORT_DWORD1_SWIZ (page 8-15).

Instruction Field CF_INST == CF_INST_MEM_EXPORT, opcode 58 (0x3A).

Access Reduction Buffer

Instructions **MEM_REDUCTION**

Description Perform a memory read or write on a reduction buffer.

Microcode

B	WQM	CF_INST	VPM	EOP	BC	EL	COMP_MASK	ARRAY_SIZE	+4

ES	INDEX_GPR	RR	RW_GPR	TYPE	ARRAY_BASE	+0

Format CF_ALLOC_EXPORT_DWORD0 (page 8-10) and either CF_ALLOC_EXPORT_DWORD1_BUF (page 8-12) or CF_ALLOC_EXPORT_DWORD1_SWIZ (page 8-15).

Instruction Field CF_INST == CF_INST_MEM_REDUCTION, opcode 37 (0x25).

Write Ring Buffer

Instructions **MEM_RING**

Description Perform a memory write on a ring buffer. Used for DC and GS output.

Microcode

B	WQM	CF_INST	VPM	EOP	BC	EL	COMP_MASK	ARRAY_SIZE	+4

ES	INDEX_GPR	RR	RW_GPR	TYPE	ARRAY_BASE	+0

Format CF_ALLOC_EXPORT_DWORD0 (page 8-10) and either CF_ALLOC_EXPORT_DWORD1_BUF (page 8-12) or CF_ALLOC_EXPORT_DWORD1_SWIZ (page 8-15).

Instruction Field CF_INST == CF_INST_MEM_RING, opcode 38 (0x26).

Access Scratch Buffer

Instructions **MEM_SCRATCH**

Description Perform a memory read or write on the scratch buffer.

Microcode

B	WQM	CF_INST	VPM	EOP	BC	EL	COMP_MASK	ARRAY_SIZE	+4

ES	INDEX_GPR	RR	RW_GPR	TYPE	ARRAY_BASE	+0

Format `CF_ALLOC_EXPORT_DWORD0` (page 8-10) and either `CF_ALLOC_EXPORT_DWORD1_BUF` (page 8-12) or `CF_ALLOC_EXPORT_DWORD1_SWIZ` (page 8-15).

Instruction Field `CF_INST == CF_INST_MEM_SCRATCH`, opcode 36 (0x24).

Write Steam Buffer 0

Instructions **MEM_STREAM0**

Description Write vertex or pixel data to stream buffer 0 in memory (write-only). Used by vertex shader (VS) output for DirectX10 compliance.

Microcode

B	WQM	CF_INST	VPM	EOP	BC	EL	COMP_MASK	ARRAY_SIZE	+4

ES	INDEX_GPR	RR	RW_GPR	TYPE	ARRAY_BASE	+0

Format `CF_ALLOC_EXPORT_DWORD0` (page 8-10) and either `CF_ALLOC_EXPORT_DWORD1_BUF` (page 8-12) or `CF_ALLOC_EXPORT_DWORD1_SWIZ` (page 8-15).

Instruction Field `CF_INST == CF_INST_MEM_STREAM0`, opcode 32 (0x20).

Write Steam Buffer 1

Instructions	**`MEM_STREAM1`**
Description	Write vertex or pixel data to stream buffer 1 in memory (write-only). Used by vertex shader (VS) output for DirectX10 compliance.

Microcode

B	WQM	CF_INST	VPM	EOP	BC	EL	COMP_MASK	ARRAY_SIZE	+4
ES	INDEX_GPR	RR	RW_GPR	TYPE	ARRAY_BASE				+0

Format	`CF_ALLOC_EXPORT_DWORD0` (page 8-10) and either `CF_ALLOC_EXPORT_DWORD1_BUF` (page 8-12) or `CF_ALLOC_EXPORT_DWORD1_SWIZ` (page 8-15).
Instruction Field	`CF_INST == CF_INST_MEM_STREAM1`, opcode 33 (0x21).

Write Steam Buffer 2

Instructions **MEM_STREAM2**

Description Write vertex or pixel data to stream buffer 2 in memory (write-only). Used by vertex shader (VS) output for DirectX10 compliance.

Microcode

B	WQM	CF_INST	VPM	EOP	BC	EL	COMP_MASK	ARRAY_SIZE	+4
ES	INDEX_GPR	RR	RW_GPR	TYPE	ARRAY_BASE				+0

Format CF_ALLOC_EXPORT_DWORD0 (page 8-10) and either CF_ALLOC_EXPORT_DWORD1_BUF (page 8-12) or CF_ALLOC_EXPORT_DWORD1_SWIZ (page 8-15).

Instruction Field CF_INST == CF_INST_MEM_STREAM2, opcode 34 (0x22).

Write Steam Buffer 3

Instructions **MEM_STREAM3**

Description Write vertex or pixel data to stream buffer 3 in memory (write-only). Used by vertex shader (VS) output for DirectX10 compliance.

Microcode

B	WQM	CF_INST	VPM	EOP	BC	EL	COMP_MASK	ARRAY_SIZE	+4
ES		INDEX_GPR	RR	RW_GPR			TYPE	ARRAY_BASE	+0

Format `CF_ALLOC_EXPORT_DWORD0` (page 8-10) and either `CF_ALLOC_EXPORT_DWORD1_BUF` (page 8-12) or `CF_ALLOC_EXPORT_DWORD1_SWIZ` (page 8-15).

Instruction Field `CF_INST == CF_INST_MEM_STREAM3`, opcode 35 (0x32).

No Operation

Instructions **NOP**

Description No operation. It ignores all fields in the CF_DWORD[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 7-118.

Microcode

B	WQM	CF_INST	VPM	EOP	Rsvd	CALL_COUNT	COUNT	COND	CF_CONST	PC	+4
ADDR											+0

Format CF_DWORD0 (page 8-3) and CF_DWORD1 (page 8-4).

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

B	WQM	CF_INST	VPM	EOP	Rsvd	CALL_COUNT	COUNT	COND	CF_CONST	PC	+4
ADDR											+0

Format CF_DWORD0 (page 8-3) and CF_DWORD1 (page 8-4).

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

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

B	WQM	CF_INST	VPM	EOP	Rsvd	CALL_COUNT	COUNT	COND	CF_CONST	PC	+4
ADDR											+0

Format CF_DWORD0 (page 8-3) and CF_DWORD1 (page 8-4).

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

Push State To Stack and Invert State

Instructions **`PUSH_ELSE`**

Description Push current per-pixel state (active Mask) onto the stack and compute new active Mask. The instruction can be used to implement the ELSE part of a higher-level IF statement.

Microcode

B	WQM	CF_INST	VPM	EOP	Rsvd	CALL_COUNT	COUNT	COND	CF_CONST	PC	+4
ADDR											+0

Format `CF_DWORD0` (page 8-3) and `CF_DWORD1` (page 8-4).

Instruction Field `CF_INST == CF_INST_PUSH_ELSE`, 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

B	WQM	CF_INST	VPM	EOP	Rsvd	CALL_COUNT	COUNT	COND	CF_CONST	PC	+4
ADDR											+0

Format CF_DWORD0 (page 8-3) and CF_DWORD1 (page 8-4).

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

Initiate Texture-Fetch Clause

Instructions **TEX**

Description Initiates a texture-fetch or 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 texture fetch, and there are no special fields in the instruction for texture clause execution. The texture-fetch instructions within a texture-fetch clause are described in Section Chapter 6, “Texture-Fetch Clauses,” page 6-1 and Section 7.4, “Texture-Fetch Instructions,” page 7-183.

Microcode

B	WQM	CF_INST	VPM	EOP	Rsvd	CALL_COUNT	COUNT	COND	CF_CONST	PC	+4
ADDR											+0

Format `CF_DWORD0` (page 8-3) and `CF_DWORD1` (page 8-4).

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

Initiate Vertex-Fetch Clause

Instructions **VTX**

Description Initiate a vertex-fetch clause, starting at the double-quadword-aligned *(*128-bit) offset in the `ADDR` field and containing `COUNT + 1` instructions. The `VTX_TC` instruction issues the vertex fetch through the texture cache (TC) and is useful for systems that lack a vertex cache (VC). The vertex-fetch instructions within a vertex-fetch clause are described in Section Chapter 5, "Vertex-Fetch Clauses," page 5-1 and Section 7.3, "Vertex-Fetch Instructions," page 7-181.

Microcode

B	WQM	CF_INST	VPM	EOP	Rsvd	CALL_COUNT	COUNT	COND	CF_CONST	PC	+4
ADDR											+0

Format `CF_DWORD0` (page 8-3) and `CF_DWORD1` (page 8-4).

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

Initiate Vertex-Fetch Clause Through Texture Cache

Instructions **VTX_TC**

Description Initiate a vertex-fetch clause, starting at the double-quadword-aligned *(*128-bit) offset in the `ADDR` field and containing `COUNT + 1` instructions. It is used for systems lacking a vertex cache (VC). The `VTX_TC` instruction issues the vertex fetch through the texture cache (TC) and is useful for systems that do not have a vertex cache (VC). The vertex-fetch instructions within a vertex-fetch clause are described in Section Chapter 5, “Vertex-Fetch Clauses,” page 5-1 and Section 7.3, “Vertex-Fetch Instructions,” page 7-181.

Microcode

B	WQM	CF_INST	VPM	EOP	Rsvd	CALL_COUNT	COUNT	COND	CF_CONST	PC	+4
ADDR											+0

Format `CF_DWORD0` (page 8-3) and `CF_DWORD1` (page 8-4).

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

7.2 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.

Add Floating-Point

Instructions **ADD**

Description Floating-point add.

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

Microcode

C	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).

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 7.1 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

C	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_ADD_64, opcode 23 (0x17).

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 element (slot 0) or Z element (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 element (slot 0) or Z element (slot 2).

Add Integer

Instructions **ADD_INT**

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

`dst = src0 + src1;`

Microcode

C	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).

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

AND Bitwise

Instructions **`AND_INT`**

Description Logical bit-wise AND.

`dst = src0 & src1;`

Microcode

C	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).

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. `src1` is interpreted as an unsigned integer. If `src1` is > 31, the result is either 0 or -1, depending on the sign of `src0`.

`dst = src0 >> src1`

Microcode

C	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A		+4
L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL			+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).

Instruction Field `ALU_INST == OP2_INST_ASHR_INT`, opcode 112 (0x70).

Floating-Point Ceiling

Instructions **CEIL**

Description Floating-point ceiling.

```
dst = TRUNC(src0);
If ( (src0 > 0.0f) && (src0 != dst) ) {
    dst += 1.0f;
}
```

Microcode

C	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).

Instruction Field `ALU_INST == OP2_INST_CEIL`, opcode 18 (0x12).

Floating-Point Conditional Move If Equal

Instructions **CMOVE**

Description Floating-point conditional move if equal.

```
If (src0 == 0.0f) {
    dst = src1;
}
Else {
    dst = src2;
}
```

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.

Microcode

C	D E	D R	DST_GPR					B S	ALU_INST (**11000**)	S 2 N	S 2 E	S 2 R	SRC2_SEL	+4
L	P S	I M		S 1 N	S 1 E	S 1 R	SRC1_SEL			S 0 N	S 0 E	S 0 R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP3 (page 8-23).

Instruction Field ALU_INST == OP3_INST_CMOVE, opcode 24 (0x18).

Integer Conditional Move If Equal

Instructions **`CMOVE_INT`**

Description Integer conditional move if equal, based on signed or unsigned integer operand. Compare `CMOVE` on page 7-49.

```
If (src0 == 0x0) {
    dst = src1;
}
Else {
    dst = src2;
}
```

Microcode

C	D E	D R	DST_GPR	B S	ALU_INST (**11000**)	S 2 N	S 2 E	S 2 R	SRC2_SEL	+4

L	P S	I M	S 1 N	S 1 E	S 1 R	SRC1_SEL	S 0 N	S 0 E	S 0 R	SRC0_SEL	+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP3` (page 8-23).

