Reference Guide

AMD Accelerated Parallel Processing TECHNOLOGY

Sea Islands Series Instruction Set Architecture

December 2013

Revision 1.3

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

For AMD Accelerated Parallel Processing:

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

Contents

Chapter 12 Instruction Set

Chapter 13 Microcode Formats

Figures

Tables

Preface

About This Document

This document describes the environment, organization, and program state of the AMD Sea Islands series of devices. It details the instruction set and the microcode formats native to this family of processors that are accessible to programmers and compilers.

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

The main purposes of this document are to:

1. Specify the language constructs and behavior, including the organization, of each type of instruction in both text syntax and binary format.
2. Provide a reference of instruction operation that compiler writers can use to maximize performance of the processor.

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 AMD Sea Islands series of processors' hardware and programming environment (Chapter 1). Chapter 2 describes the organization of Sea Islands series programs. Chapter 3 describes the program state that is maintained. Chapter 4 describes the program flow. Chapter 5 describes the scalar ALU operations. Chapter 6 describes the vector ALU operations. Chapter 7 describes the scalar memory operations. Chapter 10 describes the data share operations. Chapter 11 describes exporting the parameters of pixel color and vertex shaders. Chapter 8 describes the vector memory operations. Chapter 12 describes instruction details, first by the microcode format to which they belong, then in alphabetic order. Finally, Chapter 13 provides a detailed specification of each microcode format.

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

- *Intermediate Language (IL) Reference Manual*. Published by AMD.
- *AMD Accelerated Parallel Processing OpenCL Programming Guide*. Published by AMD.
- *The OpenCL Specification*. Published by Khronos Group. Aaftab Munshi, editor.
- *OpenGL Programming Guide*, at http://www.glprogramming.com/red/
- *Microsoft DirectX Reference Website*, at http://msdn.microsoft.com/archive/default.asp?url=/archive/en-us/directx9_c_Summer_04/directx/graphics/reference/reference.asp
- GPGPU: http://www.gpgpu.org

Differences Between Southern Islands and Sea Islands Devices

Important differences between S.I. and C.I. GPUs

- Multi queue compute

  Lets multiple user-level queues of compute workloads be bound to the device and processed simultaneous. Hardware supports up to eight compute pipelines with up to eight queues bound to each pipeline.

- System unified addressing

  Allows GPU access to process coherent address space.

- Device unified addressing

  Lets a kernel view LDS and video memory as a single addressable memory. It also adds shader instructions, which provide access to “flat” memory space.

- Memory address watch

  Lets a shader determine if a region of memory has been accessed.

- Conditional debug

  Adds the ability to execute or skip a section of code based on state bits under control of debugger software. This feature adds two bits of state to each wavefront; these bits are initialized by the state register values set by the debugger, and they can be used in conditional branch instructions to skip or execute debug-only code in the kernel.

- Support for unaligned memory accesses
- Detection and reporting of violations in memory accesses

Summary of kernel instruction change from S.I. to C.I.

New instruction formats:

- FLAT

New instructions:

- FLAT_* (*entire family of operations)*
- S_CBRANCH_CDBGUSER
- S_CBRANCH_CDBGSYS
- S_CBRANCH_CDBGSYS_OR_USER
- S_CBRANCH_CDBGSYS_AND_USER
- S_DCACHE_INV_VOL
- V_TRUNC_F64
- V_CEIL_F64
- V_FLOOR_F64
- V_RNDNE_F64

- V_QSAD_PK_U16_U8
- V_MQSAD_U16_U8
- V_MQSAD_U32_U8
- V_MAD_U64_U32
- V_MAD_I64_I32
- V_EXP_LEGACY_F32
- V_LOG_LEGACY_F32
- DS_NOP
- DS_GWS_SEMA_RELEASE_ALL
- DS_WRAP_RTN_B32
- DS_CNDXCHG32_RTN_B64
- DS_WRITE_B96
- DS_WRITE_B128
- DS_CONDXCHG32_RTN_B128
- DS_READ_B96
- DS_READ_B128
- BUFFER_LOAD_DWORDX3
- BUFFER_STORE_DWORDX3

Removed instructions:

- V_QSAD_U8
- V_MQSAD_U8
- BUFFER_ATOMIC_RSUB, _X2
- IMAGE_ATOMIC_RSUB, _X2

Contact Information

To submit questions or comments concerning this document, contact our technical documentation staff at:

For questions concerning AMD Accelerated Parallel Processing developing, please email: **developer.amd.com/**.

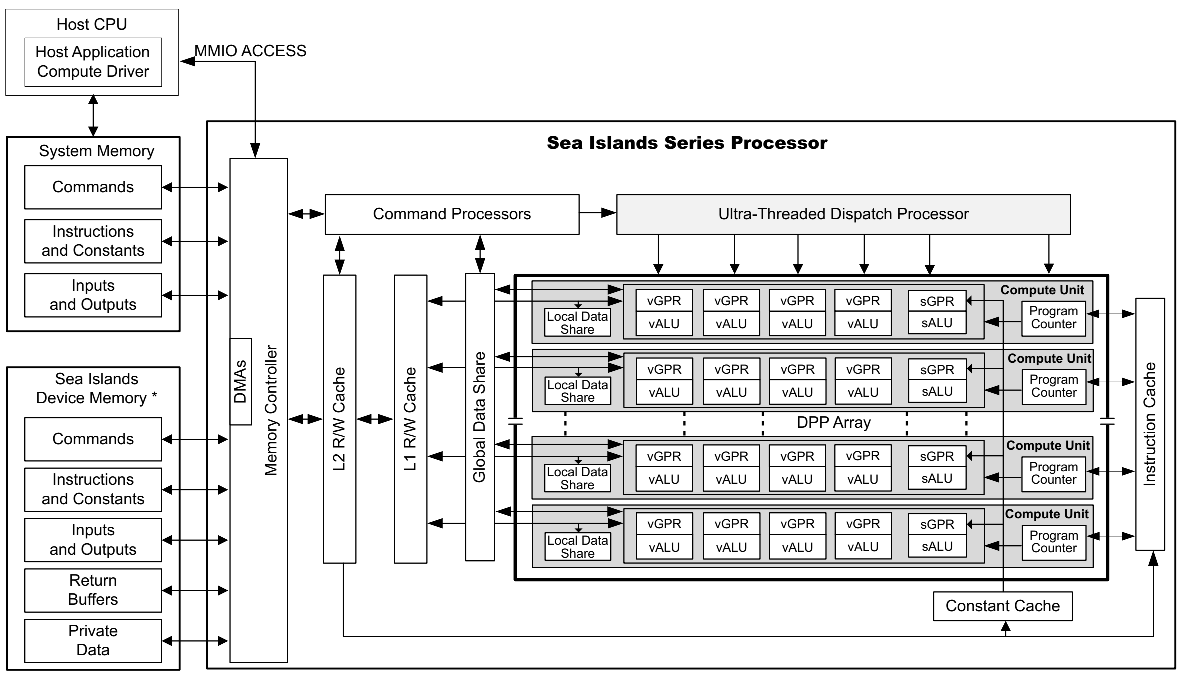
For questions about developing with AMD Accelerated Parallel Processing, please email: **developer.amd.com/appsdk.**

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

Chapter 1 Introduction

The AMD Sea Islands series of processors implements a parallel microarchitecture that provides an excellent platform not only for computer graphics applications but also for general-purpose data parallel applications. Any data-intensive application that exhibits high bandwidth needs or significant computational requirements is a candidate for running on an AMD Sea Islands series processor.

Figure 1.1 shows a block diagram of the AMD Sea Islands series processors.



*Discrete GPU – Physical Device Memory; APU – Region of system for GPU direct access

Figure 1.1 AMD Sea Islands Series Block Diagram

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

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

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

In the Sea Islands environment, a complete application includes two parts:

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

The Sea Islands programs are controlled by host commands, which

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

The Sea Islands driver program runs on the host.

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

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

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

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

Conceptually, each compute unit pipeline maintains a separate interface to memory, consisting of index pairs and a field identifying the type of request (program instruction, floating-point constant, integer constant, boolean constant, input read, or output write). The index pairs for inputs, outputs, and constants are specified by the requesting Sea Islands instructions from the hardware-maintained program state in the pipelines.

The AMD Sea Islands series of devices can detect floating point exceptions and can generate interrupts. In particular, it detects IEEE floating-point exceptions in hardware; these can be recorded for post-execution analysis. The software interrupts shown in Figure 1.1 from the command processor to the host represent hardware-generated interrupts for signalling command-completion and related management functions.

Figure 1.2 shows the dataflow for a Sea Islands application. For general-purpose applications, only one processing block performs all computation.

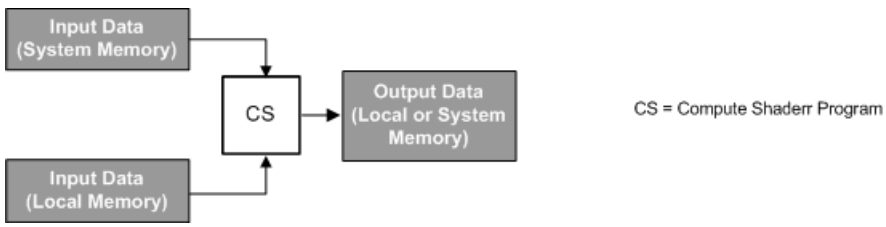


Figure 1.2 Sea Islands Dataflow

The Sea Islands processor hides memory latency by keeping track of potentially hundreds of work-items in different stages of execution, and by overlapping compute operations with memory-access operations.

Chapter 2
Program Organization

Sea Islands programs consist of scalar instructions that operate on one value per wavefront, and vector instructions that operate on one value per thread. All program flow control is handled through scalar instructions. Vector ALU instructions operate on Vector General Purpose Registers (VGPRs) and can take up to three operands and produce both a value and mask result. The mask can be a condition code or a carry out. Scalar ALU instructions operate on Scalar GPRs and can take up to two inputs and produce a single integer or bit-mask output. Both vector and scalar instructions can modify the EXECute mask; this mask controls which threads are active and execute instructions at a given point in the kernel. Programs typically use instructions for fetching data through the texture cache for data loads.

2.1 The Compute Shader Program Type

The program type commonly run on the Sea Islands GPU (see Figure 1.2, on page 1-3) is the Compute Shader (CS), which is a generic program (compute kernel) that uses an input work-item ID as an index to perform:

– gather reads on one or more sets of input data,
– arithmetic computation, and
– scatter writes to one or more set of output data to memory.

Compute shaders can write to multiple (up to eight) surfaces, which can be a mix of multiple render targets (MRTs), unordered access views (UAVs), and flat address space.

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.

2.2 Instruction Terminology

Table 2.1 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 13.

Table 2.1 Basic Instruction-Related Terms

Term	Size (bits)	Description
Microcode format	32	One of several encoding formats for all instructions. They are described in Chapter 13, "Microcode Formats."
Instruction	32 or 64	Every instruction is described with either 32 bits or 64 bits of microcode. • Vector Memory instructions are 64 bits. • Exports are 64 bits. • LDS and GDS are 64 bits. • Scalar ALU instructions are 32 bits but can have an additional 32 bits of literal constant data. • Vector ALU instructions can be 32 bits or 64 bits. The 32-bit versions can have an additional 32 bits of literal constant data.
Literal Constant	32	Literal constants specify a 32-bit constant value, either integer or float, in the instruction stream that supplies a value to a single 32-bit instruction.
Export	n/a	Copying, or the instruction to copy, from one or more VGPRs to one of the following output buffers: Pixel Color, Pixel Depth (Z), Vertex Parameter, or Vertex Position.
Fetch	n/a	Load data into VGPRs (vector fetch) or into SGPRs (scalar fetch), from the texture cache.
Quad	n/a	Four related pixels (for general-purpose programming: [x,y] data elements) in an aligned 2x2 space.
Fragment	n/a	A set of (x,y) data elements.
Pixel	n/a	A set of (x,y) data elements.
Thread	n/a	A work-item.

Table 2.2 summarizes the constant state data used by kernels for accessing memory buffer and image objects.

Table 2.2 Buffer, Texture, and Constant State

State	Access by Sea Islands S/W	Access by Host S/W	Width (bits)	Description
Texture Samplers	R	W	128	A texture sampler describes how a texture map sample instruction (IMAGE) filters texel data and handles mip-maps. Texture samplers must first be loaded into four contiguous SGPRs prior to use by an IMAGE instruction.
Texture Resources	R	W	128 or 256	A texture resource describes the location, layout, and data type of a texture map in memory. A texture resource must be loaded into four or eight contiguous SGPRs prior to use by an IMAGE instruction.

Table 2.2 Buffer, Texture, and Constant State (Cont.)

State	Access by Sea Islands S/W	Access by Host S/W	Width (bits)	Description
Constant Buffer	R	W	128	A specific usage of a buffer resource to describe an array of constant values that are provided by the application and loaded into memory prior to kernel invocation.
Border Color	No	W	128 (4 x 32 bits)	This is stored in the texture cache and referenced by texture samplers.
Bicubic Weights	No	W	176	These define the weights, one horizontal and one vertical, for bicubic interpolation. The state is stored in the texture pipeline, but referenced in fetch instructions.
Kernel Size for Cleartype Filtering	No	W	3	These define the kernel sizes, one horizontal and one vertical, for filtering with Microsoft's Cleartype™ sub-pixel rendering display technology. The state is stored in the texture pipeline, but referenced in fetch instructions.

2.3 Data Sharing

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

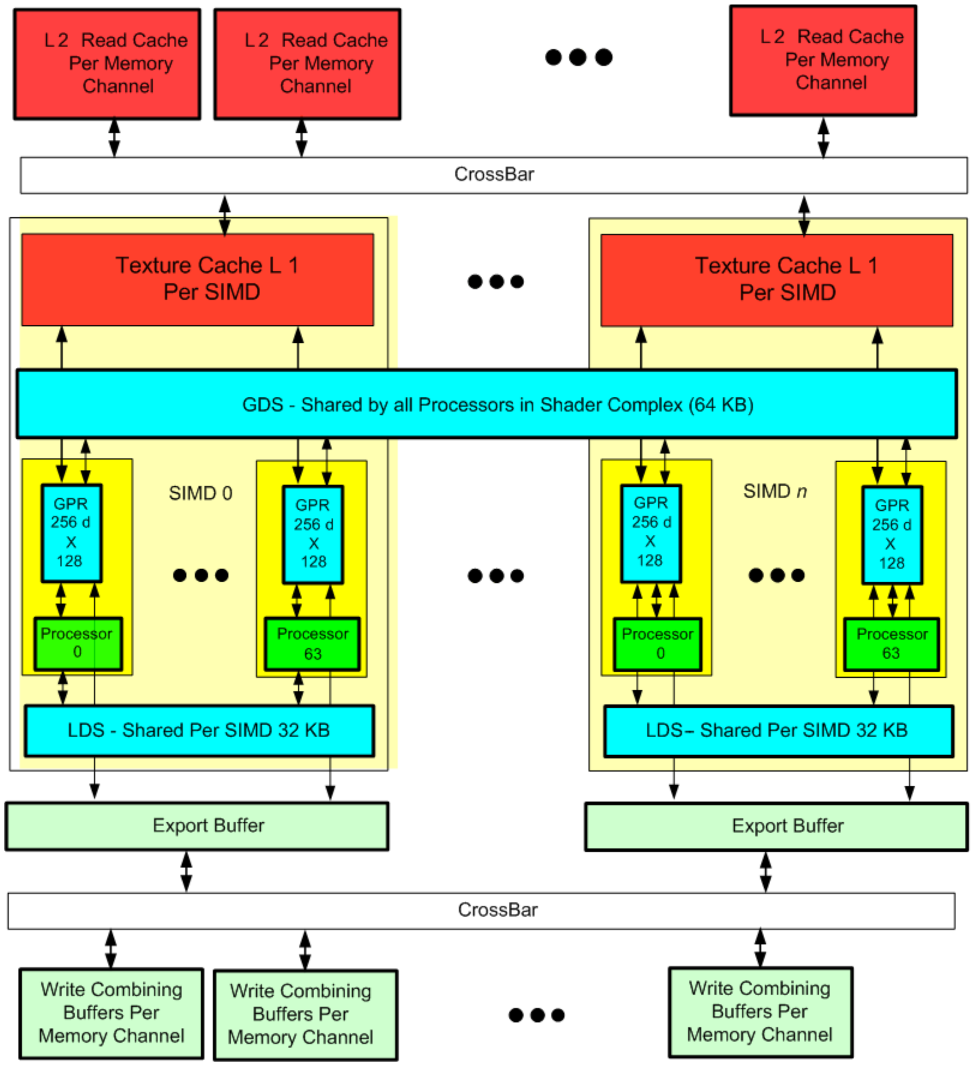


Figure 2.1 Shared Memory Hierarchy on the AMD Sea Islands Series of Stream Processors

2.3.1 Local Data Share (LDS)

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

2.3.2 Global Data Share (GDS)

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

2.4 Device Memory

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

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

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

Chapter 3
Kernel State

This chapter describes the kernel states visible to the shader program.

3.1 State Overview

Table 3.1 describes all of the hardware states readable or writable by a shader program.

Table 3.1 Readable and Writable Hardware States

Abbrev.	Name	Size	Description
PC	Program Counter	40 bits	Points to the memory address of the next shader instruction to execute.
V0-V255	VGPR	32 bits	Vector general-purpose register.
S0-S103	SGPR	32 bits	Scalar general-purpose register.
LDS	Local Data Share	32 kB	Local data share is a scratch RAM with built-in arithmetic capabilities that allow data to be shared between threads in a workgroup.
EXEC	Execute Mask	64 bits	A bit mask with one bit per thread, which is applied to vector instructions and controls that threads execute and that ignore the instruction.
EXECZ	EXEC is zero	1 bit	A single bit flag indicating that the EXEC mask is all zeros.
VCC	Vector Condition Code	64 bits	A bit mask with one bit per thread; it holds the result of a vector compare operation.
VCCZ	VCC is zero	1 bit	A single bit-flag indicating that the VCC mask is all zeros.
SCC	Scalar Condition Code	1 bit	Result from a scalar ALU comparison instruction.
STATUS	Status	32 bits	Read-only shader status bits.
MODE	Mode	32 bits	Writable shader mode bits.
M0	Memory Reg	32 bits	A temporary register that has various uses, including GPR indexing and bounds checking.
TRAPSTS	Trap Status	32 bits	Holds information about exceptions and pending traps.
TBA	Trap Base Address	64 bits	Holds the pointer to the current trap handler program.
TMA	Trap Memory Address	64 bits	Temporary register for shader operations. For example, can hold a pointer to memory used by the trap handler.
TTMP0-TTMP11	Trap Temporary SGPRs	32 bits	12 SGPRs available only to the Trap Handler for temporary storage.

Table 3.1 Readable and Writable Hardware States

Abbrev.	Name	Size	Description
VMCNT	Vector memory instruction count	4 bits	Counts the number of VMEM instructions issued but not yet completed.
EXPCNT	Export Count	3 bits	Counts the number of Export and GDS instructions issued but not yet completed. Also counts VMEM writes that have not yet sent their write-data to the TC.
LGKMCNT	LDS, GDS, Constant and Message count	5 bits	Counts the number of LDS, GDS, constant-fetch (scalar memory read), and message instructions issued but not yet completed.

3.2 Program Counter (PC)

The program counter (PC) is a byte address pointing to the next instruction to execute. When a wavefront is created, the PC is initialized to the first instruction in the program.

The PC interacts with three instructions: `S_GET_PC`, `S_SET_PC`, `S_SWAP_PC`. These transfer the PC to, and from, an even-aligned SGPR pair.

Branches jump to (PC_of_the_instruction_after_the_branch + offset). The shader program cannot directly read from, or write to, the PC. Branches, `GET_PC` and `SWAP_PC`, are PC-relative to the next instruction, not the current one. S_TRAP saves the PC of the S_TRAP instruction itself.

3.3 EXECute Mask

The Execute mask (64-bit) determines which threads in the vector are executed: 1 = execute, 0 = do not execute.

`EXEC` can be read from, and written to, through scalar instructions; it also can be written as a result of a vector-ALU compare. This mask affects vector-ALU, vector-memory, LDS, and export instructions. It does not affect scalar execution or branches.

A helper bit (`EXECZ`) can be used as a condition for branches to skip code when `EXEC` is zero.

> Performance Note: unlike previous generations, the S.I. hardware does no optimization when `EXEC` = 0. The shader hardware executes every instruction, wasting instruction issue bandwidth. Use `CBRANCH` or `VSKIP` to more rapidly skip over code when it is likely that the EXEC mask is zero.

3.4 Status Registers

Status register fields can be read, but not written to, by the shader. These bits are initialized at wavefront-creation time. Table 3.2 lists and briefly describes the status register fields.