Instruction Field `ALU_INST == OP3_INST_CMOVE_INT`, opcode 28 (0x1C).

Floating-Point Conditional Move If Greater Than Or Equal

Instructions **CMOVGE**

Description Floating-point conditional move if greater than or equal. Compare CMOVE on page 7-49.

```
If (src0 >= 0.0f) {
    dst = src1;
}
Else {
    dst = src2;
}
```

Microcode

C	D E	D R	DST_GPR	B S	ALU_INST (**11000**)	S 2 N	S 2 E	S 2 R	SRC2_SEL	+4

L	P S	I M	S 1 N	S 1 E	S 1 R	SRC1_SEL	S 0 N	S 0 E	S 0 R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP3 (page 8-23).

Instruction Field ALU_INST == OP3_INST_CMOVGE, opcode 26 (0x1A).

Integer Conditional Move If Greater Than Or Equal

Instructions **CMOVGE_INT**

Description Integer conditional move if greater than or equal, based on signed integer operand. Compare CMOVE on page 7-49.

```
If (src0 >= 0x0) {
    dst = src1;
}
Else {
    dst = src2;
}
```

Microcode

C	D E	D R	DST_GPR		B S	ALU_INST (**11000**)	S 2 N	S 2 E	S 2 R	SRC2_SEL	+4
L	P S	I M	S 1 N	S 1 E	S 1 R	SRC1_SEL	S 0 N	S 0 E	S 0 R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP3 (page 8-23).

Instruction Field ALU_INST == OP3_INST_CMOVGE_INT, opcode 30 (0x1E).

Floating-Point Conditional Move If Greater Than

Instructions **CMOVGT**

Description Floating-point conditional move if greater than. Compare CMOVE on page 7-49.

```
If (src0 > 0.0f) {
    dst = src1;
}
Else {
    dst = src2;
}
```

Microcode

C	D E	D R	DST_GPR	B S	ALU_INST **(11000)**	S 2 N	S 2 E	S 2 R	SRC2_SEL	+4

L	P S	I M	S 1 N	S 1 E	S 1 R	SRC1_SEL	S 0 N	S 0 E	S 0 R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP3 (page 8-23).

Instruction Field ALU_INST == OP3_INST_CMOVGT, opcode 25 (0x19).

Integer Conditional Move If Greater Than

Instructions **CMOVGT_INT**

Description Integer conditional move if greater than, based on signed integer operand. Compare CMOVE on page 7-49.

```
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_DWORD0 (page 8-16) and ALU_DWORD1_OP3 (page 8-23).

Instruction Field ALU_INST == OP3_INST_CMOVGT_INT, opcode 29 (0x1D).

Scalar Cosine

Instructions **`COS`**

Description Scalar cosine. Valid input domain [-PI, +PI].

`dst = ApproximateCos(src0);`

Microcode

C	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).

Instruction Field `ALU_INST == OP2_INST_COS`, opcode 111 (0x6F).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).

Instruction Field `ALU_INST == OP2_INST_CUBE`, opcode 82 (0x52).

Four-Element Dot Product

Instructions **DOT4**

Description Four-element dot product. 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.

Only the PV.X register element 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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_DOT4, opcode 80 (0x50).

Four-Element Dot Product, IEEE

Instructions **DOT4_IEEE**

Description Four-element dot product that uses IEEE rules for zero times anything. 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.

Only the PV.X register element 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	D E	D R	DST_GPR	B S	ALU_INST	OMO D	F M	W M	U P	U E M	S 1 A	S 0 A	+4

L	P S	I M	S 1 N	S 1 E	S 1 R	SRC1_SEL	S 0 N	S 0 E	S 0 R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_DOT4_IEEE, opcode 81 (0x51).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_EXP_IEEE, opcode 97 (0x61).

Floating-Point Floor

Instructions **FLOOR**

Description Floating-point floor.

```
dst = TRUNC(src0);
If ( (src0 < 0.0f) && (src0 != dst) ) {
    dst += -1.0f;
}
```

Microcode

C	D E	D R	DST_GPR	B S	ALU_INST	OMO D	F M	W M	U P	U E M	S 1 A	S 0 A	+4

L	P S	I M	S 1 N	S 1 E	S 1 R	SRC1_SEL	S 0 N	S 0 E	S 0 R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_FLOOR, opcode 20 (0x14).

Floating-Point To Integer

Instructions **`FLT_TO_INT`**

Description 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.

`dst = (int)src0`

Microcode

C	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4
L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL			+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).

Instruction Field `ALU_INST == OP2_INST_FLT_TO_INT`, opcode 107 (0x6B).

Floating-Point 32-Bit To Floating-Point 64-Bit

Instructions **FLT32_T0_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 7.2 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_T0_FLT64 is a two-slot instruction. The following coissue scenarios are possible:.

- A single FLT32_T0_FLT64 instruction in slots 0 and 1, and any valid instructions in slots 2, 3, and 4.
- A single FLT32_T0_FLT64 instruction in slots 2 and 3, and any valid instructions in slots 0, 1, and 4.
- Two FLT32_T0_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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).

Instruction Field `ALU_INST == OP2_INST_FLT32_TO_FLT64`, opcode 29 (0x1D).

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 element (slot 0) or Z element (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 element (slot 0) or Z element (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 7.3 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	D E	D R	DST_GPR	B S	ALU_INST	OMO D	F M	W M	U P	U E M	S 1 A	S 0 A	+4

L	P S	I M	S 1 N	S 1 E	S 1 R	SRC1_SEL	S 0 N	S 0 E	S 0 R	SRC0_SEL	+0

Format	ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).
Instruction Field	ALU_INST == OP2_INST_FLT64_TO_FLT32, opcode 28 (0x1C).

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 element (slot 0) or Z element (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 element (slot 0) or Z element (slot 2).

Floating-Point Fractional

Instructions **FRACT**

Description Floating-point fractional part of source operand.

```
dst = src0 - FLOOR(src0);
```

Microcode

C	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

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 7.4 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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4
L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0		

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).

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

Instruction Field ALU_INST == OP2_INST_FRACT_64, opcode 123 (0x7B).

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 element (slot 0) or Z element (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 element (slot 0) or Z element (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 7.5 Result of FREXP_64 Instruction

dst	src0			
	-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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_FREXP_64, opcode 7 (0x7).

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 element (slot 0). This slot contains the sign bit of the source.

Output Modifiers The instruction does not take output modifiers.

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

C	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_INT_TO_FLT, opcode 108 (0x6C).

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	D E	D R	DST_GPR	B S	ALU_INST	OMO D	F M	W M	U P	U E M	S 1 A	S 0 A	+4

L	P S	I M	S 1 N	S 1 E	S 1 R	SRC1_SEL	S 0 N	S 0 E	S 0 R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_KILLE, opcode 44 (0x2C).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_KILLGE, opcode 46 (0x2E).

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	D E	D R	DST_GPR	B S	ALU_INST	OMO D	F M	W M	U P	U E M	S 1 A	S 0 A	+4

L	P S	I M	S 1 N	S 1 E	S 1 R	SRC1_SEL	S 0 N	S 0 E	S 0 R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_KILLGT, opcode 45 (0x2D).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_KILLNE, opcode 47 (0x2F).

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 7.6 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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4
L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0		

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_LDEXP_64, opcode 122 (0x7A).

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 element (slot 0) or Z element (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 element (slot 0) or Z element (slot 2).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_LOG_CLAMPED, opcode 98 (0x62).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4		
L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0				

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_LOG_IEEE, opcode 99 (0x63).

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, then the result is 0.

```
dst = src0 << src1
```

Microcode

C	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).

Instruction Field `ALU_INST == OP2_INST_LSHL_INT`, opcode 114 (0x72).

Scalar Logical Shift Right

Instructions **`LSHR_INT`**

Description Scalar logical shift right. Zero is shifted into the vacated locations. `src1` is interpreted as an unsigned integer. If `src1` is > 31, then the result is 0.

`dst = src0 << src1`

Microcode

C	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).

Instruction Field `ALU_INST == OP2_INST_LSHR_INT`, opcode 113 (0x71).

Floating-Point Maximum

Instructions **MAX**

Description Floating-point maximum.

```
If (src0 >= src1) {
    dst = src0;
}
Else {
    dst = src1;
}
```

Microcode

C	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).

Instruction Field `ALU_INST == OP2_INST_MAX`, opcode 3 (0x3).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4	
L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0			

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).

Instruction Field `ALU_INST == OP2_INST_MAX_UINT`, opcode 56 (0x38).

Four-Element Maximum

Instructions **MAX4**

Description Four-element maximum. The result is replicated in all four vector elements. 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.

Only the PV.X register element 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

C	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_MAX4, opcode 83 (0x53).

Floating-Point Minimum

Instructions **MIN**

Description Floating-point minimum.

```
If (src0 < src1) {
    dst = src0;
}
Else {
    dst = src1;
}
```

Microcode

C	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4
L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL			+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_MIN, opcode 4 (0x4).

Floating-Point Minimum, DirectX 10

Instructions **`MIN_DX10`**

Description Floating-point minimum. This instruction uses the DirectX 10 method of handling of `NaN`s.

```
If (src0 < src1) {
    dst = src0;
}
Else {
    dst = src1;
}
```

Microcode

C	D E	D R	DST_GPR	B S	ALU_INST	OMO D	F M	W M	U P	U E M	S 1 A	S 0 A	+4

L	P S	I M	S 1 N	S 1 E	S 1 R	SRC1_SEL	S 0 N	S 0 E	S 0 R	SRC0_SEL	+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format	ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).
Instruction Field	ALU_INST == OP2_INST_MOV, opcode 25 (0x19).

Copy Rounded Floating-Point To Integer in AR and GPR

Instructions **MOVA**

Description Round floating-point to the nearest integer in the range [-256, +255], and copy the result to the address register (AR) and to a GPR.

When the destination is a GPR, the destination contains a 1-element scalar address that is used for GPR-relative addressing in the ALU. This GPR-index state only persists for one ALU clause, and it is only available for relative addressing within the ALU (it is not available for relative texture-fetch, vertex-fetch, or export addressing).

When the destination is the AR register, the instruction copies the four elements of a source GPR into the AR register; they are used as the index value for constant-file relative addressing (constant waterfalling). The MOVA* instructions write vector elements of the AR register. 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. MOVA* issued in ALU.X writes AR.X regardless of any GPR write mask used. Predication is supported.

MOVA* instructions must not be used in an instruction group that uses GPR or 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 a GPR destination write, clear the WRITE_MASK field for the MOVA* instruction. Do not use the corresponding PV vector element(s) in the following ALU instruction group.

```
dst = Undefined
dstF = FLOOR(src0 + 0.5f);
If (dstF >= -256.0f) {
    dstF = dstF;
}
Else {
    dstF = -256.0f;
}
If (dstF > 255.0f) {
    dstF = -256.0f;
}
dstI = truncate_to_int(dstF);
Export(dstI); // signed 9-bit integer
```

Microcode

D E	D R	DST_GPR	B S	ALU_INST	OMO D	F M	W M	U P	U E M	S 1 A	S 0 A		+4

P S	I M	S 1 N	S 1 E	S 1 R	SRC1_SEL	S 0 N	S 0 E	S 0 R	SRC0_SEL		+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_MOVA, opcode 21 (0x15).

Copy Truncated Floating-Point To Integer in AR and GPR

Instructions **MOVA_FLOOR**

Description Truncate the floating-point to the nearest integer in the range [-256, +255], and copy the result to the address register (AR) and to a GPR. See MOVA on page 7-92 for additional details.

```
dst = Undefined
dstF = FLOOR(src0);
If (dstF >= -256.0f) {
    dstF = dstF;
}
Else {
    dstF = -256.0f;
}
If (dstF > 255.0f) {
    dstF = -256.0f;
}
dstI = truncate_to_int(dstF);
Export(dstI); // signed 9-bit integer
```

Microcode

C	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_MOVA_FLOOR, opcode 22 (0x16).

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. See MOVA on page 7-92 for additional details.

```
dst = Undefined;
dstI = src0;
If (dstI < -256) {
   dstI = 0x800; //-256
}
If (dstI > 0xFF) {
   dstI = 0x800 //-256
}
Export(dstI); // signed 9-bit integer
```

Microcode

C	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_MOVA_INT, opcode 24 (0x18).

Floating-Point Multiply

Instructions **`MUL`**

Description Floating-point multiply. Zero times anything equals zero.

```
dst = src0 * src1;
```

Microcode

C	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).

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 7.7 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

DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A		+4

PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL		+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

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

Instruction Field ALU_INST == OP2_INST_MUL_64, opcode 27 (0x1B).

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 element (slot 0), Y element (slot 1), or Z element (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 element (slot 0) or Z element (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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_MUL_IEEE, opcode 2 (0x2).

Scalar Multiply Emulating LIT Operation

Instructions **MUL_LIT**

Description Scalar multiply with result replicated in all four vector elements. 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	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_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP3_INST_MUL_LIT, opcode 12 (0xC).

Scalar Multiply Emulating LIT, Divide By 2

Instructions **MUL_LIT_D2**

Description A MUL_LIT operation, followed by divide by 2.

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 * .5;
}
Else {
    dst = (src0 * src1) * .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_DWORD0 (page 8-16) and ALU_DWORD1_OP3 (page 8-23).

Instruction Field ALU_INST == OP3_INST_MUL_LIT_D2, opcode 15 (0xF).

Scalar Multiply Emulating LIT, Multiply By 2

Instructions **MUL_LIT_M2**

Description A MUL_LIT operation, followed by multiply by 2.

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 * 2;
}
Else {
    dst = (src0 * src1) * 2;

}
```

Microcode

C	D E	D R	DST_GPR	B S	ALU_INST (**11000**)	S 2 N	S 2 E	S 2 R	SRC2_SEL	+4

L	P S	I M	S 1 N	S 1 E	S 1 R	SRC1_SEL	S 0 N	S 0 E	S 0 R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP3 (page 8-23).

Instruction Field ALU_INST == OP3_INST_MUL_LIT_M2, opcode 13 (0xD).

Scalar Multiply Emulating LIT, Multiply By 4

Instructions **`MUL_LIT_M4`**

Description A `MUL_LIT` operation, followed by multiply by 4.

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 * 4;
}
Else {
    dst = (src0 * src1) * 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_DWORD0` (page 8-16) and `ALU_DWORD1_OP3` (page 8-23).

Instruction Field `ALU_INST == OP3_INST_MUL_LIT_M4`, opcode 14 (0xE).

Floating-Point Multiply-Add

Instructions **`MULADD`**

Description Floating-point multiply-add (MAD).

```
dst = src0 * src1 + src2;
```

Microcode

C	D E	D R	DST_GPR	B S	ALU_INST (**11000**)	S 2 N	S 2 E	S 2 R	SRC2_SEL	+4

L	P S	I M	S 1 N	S 1 E	S 1 R	SRC1_SEL	S 0 N	S 0 E	S 0 R	SRC0_SEL	+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP3` (page 8-23).

Instruction Field `ALU_INST == OP3_INST_MULADD`, opcode 16 (0x10).

Floating-Point Multiply-Add, 64-Bit

Instructions **MULADD_64**

Description Floating-point 64-bit multiply-add. Multiplies the double-precision value in `src0.YX` by the double-precision value in `src1.YX`, adds the lower 64 bits of the result to a double-precision value in `src2.YX`, and places this result in `dst.YX` and `dst.WZ`.

```
dst = src0 * src1 + src2;
```

Table 7.8 Result of MULADD_64 Instruction (IEEE Single-Precision Multiply)

	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.

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

Table 7.9 Result of MULADD_64 Instruction (IEEE Add)

	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	-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	-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.

Coissue The `MULADD_64` instruction is a four-slot instruction. Therefore, a single `MULADD_64` instruction can be issued in slots 0, 1, 2, and 3. Slot 4 can contain any other valid instruction.

Microcode

C	D E	D R	DST_GPR	B S	ALU_INST (**11000**)	S 2 N	S 2 E	S 2 R	SRC2_SEL		+4
L	P S	I M	S 1 N	S 1 E	S 1 R	SRC1_SEL	S 0 N	S 0 E	S 0 R	SRC0_SEL	+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP3` (page 8-23).

Instruction Fields `ALU_INST == OP3_INST_MULADD_64`, opcode 8 (0x8).
`ALU_INST == OP3_INST_MULADD_64_M2`, opcode 9 (0x9).
`ALU_INST == OP3_INST_MULADD_64_M4`, opcode 10 (0xA).
`ALU_INST == OP3_INST_MULADD_64_D2`, opcode 11 (0xB).

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

Example

The following example coissues one MULADD_64 instruction in slots 0, 1, 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 re.h, l(0x40280000)   //high dword (Input 3)
mov rf.l, l(0x00000000)   //low  dword

Issue instruction:

MULADD_64  rg.x ra.h rc.h re.h; //can be any vector element
MULADD_64  rh.y ra.h rc.h re.h; //can be any vector element
MULADD_64  ri.z ra.h rc.h re.h; //can be any vector element
MULADD_64  rj.w rb.l rd.l rf.l; //can be any vector element

Result:
(3.0 * 6.0) + 12.0 = 30.0 (0x403e000000000000)

rg.x  = 0x00000000 (LSB of muladd64 result)
rh.y  = 0x403e0000 (MSB of muladd64 result)
ri.z  = 0x00000000 (LSB of muladd64 result)
rj.w  = 0x403e0000 (MSB of muladd64 result)
```

The hardware puts the result on 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 element (slot 0), Y element (slot 1), or Z element (slot 2). These slots contain the sign bits of the sources.

Output Modifiers

The OMOD output modifier (Section 4.9.1, on page 4-25) is not needed, because the MULADD_64 instruction has different opcodes for each of the OMOD values. The CLAMP output modifier can be applied to the destination during the destination X element (slot 0) or Z element (slot 2).

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_DWORD0 (page 8-16) and ALU_DWORD1_OP3 (page 8-23).

Instruction Field ALU_INST == OP3_INST_MULADD_D2, opcode 19 (0x13).

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_DWORD0` (page 8-16) and `ALU_DWORD1_OP3` (page 8-23).

Instruction Field `ALU_INST == OP3_INST_MULADD_M2`, opcode 17 (0x11).

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_DWORD0 (page 8-16) and ALU_DWORD1_OP3 (page 8-23).

Instruction Field ALU_INST == OP3_INST_MULADD_M4, opcode 18 (0x12).

IEEE Floating-Point Multiply-Add

Instructions **`MULADD_IEEE`**

Description Floating-point multiply-add (MAD). Uses IEEE rules for zero times anything.

```
dst = src0 * src1 + src2;
```

Microcode

C	D E	D R	DST_GPR	B S	ALU_INST (**11000**)	S 2 N	S 2 E	S 2 R	SRC2_SEL	+4

L	P S	I M	S 1 N	S 1 E	S 1 R	SRC1_SEL	S 0 N	S 0 E	S 0 R	SRC0_SEL	+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP3` (page 8-23).

Instruction Field `ALU_INST == OP3_INST_MULADD_IEEE`, opcode 20 (0x14).

IEEE Floating-Point Multiply-Add, Divide by 2

Instructions **MULADD_IEEE_D2**

Description Floating-point multiply-add (MAD), followed by divide by 2. Uses IEEE rules for zero times anything.

```
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_DWORD0 (page 8-16) and ALU_DWORD1_OP3 (page 8-23).

Instruction Field ALU_INST == OP3_INST_MULADD_IEEE_D2, opcode 23 (0x17).

IEEE Floating-Point Multiply-Add, Multiply by 2

Instructions **`MULADD_IEEE_M2`**

Description Floating-point multiply-add (MAD), followed by multiply by 2. Uses IEEE rules for zero times anything.

```
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_DWORD0` (page 8-16) and `ALU_DWORD1_OP3` (page 8-23).

Instruction Field `ALU_INST == OP3_INST_MULADD_IEEE_M2`, opcode 21 (0x15).

IEEE Floating-Point Multiply-Add, Multiply by 4

Instructions **MULADD_IEEE_M4**

Description Floating-point multiply-add (MAD), followed by multiply by 4. Uses IEEE rules for zero times anything.

```
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_DWORD0` (page 8-16) and `ALU_DWORD1_OP3` (page 8-23).

Instruction Field `ALU_INST == OP3_INST_MULADD_IEEE_M4`, opcode 22 (0x16).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4
L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL			+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).

Instruction Field `ALU_INST == OP2_INST_MULHI_INT`, opcode 116 (0x74).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).

Instruction Field `ALU_INST == OP2_INST_MULHI_UINT`, opcode 118 (0x76).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_MULLO_INT, opcode 115 (0x73).

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

C	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).

Instruction Field `ALU_INST == OP2_INST_MULLO_UINT`, opcode 117 (0x75).

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 elements of the PV and PS registers.

See the CF version of NOP on page 7-33.

dst is Undefined.

Previous `dst` is preserved

Microcode

C	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4
L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0		

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).

Instruction Field `ALU_INST == OP2_INST_NOT_INT`, opcode 51 (0x33).

Bit-Wise OR

Instructions **`OR_INT`**

Description Logical bit-wise OR.

`dst = src0 | src1`

Microcode

C	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).

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	D E	D R	DST_GPR	B S	ALU_INST	OMO D	F M	W M	U P	U E M	S 1 A	S 0 A	+4
L	P S	I M	S 1 N	S 1 E	S 1 R	SRC1_SEL	S 0 N	S 0 E	S 0 R	SRC0_SEL			+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

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	D E	D R	DST_GPR	B S	ALU_INST	OMOD	F M	W M	U P	U E M	S 1 A	S 0 A	+4

L	P S	I M	S 1 N	S 1 E	S 1 R	SRC1_SEL	S 0 N	S 0 E	S 0 R	SRC0_SEL	+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4
L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL			+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4
L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0		

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

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 7.10 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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_PRED_SETE_64, opcode 125 (0x7D).

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 element (slot 0) and Z element (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 **PRED_SETE_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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_PRED_SETE_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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A			+4
L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL					+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

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	D E	D R	DST_GPR	B S	ALU_INST	OMO D	F M	W M	U P	U E M	S 1 A	S 0 A	+4

L	P S	I M	S 1 N	S 1 E	S 1 R	SRC1_SEL	S 0 N	S 0 E	S 0 R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

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

C	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

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 7.11 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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_PRED_SETGE_64, opcode 126 (0x7E).

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 element (slot 0) and Z element (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	D E	D R	DST_GPR	B S	ALU_INST	OMO D	F M	W M	U P	U E M	S 1 A	S 0 A	+4

L	P S	I M	S 1 N	S 1 E	S 1 R	SRC1_SEL	S 0 N	S 0 E	S 0 R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

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

C	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_PRED_SETGE_PUSH_INT, opcode 76 (0x4C).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4		
L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0				

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

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 7.12 Result of PRED_SETGT_64 Instruction

src0	src1								
	-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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_PRED_SETGT_64, opcode 124 (0x7C).

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 element (slot 0) and Z element (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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4
L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL			+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

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	D E	D R	DST_GPR	B S	ALU_INST	OMO D	F M	W M	U P	U E M	S 1 A	S 0 A	+4

L	P S	I M	S 1 N	S 1 E	S 1 R	SRC1_SEL	S 0 N	S 0 E	S 0 R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_PRED_SETGT_PUSH_INT, opcode 75 (0x4B).