Table 3.2 Status Register Fields

Field	Bit Position	Description
SCC	1	Scalar condition code. Used as a carry-out bit. For a comparison instruction, this bit indicates failure or success. For logical operations, this is 1 if the result was non-zero.
SPI_PRIO	2:1	Wavefront priority set by the shader processor interpolator (SPI) when the wavefront is created. See the S_SETPRIO instruction (page 12-42) for details. 0 is lowest, 3 is highest priority.
WAVE_PRIO	4:3	Wavefront priority set by the shader program. See the S_SETPRIO instruction (page 12-42) for details.
PRIV	5	Privileged mode. Can only be active when in the trap handler. Gives write access to the TTMP, TMA, and TBA registers.
TRAP_EN	6	Indicates that a trap handler is present. When set to zero, traps are never taken.
TTRACE_EN	7	Indicates whether thread trace is enabled for this wavefront. If zero, also ignore any shader-generated (instruction) thread-trace data.
EXPORT_RDY	8	This status bit indicates if export buffer space has been allocated. The shader stalls any export instruction until this bit becomes 1. It is set to 1 when export buffer space has been allocated. Before a Pixel or Vertex shader can export, the hardware checks the state of this bit. If the bit is 1, export can be issued. If the bit is zero, the wavefront sleeps until space becomes available in the export buffer. Then, this bit is set to 1, and the wavefront resumes.
EXECZ	9	Exec mask is zero.
VCCZ	10	Vector condition code is zero.
IN_TG	11	Wavefront is a member of a work-group of more than one wavefront.
IN_BARRIER	12	Wavefront is waiting at a barrier.
HALT	13	Wavefront is halted or scheduled to halt. HALT can be set by the host through wavefront-control messages, or by the shader. This bit is ignored while in the trap handler (PRIV = 1); it also is ignored if a host-initiated trap is received (request to enter the trap handler).
TRAP	14	Wavefront is flagged to enter the trap handler as soon as possible.
TTRACE_CU_EN	15	Enables/disables thread trace for this compute unit (CU). This bit allows more than one CU to be outputting USERDATA (shader initiated writes to the thread-trace buffer). Note that wavefront data is only traced from one CU per shader array. Wavefront user data (instruction based) can be output if this bit is zero.
VALID	16	Wavefront is active (has been created and not yet ended).
ECC_ERR	17	An ECC error has occurred.
SKIP_EXPORT	18	For Vertex Shaders only. 1 = this shader is never allocated export buffer space; all export instructions are ignored (treated as NOPs). Formerly called VS_NO_ALLOC. Used for stream-out of multiple streams (multiple passes over the same VS), and for DS running in the VS stage for wavefronts that produced no primitives.
PERF_EN	19	Performance counters are enabled for this wavefront.
COND_DBG_USER	20	Conditional debug indicator for user mode
COND_DBG_SYS	21	Conditional debug indicator for system mode.
DATA_ATC	22	Indicates the kernel's data is located in ATC memory space. 0 = GPUVM.

Table 3.2 Status Register Fields

Field	Bit Position	Description
INST_ATC	23	Indicates the kernel instructions are located in ATC memory space. 0 = GPUVM.
DISPATCH_CACHE_CTRL	26:24	Indicates the cache policies for this dispatch. [24] = Vector L1 cache policy. [25] = L2 cache policy. [26] = Scalar data cache policy. The policies are: 0 = normal, 1 = force miss/evict for L1 and bypass for L2.
MUST_EXPORT	27	This wavefront is required to perform an export with Done=1 before terminating.

3.5 Mode Register

Mode register fields can be read from, and written to, by the shader through scalar instructions.Table 3.3 lists and briefly describes the mode register fields.

Table 3.3 Mode Register Fields

Field	Bit Position	Description
FP_ROUND	3:0	[1:0] Single precision round mode. [3:2] Double precision round mode. Round Modes: 0=nearest even, 1= +infinity, 2= -infinity, 3= toward zero.
FP_DENORM	7:4	[1:0] Single denormal mode. [3:2] Double denormal mode. Denorm modes: 0 = flush input and output denorms. 1 = allow input denorms, flush output denorms. 2 = flush input denorms, allow output denorms. 3 = allow input and output denorms.
DX10_CLAMP	8	Used by the vector ALU to force DX10-style treatment of NaNs: when set, clamp NaN to zero; otherwise, pass NaN through.
IEEE	9	Floating point opcodes that support exception flag gathering quiet and propagate signaling NaN inputs per IEEE 754-2008. Min_dx10 and max_dx10 become IEEE 754-2008 compliant due to signaling NaN propagation and quieting.
LOD_CLAMPED	10	Sticky bit indicating that one or more texture accesses had their LOD clamped.
DEBUG	11	Forces the wavefront to jump to the exception handler after each instruction is executed (but not after ENDPGM). Only works if TRAP_EN = 1.
EXCP_EN	18:12	Enable mask for exceptions. Enabled means if the exception occurs and TRAP_EN==1, a trap is taken. [12] : invalid. [13] : inputDenormal. [14] : float_div0. [15] : overflow. [16] : underflow. [17] : inexact. [18] : int_div0.
VSKIP	28	0 = normal operation. 1 = skip (do not execute) any vector instructions: valu, vmem, export, lds, gds. "Skipping" instructions occurs at high-speed (10 wavefronts per clock cycle can skip one instruction). This is much faster than issuing and discarding instructions.
CSP	31:29	Conditional branch stack pointer. See Section 4.2 on page 4-1.

3.6 GPRs and LDS

This section describes how GPR and LDS space is allocated to a wavefront, as well as how out-of-range and misaligned accesses are handled.

3.6.1 Out-of-Range Behavior

When a source or destination is out of the legal range owned by a wavefront, the behavior is different from that resulting in the Northern Islands environment.

Out-of-range can occur through GPR-indexing or bad programming. It is illegal to index from one register type into another (for example: SGPRs into trap registers or inline constants). It is also illegal to index within inline constants.

The following describe the out-of-range behavior for various storage types.

- SGPRs
  - Source or destination out-of-range = (sgpr < 0 || (sgpr >= sgpr_size)).
  - Source out-of-range: returns the value of SGPR0 (not the value 0).
  - Destination out-of-range: instruction writes no SGPR result.
- VGPRs
  - Similar to SGPRs. It is illegal to index from SGPRs into VGPRs, or vice versa.
  - Out-of-range = (vgpr < 0 || (vgpr >= vgpr_size))
  - If a source VGPR is out of range, VGPR0 is used.
  - If a destination VGPR is out-of-range, the instruction is ignored (treated as an NOP).
- LDS
  - If the LDS-ADDRESS is out-of-range (addr < 0 or > (MIN(lds_size, m0)):
    - ◊ Writes out-of-range are discarded; it is undefined if SIZE is not a multiple of write-data-size.
    - ◊ Reads return the value zero.
  - If any source-VGPR is out-of-range, use the VGPR0 value is used.
  - If the dest-VGPR is out of range, nullify the instruction (issue with exec=0)
- Memory, LDS, and GDS: Reads and atomics with returns.
  - If any source VGPR or SGPR is out-of-range, the data value is undefined.
  - If any destination VGPR is out-of-range, the operation is nullified by issuing the instruction as if the EXEC mask were cleared to 0.
    - ◊ This out-of-range check must check all VGPRs that can be returned (for example: VDST to VDST+3 for a `BUFFER_LOAD_DWORD`x4).

◊ This check must also include the extra PRT (partially resident texture) VGPR and nullify the fetch if this VGPR is out-of-range, no matter whether the texture system actually returns this value or not.

◊ Atomic operations with out-of-range destination VGPRs are nullified: issued, but with exec mask of zero.

Instructions with multiple destinations (for example: `V_ADDC`): if any destination is out-of-range, no results are written.

3.6.2 SGPR Allocation and Storage

A wavefront can be allocated 8 to 104 SGPRs, in units of 8 GPRs (dwords). These are logically viewed as SGPRs 0–103. The VCC is physically stored as part of the wavefront's SGPRs in the highest numbered two SGPRs (the source/destination VCC is an alias for those two SGPRs). When a trap handler is present, 16 additional SGPRs are reserved after VCC to hold the trap addresses, as well as saved-PC and trap-handler temps. These all are privileged (cannot be written to unless privilege is set). Note that if a wavefront allocates 16 SGPRs, 2 SGPRs are normally used as VCC, the remaining 14 are available to the shader. Shader hardware does not prevent use of all 16 SGPRs.

3.6.3 SGPR Alignment

Even-aligned SGPRs are required in the following cases.

- When 64-bit data is used. This is required for moves to/from 64-bit registers, including the PC.
- When scalar memory reads that the address-base comes from an SGPR-pair (either in SGPR).

Quad-alignment is required for the data-GPR when a scalar memory read returns four or more dwords.

When a 64-bit quantity is stored in SGPRs, the LSBs are in SGPR[n], and the MSBs are in SGPR[n+1].

3.6.4 VGPR Allocation and Alignment

VGPRs are allocated in groups of four Dwords. Operations using pairs of VGPRs (for example: double-floats) have no alignment restrictions. Physically, allocations of VGPRs can wrap around the VGPR memory pool.

3.6.5 LDS Allocation and Clamping

LDS is allocated per work-group or per-wavefront when work-groups are not in use. LDS space is allocated to a work-group or wavefront in contiguous blocks of 64 Dwords on 64-Dword alignment.

LDS allocations do not wrap around the LDS storage.

All accesses to LDS are restricted to the space allocated to that wavefront/work-group.

Clamping of LDS reads and writes is controlled by two size registers, which contain values for the size of the LDS space allocated by SPI to this wavefront or work-group, and a possibly smaller value specified in the LDS instruction (size is held in M0). The LDS operations use the smaller of these two sizes to determine how to clamp the read/write addresses.

3.7 M# Memory Descriptor

There is one 32-bit M# (M0) register per wavefront, which can be used for:

- Local Data Share (LDS)
  - Interpolation: holds { 1’b0, new_prim_mask[15:1], parameter_offset[15:0] } // in bytes
  - LDS direct-read offset and data type: { 13’b0, DataType[2:0], LDS_address[15:0] } // addr in bytes
  - LDS addressing for Memory/Vfetch → LDS: {16’h0, lds_offset[15:0]} // in bytes
  - Indexed LDS: provides SIZE in bytes { 15’h0, size[16:0] } // size in bytes
- Global Data Share (GDS)
  - { base[15:0] , size[15:0] } // base and size are in bytes
- Indirect GPR addressing for both vector and scalar instructions. M0 is an unsigned index.
- Send-message value. `EMIT`/`CUT` use M0 and EXEC as the send-message data.
- Flat: M0 provides the LDS SIZE in bytes (same as LDS-indexed case).

3.8 SCC: Scalar Condition Code

Most scalar ALU instructions set the Scalar Condition Code (SCC) bit, indicating the result of the operation.

Compare operations: 1 = true

Arithmetic operations: 1 = carry out

Bit/logical operations: 1 = result was not zero

Move: does not alter SCC

The SCC can be used as the carry-in for extended-precision integer arithmetic, as well as the selector for conditional moves and branches.

3.9 Vector Compares: VCC and VCCZ

Vector ALU comparisons always set the Vector Condition Code (VCC) register (1=pass, 0=fail). Also, vector compares have the option of setting EXEC to the VCC value.

There is also a VCC summary bit (vccz) that is set to 1 when the VCC result is zero. This is useful for early-exit branch tests. VCC is also set for selected integer ALU operations (carry-out).

Vector compares have the option of writing the result to VCC (32-bit instruction encoding) or to any SGPR (64-bit instruction encoding). VCCZ is updated every time VCC is updated: vector compares and scalar writes to VCC.

The EXEC mask determines which threads execute an instruction. The VCC indicates which executing threads passed the conditional test, or which threads generated a carry-out from an integer add or subtract.

V_CMP_* → VCC[n] = EXEC[n] & (test passed for thread[n])

VCC is always fully written; there are no partial mask updates.

NOTE: VCC physically resides in the SGPR register file, so when an instruction sources VCC, that counts against the limit on the total number of SGPRs that can be sourced for a given instruction. VCC physically resides in the highest two user SGPRs.

<u>Shader Hazard with VCC</u> The user/compiler must prevent a scalar-ALU write to the SGPR holding VCC, immediately followed by a conditional branch using VCCZ. The hardware cannot detect this, and inserts the one required wait state (hardware *does* detect it when the SALU writes to VCC, it only fails to do this when the SALU instruction references the SGPRs that happen to hold VCC).

3.10 Trap and Exception Registers

Each type of exception can be enabled or disabled independently by setting, or clearing, bits in the TRAPSTS register's `EXCP_EN` field. This section describes the contents of all hardware register fields associated with traps and exceptions.

`STATUS . TRAP_EN`

This bit tells the shader whether or not a trap handler is present. When one is not present, traps are not taken, no matter whether they're floating point, user- or host-initiated traps. When the trap handler is present, the wavefront uses an extra 16 SGPRs for trap processing.
If `trap_en` == 0, all traps and exceptions are ignored, and `s_trap` is converted by hardware to NOP.

The EXCP_EN[8:0] bit field of the TRAP_STS register contains floating-point, address watch, and memory violation exception enables. It defines which of the six float exception types and one integer exception type cause a trap.

Bit	Exception
0	Invalid.
1	Input denormal.
2	Divide by zero.
3	Overflow.
4	Underflow.
5	Inexact.
6	Integer divide by zero.
7	ADDR_WATCH. Indicates the shader has accessed a memory address that is being tracked by the address watch function.
8	MEM_VIOL. Indicates a memory violation has occurred.

The EXCP[8:0] field of the TRAP_STS register contains the status bits that indicate which exceptions have occurred. These bits are sticky and accumulate results until the shader program clears them. These bits are accumulated regardless of the setting of EXCP_EN.

The EXCP_CYCLE[5:0] field of the TRAP_STS register contains When a float exception occurs, this tells the trap handler on which cycle the exception occurred: 0-3 for normal float operations, 0-7 for double float add operations, and 0-15 for double float muladd or transcendental operations. This register records the cycle number of the first occurrence of an enabled (unmasked) exception.

Excp.cycle[5:4] – Hybrid pass – used for machines that run at lower rates.

Excp.cycle[3:2] – Multi-slot pass.

Excp.cycle[1:0] – Phase: threads 0-15 are in phase 0, 48-63 in phase 3.

The DP_RATE[2:0] field of the TRAP_STS register specifies to the shader how to interpret TRAP_STS.cycle. Different vector shader processors (VSP) process instructions at different rates.

All trap SGPRS (TBA, TMA, TTMP) are privileged for writes; they can be written only when in the trap handler (status.priv = 1). When not privileged, writes to these are ignored.

When a trap is taken (either user-, exception-, or host-initiated), the shader hardware generates an S_TRAP instruction. This loads trap information into a pair of SGPRS: {TTMP1, TTMP0} = {3'h0,pc_rewind[3:0], HT[0],trapID[7:0], PC[47:0]}.

HT is set to 1 for host-initiated traps; it is zero for user traps (s_trap) or exceptions. TRAP_ID is zero for exceptions, or the user/host trapID for those traps. When the trap handler is entered, the PC of the faulting instruction is: (PC – PC_rewind*4).

3.11 Memory Violations

A Memory Violation is reported from:

- LDS access out of range: 0 < addr < lds_size. This can occur for indexed and direct access.
- LDS alignment error.
- Memory read/write/atomic out-of-range.
- Memory read/write/atomic alignment error.
- Flat access where the address is invalid (does not fall in any aperture).
- Write to a read-only surface.
- GDS alignment or address range error.
- GWS operation aborted (semaphore or barrier not executed).

Memory violations are not reported for instruction or scalar-data accesses.

Memory Buffer to LDS does NOT return a memory violation if the LDS address is out of range, but masks off `EXEC` bits of threads that would go out of range.

When a memory access is in violation, the appropriate memory (LDS or TC) returns `MEM_VIOL` to the wave. This is stored in the wave's `TRAPSTS.mem_viol` bit. This bit is sticky, so once set to 1, it remains at 1 until the user clears it.

There is a corresponding exception enable bit (`EXCP_EN.mem_viol`). If this bit is set when the memory returns with a violation, the wave jumps to the trap handler.

Memory violations are not precise. The violation is reported when the LDS or TC processes the address; during this time, the wave may have processed many more instructions. When a `mem_viol` is reported, the Program Counter saved is that of the next instruction to execute; it has no relationship the faulting instruction.

Chapter 4
Program Flow Control

All program flow control is programmed using scalar ALU instructions. This includes loops, branches, subroutine calls, and traps. The program uses SGPRs to store branch conditions and loop counters. Constants can be fetched from the scalar constant cache directly into SGPRs.

4.1 Program Control

The instructions in Table 4.1 control the priority and termination of a shader program, as well as provide support for trap handlers.

Table 4.1 Control Instructions

Instruction	Description
S_ENDPGM	Terminates the wavefront. It can appear anywhere in the kernel and can appear multiple times.
S_NOP	Does nothing; it can be repeated in hardware up to eight times.
S_TRAP	Jumps to the trap handler.
S_RFE	Returns from the trap handler
S_SETPRIO	Modifies the priority of this wavefront: 0=lowest, 3 = highest.
S_SLEEP	Causes the wavefront to sleep for 64 – 448 clock cycles.
S_SENDMSG	Sends a message (typically an interrupt) to the host CPU.

4.2 Branching

Branching is done using one of the following scalar ALU instructions.

Table 4.2 Scalar ALU Instructions

Instruction	Description
S_BRANCH	Unconditional branch.
S_CBRANCH_<test>	Conditional branch. Branch only if <test> is true. Tests are VCCZ, VCCNZ, EXECZ, EXECNZ, SCCZ, and SCCNZ.
S_CBRANCH_CDBGSYS	Conditional branch, taken if the COND_DBG_SYS status bit is set.
S_CBRANCH_CDBGUSER	Conditional branch, taken if the COND_DBG_USER status bit is set.
S_CBRANCH_CDBGSYS_AND_USER	Conditional branch, taken only if both COND_DBG_SYS and COND_DBG_USER are set.

Table 4.2 Scalar ALU Instructions

Instruction	Description
S_SETPC	Directly set the PC from an SGPR pair.
S_SWAPPC	Swap the current PC with an address in an SGPR pair.
S_GETPC	Retrieve the current PC value (does not cause a branch).
S_CBRANCH_FORK and S_CBRANCH_JOIN	Conditional branch for complex branching.
S_SETVSKIP	Set a bit that causes all vector instructions to be ignored. Useful alternative to branching.

For conditional branches, the branch condition can be determined by either scalar or vector operations. A scalar compare operation sets the Scalar Condition Code (SCC), which then can be used as a conditional branch condition. Vector compare operations set the VCC mask, and VCCZ or VCCNZ then can be used to determine branching.

4.3 Work-Groups

Work-groups are collections of wavefronts running on the same compute unit which can synchronize and share data. Up to 16 wavefronts (1024 work-items) can be combined into a work-group. When multiple wavefronts are in a work-group, the S_BARRIER instruction can be used to force each wavefront to wait until all other wavefronts reach the same instruction; then, all wavefronts continue. Any wavefront can terminate early using S_ENDPGM, and the barrier is considered satisfied when the remaining live waves reach their barrier instruction.

4.4 Data Dependency Resolution

Shader hardware resolves most data dependencies, but a few cases must be explicitly handled by the shader program. In these cases, the program must insert S_WAITCNT instructions to ensure that previous operations have completed before continuing.

The shader has three counters that track the progress of issued instructions. S_WAITCNT waits for the values of these counters to be at, or below, specified values before continuing.

These allow the shader writer to schedule long-latency instructions, execute unrelated work, and specify when results of long-latency operations are needed.

Instructions of a given type return in order, but instructions of different types can complete out-of-order. For example, both GDS and LDS instructions use LGKM_cnt, but they can return out-of-order.

- `VM_CNT` – Vector memory count.
  Determines when memory reads have returned data to VGPRs, or memory writes have completed.
  – Incremented every time a vector-memory read or write (MIMG, MUBUF, or MTBUF format) instruction is issued.
  – Decremented for reads when the data has been written back to the VGPRs, and for writes when the data has been written to the L2 cache.

  Ordering: Memory reads and writes return in the order they were issued, including mixing reads and writes.
- `LGKM_CNT` (LDS, GDS, (K)constant, (M)essage)

  Determines when one of these low-latency instructions have completed.
  – Incremented by 1 for every LDS or GDS instruction issued, as well as by Dword-count for scalar-memory reads. For example, `smrd_fetch_time` counts the same as an `smrd_fetch_2`.
  – Decremented by 1 for LDS/GDS reads or atomic-with-return when the data has been returned to VGPRs.
  – Incremented by 1 for each `S_SENDMSG` issued. Decremented by 1 when message is sent out.
  – Decremented by 1 for LDS/GDS writes when the data has been written to LDS/GDS.
  – Decremented by 1 for each Dword returned from the data-cache (SMRD).

  Ordering
  ◊ Instructions of different types are returned out-of-order.
  ◊ Instructions of the same type are returned in the order they were issued, except scalar-memory-reads, which can return out-of-order (in which case only `S_WAITCNT` 0 is the only legitimate value).
- `EXP_CNT` – VGPR-export count.

  Determines when data has been read out of the VGPR and sent to GDS or TA, at which time it is safe to overwrite the contents of that VGPR.
  – Incremented when an Export/GDS instruction is issued from the wavefront buffer.
  – Decremented for exports/GDS when the last cycle of the export instruction is granted and executed (VGPRs read out).

  Ordering
  ◊ Exports are kept in order only within each export type (color/null, position, parameter cache).

4.5 Manually Inserted Wait States (NOPs)

The hardware does not check for the following dependencies; they must be resolved by inserting NOPs or independent instructions.

Table 4.3 Required Software-Inserted Wait States

#	First Instruction	Second Instruction	Wait	Notes
1	S_SETREG <*>	S_GETREG <same reg>	2	
2	S_SETREG <*>	S_SETREG <same reg>	1	
3	SET_VSKIP	S_GETREG MODE	2	Reads VSKIP from MODE.
4	S_SETREG MODE.vskip	any vector op	2	Requires 2 nops or non-vector instructions.
5	VALU that sets VCC or EXEC	VALU which uses EXECZ or VCCZ as a data source	5	
6	VALU writes SGPR/VCC *(readlane, cmp, add/sub, div_scale)*	V_{READ,WRITE}LANE using that SGPR/VCC as the lane select	4	
7	VALU writes VCC (including v_div_scale)	V_DIV_FMAS	4	
8	SALU writes to SGPR that holds VCC (not writes to VCC directly)	CBRANCH on VCCZ	1	For example: if VCC is in SGPR 30,31, and the shader writes to VCC, no wait is needed. But if the shader writes to SGPR31, then uses VCCZ, one wait state is needed.
9	FLAT_STORE_X3 FLAT_STORE_X4 FLAT_ATOMIC_{F}CMPSWAP_X2 BUFFER_STORE_DWORD_X3 BUFFER_STORE_DWORD_X4 BUFFER_STORE_FORMAT_XYZ BUFFER_STORE_FORMAT_XYZW BUFFER_ATOMIC_{F}CMPSWAP_X2 IMAGE_STORE_* > 64 bits IMAGE_ATOMIC_{F}CMPSWAP > 64bits	Write VGPRs holding write-data from those instructions AND if BUFFER, doesn't use an SGPR for "offset". (If the buffer instruction uses an SGPR for its offset, there is no data hazard.)	1	BUFFER_STORE_* operations that use an SGPR for 'offset' do not require any wait states. IMAGE_STORE_* and IMAGE_{F}CMPSWAP* ops with more than 2 DMASK bits set require this 1 wait state.