Integer Predicate Set If Less Than Or Equal

Instructions **PRED_SETLE_INT**

Description Integer predicate set if less than or equal. Updates predicate register.

```
If (src0 <= src1) {
    dst = 0.0f;
    SetPredicateKillReg (Execute);
}
Else {
    dst = 1.0f;
    SetPredicateKillReg (Skip);
}
```

Microcode

C	D E	D R	DST_GPR	B S	ALU_INST	OMO D	F M	W M	U P	U E M	S 1 A	S 0 A	+4

L	P S	I M	S 1 N	S 1 E	S 1 R	SRC1_SEL	S 0 N	S 0 E	S 0 R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_PRED_SETLE_INT, opcode 71 (0x47).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_PRED_SETLE_PUSH_INT, opcode 79 (0x4F).

Integer Predicate Set If Less Than Or Equal

Instructions **PRED_SETLT_INT**

Description Integer predicate set if less than. Updates predicate register.

```
If (src0 < src1) {
    dst = 0.0f;
    SetPredicateKillReg (Execute);
}
Else {
    dst = 1.0f;
    SetPredicateKillReg (Skip);
}
```

Microcode

C	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_PRED_SETLT_INT, opcode 70 (0x46).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

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	D E	D R	DST_GPR	B S	ALU_INST	OMO D	F M	W M	U P	U E M	S 1 A	S 0 A	+4

L	P S	I M	S 1 N	S 1 E	S 1 R	SRC1_SEL	S 0 N	S 0 E	S 0 R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

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	D E	D R	DST_GPR	B S	ALU_INST	OMO D	F M	W M	U P	U E M	S 1 A	S 0 A	+4

L	P S	I M	S 1 N	S 1 E	S 1 R	SRC1_SEL	S 0 N	S 0 E	S 0 R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4
L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL			+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_PRED_SETNE_PUSH_INT, opcode 77 (0x4D).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format	`ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).
Instruction Field	`ALU_INST == OP2_INST_RECIP_CLAMPED`, opcode 100 (0x64).

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	D E	D R	DST_GPR	B S	ALU_INST	OMO D	F M	W M	U P	U E M	S 1 A	S 0 A	+4
L	P S	I M	S 1 N	S 1 E	S 1 R	SRC1_SEL	S 0 N	S 0 E	S 0 R	SRC0_SEL			+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).

Instruction Field `ALU_INST == OP2_INST_RECIP_FF`, opcode 101 (0x65).

Scalar Reciprocal, IEEE Approximation

Instructions **RECIP_IEEE**

Description Scalar reciprocal.

```
If (src0 == 1.0f) {
    dst = 1.0f;
}
Else {
    dst = ApproximateRecip(src0);
}
```

Microcode

C	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_RECIP_IEEE, opcode 102 (0x66).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4
L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL			+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).

Instruction Field `ALU_INST == OP2_INST_RECIP_INT`, opcode 119 (0x77).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4
L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL			+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).

Instruction Field `ALU_INST == OP2_INST_RECIP_UINT`, opcode 120 (0x78).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_RECIPSQRT_CLAMPED, opcode 103 (0x67).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4
L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0		

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_RECIPSQRT_FF, opcode 104 (0x68).

Scalar Reciprocal Square Root, IEEE Approximation

Instructions **RECIPSQRT_IEEE**

Description Scalar reciprocal square root.

```
If (src0 == 1.0f) {
    dst = 1.0f;
}
Else {
    dst = ApproximateRecipSqrt(srcC);
}
```

Microcode

C	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).

Instruction Field `ALU_INST == OP2_INST_RECIPSQRT_IEEE`, opcode 105 (0x69).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).

Instruction Field `ALU_INST == OP2_INST_RNDNE`, opcode 19 (0x13).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).

Instruction Field `ALU_INST == OP2_INST_SETE`, opcode 8 (0x8).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4
L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL			+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).

Instruction Field `ALU_INST == OP2_INST_SETGE`, opcode 10 (0xA).

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	D E	D R	DST_GPR	B S	ALU_INST	OMOD	F M	W M	U P	U E M	S 1 A	S 0 A	+4

L	P S	I M	S 1 N	S 1 E	S 1 R	SRC1_SEL	S 0 N	S 0 E	S 0 R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

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	D E	D R	DST_GPR	B S	ALU_INST	OMOD	F M	W M	U P	U E M	S 1 A	S 0 A	+4

L	P S	I M	S 1 N	S 1 E	S 1 R	SRC1_SEL	S 0 N	S 0 E	S 0 R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).

Instruction Field `ALU_INST == OP2_INST_SETGT`, opcode 9 (0x9).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4
L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL			+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

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	D E	D R	DST_GPR	B S	ALU_INST	OMO D	F M	W M	U P	U E M	S 1 A	S 0 A	+4

L	P S	I M	S 1 N	S 1 E	S 1 R	SRC1_SEL	S 0 N	S 0 E	S 0 R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_SETNE, opcode 11 (0xB).

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	D E	D R	DST_GPR	B S	ALU_INST	OMO D	F M	W M	U P	U E M	S 1 A	S 0 A	+4

L	P S	I M	S 1 N	S 1 E	S 1 R	SRC1_SEL	S 0 N	S 0 E	S 0 R	SRC0_SEL	+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).

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	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_SETNE_INT, opcode 61 (0x3D).

Scalar Sine

Instructions **`SIN`**

Description Scalar sine. Valid input domain [-PI, +PI].

```
dst = ApproximateSin(src0);
```

Microcode

C	D E	D R	DST_GPR	B S	ALU_INST	OMO D	F M	W M	U P	U E M	S 1 A	S 0 A	+4

L	P S	I M	S 1 N	S 1 E	S 1 R	SRC1_SEL	S 0 N	S 0 E	S 0 R	SRC0_SEL	+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).

Instruction Field `ALU_INST == OP2_INST_SIN`, opcode 110 (0x6E).

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 = ApproximateRecipSqrt(srcC);
}
```

Microcode

C	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_SQRT_IEEE, opcode 106 (0x6A).

Integer Subtract

Instructions **`SUB_INT`**

Description Integer subtract, based on signed or unsigned integer source operands.

```
dst = src1 – src0;
```

Microcode

C	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4
L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0		

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).

Instruction Field `ALU_INST == OP2_INST_SUB_INT`, opcode 53 (0x35).

Floating-Point Truncate

Instructions **TRUNC**

Description Floating-point integer part of source operand.