4.6 Arbitrary Divergent Control Flow

In the Sea Islands architecture, conditional branches are handled in one of the following ways.

1. S_CBRANCH

   This case is used for simple control flow, where the decision to take a branch is based on a previous compare operation. This is the most common method for conditional branching.

2. `S_CBRANCH_I/G_FORK` and `S_CBRANCH_JOIN`

   This method, intended for more complex, irreducible control flow graphs, is described in the rest of this section. The performance of this method is lower than that for `S_CBRANCH` on simple flow control; use it only when necessary.

Conditional Branch (CBR) graphs are grouped into self-contained code blocks, denoted by `FORK` at the entrance point, and `JOIN` and the exit point (see Figure 4.1). The shader compiler must add these instructions into the code. This method uses a six-deep stack and requires three SGPRs for each fork/join block. Fork/Join blocks can be hierarchically nested to any depth (subject to SGPR requirements); they also can coexist with other conditional flow control or computed jumps.

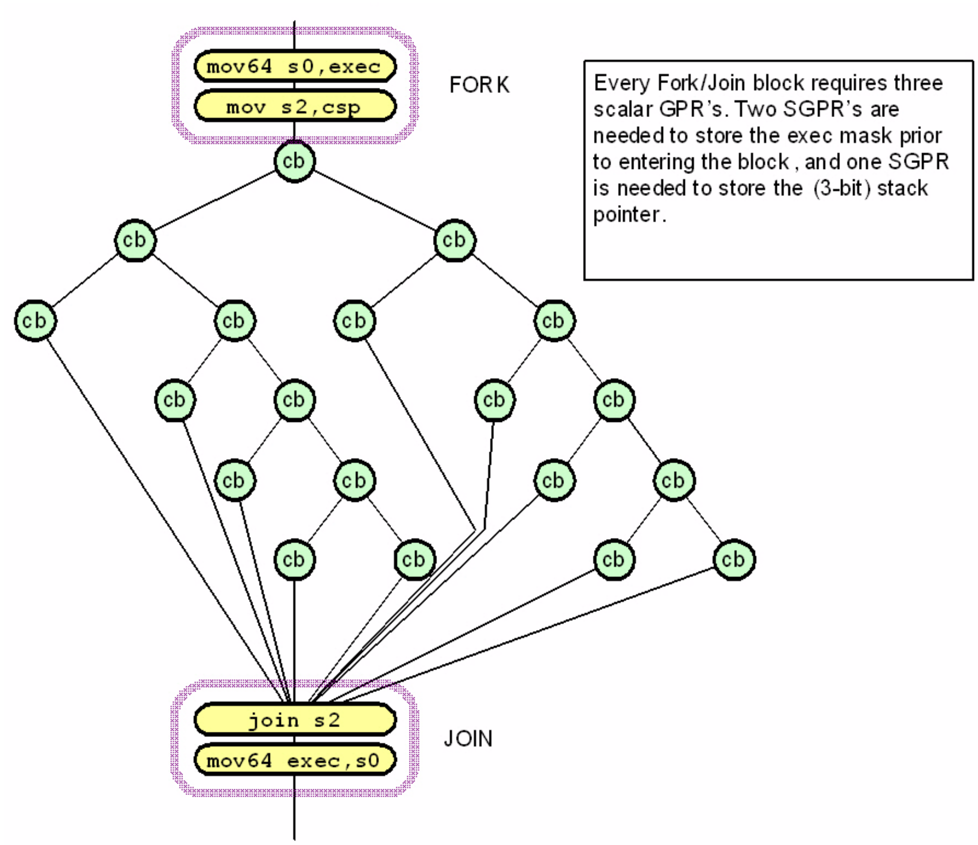


Figure 4.1 Example of Complex Control Flow Graph

The register requirements per wavefront are:

- CSP [2:0] - control stack pointer.
- Six stack entries of 128-bits each, stored in SGPRS: { exec[63:0], PC[47:2] }

This method compares how many of the 64 threads go down the PASS path instead of the FAIL path; then, it selects the path with the fewer number of threads first. This means at most 50% of the threads are active, and this limits the necessary stack depth to $\text{Log}_2 64 = 6$.

The following pseudo-code shows the details of CBRANCH Fork and Join operations.

```
S_CBRANCH_G_FORK arg0, arg1                        // arg1 is an sgpr-pair which holds 64bit
                                                      (48bit) target address
S_CBRANCH_I_FORK arg0, #target_addr_offset[17:2] // target_addr_offset is a 16b signed
                                                      immediate offset
                                                   // "PC" in this pseudo-code is pointing to
                                                      the cbranch_*_fork instruction
mask_pass  =   SGPR[arg0] & exec
    mask_fail  =  ~SGPR[arg0] & exec
    if (mask_pass == exec)
            I_FORK  : PC += 4 + target_addr_offset
G_FORK:  PC = SGPR[arg1]
    else if (mask_fail == exec)
            PC += 4
    else if (bitcount(mask_fail) < bitcount(mask_pass))
            exec = mask_fail
            I_FORK :  SGPR[CSP*4] = { (pc + 4 + target_addr_offset), mask_pass }
G_FORK: SGPR[CSP*4] = { SGPR[arg1], mask_pass }
                        CSP++
                        PC += 4
    else
            exec = mask_pass
            SGPR[CSP*4] = { (pc+4), mask_fail }
                            CSP++
            I_FORK  : PC += 4 + target_addr_offset
G_FORK:  PC = SGPR[arg1]

S_CBRANCH_JOIN arg0
    if (CSP == SGPR[arg0])        // SGPR[arg0] holds the CSP value when the FORK started
            PC += 4               // this is the 2nd time to JOIN: continue with pgm
    else
            CSP --                // this is the 1st time to JOIN: jump to other FORK path
            {PC, EXEC} = SGPR[CSP*4]    // read 128-bits from 4 consecutive SGPRs
```

Chapter 5
Scalar ALU Operations

Scalar ALU (SALU) instructions operate on a single value per wavefront. These operations consist of 32-bit integer arithmetic and 32- or 64-bit bitwise operations. The SALU also can perform operations directly on the Program Counter, allowing the program to create a call stack in SGPRs. Many operations also set the Scalar Condition Code bit (SCC) to indicate the result of a comparison, a carry-out, or whether the instruction result was zero.

5.1 SALU Instruction Formats

SALU instructions are encoded in one of 5 microcode formats, shown below:

	31	30	29 – 23	22 – 16	15 – 8	7 – 0
SOP2	1	0	OP_7	$SDST_7$	$SSRC1_8$	$SSRC0_8$

	31	30	29	28	27 – 23	22 – 16	15 – 0
SOPK	1	0	1	1	OP_5	$SDST_7$	SIMM16

	31	30	29	28	27	26	25	24	23	22 – 16	15 – 8	7 – 0
SOP1	1	0	1	1	1	1	1	0	1	$SDST_7$	OP_8	$SSRC0_8$

	31	30	29 – 23							22 – 16	15 – 8	7 – 0
SOPC	1	0	1	1	1	1	1	1	0	OP_7	$SSRC1_8$	$SSRC0_8$

	31	30	29 – 23							22 – 16	15 – 0
SOPP	1	0	1	1	1	1	1	1	1	OP_7	SIMM16

Field	Description
OP	Opcode: instruction to be executed.
SDST	Destination SGPR.
SSRC0	First source operand.
SSRC1	Second source operand.
SIMM16	Signed immediate integer constant.

The lists of similar instructions sometimes uses a condensed form using curly braces { } to express a list of possible names. For example, `S_AND_{B32, B64}` defines two legal instructions: `S_AND_B32` and `S_AND_B64`.

5.2 Scalar ALU Operands

Valid operands of SALU instructions are:

- SGPRs, including trap temporary SGPRs.
- Mode register.
- Status register (read-only).
- M0 register.
- TrapSts register.
- EXEC mask.
- VCC mask.
- SCC.
- PC.
- Inline constants: integers from -16 to 64, and a some floating point values.
- VCCZ, EXECZ, and SCCZ.
- Hardware registers.

The SALU cannot use VGPRs or LDS.

SALU instructions can use a 32-bit literal constant. This constant is part of the instruction stream and is available to all SALU microcode formats except SOPP and SOPK.

If any source SGPR is out-of-range, the value of SGPR0 is used instead.

If the destination SGPR is out-of-range, no SGPR is written with the result.

If an instruction uses 64-bit data in SGPRs, the SGPR pair must be aligned to an even boundary. For example, it is legal to use SGPRS 2 and 3 or 8 and 9 (but not 11 and 12) to represent 64-bit data.

5.3 Scalar Condition Code (SCC)

The scalar condition code (SCC) is written as a result of executing most SALU instructions.

The SCC is set by many instructions:

- Compare operations: 1 = true.
- Arithmetic operations: 1 = carry out.
  - SCC = overflow for signed add and subtract operations
    overflow = both operands are of the same sign, and the MSB (sign bit)

of the result is different than the sign of the operands. For subtract (A-B), overflow = A and B have opposite signs and the resulting sign is not the same as the sign of A.

- Bit/logical operations: 1 = result was not zero.

Table 5.1 Scalar Condition Code

		Code	Meaning	
Scalar Source (8 bits)	**Scalar Dest (7 bits)**	0 - 103	SGPR 0 to 103	Scalar GPRs.
		104	FLAT_SCR_LO	Holds the low Dword of the flat-scratch memory descriptor.
		105	FLAT_SCR_HI	Holds the high Dword of the flat-scratch memory descriptor.
		106	VCC_LO	vcc[31:0].
		107	VCC_HI	vcc[63:32].
		108	TBA_LO	Trap handler base address, [31:0].
		109	TBA_HI	Trap handler base address, [63:32].
		110	TMA_LO	Pointer to data in memory used by trap handler.
		111	TMA_HI	Pointer to data in memory used by trap handler.
		112-123	ttmp0 to ttmp11	Trap handler temps (privileged). {ttmp1,ttmp0} = PC_save{hi,lo}.
		124	M0	Temporary memory register.
		125	reserved	
		126	EXEC_LO	exec[31:0].
		127	EXEC_HI	exec[63:32].
		128	0	Immediate (constant value 0).
		129-192	int 1 to 64	Positive integer values.
		193-208	int -1 to -16	Negative integer values.
		209-239	*reserved*	*unused*
		240	0.5	single or double floats
		241	-0.5	
		242	1.0	
		243	-1.0	
		244	2.0	
		245	-2.0	
		246	4.0	
		247	-4.0	
		248-250	*reserved*	*unused*
		251	VCCZ	{ *zeros*, VCCZ }
		252	EXECZ	{ *zeros*, EXECZ }
		253	SCC	{ *zeros*, SCC }
		254	*reserved*	
		255	Literal constant	32-bit constant from instruction stream.

5.4 Integer Arithmetic Instructions

This section describes the arithmetic operations supplied by the SALU.

Table 5.2 Integer Arithmetic Instructions

Instruction	Encoding	Sets SCC?	Operation
S_ADD_I32	SOP2	y	D = S1 + S2, SCC = overflow.
S_ADD_U32	SOP2	y	D = S1 + S2, SCC= carry out.
S_ADDC_U32	SOP2	y	D = S1 + S2 + SCC.
S_SUB_I32	SOP2	y	D = S1 - S2, SCC = overflow.
S_SUB_U32	SOP2	y	D = S1 - S2, SCC = carry out.
S_SUBB_U32	SOP2	y	D = S1 - S2 - SCC.
S_ABSDIFF_I32	SOP2	y	D = abs (s1 - s2).
S_MIN_I32 S_MIN_U32	SOP2	y	D = (S1 < S2) ? S1 : S2. SCC = 1 if S1 was min.
S_MAX_I32 S_MAX_U32	SOP2	y	D = (S1 > S2) ? S1 : S2. SCC = 1 if S1 was max.
S_MUL_I32	SOP2	n	D = S1 * S2. Low 32 bits of result.
S_ADDK_I32	SOPK	y	D = D + simm16.
S_MULK_I32	SOPK	n	D = D * simm16. Return low 32bits.
S_ABS_I32	SOP1	y	D.i = abs (S1.i). SCC=result not zero.
S_SEXT_I32_I8	SOP1	n	D = { 24{S1[7]}, S1[7:0] }.
S_SEXT_I32_I16	SOP1	n	D = { 16{S1[15]}, S1[15:0] }.

5.5 Conditional Instructions

Conditional instructions use the SCC flag to determine whether to perform the operation, or (for CSELECT) which source operand to use.

Table 5.3 Conditional Instructions

Instruction	Encoding	Sets SCC?	Operation
S_CSELECT_{B32, B64}	SOP2	n	D = SCC ? S1 : S2.
S_CMOVK_I32	SOPK	n	if (SCC) D = signext(simm16).
S_CMOV_{B32,B64}	SOP1	n	if (SCC) D = S1, else NOP.

5.6 Comparison Instructions

These instructions compare two values and set the SCC to 1 if the comparison yielded a TRUE result.

Table 5.4 Comparison Instructions

Instruction	Encoding	Sets SCC?	Operation
S_CMP_{EQ,NE,GT,GE,LE,LT}_{I32,U32}	SOPC	y	Compare two source values. SCC = S1 <cond> S2.
S_CMPK_{EQ,NE,GT,GE,LE,LT}_{I32,U32}	SOPK	y	Compare Dest SGPR to a constant. SCC = DST <cond> simm16.
S_BITCMP0_{B32,B64}	SOPC	y	Test for "is a bit zero". SCC = !S1[S2].
S_BITCMP1_{B32,B64}	SOPC	y	Test for "is a bit one". SCC = S1[S2].

5.7 Bit-Wise Instructions

Bit-wise instructions operate on 32- or 64-bit data without interpreting it has having a type.

Table 5.5 Bit-Wise Instructions

Instruction	Encoding	Sets SCC?	Operation
S_MOV_{B32,B64}	SOP1	n	D = S1
S_MOVK_I32	SOPK	n	D = signext(simm16)
S_AND, S_OR, S_XOR *_{B32,B64}	SOP2	y	D = S1 & S2, S1 OR S2, S1 XOR S2
S_ANDN2, S_ORN2 *_{B32,B64}	SOP2	y	D = S1 & ~S2, S1 OR ~S2, S1 XOR ~S2,
S_NAND, S_NOR, S_XNOR *_{B32,B64}	SOP2	y	D = ~(S1 & S2), ~(S1 OR S2), ~(S1 XOR S2)
S_LSHL_{B32,B64}	SOP2	y	D = S1 << S2[4:0] , [5:0] for B64.
S_LSHR_{B32,B64}	SOP2	y	D = S1 >> S2[4:0] , [5:0] for B64.
S_ASHR_{I32,I64}	SOP2	y	D = sext(S1 >> S2[4:0]) ([5:0] for I64).
S_BFM_{B32,B64}	SOP2	n	Bit field mask. D = ((1<<S1[4:0]) -1) << S2[4:0].
S_BFE_U32, S_BFE_U64 S_BFE_I32, S_BFE_I64 (signed/unsigned)	SOP2	n	Bit Field Extract, then sign-extend result for I32/64 instructions. S1 = data, S2[5:0] = offset, S2[22:16]= width.
S_NOT_{B32,B64}	SOP1	y	D = !S1.
S_WQM_{B32,B64}	SOP1	y	D = wholeQuadMode(S1). If any bit in a group of four is set to 1, set the resulting group of four bits all to 1.
S_QUADMASK_{B32,B64}	SOP1	y	D[0] = OR(S1[3:0]), D[1]=OR(S1[7:4]), etc.

Table 5.5 Bit-Wise Instructions (Cont.)

Instruction	Encoding	Sets SCC?	Operation
S_BREV_{B32,B64}	SOP1	n	D = S1[0:31] are reverse bits.
S_BCNT0_I32_{B32,B64}	SOP1	y	D = CountZeroBits(S1).
S_BCNT1_I32_{B32,B64}	SOP1	y	D = CountOneBits(S1).
S_FF0_I32_{B32,B64}	SOP1	n	D = Bit position of first zero in S1 starting from LSB. -1 if not found.
S_FF1_I32_{B32,B64}	SOP1	n	D = Bit position of first one in S1 starting from LSB. -1 if not found.
S_FLBIT_I32_{B32,B64}	SOP1	n	Find last bit. D = the number of zeros before the first one starting from the MSB. Returns -1 if none.

S_FLBIT_I32
S_FLBIT_I32_I64
	SOP1	n	Count how many bits in a row (from MSB to LSB) are the same as the sign bit. Return -1 if the input is zero or all 1's (-1). 32-bit pseudo-code:
			`if (S0 == 0 || S0 == -1)  D = -1`
			`else`
			`D = 0`
			`for (I = 31 .. 0)`
			`if (S0[I] == S0[31])`
			`D++`
			`else break`
			This opcode behaves the same as V_FFBH_I32.

S_BITSET0_{B32,B64}	SOP1	n	D[S1] = 0
S_BITSET1_{B32,B64}	SOP1	n	D[S1] = 1
S_{and,or,xor,andn2,orn2,nand,nor,xnor} _SAVEEXEC_B64	SOP1	y	Save the EXEC mask, then apply a bitwise operation to it. D = EXEC EXEC = S1 <op> EXEC SCC = (exec != 0)
S_MOVRELS_{B32,B64} S_MOVRELD_{B32,B64}	SOP1	n	Move a value into an SGPR relative to the value in M0. MOVERELS: D = SGPR[S1+M0] MOVERELD: SGPR[D+M0] = S1 Index must be even for 64. M0 is an unsigned index.

5.8 Special Instructions

These instructions access hardware internal registers.

Table 5.6 Access Hardware Internal Register Instructions

Instruction	Encoding	Sets SCC?	Operation
`S_GETREG_B32`	SOPK*	n	Read a hardware register into the LSBs of D.
`S_SETREG_B32`	SOPK*	n	Write the LSBs of D into a hardware register. (Note that D is a source SGPR.) Must add an `S_NOP` between two consecutive `S_SETREG` to the same register.
`S_SETREG_IMM32_B32`	SOPK*	n	`S_SETREG` where 32-bit data comes from a literal constant (so this is a 64-bit instruction format).

The hardware register is specified in the DEST field of the instruction, using the values in Table 5.7.

Table 5.7 Hardware Register Values

Code	Register	Description
0	reserved	
1	MODE	R/W.
2	STATUS	Read only.
3	TRAPSTS	R/W.
4	HW_ID	Read only. Debug only.
5	GPR_ALLOC	Read only. {sgpr_size, sgpr_base, vgpr_size, vgpr_base }
6	LDS_ALLOC	Read only. {lds_size, lds_base}
7	IB_STS	Read only. {valu_cnt, lgkm_cnt, exp_cnt, vm_cnt}

The following tables describe some of the registers in Table 5.7.

Table 5.8 IB_STS

Field	Bits	Description
VM_CNT	3:0	Number of VMEM instructions issued but not yet returned.
EXP_CNT	6:4	Number of Exports issued but have not yet read their data from VGPRs.
LGKM_CNT	10:8	LDS, GDS, Constant-memory and Message instructions issued-but-not-completed count.
VALU_CNT	14:12	Number of VALU instructions outstanding for this wavefront.

Table 5.9 GPR_ALLOC

Field	Bits	Description
VGPR_BASE	5:0	Physical address of first VGPR assigned to this wavefront, as [7:2]
VGPR_SIZE	13:8	Number of VGPRs assigned to this wavefront, as [7:2]. 0=4 VGPRs, 1=8 VGPRs, etc.
SGPR_BASE	21:16	Physical address of first SGPR assigned to this wavefront, as [7:3].
SGPR_SIZE	27:24	Number of SGPRs assigned to this wave, as [7:3]. 0=8 SGPRs, 1=16 SGPRs, etc.

Table 5.10 LDS_ALLOC

Field	Bits	Description
LDS_BASE	7:0	Physical address of first LDS location assigned to this wavefront, in units of 64 Dwords.
LDS_SIZE	20:12	Amount of LDS space assigned to this wavefront, in units of 64 Dwords.

Chapter 6
Vector ALU Operations

Vector ALU instructions (VALU) perform an arithmetic or logical operation on data for each of 64 threads and write results back to VGPRs, SGPRs or the EXEC mask.

Parameter interpolation is a mixed VALU and LDS instruction, and is described in the Data Share chapter.

6.1 Microcode Encodings

Most VALU instructions are available in two encodings: VOP3 which uses 64-bits of instruction and has the full range of capabilities, and one of three 32-bit encodings that offer a restricted set of capabilities. A few instructions are only available in the VOP3 encoding. The only instructions that cannot use the VOP3 format are the parameter interpolation instructions.

When an instruction is available in two microcode formats, it is up to the user to decide which to use. It is recommended to use the 32-bit encoding whenever possible.

The microcode encodings are shown below.

VOP2 is for instructions with two inputs and a single vector destination. Instructions that have a carry-out implicitly write the carry-out to the VCC register.

0	OP	VDST	VSRC1	SRC0	+0

VOP1 is for instructions with no inputs or a single input and one destination.

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

VOPC is for comparison instructions.

0	**1**	**1**	**1**	**1**	**1**	**0**	OP	VSRC1	SRC0	+0

VINTRP is for parameter interpolation instructions.

1	**1**	**0**	**0**	**1**	**0**	VDST	OP	ATTR	ATTR-CHAN	VSRC (I, J)	+0

VOP3 is for instructions with up to three inputs, input modifiers (negate and absolute value), and output modifiers. There are two forms of VOP3: one which uses a scalar destination field (used only for div_scale, integer add and subtract); this is designated VOP3b. All other instructions use the common form, designated VOP3a.

VOP3a:

NEG	OMOD	SRC2	SRC1	SRC0	+4

1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

VOP3b:

NEG	OMOD	SRC2	SRC1	SRC0	+4

1	**1**	**0**	**1**	**0**	**0**	OP	r	SDST	VDST	+0

Any of the 32-bit microcode formats may use a 32-bit literal constant, but not VOP3.

6.2 Operands

All VALU instructions take at least one input operand (except `V_NOP` and `V_CLREXCP`). The data-size of the operands is explicitly defined in the name of the instruction. For example, `V_MAD_F32` operates on 32-bit floating point data.

6.2.1 Instruction Inputs

VALU instructions can use any of the following sources for input, subject to restrictions listed below:

- VGPRs.
- SGPRs.
- Inline constants – a constant selected by a specific VSRC value (see Table 6.1).
- Literal constant – a 32-bit value in the instruction stream. When a literal constant is used with a 64bit instruction, the literal is expanded to 64 bits by:

padding the LSBs with zeros for floats, padding the MSBs with zeros for unsigned ints, and by sign-extending signed ints.

- LDS direct data read.
- M0.
- EXEC mask.

Limitations

- At most one SGPR can be read per instruction, but the value can be used for more than one operand.
- At most one literal constant can be used, and only when an SGPR or M0 is not used as a source.
- Only SRC0 can use LDS_DIRECT (see Chapter 10).

Instructions using the VOP3 form and also using floating-point inputs have the option of applying absolute value (ABS field) or negate (NEG field) to any of the input operands.

6.2.2 Instruction Outputs

VALU instructions typically write their results to VGPRs specified in the VDST field of the microcode word. A thread only writes a result if the associated bit in the EXEC mask is set to 1.

All V_CMPX instructions write the result of their comparison (one bit per thread) to both an SGPR (or VCC) and the EXEC mask.

Instructions producing a carry-out (integer add and subtract) write their result to VCC when used in the VOP2 form, and to an arbitrary SGPR-pair when used in the VOP3 form.

When the VOP3 form is used, instructions with a floating-point result can apply an output modifier (OMOD field) that multiplies the result by: 0.5, 1.0, 2.0 or 4.0. Optionally, the result can be clamped (CLAMP field) to the range [-1.0, +1.0], as indicated in Table 6.1.

In Table 6.1, all codes can be used when the vector source is nine bits; codes 0 to 255 can be the scalar source if it is eight bits; codes 0 to 127 can be the scalar source if it is seven bits; and codes 256 to 511 can be the vector source or destination.

Table 6.1 Instruction Operands

Field	Bit Position	Description
0 – 103	SGPR 0 .. 103	
104	FLATSCR_LO	Flat Scratch[31:0].
105	FLATSCR_HI	Flat Scratch[63:32].
106	VCC_LO	vcc[31:0].

Table 6.1 Instruction Operands (Cont.)

Field	Bit Position	Description
107	VCC_HI	vcc[63:32].
108	TBA_LO	Trap handler base address, [31:0].
109	TBA_HI	Trap handler base address, [63:32].
110	TMA_LO	Pointer to data in memory used by trap handler.
111	TMA_HI	Pointer to data in memory used by trap handler.
112-123	ttmp0..ttmp11	Trap handler temps (privileged). {ttmp1,ttmp0} = PC_save{hi,lo}
124	M0	
125	reserved	
126	EXEC_LO	exec[31:0].
127	EXEC_HI	exec[63:32].
128	0	
129-192	int 1.. 64	Integer inline constants.
193-208	int -1 .. -16	
209-239	reserved	Unused.
240	0.5	Single or double inline floats.
241	-0.5	
242	1.0	
243	-1.0	
244	2.0	
245	-2.0	
246	4.0	
247	-4.0	
248-250	reserved	Unused.
251	VCCZ	{ *zeros*, VCCZ }
252	EXECZ	{ *zeros*, EXECZ }
253	SCC	{ *zeros*, SCC }
254	LDS direct	Use LDS direct read to supply 32-bit value *Vector-alu instructions only.*
255	Literal constant	32-bit constant from instruction stream.
256 – 511	VGPR 0 .. 255	

6.2.3 Out-of-Range GPRs

When a source VGPR is out-of-range, the instruction uses as input the value from VGPR0.

When the destination GPR is out-of-range, the instruction executes but does not write the results.

6.2.4 GPR Indexing

The M0 register can be used along with the `V_MOVRELS` and `V_MOVERELD` instructions to provide indexed access to VGPRs.

- `V_MOVRELS` performs: VGPR[dst] = VGPR[src + m0]
- `V_MOVRELD` performs: VGPR[dst+m0] = VGPR[src]

6.3 Instructions

Table 6.2 lists the complete VALU instruction set by microcode encoding.

Table 6.2 VALU Instruction Set

VOP3	VOP3 – 1-2 operand opcodes	VOP2	VOP1
V_MAD_LEGACY_F32	V_ADD_F64	V_READLANE_B32	V_NOP
V_MAD_F32	V_MUL_F64	V_WRITELANE_B32	V_MOV_B32
V_MAD_I32_I24	V_MIN_F64	V_ADD_{F32, I32}	V_READFIRSTLANE_B32
V_MAD_U32_U24	V_MAX_F64	V_SUB_{F32, I32}	V_CVT_F32_{I32,U32,F16,F64}
V_CUBEID_F32	V_LDEXP_F64	V_SUBREV_{F32, I32}	V_CVT_{I32,U32,F16, F64}_F32
V_CUBESC_F32	V_MUL_{LO,HI}_{I32, U32}	V_MAC_LEGACY_F32	V_CVT_{I32,U32}_F64
V_CUBETC_F32	V_LSHL_B64	V_ADDC_U32	V_CVT_F64_{I32,U32}
V_CUBEMA_F32	V_LSHR_B64	V_SUBB_U32	V_CVT_F32_UBYTE{0,1,2,3}
V_BFE_{U32 , I32 }	V_ASHR_I64	V_SUBBREV_U32	V_CVT_RPI_I32_F32
V_FMA_{F32 , F64}		V_MUL_LEGACY_F32	V_CVT_FLR_I32_F32
V_BFI_B32		V_MUL_F32	V_CVT_OFF_F32_I4
V_LERP_U8		V_MUL_I32_I24	V_FRACT_{F32,F64}
V_ALIGNBIT_B32		V_MUL_HI_I32_I24	V_TRUNC_{F32, F64}
V_ALIGNBYTE_B32		V_MUL_U32_U24	V_CEIL_{F32, F64}
V_MULLIT_F32		V_MUL_HI_U32_U24	V_RNDNE_{F32, F64}
V_MIN3_{F32,I32,U32}		V_MIN_LEGACY_F32	V_FLOOR_{F32, F64}
V_MAX3_{F32,I32,U32}		V_MAX_LEGACY_F32	V_EXP_F32
V_MED3_{F32,I32,U32}		V_MIN_{F32,I32,U32}	V_LOG_CLAMP_F32
V_SAD_{U8, HI_U8, U16, U32}		V_MAX_{F32,I32,U32}	V_LOG_F32
V_CVT_PK_U8_F32		V_LSHR_B32	V_RCP_{F32,F64}
V_DIV_FIXUP_{F32,F64}		V_LSHRREV_B32	V_RCP_CLAMP_{F32,F64}
V_DIV_SCALE_{F32,F64}		V_ASHR_I32	V_RCP_LEGACY_F32
V_DIV_FMAS_{F32,F64}		V_ASHRREV_I32	V_RCP_IFLAG_F32
V_QSAD_U8		V_LSHL_B32	V_RSQ_CLAMP_{F32,F64}
V_MSAD_U8		V_LSHLREV_B32	V_RSQ_LEGACY_F32
V_QMSAD_U8		V_AND_B32	V_RSQ_{F32, F64}
V_QSAD_PK_U16_U8		V_OR_B32	V_SQRT_{F32,F64}

V_MQSAD_PK_U16_U8		V_XOR_B32	V_SIN_F32
V_TRIG_PREOP_F64		V_BFM_B32	V_COS_F32
V_MQSAD_U32_U8		V_MAC_F32	V_NOT_B32
V_MAD{U64_U32, I64_I32}		V_MADMK_F32	V_BFREV_B32
		V_MADAK_F32	V_FFBH_U32
		V_BCNT_U32_B32	V_FFBL_B32
		V_MBCNT_LO_U32_B32	V_FFBH_I32
		V_MBCNT_HI_U32_B32	V_FREXP_EXP_I32_F64
		V_LDEXP_F32	V_FREXP_MANT_F64
		V_CVT_PKACCUM_U8_F32	V_FREXP_EXP_I32_F32
		V_CVT_PKNORM_I16_F32	V_FREXP_MANT_F32
		V_CVT_PKNORM_U16_F32	V_CLREXCP
		V_CVT_PKRTZ_F16_F32	V_MOVRELD_B32
		V_CVT_PK_U16_U32	V_MOVRELS_B32
		V_CVT_PK_I16_I32	V_MOVRELSD_B32
		V_CNDMASK_B32	

Table 6.3 lists the compare instruction.

Table 6.3 Compare Operations

All are VOPC or VOP3 : VOPC writes to VCC, VOP3 writes compare result to any SGPR			
V_CMP	I32, I64, U32, U64	F, LT, EQ, LE, GT, LG, GE, T	
V_CMPX			write exec
V_CMP	F32, F64	F, LT, EQ, LE, GT, LG, GE, T, O, U, NGE, NLG, NGT, NLE, NEQ, NLT (o = total order, u = unordered, "N" = NaN or normal compare)	
V_CMPX	F32, F64		write exec
V_CMPS	F32, F64		signal NaNs
V_CMPSX	F32, F64		write exec, signal NaNs

6.4 Denormals and Rounding Modes

The shader program has explicit control over the rounding mode applied and the handling of denormalized inputs and results. The MODE register is set using the `S_SETREG` instruction; it has separate bits for controlling the behavior of single- and double-precision floating-point numbers (see Table 6.4).

Table 6.4 MODE Register FP Bits

Field	Bit Position	Description
FP_ROUND	3:0	[1:0] Single-precision round mode. [3:2] Double-precision round mode. Round Modes: 0=nearest even; 1= +infinity; 2= -infinity,; 3= toward zero.
FP_DENORM	7:4	[5:4] Single-precision denormal mode. [7:6] Double-precision denormal mode. Denormal modes: 0 = Flush input and output denorms. 1 = Allow input denorms, flush output denorms. 2 = Flush input denorms, allow output denorms. 3 = Allow input and output denorms.

Chapter 7
Scalar Memory Operations

Scalar Memory Read (SMRD) instructions allow a shader program to load data from memory into SGPRs through the Scalar Constant Cache. Instructions can fetch from 1 to 16 dwords at a time. Data is read directly into SGPRs with any format conversion.

The scalar unit can read 1-16 consecutive dwords from memory into the SGPRs. This is intended primarily for loading ALU constants and for indirect T#/S# lookup. No data formatting is supported, nor is byte or short data.

7.1 Microcode Encoding

Scalar memory read instructions are encoded using the SMRD microcode format.

1	**1**	**0**	**0**	**0**	OP	SDST	SBASE	IMM	OFFSET	+0

The fields are described in Table 7.1

Table 7.1 SMRD Encoding Field Descriptions

Field	Bits	Description
OP	26:22	Opcode S_LOAD_DWORD{“”,X2,X4,X8,X16} S_BUFFER_LOAD_DWORD{“”,X2,X4,X8,X16} S_DCACHE_INV, S_DCACHE_INV_VOL S_MEMTIME
SDST	21:16	Destination SGPR. Specifies which SGPR to receives the data from this instruction. When the instruction fetches multiple Dwords, this specifies the first of N consecutive SGPRs. SDST must be an even number for reads of two dwords; it must be a multiple of 4 for reads of four or more Dwords. SDST can only be an SGPR or VCC (not EXEC or M0). SDST must not be a trap-temporary register.

Table 7.1 SMRD Encoding Field Descriptions (Cont.)

Field	Bits	Description
SBASE	15:9	S_LOAD_DWORD*: Specifies the SGPR-pair that holds the base byte-address for the fetch. The LSBs of the address are in the lower numbered SGPR. S_BUFFER_LOAD_DWORD*: specifies the first of four consecutive SGPRs that contain the buffer constant. Must be a multiple of 2. This field is missing the LSB; so, for example, when SBASE=2, it means use SGPRs 4 and 5.
IMM	8	Determines the meaning of the OFFSET field. 1 = OFFSET is an eight-bit unsigned Dword offset to the address. 0 = OFFSET is the address of an SGPR that supplies an unsigned byte offset to the address. Inline constants are not allowed.
OFFSET	7:0	Depending on the IMM field, either supplies a Dword offset from the immediate field, or the address of an SGPR that contains a byte offset. This field also can be set to SQ_SRC_LITERAL, indicating that the offset is specified in a 32-bit literal constant.

7.2 Operations

7.2.1 S_LOAD_DWORD

These instructions load 1-16 Dwords from memory at the address specified in the SBASE register plus the offset. SBASE holds a 64-bit byte-address.

Memory Address = BASE + OFFSET, truncated to a Dword address

7.2.2 S_BUFFER_LOAD_DWORD

These instructions also load 1-16 Dwords from memory, but they use a buffer resource (V#, described in Chapter 8). The resource provides:

- Base address
- Stride
- Num_records

All other buffer resource fields are ignored.

Memory Address = Base (from V#) + OFFSET, truncated to a DWORD address.

The address is clamped if:

- Stride is zero: clamp if (offset >= num_records)
- Stride is non-zero: clamp if (offset > (stride * num_records))

7.2.3 S_DCACHE_INV

This instruction invalidates the entire data cache. It does not return anything to SDST.

7.2.4 S_DCACHE_INV_VOL

This instruction invalidates the entire data cache of any data marked as “volatile.” It does not return anything to SDST.

7.2.5 S_MEM_TIME

This instruction reads a 64-bit clock counter into a pair of SGPRs: SDST and SDST+1.

7.3 Dependency Checking

Scalar memory reads can return data out-of-order from how they were issued; they can return partial results at different times when the read crosses two cache lines. The shader program uses the LGKM_CNT counter to determine when the data has been returned to the SDST SGPRs. This is done as follows.

- LGKM_CNT is incremented by 1 for every fetch of a single Dword.
- LGKM_CNT is incremented by 2 for every fetch of two or more Dwords.
- LGKM_CNT is decremented by an equal amount when each instruction completes.

Because the instructions can return out-of-order, the only sensible way to use this counter is to implement `S_WAITCNT 0`; this imposes a wait for all data to return from previous SMRDs before continuing.

`S_MEM_TIME` executes like any other scalar memory read. `S_WAITCNT 0` must be executed before the result from `S_MEM_TIME` is available for use.

7.4 Alignment and Bounds Checking

SDST – The value of SDST must be even for fetches of two Dwords (including S_MEMTIME), or a multiple of four for larger fetches. If this rule is not followed, invalid data can result. If SDST is out-of-range, the instruction is not executed.

SBASE – The value of SBASE must be even for S_BUFFER_LOAD (specifying the address of an SGPR which is a multiple of four). If SBASE is out-of-range, the value from SGPR0 is used.

OFFSET – The value of OFFSET has no alignment restrictions.

Memory Address – If the memory address is out-of-range (clamped), the operation is not performed for any Dwords that are out-of-range.

Chapter 8
Vector Memory Operations

Vector Memory (VMEM) instructions read or write one piece of data separately for each work-item in a wavefront into, or out of, VGPRs. This is in contrast to Scalar Memory instructions, which move a single piece of data that is shared by all threads in the wavefront. All Vector Memory (VM) operations are processed by the texture cache system (level 1 and level 2 caches).

Software initiates a load, store or atomic operation through the texture cache through one of three types of VMEM instructions:

- MTBUF – Memory typed-buffer operations.
- MUBUF – Memory untyped-buffer operations.
- MIMG – Memory image operations.

These instruction types are described by one of three 64-bit microcode formats (see Section 13.6, "Vector Memory Buffer Instructions," page 13-41 and Section 13.7, "Vector Memory Image Instruction," page 13-48). The instruction defines which VGPR(s) supply the addresses for the operation, which VGPRs supply or receive data from the operation, and a series of SGPRs that contain the memory buffer descriptor (`V#` or `T#`). Also, MIMG operations supply a texture sampler from a series of four SGPRs; this sampler defines texel filtering operations to be performed on data read from the image.

8.1 Vector Memory Buffer Instructions

Vector-memory (VM) operations transfer data between the VGPRs and buffer objects in memory through the texture cache (TC). "Vector" means that one or more piece of data is transferred uniquely for every thread in the wavefront, in contrast to scalar memory reads, which transfer only one value that is shared by all threads in the wavefront.

Buffer reads have the option of returning data to VGPRs or directly into LDS.

Examples of buffer objects are vertex buffers, raw buffers, stream-out buffers, and structured buffers.

Buffer objects support both homogenous and heterogeneous data, but no filtering of read-data (no samplers). Buffer instructions are divided into two groups:

- MUBUF – Untyped buffer objects.
  - Data format is specified in the resource constant.

  – Load, store, atomic operations, with or without data format conversion.
- MTBUF – Typed buffer objects.
  – Data format is specified in the instruction.
  – The only operations are Load and Store, both with data format conversion.

Atomic operations take data from VGPRs and combine them arithmetically with data already in memory. Optionally, the value that was in memory before the operation took place can be returned to the shader.

All VM operations use a buffer resource constant (T#) which is a 128-bit value in SGPRs. This constant is sent to the texture cache when the instruction is executed. This constant defines the address and characteristics of the buffer in memory. Typically, these constants are fetched from memory using scalar memory reads prior to executing VM instructions, but these constants also can be generated within the shader.

8.1.1 Simplified Buffer Addressing

The equation in Figure 8.1 shows how the hardware calculates the memory address for a buffer access.

ADDR =	Base	+	baseOffset	+	Ioffset	+	Voffset	+	Stride	*	(Vindex	+	TID)
	T#		SGPR		Instr		VGPR		T#		VGPR		0..63

Voffset is ignored when instruction bit “OFFEN” == 0
Vindex is ignored when instructino bit “IDXEN” == 0
TID is a constant value (0..63) unique to each thread in the wave. It is ignored when resource bit ADD_TID_ENABLE == 0

Figure 8.1 Buffer Address Components

8.1.2 Buffer Instructions

Buffer instructions (MTBUF and MUBUF) allow the shader program to read from, and write to, linear buffers in memory. These operations can operate on data as small as one byte, and up to four Dwords per work-item. Atomic arithmetic operations are provided that can operate on the data values in memory and, optionally, return the value that was in memory before the arithmetic operation was performed.

Table 8.1 Buffer Instructions

Instruction	Description
MTBUF Instructions	
TBUFFER_LOAD_FORMAT_{x,xy,xyz,xyzw} TBUFFER_STORE_FORMAT_{x,xy,xyz,xyzw}	Read from, or write to, a typed buffer object. Also used for a vertex fetch.
MUBUF Instructions	
BUFFER_LOAD_FORMAT_{x,xy,xyz,xyzw} BUFFER_STORE_FORMAT_{x,xy,xyz,xyzw} BUFFER_LOAD_<**size**> BUFFER_STORE_<**size**>	Read to, or write from, an untyped buffer object. <size> = byte, ubyte, short, ushort, Dword, Dwordx2, Dwordx3, Dwordx4
BUFFER_ATOMIC_<op> BUFFER_ATOMIC_<op>_ x2	Buffer object atomic operation. Always globally coherent. Operates on 32-bit or 64-bit values (x2 = 64 bits).

Table 8.2 Microcode Formats

Field	Bit Size	Description
OP	3 7	MTBUF: Opcode for Typed buffer instructions. MUBUF: Opcode for Untyped buffer instructions.
VADDR	8	Address of VGPR to supply first component of address (offset or index). When both index and offset are used, index is in the first VGPR, offset in the second.
VDATA	8	Address of VGPR to supply first component of write data or receive first component of read-data.
SOFFSET	8	SGPR to supply unsigned byte offset. Must be an SGPR, M0, or inline constant.
SRSRC	5	Specifies which SGPR supplies T# (resource constant) in four or eight consecutive SGPRs. This field is missing the two LSBs of the SGPR address, since this address must be aligned to a multiple of four SGPRs.
DFMT	4	Data Format of data in memory buffer: 0 invalid — 8 10_10_10_2 1 8 — 9 2_10_10_10 2 16 — 10 8_8_8_8 3 8_8 — 11 32_32 4 32 — 12 16_16_16_16 5 16_16 — 13 32_32_32 6 10_11_11 — 14 32_32_32_32
NFMT	3	Numeric format of data in memory. 0 unorm 1 snorm 2 uscaled 3 sscaled 4 uint 5 sint 6 snorm_ogl 7 float
OFFSET	12	Unsigned byte offset.
OFFEN	1	1 = Supply an offset from VGPR (VADDR). 0 = Do not (offset = 0).

Table 8.2 Microcode Formats (Cont.)

Field	Bit Size	Description
`IDXEN`	1	1 = Supply an index from VGPR (VADDR). 0 = Do not (index = 0).
`ADDR64`	1	Address size is 64-bit. 0 = 32 bit offset is added to base-address from the resource. 1 = VGPR supplies 64-bit address, ignores size in resource. IDXEN and OFFEN must be zero in this mode. Stride (from resource) is ignored. Address = base(T#) + vgpr_addr[63:0] + instr_offset[11:0] + SOFFSET. No range checking. It is illegal to use ADDR64==1 and add_tid_enable==1 together.
`GLC`	1	Globally Coherent. Controls how reads and writes are handled by the L1 texture cache. READ GLC = 0 Reads can hit on the L1 and persist across wavefronts GLC = 1 Reads always miss the L1 and force fetch to L2. No L1 persistence across waves. WRITE GLC = 0 Writes miss the L1, write through to L2, and persist in L1 across wavefronts. GLC = 1 Writes miss the L1, write through to L2. No persistence across wavefronts. ATOMIC GLC = 0 Previous data value is not returned. No L1 persistence across wavefronts. GLC = 1 Previous data value is returned. No L1 persistence across wavefronts. Note: GLC means “return pre-op value” for atomics.
`SLC`	1	System Level Coherent. When set, accesses are forced to miss in level 2 texture cache and are coherent with system memory.
`TFE`	1	Texel Fail Enable for PRT (partially resident textures). When set to 1, fetch can return a NACK that causes a VGPR write into DST+1 (first GPR after all fetch-dest GPRs).
`LDS`	1	MUBUF-ONLY: 0 = Return read-data to VGPRs. 1 = Return read-data to LDS instead of VGPRs.