```
dst = trunc(src0);
```

Microcode

C	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4
L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0		

Format ALU_DWORD0 (page 8-16) and ALU_DWORD1_OP2 (page 8-18).

Instruction Field ALU_INST == OP2_INST_TRUNC, opcode 17 (0x11).

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

C	D E	D R	DST_GPR	B S	ALU_INST	OMO D	F M	W M	U P	U E M	S 1 A	S 0 A	+4

L	P S	I M	S 1 N	S 1 E	S 1 R	SRC1_SEL	S 0 N	S 0 E	S 0 R	SRC0_SEL	+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).

Instruction Field `ALU_INST == OP2_INST_UINT_TO_FLT`, opcode 109 (0x6D).

Bit-Wise XOR

Instructions **`XOR_INT`**

Description Logical bit-wise XOR.

`dst = src0 ^ src1`

Microcode

C	DE	DR	DST_GPR	BS	ALU_INST	OMOD	FM	WM	UP	UEM	S1A	S0A	+4

L	PS	IM	S1N	S1E	S1R	SRC1_SEL	S0N	S0E	S0R	SRC0_SEL	+0

Format `ALU_DWORD0` (page 8-16) and `ALU_DWORD1_OP2` (page 8-18).

Instruction Field `ALU_INST == OP2_INST_XOR_INT`, opcode 50 (0x32).

7.3 Vertex-Fetch Instructions

All of the instructions in this section have a mnemonic that begins with VTX_INST_ in the VTX_INST field of their microcode formats.

Vertex Fetch

Instructions **FETCH**

Description Vertex fetch (X = unsigned integer index). These fetches specify the destination GPR directly.

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	0	+12
Reserved												MF	CBNS	ES		OFFSET																+8
SMA	FCA	NFA		DATA_FORMAT						UCF	DSW			DSZ			DSY			DSX				DR	DST_GPR							+4
MFC						SSX		SR	SRC_GPR							BUFFER_ID								FWQ	FT		VTX_INST					+0

Format VTX_DWORD0 (page 8-25), VTX_DWORD1_GPR (page 8-29), and VTX_DWORD2 (page 8-33).

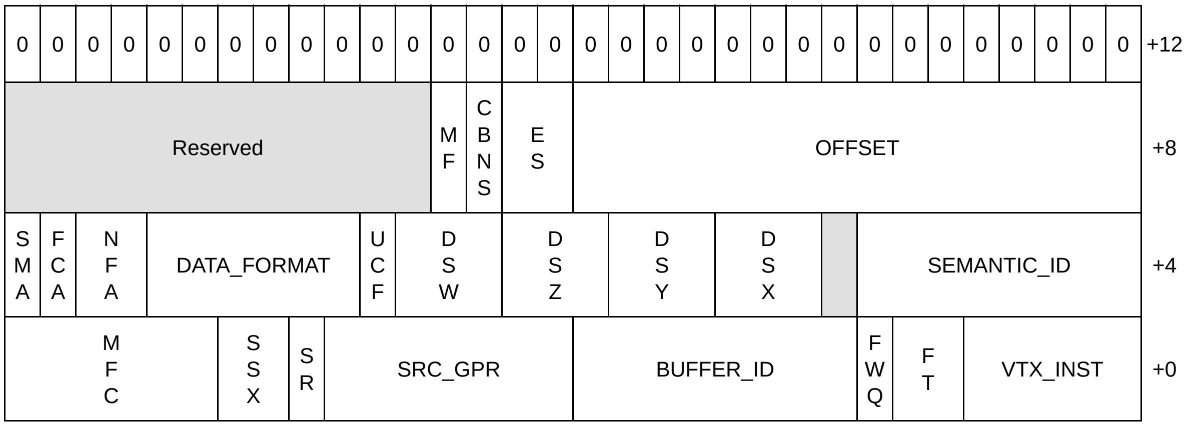
Instruction Field VTX_INST == VTX_INST_FETCH, opcode 0 (0x0).

Semantic Vertex Fetch

Instructions **SEMANTIC**

Description Semantic vertex fetch. 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_DWORD0` (page 8-25), `VTX_DWORD1_SEM` (page 8-27), and `VTX_DWORD2` (page 8-33).

Instruction Field `VTX_INST == VTX_INST_SEMANTIC`, opcode 1 (0x1).

7.4 Texture-Fetch Instructions

All of the instructions in this section have a mnemonic that begins with TEX_INST_ in the TEX_INST field of their microcode formats.

Get Computed Level of Detail For Pixels

Instructions **GET_COMP_TEX_LOD**

Description Computed level of detail (LOD) for all pixels in quad.

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	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	▨	DR	DST_GPR																					+4
Reserved	SR	SRC_GPR	RESOURCE_ID	FWQ	▨	BFM	TEX_INST																									+0

Format TEX_DWORD0 (page 8-34), TEX_DWORD1 (page 8-36), and TEX_DWORD2 (page 8-37).

Instruction Field TEX_INST == TEX_INST_GET_COMP_TEX_LOD, opcode 6 (0x6).

Get Slopes Relative To Horizontal

Instructions **GET_GRADIENTS_H**

Description Retrieve lopes relative to horizontal: X = dx/dh, Y = dy/dh, Z = dz/dh, W = dw/dh.

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	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			▨	DR	DST_GPR							+4
Reserved								SR	SRC_GPR							RESOURCE_ID								FWQ	▨	BFM	TEX_INST					+0

Format TEX_DWORD0 (page 8-34), TEX_DWORD1 (page 8-36), and TEX_DWORD2 (page 8-37).

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

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	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	▨	DR	DST_GPR	+4

Reserved	SR	SRC_GPR	RESOURCE_ID	FWQ	▨	BFM	TEX_INST	+0

Format TEX_DWORD0 (page 8-34), TEX_DWORD1 (page 8-36), and TEX_DWORD2 (page 8-37).

Instruction Field TEX_INST == TEX_INST_GET_GRADIENTS_V, opcode 8 (0x8).

Get Linear-Interpolation Weights

Instructions **GET_LERP_FACTORS**

Description Retrieve linear interpolation (LERP) weights used for bilinear fetch, X = horizontal LERP, Y = vertical LERP.

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	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				DR	DST_GPR							+4
Reserved								SR	SRC_GPR							RESOURCE_ID								FWQ		BFM	TEX_INST					+0

Format TEX_DWORD0 (page 8-34), TEX_DWORD1 (page 8-36), and TEX_DWORD2 (page 8-37).

Instruction Field TEX_INST == TEX_INST_GET_LERP_FACTORS, opcode 9 (0x9).

Get Number of Samples

Instructions **GET_NUMBER_OF_SAMPLES**

Description Gets and returns the number of samples.

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	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			▨	DR	DST_GPR							+4
Reserved								SR	SRC_GPR							RESOURCE_ID								FWQ	▨	BFM	TEX_INST					+0

Format `TEX_DWORD0` (page 8-34), `TEX_DWORD1` (page 8-36), and `TEX_DWORD2` (page 8-37).

Instruction Field `TEX_INST == TEX_INST_GET_LERP_FACTORS`, opcode 5 (0x5).

Get Texture Resolution

Instructions **GET_TEXTURE_RESINFO**

Description Retrieve width, height, depth, and number of mipmap levels.

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	0	+12
S S W			S S Z			S S Y			S S X			SAMPLER_ID					OFFSET_Z					OFFSET_Y					OFFSET_X					+8
C T W	C T Z	C T Y	C T X	LOD_BIAS							D S W			D S Z			D S Y			D S X			▨	D R	DST_GPR							+4
Reserved								S R	SRC_GPR							RESOURCE_ID								F W Q	▨	B F M	TEX_INST					+0

Format TEX_DWORD0 (page 8-34), TEX_DWORD1 (page 8-36), and TEX_DWORD2 (page 8-37).

Instruction Field TEX_INST == TEX_INST_GET_TEXTURE_RESINFO, opcode 4 (0x4).

Load Texture Elements

Instructions **LD**

Description Load texture element (texel). The elements X, Y, Z, W are unsigned integers.

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	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				DR	DST_GPR							+4
Reserved								SR	SRC_GPR							RESOURCE_ID								FWQ		BFM	TEX_INST					+0

Format `TEX_DWORD0` (page 8-34), `TEX_DWORD1` (page 8-36), and `TEX_DWORD2` (page 8-37).

Instruction Field `TEX_INST == TEX_INST_LD`, opcode 3 (0x3).

Return Memory Address

Instructions **PASS**

Description Returns the address read in memory.

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	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				DR	DST_GPR							+4
Reserved								SR	SRC_GPR							RESOURCE_ID								FWQ		BFM	TEX_INST					+0

Format `TEX_DWORD0` (page 8-34), `TEX_DWORD1` (page 8-36), and `TEX_DWORD2` (page 8-37).

Instruction Field `TEX_INST == TEX_INST_PASS`, opcode 13 (0xD).

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

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	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			▨	DR	DST_GPR							+4
Reserved								SR	SRC_GPR							RESOURCE_ID								FWQ	▨	BFM	TEX_INST					+0

Format TEX_DWORD0 (page 8-34), TEX_DWORD1 (page 8-36), and TEX_DWORD2 (page 8-37).

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

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	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		DR	DST_GPR																					+4
Reserved	SR	SRC_GPR	RESOURCE_ID	FWQ		BFM	TEX_INST																									+0

Format TEX_DWORD0 (page 8-34), TEX_DWORD1 (page 8-36), and TEX_DWORD2 (page 8-37).

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

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	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				DR	DST_GPR							+4
Reserved								SR	SRC_GPR							RESOURCE_ID								FWQ		BFM	TEX_INST					+0

Format `TEX_DWORD0` (page 8-34), `TEX_DWORD1` (page 8-36), and `TEX_DWORD2` (page 8-37).

Instruction Field `TEX_INST == TEX_INST_SAMPLE_C_G`, opcode 28 (0x1C).

Sample Texture with Comparison, Gradient, and LOD

Instructions **SAMPLE_C_G_L**

Description This instruction behaves exactly like the SAMPLE_C_G 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

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	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				DR	DST_GPR							+4
Reserved								SR	SRC_GPR							RESOURCE_ID								FWQ		BFM	TEX_INST					+0

Format TEX_DWORD0 (page 8-34), TEX_DWORD1 (page 8-36), and TEX_DWORD2 (page 8-37).

Instruction Field TEX_INST == TEX_INST_SAMPLE_C_G_L, opcode 29 (0x1D).

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

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	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	▨	DR	DST_GPR																					+4
Reserved	SR	SRC_GPR	RESOURCE_ID	FWQ	▨	BFM	TEX_INST																									+0

Format TEX_DWORD0 (page 8-34), TEX_DWORD1 (page 8-36), and TEX_DWORD2 (page 8-37).

Instruction Field TEX_INST == TEX_INST_SAMPLE_C_G_LB, opcode 30 (0x1E).

Sample Texture with Comparison, Gradient, and LOD Zero

Instructions **SAMPLE_C_G_LZ**

Description This instruction behaves exactly like the SAMPLE_C_G instruction, except that the mipmap level of detail (LOD) and fraction are forced to zero before level-clamping.

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	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				DR	DST_GPR							+4
Reserved								SR	SRC_GPR							RESOURCE_ID								FWQ		BFM	TEX_INST					+0

Format TEX_DWORD0 (page 8-34), TEX_DWORD1 (page 8-36), and TEX_DWORD2 (page 8-37).

Instruction Field TEX_INST == TEX_INST_SAMPLE_C_G_LZ, opcode 31 (0x1F).

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

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	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				DR	DST_GPR							+4
Reserved								SR	SRC_GPR							RESOURCE_ID								FWQ		BFM	TEX_INST					+0

Format TEX_DWORD0 (page 8-34), TEX_DWORD1 (page 8-36), and TEX_DWORD2 (page 8-37).

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	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				DR	DST_GPR							+4
Reserved								SR	SRC_GPR							RESOURCE_ID								FWQ		BFM	TEX_INST					+0

Format TEX_DWORD0 (page 8-34), TEX_DWORD1 (page 8-36), and TEX_DWORD2 (page 8-37).

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

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	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		DR	DST_GPR	+4																				
Reserved	SR	SRC_GPR	RESOURCE_ID	FWQ		BFM	TEX_INST	+0																								

Format TEX_DWORD0 (page 8-34), TEX_DWORD1 (page 8-36), and TEX_DWORD2 (page 8-37).

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

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	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			▨	DR	DST_GPR							+4
Reserved								SR	SRC_GPR							RESOURCE_ID								FWQ	▨	BFM	TEX_INST					+0

Format TEX_DWORD0 (page 8-34), TEX_DWORD1 (page 8-36), and TEX_DWORD2 (page 8-37).

Instruction Field TEX_INST == TEX_INST_SAMPLE_G, opcode 20 (0x14).

Sample Texture with Gradient and LOD

Instructions **SAMPLE_G_L**

Description This instruction behaves exactly like the SAMPLE_G 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

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	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				DR	DST_GPR							+4
Reserved								SR	SRC_GPR							RESOURCE_ID								FWQ		BFM	TEX_INST					+0

Format TEX_DWORD0 (page 8-34), TEX_DWORD1 (page 8-36), and TEX_DWORD2 (page 8-37).

Instruction Field TEX_INST == TEX_INST_SAMPLE_G_L, opcode 21 (0x15).

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

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	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	▨	DR	DST_GPR																					+4
Reserved	SR	SRC_GPR	RESOURCE_ID	FWQ	▨	BFM	TEX_INST																									+0

Format TEX_DWORD0 (page 8-34), TEX_DWORD1 (page 8-36), and TEX_DWORD2 (page 8-37).

Instruction Field TEX_INST == TEX_INST_SAMPLE_G_LB, opcode 22 (0x16).

Sample Texture with Gradient and LOD Zero

Instructions **SAMPLE_G_LZ**

Description This instruction behaves exactly like the SAMPLE_G instruction, except that the mipmap level of detail (LOD) and fraction are forced to zero before level-clamping.

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	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			▨	DR	DST_GPR							+4
Reserved								SR	SRC_GPR							RESOURCE_ID								FWQ	▨	BFM	TEX_INST					+0

Format TEX_DWORD0 (page 8-34), TEX_DWORD1 (page 8-36), and TEX_DWORD2 (page 8-37).

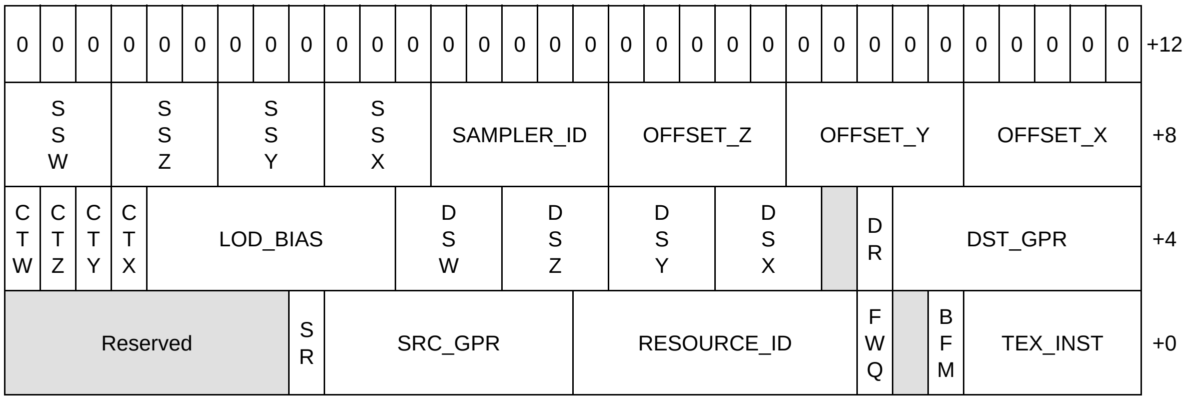
Instruction Field TEX_INST == TEX_INST_SAMPLE_G_LZ, opcode 23 (0x17).

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_DWORD0 (page 8-34), TEX_DWORD1 (page 8-36), and TEX_DWORD2 (page 8-37).

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

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	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	▨	DR	DST_GPR	+4																				
Reserved	SR	SRC_GPR	RESOURCE_ID	FWQ	▨	BFM	TEX_INST	+0																								

Format TEX_DWORD0 (page 8-34), TEX_DWORD1 (page 8-36), and TEX_DWORD2 (page 8-37).

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

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	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	▨	DR	DST_GPR																					+4
Reserved	SR	SRC_GPR	RESOURCE_ID	FWQ	▨	BFM	TEX_INST																									+0