8.1.3 VGPR Usage

VGPRs supply address and write-data; also, they can be the destination for return data (the other option is LDS).

Address – Zero, one or two VGPRs are used, depending of the offset-enable (OFFEN) and index-enable (IDXEN) in the instruction word, as shown in Table 8.3. For 64-bit addresses (ADDR64=1), the LSBs are in VGPRn, and the MSBs are in VGPRn+1.

Table 8.3 Address VGPRs

IDXEN	OFFEN	VGPRn	VGPRn+1
0	0	*nothing*	
0	1	uint offset	
1	0	uint index	
1	1	uint index	uint offset

Write Data – *N* consecutive VGPRs, starting at VDATA. The data format specified in the instruction word (NFMT, DFMT for MTBUF, or encoded in the opcode field for MUBUF) determines how many Dwords to write.

Read Data – Same as writes. Data is returned to consecutive GPRs.

Read Data Format – Read data is always 32 bits, based on the data format in the instruction or resource. Float or normalized data is returned as floats; integer formats are returned as integers (signed or unsigned, same type as the memory storage format). Memory reads of data in memory that is 32 or 64 bits do not undergo any format conversion.

Atomics with Return – Data is read out of the VGPR(s) starting at VDATA to supply to the atomic operation. If the atomic returns a value to VGPRs, that data is returned to those same VGPRs starting at VDATA.

8.1.4 Buffer Data

The amount and type of data that is read or written is controlled by the following: data-format (`dfmt`), numeric-format (`nfmt`), destination-component-selects (`dst_sel`), and the opcode. `Dfmt` and `nfmt` can come from the resource, instruction fields, or the opcode itself. `Dst_sel` comes from the resource, but is ignored for many operations.

Table 8.4 Buffer Instructions

Instruction	Data Format	Num Format	DST SEL
`TBUFFER_LOAD_FORMAT_*`	instruction	instruction	identity
`TBUFFER_STORE_FORMAT_*`	instruction	instruction	identity
`BUFFER_LOAD_<type>`	derived	derived	identity
`BUFFER_STORE_<type>`	derived	derived	identity
`BUFFER_LOAD_FORMAT_*`	resource	resource	resource
`BUFFER_STORE_FORMAT_*`	resource	resource	resource
`BUFFER_ATOMIC_*`	derived	derived	identity

Instruction – The instruction's `dfmt` and `nfmt` fields are used instead of the resource's fields.

Data format derived – The data format is derived from the opcode and ignores the resource definition. For example, `buffer_load_ubyte` sets the data-format to `8` and number-format to `uint`.

NOTE: The resource's data format must not be INVALID; that format has special meaning (unbound resource), and for that case the data format is not replaced by the instruction's implied data format.

DST_SEL identity – Depending on the number of components in the data-format, this is: X000, XY00, XYZ0, or XYZW.

The MTBUF derives the data format from the instruction. The MUBUF `BUFFER_LOAD_FORMAT` and `BUFFER_STORE_FORMAT` instructions use `dst_sel` from the resource; other MUBUF instructions derive data-format from the instruction itself.

8.1.5 Buffer Addressing

Buffers are addressed with an index and an offset. The index points to a particular record of size stride bytes; the offset is the byte offset within the record, the stride always comes from the resource, while index and offset can come from a variety of sources as specified in the `BUFFER_*` instruction.

`BUFFER_*` instruction fields for addressing are:

- `inst_offen` – Boolean, get offset from vector GPR.
- `inst_idxen` – Boolean, get index from vector GPR.
- `inst_offset[11:0]` – Literal byte offset in the instruction.
- `inst_element_size` – Decoded from the instruction (1, 2, 4, 8, or 16 bytes).
  - For `UBUFFER_*` instructions, this comes from the opcode (from resource for `_format_` instructions).
  - For `TBUFFER_*` instructions, this is decoded from the format field.

Resource Constant fields used for buffer addressing are:

- `const_base[39:0]` – Base address of the resource.
- `const_stride[12:0]` – Stride of record in bytes (range 0 to 4 kB).
- `const_num_records[31:0]` – Number of records in the buffer of size stride (if stride<=1, this is bytes).
- `const_add_tid_enable` – Boolean, within wavefront add `thread_id` to the index.
- `const_swizzle_en` – Boolean, indicates if the surface is swizzled.
- `const_element_size` – Number of contiguous bytes of a record for a given index (2, 4, 8, or 16 bytes).
  - Must be >= maximum element size.
  - `const_stride` must be an integer multiple of `const_element_size`

  - only used for `const_swizzle_en` == true
- `const_index_stride` – number of contiguous indices for a single element (of `const_element_size`) before switching to next element (8, 16, 32, or 64). Only used for `const_swizzle_en` == true.

Address values from GPRs are:

- `sgpr_offset[31:0]` – an offset to add to the base address that comes from an SGPR.
- `vgpr_index[31:0]` – Index per thread from a vector GPR.
- `vgpr_offset[31:0]` – A byte offset per thread from a vector GPR.

Then, the base, index, and offset is calculated.

- base = `const_base + sgpr_offset`
- index = `(inst_idxen ? vgpr_index : 0) + (const_add_tid_enable ? thread_id[5:0] : 0)`
- offset = `(inst_offen ? vgpr_offset : 0) + inst_offset`

For coalescing, the hardware looks at the offset for stride==0; otherwise; it is always based on the index.

- raw = (`const_stride` <= 1)
- coalesce_enable = (raw | (`const_swizzle_en` & (all offsets equal) & (`const_element_size` == `inst_element_size`)))

If `coalesce_enable` is true, the hardware can coalesce across any set of contiguous indices (except TCP-required boundaries) for raw buffers. For swizzled buffers, it cannot coalesce across `const_index_stride` boundaries.

8.1.5.1 Linear Buffer Addressing

The linear buffer address calculation is an AOS-based calculation:

```
buffer_offset = index * const_stride + offset
```

8.1.5.2 Swizzled Buffer Addressing

When `const_swizzle_en` is set to 1, the address calculation below is performed, instead of the simple “linear buffer” calculation. This modified equation supports swizzled buffers.

`index_msb` = index / const_index_stride.

`index_lsb` = index % const_index_stride.

`offset_msb` = offset / const_element_size.

`offset_lsb` = offset % const_element_size.

```
buffer_offset = (index_msb * const_stride + offset_msb *
                const_element_size) * const_index_stride + index_lsb
                * const_element_size + offset_lsb
```

8.1.5.3 Range Checking

The hardware checks the range in the following way:

When the const_stride == 0, the const_num_records is the size of the buffer in bytes.

- If ((const_stride == 0) && (buffer_offset >= (const_num_records – sgpr_offset))
  - If op is a write or atomic, drop the write.
  - If op is a read or atomic, return 0.
- If (const_stride != 0 &&
  ((index >= const_num_records) || ((inst_idxen | const_add_tid_enable) && (offset >= const_stride)))
  - If op is a write or atomic, drop the write.
  - If op is a read or atomic, return 0.

Load/store-format-* instructions and atomics are either completely in-range or completely discarded. Load/store-dword_x2 and x4 are range-checked per component.

8.1.5.4 Base Add

The base address is added to construct the final byte address:

address = base + buffer_offset.

8.1.6 Alignment

For Dword or larger reads or writes, the two LSBs of the byte-address are ignored, thus forcing Dword alignment.

8.1.7 Buffer Resource

The buffer resource describes the location of a buffer in memory and the format of the data in the buffer. It is specified in four consecutive SGPRs (four aligned SGPRs) and sent to the texture cache with each buffer instruction.

Table 8.5 details the fields that make up the buffer resource descriptor.

Table 8.5 Buffer Resource Descriptor[1]

Bits	Size	Name	Description
47:0	48	Base address	Byte address. (In the Northern Islands environment, this was 40 bits.)
61:48	14	Stride	Bytes 0 to 16383
62	1	Cache swizzle	Buffer access. Optionally, swizzle texture cache TC L1 cache banks.
63	1	Swizzle enable	Swizzle AOS according to stride, index_stride, and element_size, else linear (stride * index + offset).
95:64	32	Num_records	In units of stride.
98:96	3	Dst_sel_x	Destination channel select: 0=0, 1=1, 4=R, 5=G, 6=B, 7=A
101:99	3	Dst_sel_y	
104:102	3	Dst_sel_z	
107:105	3	Dst_sel_w	
110:108	3	Num format	Numeric data type (float, int, …). See instruction encoding for values.
114:111	4	Data format	Number of fields and size of each field. See instruction encoding for values.
116:115	2	Element size	2, 4, 8, or 16 bytes (NI = 4). Used for swizzled buffer addressing.
118:117	2	Index stride	8, 16, 32, or 64 (NI = 16). Used for swizzled buffer addressing.
119	1	Add tid enable	Add thread ID to the index for to calculate the address.
120	1	ATC	0: resource is in GPUVM memory space. 1 = resource is in ATC memory space.
121	1	Hash enable	1 = buffer addresses are hashed for better cache performance.
122	1	Heap	1 = buffer is a heap. out-of-range if offset = 0 or >= num_records.
125:123	*3*	MTYPE	Memory type - controls cache behavior.
127:126	*2*	Type	value == 0 for buffer. Overlaps upper two bits of four-bit TYPE field in 128-bit T# resource.

1.A resource set to all zeros acts as an unbound texture or buffer (return 0,0,0,0). Buffer Size (in bytes) = (stride==0) ? num_elements : stride * num_elements.

8.1.8 Memory Buffer Load to LDS

The MUBUF instruction format allows reading data from a memory buffer directly into LDS without passing through VGPRs. This is supported for the following subset of MUBUF instructions.

- `BUFFER_LOAD_{ubyte, sbyte, ushort, sshort, dword, format_x}`.
- It is illegal to set the instruction's TFE bit for loads to LDS.

LDS_offset = 16-bit unsigned byte offset from M0[15:0].

Mem_offset = 32-bit unsigned byte offset from an SGPR (the SOFFSET SGPR).

idx_vgpr = index value from a VGPR (located at VADDR). (Zero if idxen=0.)

off_vgpr = offset value from a VGPR (located at VADDR or VADDR+1). (Zero if offen=0.)

Figure 8.2 shows the components of the LDS and memory address calculation.

LDS_ADDR =	LDSbase +	LDS_offset +	(TIDinWave	* 4)
	Alloc	M0[15:0]	0..63	bytes-per-dword

MEM_ADDR =	Base +	mem_offset +	inst_offset +	off_vgpr +	stride *	(idx_vgpr +	TIDinWave)
	T#	SGPR (soffset)	Instr.	VGPR	T#	VGPR	0..63

Figure 8.2 Components of Addresses for LDS and Memory

TIDinWave is only added if the resource (T#) has the ADD_TID_ENABLE field set to 1. LDS always adds it.

The MEM_ADDR M# is in the VDATA field; it specifies M0.

8.1.8.1 Clamping Rules

Memory address clamping follows the same rules as any other buffer fetch.

LDS address clamping: the return data must not be written outside the LDS space allocated to this wave.

- Set the active-mask to limit buffer reads to those threads that return data to a legal LDS location.
- The LDSbase (alloc) is in units of 32 Dwords, as is LDSsize.
- M0[15:0] is in bytes.

8.1.9 GLC Bit Explained

The GLC bit means different things for loads, stores, and atomic ops.

<u>GLC Meaning for Loads</u>

- For GLC==0
  - The load can read data from the GPU L1.
  - Typically, all loads (except load-acquire) use GLC==0.
- For GLC==1
  - The load intentionally misses the GPU L1 and reads from L2.

    If there was a line in the GPU L1 that matched, it is invalidated; L2 is re-read.
  - NOTE: L2 is not re-read for every work-item in the same wave-front for a single load instruction. For example:

    b=uav[N+tid] // assume this is a byte read w/ glc==1 and N is aligned to 64B

In the above op, the first Tid of the wavefront brings in the line from L2 or beyond, and all 63 of the other Tids read from same 64 B cache line in the L1.

GLC Meaning for Stores

- For GLC==0

  This causes a write-combine across work-items of the wavefront store op; dirtied lines are written to the L2 automatically.

  – If the store operation dirtied all bytes of the 64 B line, it is left clean and valid in the L1; subsequent accesses to the cache are allowed to hit on this cache line.
  – Else do not leave write-combined lines in L1.
- For GLC==1

  Same as GLC==0, except the write-combined lines are not left in the line, even if all bytes are dirtied.

Atomic

- For GLC == 0

  No return data (this is “write-only” atomic op).
- For GLC == 1

  Returns previous value in memory (before the atomic operation).

8.2 Vector Memory (VM) Image Instructions

Vector Memory (VM) operations transfer data between the VGPRs and memory through the texture cache (TC). Vector means the transfer of one or more pieces of data uniquely for every work-item in the wavefront. This is in contrast to scalar memory reads, which transfer only one value that is shared by all work-items in the wavefront.

Examples of image objects are texture maps and typed surfaces.

Image objects are accessed using from one to four dimensional addresses; they are composed of homogenous data of one to four elements. These image objects are read from, or written to, using `IMAGE_*` or `SAMPLE_*` instructions, all of which use the MIMG instruction format. `IMAGE_LOAD` instructions read an element from the image buffer directly into VGPRS, and `SAMPLE` instructions use sampler constants (S#) and apply filtering to the data after it is read. `IMAGE_ATOMIC` instructions combine data from VGPRs with data already in memory, and optionally return the value that was in memory before the operation.

All VM operations use an image resource constant (T#) that is a 128- or 256-bit value in SGPRs. This constant is sent to the texture cache when the instruction is executed. This constant defines the address, data format, and characteristics of the surface in memory. Some image instructions also use a sampler constant that is a 128-bit constant in SGPRs. Typically, these constants are fetched from memory using scalar memory reads prior to executing VM instructions, but these constants can also be generated within the shader.

Texture fetch instructions have a data mask (DMASK) field. DMASK specifies how many data components it receives. If DMASK is less than the number of components in the texture, the texture unit only sends DMASK components, starting with R, then G, B, and A. if DMASK specifies more than the texture format specifies, the shader receives zero for the missing components.

8.2.1 Image Instructions

This section describes the image instruction set, and the microcode fields available to those instructions.

Table 8.6 Image Instructions

MIMG Instruction	Description
SAMPLE_*	Read and filter data from a image object.
IMAGE_LOAD_<op>	Read data from an image object using one of the following: image_load, image_load_mip, image_load_{pck, pck_sgn, mip_pck, mip_pck_sgn}.
IMAGE_STORE IMAGE_STORE_MIP	Store data to an image object. Store data to a specific mipmap level.
IMAGE_ATOMIC_<op>	Image atomic operation, which is one of the following: swap, cmpswap, add, sub, rsub, {u,s}{min,max}, and, or, xor, inc, dec, fcmpswap, fmin, fmax.

Table 8.7 Instruction Fields

Instruction	Bit Size	Description
OP	7	Opcode.
VADDR	8	Address of VGPR to supply first component of address.
VDATA	8	Address of VGPR to supply first component of write data or receive first component of read-data.
SSAMP	5	SGPR to supply S# (sampler constant) in four consecutive SGPRs. Missing two LSBs of SGPR-address since must be aligned to a multiple of four SGPRs.
SRSRC	5	SGPR to supply T# (resource constant) in four or eight consecutive SGPRs. Missing two LSBs of SGPR-address since must be aligned to a multiple of four SGPRs.
UNRM	1	Force address to be un-normalized regardless of T#. Must be set to 1 for image stores and atomics.
R128	1	Texture resource size: 1 = 128 bits, 0 = 256 bits.
DA	1	Shader declared an array resource to be used with this fetch. When 1, the shader provides an array-index with the instruction. When 0, no array index is provided.

Table 8.7 Instruction Fields (Cont.)

Instruction	Bit Size	Description
DMASK	4	Data VGPR enable mask: one to four consecutive VGPRs. Reads: defines which components are returned. 0 = red, 1 = green, 2 = blue, 3 = alpha Writes: defines which components are written with data from VGPRs (missing components get 0). Enabled components come from consecutive VGPRs. For example: DMASK=1001 : Red is in VGPRn and alpha in VGPRn+1. If DMASK==0, the TA overrides the data format to “invalid,” and forces dst_sels to return 0.
GLC	1	Globally Coherent. Controls how reads and writes are handled by the L1 texture cache. READ GLC = 0 Reads can hit on the L1 and persist across waves. GLC = 1 Reads always miss the L1 and force fetch to L2. No L1 persistence across waves. WRITE GLC = 0 Writes miss the L1, write through to L2, and persist in L1 across wavefronts. GLC = 1 Writes miss the L1, write through to L2. No persistence across wavefronts. ATOMIC GLC = 0 Previous data value is not returned. No L1 persistence across wavefronts. GLC = 1 Previous data value is returned. No L1 persistence across wavefronts.
SLC	1	System Level Coherent. When set, accesses are forced to miss in level 2 texture cache and are coherent with system memory.
TFE	1	Texel Fail Enable for PRT (partially resident textures). When set, a fetch can return a NACK, which causes a VGPR write into DST+1 (first GPR after all fetch-dest GPRs).
LWE	1	Force data to be un-normalized, regardless of T#.

8.2.2 Image Opcodes with No Sampler

For image opcodes with no sampler, all VGPR address values are taken as uint. For cubemaps, face_id = slice * 8 + face.

Table 8.8 shows the contents of address VGPRs for the various image opcodes.

Table 8.8 Image Opcodes with No Sampler

Image Opcode (Resource w/o Sampler)	Acnt	dim	VGPRn	VGPRn+1	VGPRn+2	VGPRn+3
get_resinfo	0	Any	mipid			
load / store / atomics	0	1D	x			
	1	1D Array	x	slice		
	1	2D	x	y		
	2	2D MSAA	x	y	fragid	
	2	2D Array	x	y	slice	
	3	2D Array MSAA	x	y	slice	fragid
	2	3D	x	y	z	
	2	Cube	x	y	face_id	
load_mip / store_mip	1	1D	x	mipid		
	2	1D Array	x	slice	mipid	
	2	2D	x	y	mipid	
	3	2D Array	x	y	slice	mipid
	3	3D	x	y	z	mipid
	3	Cube	x	y	face_id	mipid

8.2.3 Image Opcodes with Sampler

For image opcodes with a sampler, all VGPR address values are taken as float. For cubemaps, face_id = slice * 8 + face.

Certain sample and gather opcodes require additional values from VGPRs beyond what is shown in Table 8.9. These values are: offset, bias, z-compare, and gradients. See Section 8.2.4, “VGPR Usage,” page 8-16, for details.

Table 8.9Image Opcodes with Sampler

Image Opcode (w/ Sampler)	Acnt	dim	VGPRn	VGPRn+1	VGPRn+2	VGPRn+3
sample[1]	0	1D	x			
	1	1D Array	x	slice		
	1	2D	x	y		
	2	2D interlaced	x	y	field	
	2	2D Array	x	y	slice	
	2	3D	x	y	z	
	2	Cube	x	y	face_id	

Table 8.9Image Opcodes with Sampler (Cont.)

Image Opcode (w/ Sampler)	Acnt	dim	VGPRn	VGPRn+1	VGPRn+2	VGPRn+3
sample_l[2]	1	1D	x	lod		
	2	1D Array	x	slice	lod	
	2	2D	x	y	lod	
	3	2D interlaced	x	y	field	lod
	3	2D Array	x	y	slice	lod
	3	3D	x	y	z	lod
	3	Cube	x	y	face_id	lod
sample_cl[3]	1	1D	x	clamp		
	2	1D Array	x	slice	clamp	
	2	2D	x	y	clamp	
	3	2D interlaced	x	y	field	clamp
	3	2D Array	x	y	slice	clamp
	3	3D	x	y	z	clamp
	3	Cube	x	y	face_id	clamp
gather4[4]	1	2D	x	y		
	2	2D interlaced	x	y	field	
	2	2D Array	x	y	slice	
	2	Cube	x	y	face_id	
gather4_l	2	2D	x	y	lod	
	3	2D interlaced	x	y	field	lod
	3	2D Array	x	y	slice	lod
	3	Cube	x	y	face_id	lod
gather4_cl	2	2D	x	y	clamp	
	3	2D interlaced	x	y	field	clamp
	3	2D Array	x	y	slice	clamp
	3	Cube	x	y	face_id	clamp

1. sample includes sample, sample_d, sample_b, sample_lz, sample_c, sample_c_d, sample_c_b, sample_c_lz, and getlod
2. sample_l includes sample_l and sample_c_l.
3. sample_cl includes sample_cl, sample_d_cl, sample_b_cl, sample_c_cl, sample_c_d_cl, and sample_c_b_cl.
4. gather4 includes gather4, gather4_lz, gather4_c, and gather4_c_lz.

Table 8.10 lists and briefly describes the legal suffixes for image instructions.

Table 8.10Sample Instruction Suffix Key

Suffix	Meaning	Extra Addresses	Description
_L	LOD	-	LOD is used instead of TA computed LOD.
_B	LOD BIAS	1: lod bias	Add this BIAS to the LOD TA computes.
_CL	LOD CLAMP	-	Clamp the LOD to be no larger than this value.
_D	Derivative	2,4 or 6: slopes	Send dx/dv, dx/dy, etc. slopes to TA for it to used in LOD computation.
_CD	Coarse Derivative		Send dx/dv, dx/dy, etc. slopes to TA for it to used in LOD computation.
_LZ	Level 0	-	Force use of MIP level 0.
_C	PCF	1: z-comp	Percentage closer filtering.
_O	Offset	1: offsets	Send X, Y, Z integer offsets (packed into 1 Dword) to offset XYZ address.

8.2.4 VGPR Usage

- Address: The address consists of up to four parts:

  { offset } { bias } { z-compare } { derivative } { body }

These are all packed into consecutive VGPRs.