Format TEX_DWORD0 (page 8-34), TEX_DWORD1 (page 8-36), and TEX_DWORD2 (page 8-37).

Instruction Field TEX_INST == TEX_INST_SAMPLE_LZ, opcode 19 (0x13).

Set Cubemap Index

Instructions **SET_CUBEMAP_INDEX**

Description Sets the index of the cubemap.

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	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				DR	DST_GPR							+4
Reserved								SR	SRC_GPR							RESOURCE_ID								FWQ		BFM	TEX_INST					+0

Format TEX_DWORD0 (page 8-34), TEX_DWORD1 (page 8-36), and TEX_DWORD2 (page 8-37).

Instruction Field TEX_INST == TEX_INST_SET_CUBEMAP_INDEX, opcode 14 (0xE).

Set Horizontal Gradients

Instructions **SET_GRADIENTS_H**

Description Set horizontal gradients specified by X, Y, Z coordinates.

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	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				DR	DST_GPR							+4
Reserved								SR	SRC_GPR							RESOURCE_ID								FWQ		BFM	TEX_INST					+0

Format TEX_DWORD0 (page 8-34), TEX_DWORD1 (page 8-36), and TEX_DWORD2 (page 8-37).

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

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	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			▨	DR	DST_GPR							+4
Reserved								SR	SRC_GPR							RESOURCE_ID								FWQ	▨	BFM	TEX_INST					+0

Format TEX_DWORD0 (page 8-34), TEX_DWORD1 (page 8-36), and TEX_DWORD2 (page 8-37).

Instruction Field TEX_INST == TEX_INST_SET_GRADIENTS_V, opcode 12 (0xC).

Chapter 8
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 8.1 summarizes the microcode formats and their widths. The sections that follow provide details.

Table 8.1 Summary of Microcode Formats

Microcode Formats	Reference	Width (bits)	Function
Control Flow (CF) Instructions			
CF_DWORD0 and CF_DWORD1	page 8-3 page 8-4	64	Implements general control-flow instructions.
CF_ALU_DWORD0 and CF_ALU_DWORD1	page 8-7 page 8-8	64	Initiates ALU clauses.
CF_ALLOC_EXPORT_DWORD0 and CF_ALLOC_EXPORT_DWORD1_ {BUF, SWIZ}	page 8-10 page 8-12, page 8-14, page 8-15	64	Initiates and implements allocation, import, and export instructions.
ALU Clause Instructions			
ALU_DWORD0 and ALU_DWORD1_OP2 or ALU_DWORD1_OP3	page 8-16 page 8-18, page 8-23	64	Implements ALU instructions.
Vertex-Fetch Clause Instructions			
VTX_DWORD0 and VTX_DWORD1_{GPR, SEM} and VTX_DWORD2	page 8-25 page 8-27, page 8-29 page 8-33	96, padded to 128	Implements vertex-fetch instructions.
Texture-Fetch Clause Instructions			
TEX_DWORD0 and TEX_DWORD1 and TEX_DWORD2	page 8-34 page 8-36 page 8-37	96, padded to 128	Implements texture-fetch 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_DWORD1_BUF` or the `CF_ALLOC_EXPORT_DWORD1_SWIZ` formats are the only exceptions). The default value of all fields is zero.

8.1 Control Flow (CF) Instructions

Control flow (CF) instructions include:

- General control flow instructions (conditional jumps, loops, subroutines).
- Allocate, import, or export instructions.
- Clause-initiation instructions for ALU, texture-fetch, vertex-fetch clauses.

All CF microcode formats are 64 bits wide.

Control Flow Doubleword 0

Instructions	**CF_DWORD0**
Description	This is the low-order (least-significant) doubleword in the 64-bit microcode-format pair formed by CF_DWORD[0,1]. This format pair is the default format for CF instructions.
Access	Read-write

Opcode

Field Name	*Bits*	*Format*
ADDR	[31:0]	int(32)

- (producing a quadword-aligned value) of the beginning of the clause in memory.
- For control flow instructions: Bits [34:3] of the byte offset (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.

Related CF_DWORD1

Control Flow Doubleword 1

Instructions **CF_DWORD1**

Description This is the high-order (most-significant) doubleword in the 64-bit microcode-format pair formed by CF_DWORD[0,1]. This format pair is the default format for CF instructions.

Access Read-write

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 stack. Can be zero to indicate no pop operation.	
CF_CONST	[7:3]	int(5)
	Specifies the CF constant to use for flow control statements. For LOOP_START_* and LOOP_END, this 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 loop index (aL) is maintained by hardware in the aL register. For instructions using the COND field, this specifies the index of the boolean constant. See Section 3.7.3, on page 3-18 for details.	
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.) 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	[12:10]	int(3)
	Number of instruction slots in the range [1,8] to execute in the clause, minus one (clause instructions only).	
CALL_COUNT	[18:13]	int(6)
	Amount to increment call nesting counter by when executing a CALL statement; a CALL is skipped if the current nesting depth + CALL_COUNT > 32. This field is interpreted in the range [0,31], and has no effect for other instruction types.	
RSVD	[19:20]	Reserved
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.)

VALID_PIXEL_MODE (VPM)	22	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 are inactive. This is the antonym of WHOLE_QUAD_MODE. Caution: VALID_PIXEL_MODE is not the default mode; this bit is cleared by default.
CF_INST	[29:23]	enum(7)
	0	CF_INST_NOP: perform no operation.
	1	CF_INST_TEX: execute texture-fetch or constant-fetch clause.
	2	CF_INST_VTX: execute vertex-fetch clause
	3	CF_INST_VTX_TC: execute vertex-fetch clause through the texture cache (for systems lacking VC).
	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 don't 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.
	12	CF_INST_PUSH_ELSE: execute push/else statement. Always pushes per-pixel state onto the stack.
	13	CF_INST_ELSE: execute else statement (can be conditional).
	14	CF_INST_POP: pop current per-pixel state from the stack.
	15	CF_INST_POP_JUMP: pop current per-pixel state from the stack; then, execute CF_INST_JUMP with pop count = 0.
	16	CF_INST_POP_PUSH: pop current per-pixel state from the stack; then, execute CF_INST_PUSH with pop count = 0.
	17	CF_INST_POP_PUSH_ELSE: pop current per-pixel state from the stack; then, execute CF_INST_PUSH_ELSE.
	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.
	20	CF_INST_RETURN: execute subroutine return instruction (pop stack). Pair with CF_INST_CALL only.
	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.
	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.)

	`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 instructions. 1 All prior instructions must complete before this instruction executes.	
Related	`CF_DWORD0`		

Control Flow ALU Doubleword 0

Instructions **CF_ALU_DWORD0**

Description This is the low-order (least-significant) doubleword in the 64-bit microcode-format pair formed by CF_ALU_DWORD[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 8.2, on page 8-15.

Access Read-write

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[0.1], ADDR. 2 CF_KCACHE_LOCK_2: lock cache lines KCACHE_BANK[0.1], ADDR and KCACHE_BANK[0.1], ADDR+1. 3 CF_KCACHE_LOCK_LOOP_INDEX: lock cache lines KCACHE_BANK[0.1], LOOP/16+ADDR and KCACHE_BANK[0.1], LOOP/16+ADDR+1, where LOOP is the current loop index (aL).	

Related CF_ALU_DWORD1

Control Flow ALU Doubleword 1

Instructions **CF_ALU_DWORD1**

Description This is the high-order (most-significant) doubleword in the 64-bit microcode-format pair formed by CF_ALU_DWORD[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 8.2, on page 8-15.

Access Read-write

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[0.1], ADDR. 2 CF_KCACHE_LOCK_2: lock cache lines KCACHE_BANK[0.1], ADDR+1. 3 CF_KCACHE_LOCK_LOOP_INDEX: lock cache lines KCACHE_BANK[0.1], 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 16 128-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 instruction slots (64-bit slots) in the range [1,128] to execute in the clause, minus one.	
USES_WATERFALL (UW)	25	int(1)
	0 This ALU clause does not use waterfall constants. 1 This ALU clause uses waterfall constants (GPR-based indexing).	
CF_INST	[29:26]	enum(4)
	Instruction. 8 CF_INST_ALU: each PRED_SET* instruction updates the active state but does not update the stack. 9 CF_INST_ALU_PUSH_BEFORE: each PRED_SET* causes a stack push first; then updates the active state. 10 CF_INST_ALU_POP_AFTER: pop the stack after the clause completes execution. 11 CF_INST_ALU_POP2_AFTER: pop the stack twice after the clause completes execution. 12 Reserved 13 CF_INST_ALU_CONTINUE: each PRED_SET* causes a continue operation on the unmasked pixels. 14 CF_INST_ALU_BREAK: each PRED_SET* causes a break operation on the unmasked pixels. 15 CF_INST_ALU_ELSE_AFTER: behaves like PUSH_BEFORE, but also performs an ELSE operation after the clause completes execution, which inverts the pixel state.	

Control Flow ALU Doubleword 1 (Cont.)

	WHOLE_QUAD_MODE (WQM)	30	int(1)
		Active pixels. 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.	
	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_DWORD0		

Control Flow Allocate, Import, or Export Doubleword 0

Instructions **CF_ALLOC_EXPORT_DWORD0**

Description This is the low-order (least-significant) doubleword in the 64-bit microcode-format pair formed by CF_ALLOC_EXPORT_DWORD0 and CF_ALLOC_EXPORT_DWORD1_{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.

Access Read-write

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 (frame buffer, no fog: [0, 7]; frame buffer, with fog: [16, 23]; 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 CF_INST_EXPORT* instruction, and the second value (WRITE, WRITE_IND, READ, and READ_IND) is used with CF_INST_MEM* instruction: 0 EXPORT_PIXEL: write pixel. Avaliable 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). IMPORT_READ: read from memory buffer (scratch and reduction buffers only). 3 Unused. IMPORT_READ_IND: read from memory buffer, use offset in INDEX_GPR (scratch and reduction buffers only).	
RW_GPR	[21:15]	int(7)
	GPR register to read data from or write data to.	
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 in the computation for determining the address of the first import or 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 [1,4]. The value from INDEX_GPR and the loop index (aL) 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 and reduction, one doubleword for other types.	
Related	CF_ALLOC_EXPORT_DWORD1_BUF CF_ALLOC_EXPORT_DWORD1_SWIZ		

Control Flow Allocate, Import, or Export Doubleword 1

Instructions **CF_ALLOC_EXPORT_DWORD1**

Description Word 1 of the control flow instruction for allocation/export is the bitwise OR of Word1 | Word1_BUF,SWIZ. This part contains firlds that are always defined.

Access Read-write, except for the CF_INST field, in which some values are write-only.

Opcode

Field Name	*Bits*	*Format*
	[16:0] Reserved.	
BURST_COUNT	[20:17] Number of MRTs, positions, parameters, or logical export values to allocate and/or export, minus one. This field is interpreted as a value in [16:1].	int(4)
END_OF_PROGRAM	21 0 This is not the last instruction in the CF program. 1 This instruction is the last one of the CF program. Execution ends after this instruction is issued.	int(1)
VALID_PIXEL_MODE	22 Antonym of WHOLE_QUAD_MODE. 0 Execute this instruction/clause as if invalid pixels are active. 1 Execute this instruction/clause as if invalid pixels are inactive. Set the default of this field to 0.	int(1)
CF_INST	[29:23] 32 CF_INST_MEM_STREAM0: perform a memory operation on stream buffer 0 (write-only). 33 CF_INST_MEM_STREAM1: perform a memory operation on stream buffer 1 (write-only). 34 CF_INST_MEM_STREAM2: perform a memory operation on stream buffer 2 (write-only). 35 CF_INST_MEM_STREAM3: perform a memory operation on stream buffer 3 (write-only). 36 CF_INST_MEM_SCRATCH: perform a memory operation on the scratch buffer (read-write). 37 CF_INST_MEM_REDUCTION: perform a memory operation on the reduction buffer (read-write). 38 CF_INST_MEM_RING: perform a memory operation on the ring buffer (write-only). 39 CF_INST_EXPORT:: export only (not last). Used for PIXEL, POS, PARAM exports. 40 CF_INST_EXPORT_DONE: export only (last export). Used for PIXEL, POS, PARAM exports. 41 CF_INST_MEM_EXPORT: perform a memory operation on the shard buffer (read-write).	int(7)
WHOLE_QUAD_MODE (WQM)	30 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 at most one of these bits.	int(1)

Control Flow Allocate, Import, or Export Doubleword 1 (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_DWORD0`
	`CF_ALLOC_EXPORT_DWORD1_SWIZ`

Control Flow Allocate, Import, or Export Doubleword 1 Buffer

Instructions **CF_ALLOC_EXPORT_DWORD1_BUF**

Description Word 1 of the control flow instruction . This subencoding is used by allocations/exports for all input/outputs to scratch, ring, stream, and reduction buffers.

Access Read-write.

Opcode

Field Name	*Bits*	*Format*
	[11:0]	
	Array size (elem-size units). Represents values [1:4096] when ELEMSIZE=0, [4:16384] when ELEMSIZE=3.	
COMP_MASK	[15:12]	int(4)
	XYZW component mask (X is the LSB). Write the component iff the corresponding bit is 1. Applies only to writes, not reads in the RV600, RV610, and RV630. In the RV670 and beyond, component mask is used for SMX reads and writes.	
	16	
	Unused. Must be set to 0.	
	[31:17]	
	Described in CF_ALLOC_EXPORT_DWORD1.	

Related CF_ALLOC_EXPORT_DWORD1
CF_ALLOC_EXPORT_DWORD1_SWIZ

Control Flow Allocate, Import, or Export Doubleword 1 Swizzle

Instructions **CF_ALLOC_EXPORT_DWORD1_SWIZ**

Description Word 1 of the control flow instruction. This subencoding is used by allocations/exports for PIXEL, POS, and PARAM.

Access Read-write

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.

[16:12]

Unused. Must be set to 0.

[31:17]

Described in CF_ALLOC_EXPORT_DWORD1.

Related CF_ALLOC_EXPORT_DWORD0
CF_ALLOC_EXPORT_DWORD1_BUF

8.2 ALU Instructions

ALU clauses are initiated using the CF_ALU_DWORD[0,1] format pair, described in Section 8.1, on page 8-2. 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_DWORD0**

Description This is the low-order (least-significant) doubleword in the 64-bit microcode-format pair formed by ALU_DWORD0 and ALU_DWORD1_{OP2, OP3}. Each instruction using this format pair has either an OP2 or an OP3 version (not both).

Access Read-write

Opcode

Field Name	*Bits*	*Format*
SRC0_SEL SRC1_SEL	[8:0] [21:13]	enum(9) 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. [511:256] cfile constants c[255:0]. Other special values are shown in the list below. 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] result. 255 ALU_SRC_PS: the previous ALU.Trans (scalar) result.	
SRC0_REL (S0R) SRC1_REL (S1R)	9 22	enum(1) enum(1)
	Addressing mode for this source operand. 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 channel 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.	

ALU Doubleword 0 (Cont.)

	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: add AR.X. 1 INDEX_AR_Y - For constants: add AR.Y. 2 INDEX_AR_Z - For constants: add AR.Z. 3 INDEX_AR_W - For constants: add AR.W. 4 INDEX_LOOP - add loop index (aL).
	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_DWORD1_OP2 ALU_DWORD1_OP3	

ALU Doubleword 1 Zero to Two Source Operands

Instructions **ALU_DWORD1_OP2**

Description This is the high-order (most-significant) doubleword in the 64-bit microcode-format pair formed by ALU_DWORD0 and ALU_DWORD1_{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.

Bits [31:18] of this format are identical to those in the ALU_DWORD1_OP3 format.

Access Read-write

Opcode

Field Name	*Bits*	*Format*
SRC0_ABS (S0A) SRC1_ABS (S1A)	0 1	int(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_EXECUTE_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.	
FOG_MERGE (FM)	5	int(1)
	Export fog value. 0 Do not export fog value. 1 Export fog value by merging the transcendental ALU result into the low-order bits of the vector destination. The vector results lose some precision.	
OMOD	[7:6]	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 [17:8] enum(10)

Instruction. The top three bits of this field must be zero. Gaps in opcode values are not marked in the list below. See Chapter 7 for descriptions of each instruction.

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	OP2_INST_FREXP_64
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_MOVA
22	OP2_INST_MOVA_FLOOR
23	OP2_INST_ADD_64
24	OP2_INST_MOVA_INT
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
43	OP2_INST_PRED_SETNE_PUSH

ALU Doubleword 1 Zero to Two Source Operands (Cont.)

ALU_INST	[17:8]	enum(10)
	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_PRED_SETE_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_DOT4
	81	OP2_INST_DOT4_IEEE
	82	OP2_INST_CUBE
	83	OP2_INST_MAX4
	95:84	*reserved*
	96	OP2_INST_MOVA_GPR_INT
	97	OP2_INST_EXP_IEEE
	98	OP2_INST_LOG_CLAMPED
	99	OP2_INST_LOG_IEEE
	100	OP2_INST_RECIP_CLAMPED
	101	OP2_INST_RECIP_FF
	102	OP2_INST_RECIP_IEEE
	103	OP2_INST_RECIPSQRT_CLAMPED
	104	OP2_INST_RECIPSQRT_FF

ALU Doubleword 1 Zero to Two Source Operands (Cont.)

ALU_INST	[17:8]	enum(10)
	105	OP2_INST_RECIPSQRT_IEEE
	106	OP2_INST_SQRT_IEEE
	107	OP2_INST_FLT_TO_INT
	108	OP2_INST_INT_TO_FLT
	109	OP2_INST_UINT_TO_FLT
	110	OP2_INST_SIN
	111	OP2_INST_COS
	112	OP2_INST_ASHR_INT
	113	OP2_INST_LSHR_INT
	114	OP2_INST_LSHL_INT
	115	OP2_INST_MULLO_INT
	116	OP2_INST_MULHI_INT
	117	OP2_INST_MULLO_UINT
	118	OP2_INST_MULHI_UINT
	119	OP2_INST_RECIP_INT
	120	OP2_INST_RECIP_UINT
	121	OP2_INST_FLT_TO_UINT
	122	OP2_INST_LDEXP_64
	123	OP2_INST_FRACT_64
	124	OP2_INST_PRED_SETGT_64
	125	OP2_INST_PRED_SETE_64
	126	OP2_INST_PRED_SETGE_64

BANK_SWIZZLE (BS)	[20:18]	enum(3)	
	Specifies how to load source operands.		
		Vector Instruction Slot	Scalar Instruction Slot
	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.	
	See Section 4.7.7, on page 4-12 for details.		

DST_GPR	[27:21]	int(7)
	Destination GPR address to which result is written.	
DST_REL (DR)	28	enum(1)
	Addressing mode for the destination GPR address.	
	0	ABSOLUTE: no relative addressing.
	1	RELATIVE: add index from INDEX_MODE to this address.
DST_ELEM (DE)	[30:29]	enum(2)
	Vector element of DST_GPR to which the result is written.	
	0	ELEM_X: write to X element.
	1	ELEM_Y: write to Y element.
	2	ELEM_Z: write to Z element.
	3	ELEM_W: write to W element.

ALU Doubleword 1 Zero to Two Source Operands (Cont.)

	`CLAMP (C)`	31	int(1)
		Clamp result. 0 Do not clamp the result. 1 Clamp the result to [0.0, 1.0]. Not mathematically defined for instructions that produce integer results.	
Related	`ALU_DWORD0` `ALU_DWORD1_OP3`		

ALU Doubleword 1 Three Source Operands

Instructions **ALU_DWORD1_OP3**

Description This is the high-order (most-significant) doubleword in the 64-bit microcode-format pair formed by ALU_DWORD0 and ALU_DWORD1_{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.

Bits [31:18] of this format are identical to those in the ALU_DWORD1_OP2 format.

Access Read-write

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. [511:256] cfile constants c[255:0]. Other special values are shown below. 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	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_DWORD0, on page 8-16, for the specification of INDEX_MODE.	
SRC2_CHAN (S2C)	[11:10]	enum(2)
	Source channel 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	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.)

Field	Bits	Type / Description
ALU_INST	[17:13]	enum(5)
		Instruction. Gaps in opcode values are not marked in the list below. See Chapter 7 for descriptions of each instruction. Note: opcode values do not begin at zero.
		8 OP3_INST_MULADD_64
		9 OP3_INST_MULADD_64_M2
		10 OP3_INST_MULADD_64_M4
		11 OP3_INST_MULADD_64_D2
		12 OP3_INST_MUL_LIT
		13 OP3_INST_MUL_LIT_M2
		14 OP3_INST_MUL_LIT_M4
		15 OP3_INST_MUL_LIT_D2
		16 OP3_INST_MULADD
		17 OP3_INST_MULADD_M2
		18 OP3_INST_MULADD_M4
		19 OP3_INST_MULADD_D2
		20 OP3_INST_MULADD_IEEE
		21 OP3_INST_MULADD_IEEE_M2
		22 OP3_INST_MULADD_IEEE_M4
		23 OP3_INST_MULADD_IEEE_D2
		24 OP3_INST_CNDE
		25 OP3_INST_CNDGT
		26 OP3_INST_CNDGE
		27 *Reserved*
		28 OP3_INST_CNDE_INT
		29 OP3_INST_CMNDGT_INT
		30 OP3_INST_CNDGE_INT
		31 *Reserved*
BANK_SWIZZLE (BS)	[20:18]	enum(3)
		Specifies how to load source operands.
		Vector Instruction Slot / Scalar Instruction Slot
		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.
		See Section 4.7.7, on page 4-12.
DST_GPR	[27:21]	int(7)
		Destination GPR address to which result is written.
DST_REL (DR)	28	enum(1)
		Addressing mode for the destination GPR address.
		0 ABSOLUTE: no relative addressing.
		1 RELATIVE: add index from INDEX_MODE to this address. See ALU_DWORD0, on page 8-16, for the specification of INDEX_MODE.

ALU Doubleword 1 Three Source Operands (Cont.)

	DST_ELEM (DE)	[30:29]	enum(2)
		Vector element of DST_GPR to which the result is written. 0 ELEM_X: write to X element. 1 ELEM_Y: write to Y element. 2 ELEM_Z: write to Z element. 3 ELEM_W: write to W element.	
	CLAMP (C)	31	int(1)
		Clamp result. 0 Do not clamp the result. 1 Clamp the result to [0.0, 1.0]. Not mathematically defined for instructions that produce integer results.	
Related	ALU_DWORD0 ALU_DWORD1_OP2		

8.3 Vertex-Fetch Instructions

Vertex-fetch clauses are specified in the CF_DWORD0 and CF_DWORD1 formats, described in Section 8.1, on page 8-2. 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 texture-fetch instructions to load all data.

All vertex-fetch microcode formats are 64 bits wide.

Vertex Fetch Doubleword 0

Instructions	**VTX_DWORD0**		
Description	This is the low-order (least-significant) doubleword in the 128-bit 4-tuple formed by VTX_DWORD0, VTX_DWORD1_{SEM, GPR}, VTX_DWORD2, 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_DWORD0 doubleword.		
Access	Read-write		
Opcode	*Field Name*	*Bits*	*Format*
	VTX_INST	[4:0]	enum(5)
		Instruction. 0 VTX_INST_FETCH: vertex fetch (X = uint32 index). Use VTX_DWORD1_GPR (page 8-29). 1 VTX_INST_SEMANTIC: semantic vertex fetch. Use VTX_DWORD1_SEM (page 8-27).	
	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	

Vertex Fetch Doubleword 0

FETCH_WHOLE_QUAD (FWQ)	7	int(1)
	0	Texture instruction can ignore invalid pixels.
	1	Texture instruction must fetch data for all pixels (result can be used as source coordinate of a dependent read).
BUFFER_ID	[15:8]	int(8)
	Constant ID to use for this vertex fetch (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.
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_DWORD1_GPR
	VTX_DWORD1_SEM
	VTX_DWORD2

Vertex Fetch Doubleword 1

Instructions **VTX_DWORD1**

Description This doubleword is part of the 128-bit 4-tuple formed by VTX_DWORD0, VTX_DWORD1_{SEM, GPR}, VTX_DWORD2, 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_DWORD0 doubleword. This SEM format is used by SEMANTIC instructions that specify a destination using a semantic table.

Access Read-write

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)	[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.	
DATA_FORMAT	[27:22]	int(6)
	Specifies vertex data format (ignored if USE_CONST_FIELDS is set).	
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	

Vertex Fetch Doubleword 1 (Cont.)

	SRF_MODE_ALL (SMA)	31 enum(1) Mapping to use when converting from signed RF to float (ignored if USE_CONST_FIELDS is set). 0 SRF_MODE_ZERO_CLAMP_MINUS_ONE: representation with two -1 representations (one is slightly past -1 but clamped). 1 SRF_MODE_NO_ZERO: OpenGL format lacking representation for zero.
Related	VTX_DWORD0 VTX_DWORD1_GPR VTX_DWORD2	

Vertex Fetch Doubleword 1 GPR Specification

Instructions **VTX_DWORD1_GPR**

Description This doubleword is part of the 128-bit 4-tuple formed by VTX_DWORD0, VTX_DWORD1_{SEM, GPR}, VTX_DWORD2, plus a doubleword filled with zeros (DWROD3), 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_DWORD0 doubleword. This GPR format is used by FETCH instructions that specify a destination GPR directly. See the next format for the semantic-table option.

Access Read-write

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.	
DATA_FORMAT	[27:22]	int(6)
	Specifies vertex data format (ignored if USE_CONST_FIELDS is set).	
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).	

Vertex Fetch Doubleword 1 GPR Specification (Cont.)

	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
	SRF_MODE_ALL (SMA)	31 enum(1) Mapping to use when converting from signed RF to float (ignored if USE_CONST_FIELDS is set). 0 SRF_MODE_ZERO_CLAMP_MINUS_ONE: representation with two -1 representations (one is slightly past -1 but clamped). 1 SRF_MODE_NO_ZERO: OpenGL format lacking representation for zero.
Related	VTX_DWORD0 VTX_DWORD1_SEM VTX_DWORD2	

Vertex Fetch Doubleword 1 Semantic-Table Specification

Instructions **VTX_DWORD1_SEM**

Description This doubleword is part of the 128-bit 4-tuple formed by VTX_DWORD0, VTX_DWORD1_{SEM, GPR}, VTX_DWORD2, 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_DWORD0 doubleword. This SEM format is used by SEMANTIC instructions that specify a destination using a semantic table.

Access Read-write

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)	[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.	
DATA_FORMAT	[27:22]	int(6)
	Specifies vertex data format (ignored if USE_CONST_FIELDS is set).	
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	

Vertex Fetch Doubleword 1 Semantic-Table Specification (Cont.)

	SRF_MODE_ALL (SMA)	31	enum(1)
		Mapping to use when converting from signed RF to float (ignored if USE_CONST_FIELDS is set).	
		0	SRF_MODE_ZERO_CLAMP_MINUS_ONE: representation with two -1 representations (one is slightly past -1 but clamped).