- Offset: `SAMPLE*_O_*`, `GATHER*_O_*`
  One Dword of offset_xyz. The offsets are six-bit signed integers: X=[5:0], Y=[13:8], and Z=[21:16].
- Bias: `SAMPLE*_B_*`, `GATHER*_B_*`. One Dword float.
- Z-compare: `SAMPLE*_C_*`, `GATHER*_C_*`. One Dword.
- Derivatives (sample_d, sample_cd): 2, 4, or 6 Dwords, packed one Dword per derivative as:

Image Dim	VGPR N	N+1	N+2	N+3	N+4	N+5
1D	DX/DH	DX/DV	-	-	-	-
2D	dx/dh	DY/DH	DX/DV	DY/DV	-	--
3D	dx/dh	DY/DH	DZ/DH	DX/DV	DY/DV	DZ/DV

- Body: One to four Dwords, as defined by Table 8.9.
  Address components are X,Y,Z,W with X in VGPR_M, Y in VGPR_M+1, etc.
- Data: Written from, or returned to, one to four consecutive VGPRs. The amount of data read or written is determined by the DMASK field of the instruction.
- Reads: DMASK specifies which elements of the resource are returned to consecutive VGPRs. The texture system reads data from memory and based on the data format expands it to a canonical RGBA form, filling in zero or one for missing components. Then, DMASK is applied, and only those components selected are returned to the shader.

- Writes: When writing an image object, it is only possible to write an entire element (all components), not just individual components. The components come from consecutive VGPRs, and the texture system fills in the value zero for any missing components of the image's data format; it ignores any values that are not part of the stored data format. For example, if the DMASK=1001, the shader sends Red from VGPR_N, and Alpha from VGPR_N+1, to the texture unit. If the image object is RGB, the texel is overwritten with Red from the VGPR_N, Green and Blue set to zero, and Alpha from the shader ignored.
- Atomics: Image atomic operations are supported only on 32- and 64-bit-per-pixel surfaces. The surface data format is specified in the resource constant. Atomic operations treat the element as a single component of 32- or 64-bits. For atomic operations, DMASK is set to the number of VGPRs (Dwords) to send to the texture unit.

  DMASK legal values for atomic image operations: no other values of DMASK are legal.

  0x1 = 32-bit atomics except cmpswap.

  0x3 = 32-bit atomic cmpswap.

  0x3 = 64-bit atomics except cmpswap.

  0xf = 64-bit atomic cmpswap.
- Atomics with Return: Data is read out of the VGPR(s), starting at VDATA, to supply to the atomic operation. If the atomic returns a value to VGPRs, that data is returned to those same VGPRs starting at VDATA.

8.2.5 Image Resource

The image resource (also referred to as T#) defines the location of the image buffer in memory, its dimensions, tiling, and data format. These resources are stored in four or eight consecutive SGPRs and are read by MIMG instructions.

Table 8.11 Image Resource Definition

Bits	Size	Name	Comments
	128-bit Resource: 1D-tex, 2d-tex, 2d-msaa (multi-sample auto-aliasing)		
39:0	40	base address	256-byte aligned. Also used for fmask-ptr.
51:40	12	min lod	4.8 (four uint bits, eight fraction bits) format.
57:52	6	data format	Number of comps, number of bits/comp.
61:58	4	num format	Numeric format.
63:62	2	MTYPE[1:0]	Memory type - controls cache behavior.
77:64	14	width	
91:78	14	height	
94:92	3	perf modulation	Scales sampler's perf_z, perf_mip, aniso_bias, lod_bias_sec.
95	1	interlaced	

Table 8.11 Image Resource Definition (Cont.)

Bits	Size	Name	Comments
98:96	3	dst_sel_x	0 = 0, 1 = 1, 4 = R, 5 = G, 6 = B, 7 = A.
101:99	3	dst_sel_y	
104:102	3	dst_sel_z	
107:105	3	dst_sel_w	
111:108	4	base level	
115:112	4	last level	For msaa, holds number of samples
120:116	5	Tiling index	Lookuptable: 32 x 16 bank_width[2], bank_height[2], num_banks[2], tile_split[2], macro_tile_aspect[2], micro_tile_mode[2], array_mode[4].
121	1	pow2pad	Memory footprint is padded to pow2 dimensions
122	1	MTYPE[2]	Bit 2 of the MTYPE field.
123	1	ATC	0 = image is in GPUVM memory; 1 = image is in ATC memory.
127:124	4	type	0 = buf, 8 = 1d, 9 = 2d, 10 = 3d, 11 = cube, 12 = 1d-array, 13 = 2d-array, 14 = 2d-msaa, 15 = 2d-msaa-array. 1-7 are reserved.
	256-bit Resource: 1d-array, 2d-array, 3d, cubemap, MSAA		
140:128	13	depth	
154:141	14	pitch	In texel units.
159:155	5	*unused*	
172:160	13	base array	
185:173	13	last array	
191:186	6	*unused*	
203:192	12	min_lod_warn	feedback trigger for lod
255:204	52	*unused*	

All image resource view descriptors (T#'s) are written by the driver as 256 bits. It is permissible to use only the first 128 bits when a simple 1D or 2D (not an array) is bound. This is specified in the MIMG `R128` instruction field.

The MIMG-format instructions have a DeclareArray (DA) bit that reflects whether the shader was expecting an array-texture or simple texture to be bound. When DA is zero, the hardware does not send an array index to the texture cache. If the texture map was indexed, the hardware supplies an index value of zero. Indices sent for non-indexed texture maps are ignored.

8.2.6 Sampler Resource

The sampler resource (also referred to as S#) defines what operations to perform on texture map data read by "sample" instructions. These are primarily address clamping and filter options. Sampler resources are defined in four consecutive SGPRs and are supplied to the texture cache with every sample instruction.

Table 8.12 Sampler Resource Definition

Bits	Size	Name	Description
2:0	3	clamp x	Clamp/wrap mode.
5:3	3	clamp y	
8:6	3	clamp z	
11:9	3	max aniso ratio	
14:12	3	depth compare func	
15	1	force unnormalized	Force address cords to be unorm.
18:16	3	aniso threshold	
19	1	mc coord trunc	
20	1	force degamma	
26:21	6	aniso bias	u1.5.
27	1	trunc coord	
28	1	disable cube wrap	
30:29	2	filter_mode	Normal lerp, min, or max filter.
31	*1*	*unused*	
43:32	12	min lod	u4.8.
55:44	12	max lod	u4.8.
59:56	4	perf_mip	
63:60	4	perf z	
77:64	14	lod bias	s5.8.
83:78	6	lod bias sec	s1.4.
85:84	2	xy mag filter	Magnification filter.
87:86	2	xy min filter	Minification filter.
89:88	2	z filter	
91:90	2	mip filter	
92	1	mip_point_preclamp	When mipfilter = point, add 0.5 before clamping.
93	1	disable_lsb_ceil	Disable ceiling logic in filter (rounds up).
95:94	*4*	*unused*	
107:96	12	border color ptr	
125:108	*18*	*unused*	
127:126	2	border color type	Opaque-black, transparent-black, white, use border color ptr.

8.2.7 Data Formats

Data formats 0-15 are available to buffer resources, and all formats are available to image formats. Table 8.13 details all the data formats that can be used by image and buffer resources.

Table 8.13 Data and Image Formats

data_format								shader num_format					
value	**encode**	**buffer r**	**buffer w**	**image r**	**image w**	**MRT (CB)**		**value**	**encode**	**buffer r**	**buffer w**	**image r**	**image w**
0	invalid	yes	yes	yes	yes	yes		0	unorm	yes	yes	yes	yes
1	8	yes	yes	yes	yes	yes		1	snorm	yes	yes	yes	yes
2	16	yes	yes	yes	yes	yes		2	uscaled	yes	no	yes	no
3	8_8	yes	yes	yes	yes	yes		3	sscaled	yes	no	yes	no
4	32	yes	yes	yes	yes	yes		4	uint	yes	yes	yes	yes
5	16_16	yes	yes	yes	yes	yes		5	sint	yes	yes	yes	yes
6	10_11_11	yes	yes	yes	yes	yes		6	snorm_nz	yes	no	yes	no
7	11_11_10	yes	yes	yes	yes	yes		7	float	yes	yes	yes	yes
8	10_10_10_2	yes	yes	yes	yes	yes		8	reserved				
9	2_10_10_10	yes	yes	yes	yes	yes		9	srgb	no	no	yes	no
10	8_8_8_8	yes	yes	yes	yes	yes		10	ubnorm	no	no	yes	no
11	32_32	yes	yes	yes	yes	yes		11	ubnorm_nz	no	no	yes	no
12	16_16_16_16	yes	yes	yes	yes	yes		12	ubint	no	no	yes	no
13	32_32_32	yes	yes	yes	no	no		13	ubscaled	no	no	yes	no
14	32_32_32_32	yes	yes	yes	yes	yes							
15	reserved												
16	5_6_5	no	no	yes	yes	yes							
17	1_5_5_5	no	no	yes	yes	yes							
18	5_5_5_1	no	no	yes	yes	yes							
19	4_4_4_4	no	no	yes	yes	yes							
20	8_24	no	no	yes	no	yes							
21	24_8	no	no	yes	no	yes							
22	X24_8_32	no	no	yes	no	yes							
23-31	reserved												
32	GB_GR	no	no	yes	no	no							
33	BG_RG	no	no	yes	no	no							
34	5_9_9_9	no	no	yes	no	no							
35	BC1	no	no	yes	no	no							
36	BC2	no	no	yes	no	no							
37	BC3	no	no	yes	no	no							
38	BC4	no	no	yes	no	no							
39	BC5	no	no	yes	no	no							

Table 8.13 Data and Image Formats (Cont.)

data_format								shader num_format					
value	**encode**	**buffer r**	**buffer w**	**image r**	**image w**	**MRT (CB)**		**value**	**encode**	**buffer r**	**buffer w**	**image r**	**image w**
40	BC6	no	no	yes	no	no							
41	BC7	no	no	yes	no	no							
42-46	reserved												
47	FMASK_8_1	no	no	yes	yes	no		8-bits FMASK, 1 fragment per sample					
48	FMASK_8_2	no	no	yes	yes	no		8-bits FMASK, 2 fragments per sample					
49	FMASK_8_4	no	no	yes	yes	no		8-bits FMASK, 4 fragment per sample					
50	FMASK_16_1	no	no	yes	yes	no		16-bits FMASK, 1 fragment per sample					
51	FMASK_16_2	no	no	yes	yes	no		16-bits FMASK, 2 fragments per sample					
52	FMASK_32_2	no	no	yes	yes	no		32-bits FMASK, 2 fragments per sample					
53	FMASK_32_4	no	no	yes	yes	no		32-bits FMASK, 4 fragments per sample					
54	FMASK_32_8	no	no	yes	yes	no		32-bits FMASK, 8 fragments per sample					
55	FMASK_64_4	no	no	yes	yes	no		64-bits FMASK, 4 fragments per sample					
56	FMASK_64_8	no	no	yes	yes	no		64-bits FMASK, 8 fragments per sample					
57	4_4	no	no	yes	no	no							
58	6_5_5	no	no	yes	no	no							
59	1	no	no	yes	no	no							
60	1_REVERSED	no	no	yes	no	no							
61	32_AS_8	no	no	yes	no	no							
62	32_AS_8_8	no	no	yes	no	no							
63	32_AS_32_32_32_32	no	no	yes	no	no							

8.2.8 Vector Memory Instruction Data Dependencies

When a VM instruction is issued, the address is immediately read out of VGPRs and sent to the texture cache. Any texture or buffer resources and samplers are also sent immediately. However, write-data is not immediately sent to the texture cache.

The shader developer's responsibility to avoid two data hazards associated with VMEM instructions include:

- Wait for VMEM read instruction completion before reading data fetched from the TC (VMCNT).
- Wait for the TC to consume write data before overwriting the VGPRs holding the write data (EXPCNT).

This is explained in Section 4.4, "Data Dependency Resolution," page 4-2.

Chapter 9
Flat Memory Instructions

Flat Memory instructions read, or write, one piece of data into, or out of, VGPRs; they do this separately for each work-item in a wavefront. Unlike buffer or image instructions, Flat instructions do not use a resource constant to define the base address of a surface. Instead, Flat instructions use a single flat address from the VGPR; this addresses memory as a single flat memory space. This memory space includes video memory, system memory, LDS memory, and scratch (private) memory. It does not include GDS memory. Parts of the flat memory space may not map to any real memory, and accessing these regions generates a memory-violation error. The determination of the memory space to which an address maps is controlled by a set of "memory aperture" base and size registers.

9.1 Flat Memory Instructions

Flat memory instructions let the kernel read or write data in memory, or perform atomic operations on data already in memory. These operations occur through the texture L2 cache. The instruction declares which VGPR holds the address (either 32- or 64-bit, depending on the memory configuration), the VGPR which sends and the VGPR which receives data. Flat instructions also use M0 as described in Table 9.1.

Table 9.1 Flat Microcode Formats

Field	Bit Size	Description			
OP	7	FLAT_LOAD_UBYTE	FLAT_STORE_BYTE	FLAT_ATOMIC_SWAP	FLAT_ATOMIC_SWAP_X2
		FLAT_LOAD_SBYTE		FLAT_ATOMIC_CMPSWAP	FLAT_ATOMIC_CMPSWAP_X2
		FLAT_LOAD_USHORT	FLAT_STORE_SHORT	FLAT_ATOMIC_ADD	FLAT_ATOMIC_ADD_X2
		FLAT_LOAD_SSHORT		FLAT_ATOMIC_SUB	FLAT_ATOMIC_SUB_X2
		FLAT_LOAD_DWORD	FLAT_STORE_DWORD	FLAT_ATOMIC_SMIN	FLAT_ATOMIC_SMIN_X2
		FLAT_LOAD_DWORDX2	FLAT_STORE_DWORDX2	FLAT_ATOMIC_UMIN	FLAT_ATOMIC_UMIN_X2
		FLAT_LOAD_DWORDX3	FLAT_STORE_DWORDX3	FLAT_ATOMIC_SMAX	FLAT_ATOMIC_SMAX_X2
		FLAT_LOAD_DWORDX4	FLAT_STORE_DWORDX4	FLAT_ATOMIC_UMAX	FLAT_ATOMIC_UMAX_X2
				FLAT_ATOMIC_AND	FLAT_ATOMIC_AND_X2
				FLAT_ATOMIC_OR	FLAT_ATOMIC_OR_X2
				FLAT_ATOMIC_XOR	FLAT_ATOMIC_XOR_X2
				FLAT_ATOMIC_INC	FLAT_ATOMIC_INC_X2
				FLAT_ATOMIC_DEC	FLAT_ATOMIC_DEC_X2
				FLAT_ATOMIC_FCMPSWAP	FLAT_ATOMIC_FCMPSWAP_X2
				FLAT_ATOMIC_FMIN	FLAT_ATOMIC_FMIN_X2
				FLAT_ATOMIC_FMAX	FLAT_ATOMIC_FMAX_X2
ADDR	8	VGPR which holds the address. For 64-bit addresses, ADDR has the LSBs, and ADDR+1 has the MSBs.			
DATA	8	VGPR which holds the first Dword of data. Instructions can use 0-4 Dwords.			
VDST	8	VGPR destination for data returned to the kernel, either from LOADs or Atomics with GLC=1 (return pre-op value).			
SLC	1	System Level Coherent. Used in conjunction with GLC and MTYPE to determine cache policies.			
GLC	1	Global Level Coherent. For Atomics, GLC: 1 means return pre-op value, 0 means do not return pre-op value.			
TFE	1	Texel Fail Enable for PRT (Partially Resident Textures). When set, fetch can return a NACK, which causes a VGPR write into DST+1 (first GPR after all fetch-dest gprs).			
(M0)	32	Implied use of M0. M0[16:0] contains the byte-size of the LDS segment. This is used to clamp the final address.			

9.2 Instructions

The FLAT instruction set is nearly identical to the Buffer instruction set, but without the FORMAT reads and writes. Unlike Buffer instructions, FLAT instructions cannot return data directly to LDS, but only to VGPRs.

FLAT instructions do not use a resource constant (V#) or sampler (S#); however, they do require a special SGPR-pair to hold scratch-space information in case any threads' address resolves to scratch space. See Section 9.6, "Scratch Space (Private)," page 9-4.

Internally, FLAT instruction are executed as both an LDS and a Buffer instruction; so, they increment both VM_CNT and LGKM_CNT and are not considered done until both have been decremented. There is no way beforehand to determine whether a FLAT instruction uses only LDS or TA memory space.

9.2.1 Ordering

Flat instructions can complete out of order with each other. If one flat instruction finds all of its data in Texture cache, and the next finds all of its data in LDS, the second instruction might complete first. If the two fetches return data to the same VGPR, the result are unknown.

9.2.2 Important Timing Consideration

Since the data for a FLAT load can come from either LDS or the texture cache, and because these units have different latencies, there is a potential race condition with respect to the VM_CNT and LGKM_CNT counters. Because of this, the only sensible S_WAITCNT value to use after FLAT instructions is zero.

9.3 Addressing

FLAT instructions support both 64- and 32-bit addressing. The address size is set using a mode register (PTR32), and a local copy of the value is stored per wave.

The addresses for the aperture check differ in 32- and 64-bit mode; however, this is not covered here.

64-bit addresses are stored with the LSBs in the VGPR at ADDR, and the MSBs in the VGPR at ADDR+1.

For scratch space, the TA takes the address from the VGPR and does the following.

```
Address = VGPR[addr] +  TID_in_wave * Size
    + SH_HIDDEN_PRIVATE_BASE_VMID
    – "private aperture base" (in SH_MEM_BASES)
    + offset (from flat_scratch)
```

9.4 Memory Error Checking

Both TA and LDS can report that an error occurred due to a bad address. This can occur for the following reasons:

- invalid address (outside any aperture)
- write to read-only surface
- misaligned data
- out-of-range address:
  - LDS access with an address outside the range:
    [0, MIN(M0, LDS_SIZE)-1]

– Scratch access with an address outside the range: [0, *scratch-size -1*]
– Heap address outside of legal range

The policy for threads with bad addresses is: writes outside this range do not write a value, and reads return zero.

Addressing errors from either LDS or TA are returned on their respective "instruction done" busses as `MEM_VIOL`. This sets the wave's `MEM_VIOL` TrapStatus bit and causes an exception (trap) if the corresponding `EXCPEN` bit is set.

9.5 Data

FLAT instructions can use zero to four consecutive Dwords of data in VGPRs and/or memory. The `DATA` field determines which VGPR(s) supply source data (if any), and the VDST VGPRs hold return data (if any). No data-format conversion is done.

9.6 Scratch Space (Private)

Scratch (thread-private memory) is an area of memory defined by the aperture registers. When an address falls in scratch space, additional address computation is automatically performed by the hardware. The kernel must provide additional information for this computation to occur in the form of the `FLAT_SCRATCH` register.

The wavefront must supply the scratch size and offset (for space allocated to this wave) with every FLAT request. This is stored in a fixed SGPR location (FLAT_SCRATCH): N_SGPRS-3 and N_SGPRS-4, as:

{ 8'h0, Offset[31:8], 13'h0, Size[18:0] }

◊ Offset is in units of 256-bytes (hence the missing eight LSBs)
◊ Size is the per-thread scratch size, in bytes.

These SGPRs are automatically sent with every FLAT request.

It is the responsibility of the kernel to initialize this SGPR-pair.

Note that in FSA32, only SIZE[15:0] are considered ([18:16] are ignored).

Chapter 10
Data Share Operations

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

10.1 Overview

Figure 10.1 shows the conceptual framework of the LDS is integration into the memory of AMD GPUs using OpenCL.

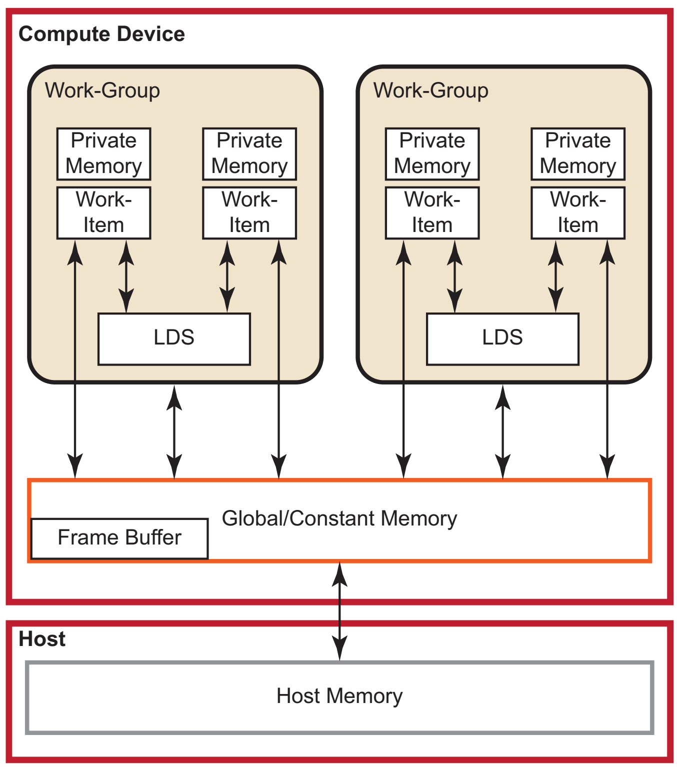


Figure 10.1 High-Level Memory Configuration

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

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

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

10.2 Dataflow in Memory Hierarchy

Figure 10.2 is a conceptual diagram of the dataflow within the memory structure.

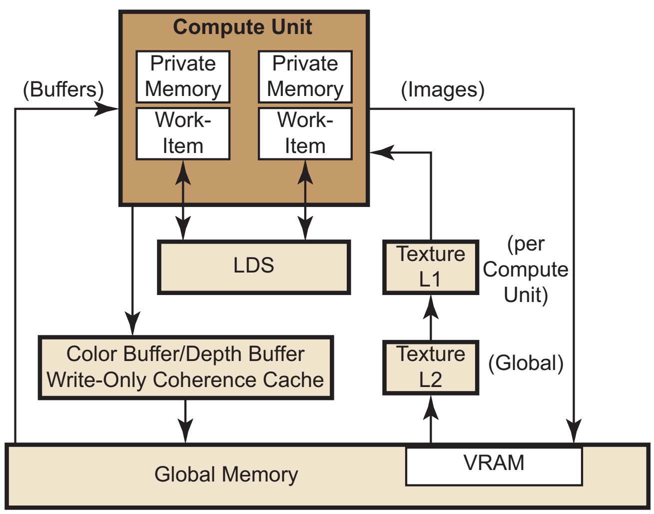


Figure 10.2 Memory Hierarchy Dataflow

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

LDS atomics are performed in the LDS hardware. (Thus, although ALUs are not directly used for these operations, latency is incurred by the LDS executing this function.) If the algorithm does not require write-to-read reuse (the data is read only), it usually is better to use the image dataflow (see right side of Figure 10.2) because of the cache hierarchy.

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

Buffers and images are written through the texture L2 cache, but this is flushed immediately after an image write.

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

Global memory can be in the high-speed GPU memory (VRAM) or in the host memory, which is accessed by the PCIe bus. A work-item can access global memory either as a buffer or a memory object. Buffer objects are generally read and written directly by the work-items. Data is accessed through the L2 and L1 data caches on the GPU. This limited form of caching provides read coalescing among work-items in a wavefront. Similarly, writes are executed through the texture L2 cache.

Global atomic operations are executed through the texture L2 cache. Atomic instructions that return a value to the kernel are handled similarly to fetch instructions: the kernel must use `S_WAITCNT` to ensure the results have been written to the destination GPR before using the data.

10.3 LDS Access

The LDS is accessed in one of three ways:

- Direct Read
- Parameter Read
- Indexed or Atomic

The following subsections describe these methods.

10.3.1 LDS Direct Reads

Direct reads are only available in LDS, not in GDS.

LDS Direct reads occur in vector ALU (VALU) instructions and allow the LDS to supply a single DWORD value which is broadcast to all threads in the wavefront and is used as the SRC0 input to the ALU operations. A VALU instruction indicates that input is to be supplied by LDS by using the `LDS_DIRECT` for the SRC0 field.

The LDS address and data-type of the data to be read from LDS comes from the M0 register:

`LDS_addr = M0[15:0]` (byte address and must be dword aligned)

`DataType = M0[18:16]`

0 – unsigned byte

1 – unsigned short

2 – dword

3 – unused

4 – signed byte

5 – signed short

10.3.2 LDS Parameter Reads

Parameter reads are only available in LDS, not in GDS.

Pixel shaders use LDS to read vertex parameter values; the pixel shader then interpolates them to find the per-pixel parameter values. LDS parameter reads occur when the following opcodes are used.

- `V_INTERP_P1_F32` D = P10 * S + P0 Parameter interpolation, first step.
- `V_INTERP_P2_F32` D = P20 * S + D Parameter interpolation, second step.
- `V_INTERP_MOV_F32` D = {P10,P20,P0}[S] Parameter load.

The typical parameter interpolation operations involves reading three parameters: P0, P10, and P20, and using the two barycentric coordinates, I and J, to determine the final per-pixel value:

Final value = P0 + P10 * I + P20 * J

Parameter interpolation instructions indicate the parameter attribute number (0 to 32) and the component number (0=x, 1=y, 2=z and 3=w).

Table 10.1 lists and briefly describes the parameter instruction fields.

Table 10.1 Parameter Instruction Fields

Field	Size	Description
VDST	8	Destination VGPR. Also acts as source for `v_interp_p2_f32`.
OP	2	Opcode: 0: `v_interp_p1_f32` VDST = P10 * VSRC + P0 1: `v_interp_p2_f32` VDST = P20 * VSRC + VDST 2: `v_interp_mov_f32` VDST = (P0, P10 or P20 selected by VSRC[1:0]) P0, P10 and P20 are parameter values read from LDS
ATTR	6	Attribute number: 0 to 32.
ATTR CHAN	2	0=X, 1=Y, 2=Z, 3=W
VSRC	8	Source VGPR supplies interpolation "I" or "J" value. For OP==`v_interp_mov_f32`: 0=P10, 1=P20, 2=P0. VSRC must not be the same register as VDST because 16-bank LDS chips implement `v_interp_p1` as a macro of two instructions.
(M0)	32	Use of the M0 register is automatic. M0 must contain: `{ 1'b0, new_prim_mask[15:1], lds_param_offset[15:0] }`

Parameter interpolation and parameter move instructions must initialize the M0 register before using it, as shown in Table 10.1. The `lds_param_offset[15:0]` is an address offset from the beginning of LDS storage allocated to this wavefront to where parameters begin in LDS memory for this wavefront. The `new_prim_mask` is a 15-bit mask with one bit per quad; a one in this mask indicates that this quad begins a new primitive, a zero indicates it uses the same primitive as the previous quad. The mask is 15 bits, not 16, since the first quad in a wavefront always begins a new primitive and so it is not included in the mask.

Figure 10.3 shows how parameters are laid out in LDS memory.

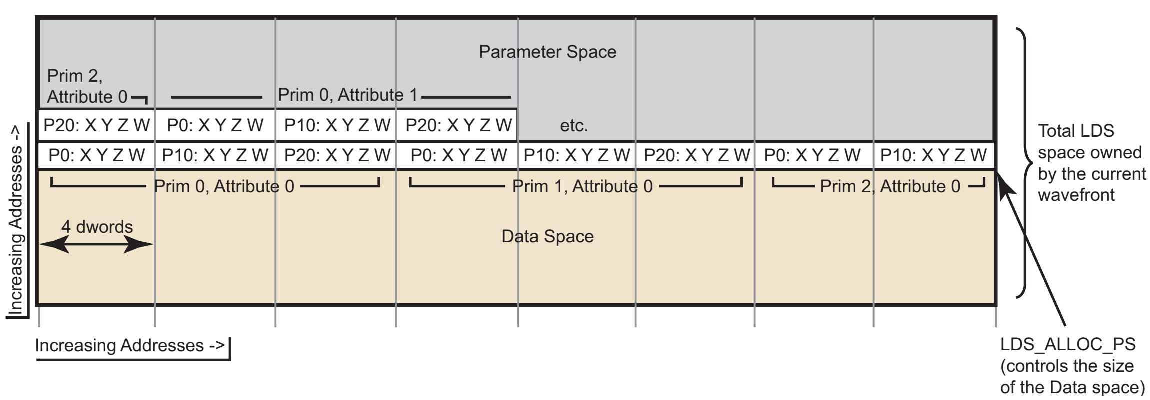


Figure 10.3 LDS Layout with Parameters and Data Share

10.3.3 Data Share Indexed and Atomic Access

Both LDS and GDS can perform indexed and atomic data share operations. For brevity, “LDS” is used in the text below and, except where noted, also applies to GDS.

Indexed and atomic operations supply a unique address per work-item from the VGPRs to the LDS, and supply or return unique data per work-item back to VGPRs. Due to the internal banked structure of LDS, operations can complete in as little as two cycles, or take as many 64 cycles, depending upon the number of bank conflicts (addresses that map to the same memory bank).

Indexed operations are simple LDS load and store operations that read data from, and return data to, VGPRs.

Atomic operations are arithmetic operations that combine data from VGPRs and data in LDS, and write the result back to LDS. Atomic operations have the option of returning the LDS “pre-op” value to VGPRs.

Table 10.2 lists and briefly describes the LDS instruction fields.

Table 10.2 LDS Instruction Fields

Field	Size	Description
OP	7	LDS opcode.
GDS	1	0 = LDS, 1 = GDS.
OFFSET0	8	Immediate offset, in bytes. Instructions with one address combine the offset fields into a single 16-bit unsigned offset: {offset1, offset0}. Instructions with two addresses (for example: READ2) use the offsets separately as two 8-bit unsigned offsets. `DS_*_SRC2_*` ops treat the offset as a 16-bit signed Dword offset.
OFFSET1	8	
VDST	8	VGPR to which result is written: either from LDS-load or atomic return value.
ADDR	8	VGPR that supplies the byte address offset.
DATA0	8	VGPR that supplies first data source.
DATA1	8	VGPR that supplies second data source.
(M0)	32	Implied use of M0. M0[16:0] contains the byte-size of the LDS segment. This is used to clamp the final address.

All LDS operations require that M0 be initialized prior to use. M0 contains a size value that can be used to restrict access to a subset of the allocated LDS range. If no clamping is wanted, set M0 to 0xFFFFFFFF.

Table 10.3 lists and describes the LDS indexed loads and stores.

Table 10.3 LDS Indexed Load/Store

DS_READ_{B32, B64, B96, B128, U8, I8, U16, I16}	Read one value per thread; sign extend to DWORD, if signed.
DS_READ2_{B32,B64}	Read two values at unique addresses.
DS_READ2ST64_{B32,B64}	Read 2 values at unique addresses, offset *= 64
DS_WRITE_{B32, B64, B96, B128, B8, B16}	Write one value.
DS_WRITE2_{B32, B64}	Write two values.
DS_WRITE2ST64_{B32, B64}	Write two values, offset *= 64.
DS_WRXCHG2_RTN_{B32, B64}	Exchange GPR with LDS-memory.
DS_WRXCHG2ST64_RTN_{B32,B64}	Exchange GPR with LDS-memory, offset *= 64.

Single Address Instructions

```
LDS_Addr = LDS_BASE + VGPR[ADDR] + {InstrOffset1,InstrOffset0}
```

Double Address Instructions

```
LDS_Addr0 = LDS_BASE + VGPR[ADDR] + InstrOffset0

LDS_Addr1 = LDS_BASE + VGPR[ADDR] + InstrOffset1
```

Note that `LDS_ADDR1` is used only for `READ2*`, `WRITE2*`, and `WREXCHG2*`.

`M0[15:0]` provides the size in bytes for this access. The size sent to LDS is `MIN(M0, LDS_SIZE)`, where `LDS_SIZE` is the amount of LDS space allocated by the shader processor interpolator, SPI, at the time the wavefront was created.

The address comes from VGPR, and both `ADDR` and `InstrOffset` are byte addresses.

At the time of wavefront creation, `LDS_BASE` is assigned to the physical LDS region owned by this wavefront or work-group.

Specify only one address by setting both offsets to the same value. This causes only one read or write to occur and uses only the first DATA0.

LDS Atomic Ops

```
DS_<atomicOp> OP, GDS=0, OFFSET0, OFFSET1, VDST, ADDR, Data0, Data1
```

Data size is encoded in `atomicOp`: byte, word, Dword, or double.

```
LDS_Addr0 = LDS_BASE + VGPR[ADDR] + {InstrOffset1,InstrOffset0}
```

ADDR is a Dword address. VGPRs 0,1 and dst are double-GPRs for doubles data.

VGPR data sources can only be VGPRs or constant values, not SGPRs.

Chapter 11
Exporting Pixel Color and Vertex Shader Parameters

The export instruction copies pixel or vertex shader data from VGPRs into a dedicated output buffer. The export instruction outputs the following types of data.

- Vertex Position
- Vertex Parameter
- Pixel color
- Pixel depth (Z)

11.1 Microcode Encoding

The export instruction uses the EXP microcode format.

<table>
<tr><td colspan="6">VSRC3</td><td>VSRC2</td><td colspan="4">VSRC1</td><td colspan="2">VSRC0</td><td>+4</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>r</td><td>VM</td><td>DONE</td><td>COMPR</td><td>TARGET</td><td colspan="2">EN</td><td>+0</td></tr>
</table>

The fields are described in Table 11.1.

Table 11.1 EXP Encoding Field Descriptions

Field	Bits	Description
VM	12	Valid Mask. When set to 1, this indicates that the EXEC mask represents the valid-mask for this wavefront. It can be sent multiple times per shader (the final value is used), but must be sent at least once per pixel shader.
DONE	11	This is the final pixel shader or vertex-position export of the program. Used only for pixel and position exports. Set to zero for parameters.
COMPR	10	Compressed data. When set, indicates that the data being exported is 16-bits per component rather than the usual 32-bit.
TARGET	10:4	Indicates type of data exported. 0..7 MRT 0..7 8 Z 9 Null (no data) 12-15 Position 0..3 32-63 Param 0..31

Table 11.1 EXP Encoding Field Descriptions (Cont.)

Field	Bits	Description
EN	3:0	**COMPR==1:** export half-dword enable. Valid values are: 0x0,3,C,F. [0] enables VSRC0 : R,G from one VGPR [2] enables VSRC1 : B,A from one VGPR **COMPR==0**: [0-3] = enables for VSRC0..3. EN can be zero (used when exporting only valid mask to NULL target).
VSRC3	63:56	VGPR from which to read data. Pos & Param: vsrc0=X, 1=Y, 2=Z, 3=W MRT: vsrc0=R, 1=G, 2=B, 3=A
VSRC2	55:48	
VSRC1	47:40	
VSRC0	39:32	

11.2 Operations

11.2.1 Pixel Shader Exports

Export instructions copy color data to the MRTs. Data always has four components (R, G, B, A). Optionally, export instructions also output depth (Z) data.

Every pixel shader must have at least one export instruction. The last export instruction executed must have the DONE bit set to one.

The EXEC mask is applied to all exports. Only pixels with the corresponding EXEC bit set to 1 export data to the output buffer. Results from multiple exports are accumulated in the output buffer.

At least one export must have the VM bit set to 1. This export, in addition to copying data to the color or depth output buffer, also informs the color buffer which pixels are valid and which have been discarded. The value of the EXEC mask communicates the pixel valid mask. If multiple exports are sent with VM set to 1, the mask from the final export is used. If the shader program wants to only update the valid mask but not send any new data, the program can do an export to the NULL target.

11.2.2 Vertex Shader Exports

The vertex shader uses export instructions to output vertex position data and vertex parameter data to the output buffer. This data is passed on to subsequent pixel shaders.

Every vertex shader must output at least one position vector (x, y, z; w is optional) to the POS0 target. The last position export must have the DONE bit set to 1. A vertex shader can export zero or more parameters. For best performance, it is best to output all position data as early as possible in the vertex shader.

11.3 Dependency Checking

Export instructions are executed by the hardware in two phases. First, the instruction is selected to be executed, and EXPCNT is incremented by 1. At this time, the hardware requests the use of internal busses needed to complete the instruction.

When access to the bus is granted, the EXEC mask is read and the VGPR data sent out. After the last of the VGPR data is sent, the EXPCNT counter is decremented by 1.

Use S_WAITCNT on EXPCNT to prevent the shader program from overwriting EXEC or the VGPRs holding the data to be exported before the export operation has completed.

Multiple export instructions can be outstanding at one time. Exports of the same type (for example: position) are completed in order, but exports of different types can be completed out of order.

If the STATUS register's SKIP_EXPORT bit is set to one, the hardware treats all EXPORT instructions as if they were NOPs.

Chapter 12 Instruction Set

This chapter lists, and provides descriptions for, all instructions in the Sea Islands environment. Instructions are grouped according to their format.

Instruction suffixes have the following definitions:

B32 Bitfield (untyped data) 32-bit
B64 Bitfield (untyped data) 64-bit
F32 floating-point 32-bit (IEEE 754 single-precision float)
F64 floating-point 64-bit (IEEE 754 double-precision float)
I32 signed 32-bit integer
I64 signed 64-bit integer
U32 unsigned 32-bit integer
U64 unsigned 64-bit integer
If an instruction has two suffixes (for example, _I32_F32), the first suffix indicates the destination type, the second the source type.

Note that .u or .i specifies to interpret the argument as an unsigned or signed float.

12.1 SOP2 Instructions

Instruction **S_ABSDIFF_I32**

Description D.i = abs(S0.i – S1.i). SCC = 1 if result is non-zero.

Microcode SOP2 Opcode 44 (0x2C)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction	**`S_ADD_I32`**
Description	D.u = S0.i + S1.i. SCC = signed overflow.
Microcode	SOP2 Opcode 2 (0x2)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction	**`S_ADD_U32`**
Description	D.u = S0.u + S1.u. SCC = unsigned carry out.
Microcode	SOP2 Opcode 0 (0x0)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction	**`S_ADDC_U32`**
Description	D.u = S0.u + S1.u + SCC. SCC = unsigned carry-out.
Microcode	SOP2 Opcode 4 (0x4)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction	**`S_AND_B32`**
Description	D.u = S0.u & S1.u. SCC = 1 if result is non-zero.
Microcode	SOP2 Opcode 14 (0xE)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction **S_AND_B64**

Description D.u = S0.u & S1.u. SCC = 1 if result is non-zero.

Microcode SOP2 Opcode 15 (0xF)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction **S_ANDN2_B32**

Description D.u = S0.u & ~S1.u. SCC = 1 if result is non-zero.

Microcode SOP2 Opcode 20 (0x14)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction **S_ANDN2_B64**

Description D.u = S0.u & ~S1.u. SCC = 1 if result is non-zero.

Microcode SOP2 Opcode 21 (0x15)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction **S_ASHR_I32**

Description D.i = signext(S0.i) >> S1.i[4:0]. SCC = 1 if result is non-zero.

Microcode SOP2 Opcode 34 (0x22)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction **S_ASHR_I64**

Description D.i = signext(S0.i) >> S1.i[5:0]. SCC = 1 if result is non-zero.

Microcode SOP2 Opcode 35 (0x23)

1	0	OP	SDST	SSRC1	SSRC0	+0

Instruction **S_BFE_I32**

Description Replace description text with:

DX11 Unsigned bitfield extract. Extract a contiguous range of bits from 32-bit source.

SRC0 = input data
SRC1 = the lowest bit position to select
SRC2 = the width of the bit field

Returns the bit starting at "offset" and ending at "offset+width-1".

The final result is sign-extended.

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

>>> means arithmetic shift right.

SCC = 1 if result is non-zero. Test sign-extended result.

Microcode SOP2 Opcode 40 (0x28)

1	0	OP	SDST	SSRC1	SSRC0	+0

Instruction **S_BFE_I64**

Description Bit field extract. S0 is data, S1[5:0] is field offset, S1[22:16] is field width. D.i = (S0.u >> S1.u[5:0]) & ((1 << S1.u[22:16]) - 1). SCC = 1 if result is non-zero. Test sign-extended result.

Microcode SOP2 Opcode 42 (0x2A)

1	0	OP	SDST	SSRC1	SSRC0	+0

Instruction	**S_BFE_U32**
Description	DX11 Unsigned bitfield extract. Extract a contiguous range of bits from 32-bit source.

SRC0 = input data
SRC1 = the lowest bit position to select
SRC2 = the width of the bit field

Returns the bit starting at "offset" and ending at "offset+width-1".

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

SCC = 1 if result is non-zero. Test sign-extended result.

Microcode SOP2 Opcode 39 (0x27)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction	**S_BFE_U64**
Description	Bit field extract. S0 is data, S1[4:0] is field offset, S1[22:16] is field width. D.u = (S0.u >> S1.u[5:0]) & ((1 << S1.u[22:16]) - 1). SCC = 1 if result is non-zero.

Microcode SOP2 Opcode 41 (0x29)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction	**S_BFM_B32**
Description	D.u = ((1 << S0.u[4:0]) - 1) << S1.u[4:0]; bitfield mask.

Microcode SOP2 Opcode 36 (0x24)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction **S_BFM_B64**

Description D.u = ((1 << S0.u[5:0]) - 1) << S1.u[5:0]; bitfield mask.

Microcode SOP2 Opcode 37 (0x25)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction **S_CBRANCH_G_FORK**

Description Conditional branch using branch stack. Arg0 = compare mask (VCC or any SGPR), Arg1 = 64-bit byte address of target instruction. See Section 4.6, on page 4-4.

Microcode SOP2 Opcode 43 (0x2B)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction **S_CSELECT_B32**

Description D.u = SCC ? S0.u : S1.u.

Microcode SOP2 Opcode 10 (0xA)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction **S_CSELECT_B64**

Description D.u = SCC ? S0.u : S1.u.

Microcode SOP2 Opcode 11 (0xB)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction **S_LSHL_B32**

Description D.u = S0.u << S1.u[4:0]. SCC = 1 if result is non-zero.

Microcode SOP2 Opcode 30 (0x1E)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction **S_LSHL_B64**

Description D.u = S0.u << S1.u[5:0]. SCC = 1 if result is non-zero.

Microcode SOP2 Opcode 31 (0x1F)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction **S_LSHR_B32**

Description D.u = S0.u >> S1.u[4:0]. SCC = 1 if result is non-zero.

Microcode SOP2 Opcode 32 (0x20)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction **S_LSHR_B64**

Description D.u = S0.u >> S1.u[5:0]. SCC = 1 if result is non-zero.

Microcode SOP2 Opcode 33 (0x21)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction **S_MAX_I32**

Description D.i = (S0.i > S1.i) ? S0.i : S1.i. SCC = 1 if S0 is max.

Microcode SOP2 Opcode 8 (0x8)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction **S_MAX_U32**

Description D.u = (S0.u > S1.u) ? S0.u : S1.u. SCC = 1 if S0 is max.

Microcode SOP2 Opcode 9 (0x9)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction **S_MIN_I32**

Description D.i = (S0.i < S1.i) ? S0.i : S1.i. SCC = 1 if S0 is min.

Microcode SOP2 Opcode 6 (0x6)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction **S_MIN_U32**

Description D.u = (S0.u < S1.u) ? S0.u : S1.u. SCC = 1 if S0 is min.

Microcode SOP2 Opcode 7 (0x7)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction	**`S_MUL_I32`**
Description	D.i = S0.i * S1.i.
Microcode	SOP2 Opcode 38 (0x26)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction	**`S_NAND_B32`**
Description	D.u = ~(S0.u & S1.u). SCC = 1 if result is non-zero.
Microcode	SOP2 Opcode 24 (0x18)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction	**`S_NAND_B64`**
Description	D.u = ~(S0.u & S1.u). SCC = 1 if result is non-zero.
Microcode	SOP2 Opcode 25 (0x19)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction	**`S_NOR_B32`**
Description	D.u = ~(S0.u | S1.u). SCC = 1 if result is non-zero.
Microcode	SOP2 Opcode 26 (0x1A)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction **S_NOR_B64**

Description D.u = ~(S0.u | S1.u). SCC = 1 if result is non-zero.

Microcode SOP2 Opcode 27 (0x1B)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction **S_OR_B32**

Description D.u = S0.u | S1.u. SCC = 1 if result is non-zero.