		1	SRF_MODE_NO_ZERO: OpenGL format lacking representation for zero.
Related	VTX_DWORD0		
	VTX_DWORD1		
	VTX_DWORD1_GPR		
	VTX_DWORD2		

Vertex Fetch Doubleword 2

Instructions	**VTX_DWORD2**
Description	This is the high-order (most-significant) doubleword in the 128-bit 4-tuple formed by VTX_DWORD0, VTX_DWORD1_{SEM, GPR}, VTX_DWORD2, plus a doubleword filled with zeros, as described in Chapter 5.
Access	Read-write

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.	
Reserved	[31:20]	
	Reserved	

Related
VTX_DWORD0
VTX_DWORD1_GPR
VTX_DWORD1_SEM

8.4 Texture-Fetch Instructions

Texture-fetch clauses are initiated using the CF_DWORD[0,1] formats, described in Section 8.1, on page 8-2. After the clause is initiated, the instructions below can be issued. Graphics programs typically use texture fetches to load texture data from memory into GPRs. General-computing programs typically use texture fetches as conventional data loads from memory into GPRs that are unrelated to textures.

All texture-fetch microcode formats are 96 bits wide, formed by three doublewords, and padded with zeros to 128 bits.

Texture Fetch Doubleword 0

Instructions **TEX_DWORD0**

Description This is the low-order (least-significant) doubleword in the 128-bit 4-tuple formed by TEX_DWORD[0,1,2] plus a doubleword filled with zeros, as described in Chapter 6.

Access Read-write

Opcode

Field Name	*Bits*	*Format*
TEX_INST	[4:0]	enum(5)

Instruction.

0	TEX_INST_VTX_FETCH: vertex fetch (X = uint32index).
1	TEX_INST_VTX_SEMANTIC: semantic vertex fetch.
2	Reserved.
3	TEX_INST_LD: fetch texel, 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 = computed LOD for all pixels in quad.
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_GET_LERP: retrieve weights used for bilinear fetch, X = horizontal lerp, Y = vertical lerp Z = volume slice, W = mipmap lerp.
10	TEX_INST_RESERVED_10: Reserved.
11	TEX_INST_SET_GRADIENTS_H: XYZ set horizontal gradients.
12	TEX_INST_SET_GRADIENTS_V: XYZ set vertical gradients.
13	TEX_INST_PASS: returns the address read in memory.
14	Z set index for array of cubemaps.
15	Fetch4/Load4 Instruction for DX 10.1.
	NOTE for the following (16 to 31): If the LOD is computed by the hardware, then these instructions are available only to the Pixel Shader (PS).
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_SAMPLE_G_L
22	TEX_INST_SAMPLE_G_LB
23	TEX_INST_SAMPLE_G_LZ
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_SAMPLE_C_G_L
30	TEX_INST_SAMPLE_C_G_LB
31	TEX_INST_SAMPLE_C_G_LZ

Texture Fetch Doubleword 0

BC_FRAC_MODE (BFM)	5	int(1)
	0	Do not force black texture data and white border to retrieve fraction of pixel that hits the border.
	1	Force black texture data and white border to retrieve fraction of pixel that hits the border.
Reserved	6	
	Reserved	
FETCH_WHOLE_QUAD (FWQ)	7	int(1)
	0	Texture instruction can ignore invalid pixels.
	1	Texture instruction must fetch data for all pixels (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.
Reserved	[31:24]	
	Reserved	

Related TEX_DWORD1
TEX_DWORD2

Texture Fetch Doubleword 1

Instructions **TEX_DWORD1**

Description This is the middle doubleword in the 128-bit 4-tuple formed by TEX_DWORD[0,1,2] plus a doubleword filled with zeros, as described in Chapter 6.

Access Read-write

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) 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 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) COORD_TYPE_Y (CTY) COORD_TYPE_Z (CTZ) COORD_TYPE_W (CTW)	28 29 30 31	enum(1) enum(1) enum(1) 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_DWORD0
TEX_DWORD2

Texture Fetch Doubleword 2

Instructions **TEX_DWORD2**

Description This is the high-order (most-significant) doubleword in the 128-bit 4-tuple formed by TEX_DWORD[0,1,2] plus a doubleword filled with zeros, as described in Chapter 6.

Access Read-write

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_DWORD0
TEX_DWORD1

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 may 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 stream processor.
aL (also AL)	Loop register. A 3-element 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 elements being operated on in the associated ALU.[X,Y,Z,W] unit.
AMD Stream™ SDK	A complete software development suite from AMD for developing applications for AMD Stream Processors. Currently, AMD Stream SDK includes Brook+ and CAL.
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, branch granularity is equal to wavefront granularity.
brcc	Source-to-source meta-compiler that translates Brook programs (.br files) into device-dependent kernels embedded in valid C++ source code that includes CPU code and stream processor device code, which later are linked into the executable.
Brook+	A high-level language derived from C which allows developers to write their applications at an abstract level without having to worry about the exact details of the hardware. This enables the developer to focus on the algorithm and not the individual instructions run on the stream processor. Brook+ is an enhancement of Brook, which is an open source project out of Stanford. Brook+ adds additional features available on AMD Stream Processors and provides a CAL backend.
brt	The Brook runtime library that executes pre-compiled kernel routines invoked from the CPU code in the application.
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 stream processor 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	An element in a vector.

Term	**Description**
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 stream processor 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”.
command	A value written by the host processor directly to the stream processor. 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 R600 that receives host commands (see Figure 1.4), interprets them, and performs the operations they indicate.
component	An element in a vector.
compute shader	Similar to a pixel shader, but exposes data sharing and synchronization.
constant buffer	Off-chip memory that contains constants. A constant buffer can hold up to 1024 4-element 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 elements 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 CAL device.
core clock	See engine clock. The clock at which the stream processor 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 stream processor.
CRs	Constant registers. There are 512 CRs, each one 128 bits wide, organized as four 32-bit values.
CS	Compute shader. A new shader type for R7xx, 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 stream processor.
DMA	Direct-memory access. Also called DMA engine. Responsible for independently transferring data to, and from, the stream processor'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	(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.
engine clock	The clock driving the stream core and memory fetch units on the stream processor stream processor core.
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.
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.
frame	A single two-dimensional screenful of data, or the storage space required for it.
frame buffer	Off-chip memory that stores a frame.
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.
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.

Term	**Description**
gather stream	Input streams are treated as a memory array, and data elements are addressed directly.
global buffer	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.
GPGPU	General-purpose stream processor. A stream processor 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 elements (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, Stream Processor, and Data Parallel Processor.
GPU engine clock frequency	Also called 3D engine speed.
GS	Geometry Shader.
GSA	GPU ShaderAnalyzer. A performance profiling tool for developing, debugging, and profiling stream kernels using high-level stream computing languages.
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 stream processors. AMD IL is designed for efficient generalization of stream processor 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.
kernel	A small, 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 stream processor 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.

Term	**Description**
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 processor 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 stream processor.
microcode format	An encoding format whose fields specify instructions and associated parameters. Micro-code 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`. The microcode formats and their fields are described in Section 8.1, on page 8-2.
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 stream processor 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.
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”.
opcode	The numeric value of the *code* field of an “instruction”. For example, the opcode for the CMOV instruction is decimal 16 (0x10).
opcode token	A 32-bit value that describes the operation of an instruction.
operation	The function performed by an “instruction”.
PaC	Parameter Cache.
page	A program-controlled cache, backing up processor-accessible memory.

Term	**Description**
PCI Express	A high-speed computer expansion card interface used by modern graphics cards, stream processors 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 Stream Processor and AMD Data Parallel Processor.
program	Unless otherwise specified, a program is a set of instructions that can run on the AMD Stream Processor/AMD Data Parallel Processor. A device program is a type of kernel.
PS	Pixel Shader.
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-element vector result from a ALU.[X,Y,Z,W] unit within a given clause.
quad	Group of 2x2 threads in the domain. 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.
RB	Ring Buffer.
register	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).
render backend unit	The hardware units in a stream processor stream processor core 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 element, unlike a vector which contains a set of two or more data elements.
scatter	Writes (by uncached memory) to arbitrary locations.

Term	**Description**
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	Also called thread processor.
shader program	User developed program. Also called kernel.
SIMD	Single instruction multiple data. – Each SIMD receives independent stream core instructions. – Each SIMD applies the instructions to multiple data elements.
SIMD Engine	A collection of thread processors, each of which executes the same instruction per cycle.
SIMD 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 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.
slot	A position, in an “istruction 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.
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 stream processor stream core handles a single instruction within the VLIW instruction.
stream operator	A node that can restructure data.
stream processor	A parallel processor capable of executing multiple threads of a kernel in order to process streams of data.
swizzling	To copy or move any element in a source vector to any element-position in a destination vector. Accessing elements in any combination.
thread	One invocation of a kernel corresponding to a single element in the domain of execution.

Term	**Description**
thread group	It contains one or more thread blocks. Threads in the same thread-group but different thread-blocks might communicate to each through global per-stream processor shared memory. This is a concept mainly for global data share (GDS) which is not discussed in this note.
thread processor	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	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 (i.e. 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.
uncached read/write unit	The hardware units in a stream processor responsible for handling uncached read or write requests from local memory on the stream processor.
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].
VLIW design	Very Long Instruction Word. – Co-issued up to 6 operations (5 stream cores + 1 FC) – 1.25 Machine Scalar operation per clock for each of 64 data elements – Independent scalar source and destination addressing
waterfall	To use the address register (AR) for indexing the GPRs. Waterfall behavior is determined by a “configuation 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.
write combining	Combining several smaller writes to memory into a single larger write to minimize any overhead associated with write commands.