Microcode SOP2 Opcode 16 (0x10)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction **S_OR_B64**

Description D.u = S0.u | S1.u. SCC = 1 if result is non-zero.

Microcode SOP2 Opcode 17 (0x11)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction **S_ORN2_B32**

Description D.u = S0.u | ~S1.u. SCC = 1 if result is non-zero.

Microcode SOP2 Opcode 22 (0x 16)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction	**`S_ORN2_B64`**
Description	D.u = S0.u | ~S1.u. SCC = 1 if result is non-zero.
Microcode	SOP2 Opcode 23 (0x17)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction	**`S_SUB_I32`**
Description	D.u = S0.i - S1.i. SCC = borrow.
Microcode	SOP2 Opcode 3 (0x3)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction	**`S_SUB_U32`**
Description	D.u = S0.u - S1.u. SCC = unsigned carry out.
Microcode	SOP2 Opcode 1 (0x1)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction	**`S_SUBB_U32`**
Description	D.u = S0.u - S1.u - SCC. SCC = unsigned carry-out.
Microcode	SOP2 Opcode 5 (0x5)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction **S_XNOR_B32**

Description D.u = ~(S0.u ^ S1.u). SCC = 1 if result is non-zero.

Microcode SOP2 Opcode 28 (0x1C)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction **S_XNOR_B64**

Description D.u = ~(S0.u ^ S1.u). SCC = 1 if result is non-zero.

Microcode SOP2 Opcode 29 (0x1D)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction **S_XOR_B32**

Description D.u = S0.u ^ S1.u. SCC = 1 if result is non-zero.

Microcode SOP2 Opcode 18 (0x12)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

Instruction **S_XOR_B64**

Description D.u = S0.u ^ S1.u. SCC = 1 if result is non-zero.

Microcode SOP2 Opcode 19 (0x13)

1	**0**	OP	SDST	SSRC1	SSRC0	+0

12.2 SOPK Instructions

Instruction **S_ADDK_I32**

Description D.i = D.i + signext(SIMM16). SCC = signed overflow.

Microcode SOPK Opcode 15 (0xF)

1	**0**	**1**	**1**	OP	SDST	SIMM	+0

Instruction **S_CBRANCH_I_FORK**

Description Conditional branch using branch-stack. Arg0(sdst) = compare mask (VCC or any SGPR), SIMM16 = signed DWORD branch offset relative to next instruction. See Section 4.6, on page 4-4.

Microcode SOPK Opcode 17 (0x11)

1	**0**	**1**	**1**	OP	SDST	SIMM	+0

Instruction **S_CMOVK_I32**

Description if (SCC) D.i = signext(SIMM16); else NOP.

Microcode SOPK Opcode 2 (0x2)

1	**0**	**1**	**1**	OP	SDST	SIMM	+0

Instruction **S_CMPK_EQ_I32**

Description SCC = (D.i == signext(SIMM16)).

Microcode SOPK Opcode 3 (0x3)

1	**0**	**1**	**1**	OP	SDST	SIMM	+0

Instruction **`S_CMPK_EQ_U32`**

Description SCC = (D.u == SIMM16).

Microcode SOPK Opcode 9 (0x9)

1	**0**	**1**	**1**	OP	SDST	SIMM	+0

Instruction **`S_CMPK_GE_I32`**

Description SCC = (D.i >= signext(SIMM16)).

Microcode SOPK Opcode 6 (0x6)

1	**0**	**1**	**1**	OP	SDST	SIMM	+0

Instruction **`S_CMPK_GE_U32`**

Description SCC = (D.u >= SIMM16).

Microcode SOPK Opcode 12 (0xC)

1	**0**	**1**	**1**	OP	SDST	SIMM	+0

Instruction **`S_CMPK_GT_I32`**

Description SCC = (D.i > signext(SIMM16)).

Microcode SOPK Opcode 5 (0x5)

1	**0**	**1**	**1**	OP	SDST	SIMM	+0

Instruction	**`S_CMPK_GT_U32`**
Description	SCC = (D.u > SIMM16).
Microcode	SOPK Opcode 11 (0xB)

1	**0**	**1**	**1**	OP	SDST	SIMM	+0

Instruction	**`S_CMPK_LE_I32`**
Description	SCC = (D.i <= signext(SIMM16)).
Microcode	SOPK Opcode 8 (0x8)

1	**0**	**1**	**1**	OP	SDST	SIMM	+0

Instruction	**`S_CMPK_LE_U32`**
Description	D.u = SCC = (D.u <= SIMM16).
Microcode	SOPK Opcode 14 (0xE)

1	**0**	**1**	**1**	OP	SDST	SIMM	+0

Instruction	**`S_CMPK_LG_I32`**
Description	SCC = (D.i != signext(SIMM16)).
Microcode	SOPK Opcode 4 (0x4)

1	**0**	**1**	**1**	OP	SDST	SIMM	+0

Instruction **`S_CMPK_LG_U32`**

Description SCC = (D.u != SIMM16).

Microcode SOPK Opcode 10 (0xA)

1	**0**	**1**	**1**	OP	SDST	SIMM	+0

Instruction **`S_CMPK_LT_I32`**

Description SCC = (D.i < signext(SIMM16)).

Microcode SOPK Opcode 7 (0x7)

1	**0**	**1**	**1**	OP	SDST	SIMM	+0

Instruction **`S_CMPK_LT_U32`**

Description SCC = (D.u < SIMM16).

Microcode SOPK Opcode 13 (0xD)

1	**0**	**1**	**1**	OP	SDST	SIMM	+0

Instruction **`S_GETREG_B32`**

Description D.u = hardware register. Read some or all of a hardware register into the LSBs of D. See Table 5.7 on page 5-7. SIMM16 = {size[4:0], offset[4:0], hwRegId[5:0]}; offset is in the range from 0 to 31, size is in the range from 1 to 32.

Microcode SOPK Opcode 18 (0x12)

1	**0**	**1**	**1**	OP	SDST	SIMM	+0

Instruction **S_MOVK_I32**

Description D.i = signext(SIMM16).

Microcode SOPK Opcode 0 (0x0)

1	**0**	**1**	**1**	OP	SDST	SIMM	+0

Instruction **S_MULK_I32**

Description D.i = D.i * signext(SIMM16). SCC = overflow.

Microcode SOPK Opcode 16 (0x10)

1	**0**	**1**	**1**	OP	SDST	SIMM	+0

Instruction **S_SETREG_B32**

Description Hardware register = D.u. Write some or all of the LSBs of D into a hardware register (note that D is a source SGPR). See Table 5.7 on page 5-7.

SIMM16 = {size[4:0], offset[4:0], hwRegId[5:0]}; offset is in the range from 0 to 31, size is in the range from 1 to 32.

Microcode SOPK Opcode 19 (0x13)

1	**0**	**1**	**1**	OP	SDST	SIMM	+0

Instruction **S_SETREG_IMM32_B32**

Description This instruction uses a 32-bit literal constant. Write some or all of the LSBs of SIMM32 into a hardware register.

SIMM16 = {size[4:0], offset[4:0], hwRegId[5:0]}; offset is 0–31, size is 1–32.

Microcode SOPK Opcode 21 (0x15)

SIMM32	+4

1	**0**	**1**	**1**	OP	SDST	SIMM	+0

12.3 SOP1 Instructions

Instruction **S_ABS_I32**

Description D.i = abs(S0.i). SCC=1 if result is non-zero.

Microcode SOP1 Opcode 52 (0x 34)

1	0	1	1	1	1	1	0	1	SDST	OP	SSRC0	+0

Instruction **S_AND_SAVEEXEC_B64**

Description D.u = EXEC, EXEC = S0.u & EXEC. SCC = 1 if the new value of EXEC is non-zero.

Microcode SOP1 Opcode 36 (0x24)

1	0	1	1	1	1	1	0	1	SDST	OP	SSRC0	+0

Instruction **S_ANDN2_SAVEEXEC_B64**

Description D.u = EXEC, EXEC = S0.u & ~EXEC. SCC = 1 if the new value of EXEC is non-zero.

Microcode SOP1 Opcode 39 (0x27)

1	0	1	1	1	1	1	0	1	SDST	OP	SSRC0	+0

Instruction **S_BCNT0_I32_B32**

Description D.i = CountZeroBits(S0.u). SCC = 1 if result is non-zero.

Microcode SOP1 Opcode 13 (0xD)

1	0	1	1	1	1	1	0	1	SDST	OP	SSRC0	+0

Instruction **S_BCNT0_I32_B64**

Description D.i = CountZeroBits(S0.u). SCC = 1 if result is non-zero.

Microcode SOP1 Opcode 14 (0xE)

1	0	1	1	1	1	1	0	1	SDST	OP	SSRC0	+0

Instruction **S_BCNT1_I32_B32**

Description D.i = CountOneBits(S0.u). SCC = 1 if result is non-zero.

Microcode SOP1 Opcode 15 (0xF)

1	0	1	1	1	1	1	0	1	SDST	OP	SSRC0	+0

Instruction **S_BCNT1_I32_B64**

Description D.i = CountOneBits(S0.u). SCC = 1 if result is non-zero.

Microcode SOP1 Opcode 16 (0x20)

1	0	1	1	1	1	1	0	1	SDST	OP	SSRC0	+0

Instruction **S_BITSET0_B32**

Description D.u[S0.u[4:0]] = 0.

Microcode SOP1 Opcode 27 (0x1B)

1	0	1	1	1	1	1	0	1	SDST	OP	SSRC0	+0

Instruction **S_BITSET0_B64**

Description D.u[S0.u[5:0]] = 0.

Microcode SOP1 Opcode 28 (0x1C)

1	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	SDST	OP	SSRC0	+0

Instruction **S_BITSET1_B32**

Description D.u[S0.u[4:0]] = 1.

Microcode SOP1 Opcode 29 (0x1D)

1	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	SDST	OP	SSRC0	+0

Instruction **S_BITSET1_B64**

Description D.u[S0.u[5:0]] = 1.

Microcode SOP1 Opcode 30 (0x1E)

1	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	SDST	OP	SSRC0	+0

Instruction **S_BREV_B32**

Description D.u = S0.u[0:31] (reverse bits).

Microcode SOP1 Opcode 11 (0xB)

1	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	SDST	OP	SSRC0	+0

Instruction **S_BREV_B64**

Description D.u = S0.u[0:63] (reverse bits).

Microcode SOP1 Opcode 12 (0xC)

1	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	SDST	OP	SSRC0	+0

Instruction **S_CBRANCH_JOIN**

Description Conditional branch join point. Arg0 = saved CSP value. No dest. See Section 4.6, on page 4-4.

Microcode SOP1 Opcode 50 (0x32)

1	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	SDST	OP	SSRC0	+0

Instruction **S_CMOV_B32**

Description if(SCC) D.u = S0.u; else NOP.

Microcode SOP1 Opcode 5 (0x5)

1	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	SDST	OP	SSRC0	+0

Instruction **S_CMOV_B64**

Description if(SCC) D.u = S0.u; else NOP.

Microcode SOP1 Opcode 6 (0x6)

1	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	SDST	OP	SSRC0	+0

Instruction	**`S_FF0_I32_B32`**
Description	D.i = FindFirstZero(S0.u) from LSB; if no zeros, return -1.
Microcode	SOP1 Opcode 17 (0x21)

1	0	1	1	1	1	1	0	1	SDST	OP	SSRC0	+0

Instruction	**`S_FF0_I32_B64`**
Description	D.i = FindFirstZero(S0.u) from LSB; if no zeros, return -1.
Microcode	SOP1 Opcode 18 (0x22)

1	0	1	1	1	1	1	0	1	SDST	OP	SSRC0	+0

Instruction	**`S_FF1_I32_B32`**
Description	D.i = FindFirstOne(S0.u) from LSB; if no ones, return -1.
Microcode	SOP1 Opcode 19 (0x23)

1	0	1	1	1	1	1	0	1	SDST	OP	SSRC0	+0

Instruction	**`S_FF1_I32_B64`**
Description	D.i = FindFirstOne(S0.u) from LSB; if no ones, return -1.
Microcode	SOP1 Opcode 20 (0x24)

1	0	1	1	1	1	1	0	1	SDST	OP	SSRC0	+0

Instruction **S_FLBIT_I32**

Description D.i = Find first bit opposite of sign bit from MSB. If S0 == -1, return -1.

Microcode SOP1 Opcode 23 (0x27)

1	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	SDST	OP	SSRC0	+0

Instruction **S_FLBIT_I32_B32**

Description D.i = FindFirstOne(S0.u) from MSB; if no ones, return -1.

Microcode SOP1 Opcode 21 (0x25)

1	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	SDST	OP	SSRC0	+0

Instruction **S_FLBIT_I32_B64**

Description D.i = FindFirstOne(S0.u) from MSB; if no ones, return -1.

Microcode SOP1 Opcode 22 (0x26)

1	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	SDST	OP	SSRC0	+0

Instruction **S_FLBIT_I32_I64**

Description D.i = Find first bit opposite of sign bit from MSB. If S0 == -1, return -1.

Microcode SOP1 Opcode 24 (0x28)

1	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	SDST	OP	SSRC0	+0

Instruction **`S_GETPC_B64`**

Description D.u = PC + 4; destination receives the byte address of the next instruction.

Microcode SOP1 Opcode 31 (0x1F)

1	0	1	1	1	1	1	0	1	SDST	OP	SSRC0	+0

Instruction **`S_MOV_B32`**

Description D.u = S0.u.

Microcode SOP1 Opcode 3 (0x3)

1	0	1	1	1	1	1	0	1	SDST	OP	SSRC0	+0

Instruction **`S_MOV_B64`**

Description Du = S0.u.

Microcode SOP1 Opcode 4 (0x4)

1	0	1	1	1	1	1	0	1	SDST	OP	SSRC0	+0

Instruction **`S_MOVRELD_B32`**

Description SGPR[D.u + M0.u] = SGPR[S0.u].

Microcode SOP1 Opcode 48 (0x30)

1	0	1	1	1	1	1	0	1	SDST	OP	SSRC0	+0

Instruction	**`S_MOVRELD_B64`**
Description	SGPR[D.u + M0.u] = SGPR[S0.u]. M0 and D.u must be even.
Microcode	SOP1 Opcode 49 (0x31)

1	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	SDST	OP	SSRC0	+0

Instruction	**`S_MOVRELS_B32`**
Description	SGPR[D.u] = SGPR[S0.u + M0.u].
Microcode	SOP1 Opcode 46 (0x2E)

1	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	SDST	OP	SSRC0	+0

Instruction	**`S_MOVRELS_B64`**
Description	SGPR[D.u] = SGPR[S0.u + M0.u]. M0 and S0.u must be even.
Microcode	SOP1 Opcode 47 (0x2F)

1	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	SDST	OP	SSRC0	+0

Instruction	**`S_NAND_SAVEEXEC_B64`**
Description	D.u = EXEC, EXEC = ~(S0.u & EXEC). SCC = 1 if the new value of EXEC is non-zero.
Microcode	SOP1 Opcode 41 (0x29)

1	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	SDST	OP	SSRC0	+0

Instruction **S_NOR_SAVEEXEC_B64**

Description D.u = EXEC, EXEC = ~(S0.u | EXEC). SCC = 1 if the new value of EXEC is non-zero.

Microcode SOP1 Opcode 42 (0x2A)

1	0	1	1	1	1	1	0	1	SDST	OP	SSRC0	+0

Instruction **S_NOT_B32**

Description D.u = ~S0.u SCC = 1 if result non-zero.

Microcode SOP1 Opcode 7 (0x7)

1	0	1	1	1	1	1	0	1	SDST	OP	SSRC0	+0

Instruction **S_NOT_B64**

Description D.u = ~S0.u SCC = 1 if result non-zero.

Microcode SOP1 Opcode 8 (0x8)

1	0	1	1	1	1	1	0	1	SDST	OP	SSRC0	+0

Instruction **S_OR_SAVEEXEC_B64**

Description D.u = EXEC, EXEC = S0.u | EXEC. SCC = 1 if the new value of EXEC is non-zero.

Microcode SOP1 Opcode 37 (0x25)

1	0	1	1	1	1	1	0	1	SDST	OP	SSRC0	+0

Instruction **S_ORN2_SAVEEXEC_B64**

Description D.u = EXEC, EXEC = S0.u | ~EXEC. SCC = 1 if the new value of EXEC is non-zero.

Microcode SOP1 Opcode 40 (0x28)

1	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	SDST	OP	SSRC0	+0

Instruction **S_QUADMASK_B32**

Description D.u = QuadMask(S0.u). D[0] = OR(S0[3:0]), D[1] = OR(S0[7:4]) .... SCC = 1 if result is non-zero.

Microcode SOP1 Opcode 44 (0x2C)

1	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	SDST	OP	SSRC0	+0

Instruction **S_QUADMASK_B64**

Description D.u = QuadMask(S0.u). D[0] = OR(S0[3:0]), D[1] = OR(S0[7:4]) .... SCC = 1 if result is non-zero.

Microcode SOP1 Opcode 45 (0x2D)

1	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	SDST	OP	SSRC0	+0

Instruction **S_RFE_B64**

Description Return from Exception; PC = S0.u. This instruction sets PRIV to 0.

Microcode SOP1 Opcode 34 (0x22)

1	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	SDST	OP	SSRC0	+0

Instruction	**`S_SETPC_B64`**
Description	PC = S0.u; S0.u is a byte address of the instruction to jump to.
Microcode	SOP1 Opcode 32 (0x20)

1	0	1	1	1	1	1	0	1	SDST	OP	SSRC0	+0

Instruction	**`S_SEXT_I32_I8`**
Description	D.i = signext(S0.i[7:0]).
Microcode	SOP1 Opcode 25 (0x29)

1	0	1	1	1	1	1	0	1	SDST	OP	SSRC0	+0

Instruction	**`S_SEXT_I32_I16`**
Description	D.i = signext(S0.i[15:0]).
Microcode	SOP1 Opcode 26 (0x1A)

1	0	1	1	1	1	1	0	1	SDST	OP	SSRC0	+0

Instruction	**`S_SWAPPC_B64`**
Description	D.u = PC + 4; PC = S0.u.
Microcode	SOP1 Opcode 33 (0x21)

1	0	1	1	1	1	1	0	1	SDST	OP	SSRC0	+0

Instruction	**`S_WQM_B32`**
Description	D.u = WholeQuadMode(S0.u). SCC = 1 if result is non-zero. Apply whole quad mode to the bitmask specified in SSRC0. Whole quad mode checks each group of four bits in the bitmask; if any bit is set to 1, all four bits are set to 1 in the result. This operation is repeated for the entire bitmask.
Microcode	SOP1 Opcode 9 (0x9)

1	0	1	1	1	1	1	0	1	SDST	OP	SSRC0	+0

Instruction	**`S_WQM_B64`**
Description	D.u = WholeQuadMode(S0.u). SCC = 1 if result is non-zero. Apply whole quad mode to the bitmask specified in SSRC0. Whole quad mode checks each group of four bits in the bitmask; if any bit is set to 1, all four bits are set to 1 in the result. This operation is repeated for the entire bitmask.
Microcode	SOP1 Opcode 10 (0xA)

1	0	1	1	1	1	1	0	1	SDST	OP	SSRC0	+0

Instruction	**`S_XNOR_SAVEEXEC_B64`**
Description	D.u = EXEC, EXEC = ~(S0.u ^ EXEC). SCC = 1 if the new value of EXEC is non-zero.
Microcode	SOP1 Opcode 43 (0x2B)

1	0	1	1	1	1	1	0	1	SDST	OP	SSRC0	+0

Instruction	**`S_XOR_SAVEEXEC_B64`**
Description	D.u = EXEC, EXEC = S0.u ^ EXEC. SCC = 1 if the new value of EXEC is non-zero.
Microcode	SOP1 Opcode 38 (0x26)

1	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	SDST	OP	SSRC0	+0

12.4 SOPC Instructions

Instruction **S_BITCMP0_B32**

Description SCC = (S0.u[S1.u[4:0]] == 0).

Microcode SOPC Opcode 12 (C)

1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	OP	SSRC1	SSRC0	+0

Instruction **S_BITCMP0_B64**

Description SCC = (S0.u[S1.u[5:0]] == 0).

Microcode SOPC Opcode 14 (0xE)

1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	OP	SSRC1	SSRC0	+0

Instruction **S_BITCMP1_B32**

Description SCC = (S0.u[S1.u[4:0]] == 1).

Microcode SOPC Opcode 13 (0xD)

1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	OP	SSRC1	SSRC0	+0

Instruction **S_BITCMP1_B64**

Description SCC = (S0.u[S1.u[5:0]] == 1).

Microcode SOPC Opcode 15 (0xF)

1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	OP	SSRC1	SSRC0	+0

Instruction **`S_CMP_EQ_I32`**

Description SCC = (S0.i == S1.i).

Microcode SOPC Opcode 0 (0x0)

1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	OP	SSRC1	SSRC0	+0

Instruction **`S_CMP_EQ_U32`**

Description SCC = (S0.u == S1.u).

Microcode SOPC Opcode 6 (0x6)

1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	OP	SSRC1	SSRC0	+0

Instruction **`S_CMP_GE_I32`**

Description SCC = (S0.i >= S1.i).

Microcode SOPC Opcode 3 (0x3)

1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	OP	SSRC1	SSRC0	+0

Instruction **`S_CMP_GE_U32`**

Description SCC = (S0.u >= S1.u).

Microcode SOPC Opcode 9 (0x9)

1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	OP	SSRC1	SSRC0	+0

Instruction **`S_CMP_GT_I32`**

Description SCC = (S0.i > S1.i).

Microcode SOPC Opcode 2 (0x2)

1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	OP	SSRC1	SSRC0	+0

Instruction **`S_CMP_GT_U32`**

Description SCC = (S0.u > S1.u).

Microcode SOPC Opcode 8 (0x8)

1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	OP	SSRC1	SSRC0	+0

Instruction **`S_CMP_LE_I32`**

Description SCC = (S0.i <= S1.i).

Microcode SOPC Opcode 5 (0x5)

1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	OP	SSRC1	SSRC0	+0

Instruction **`S_CMP_LE_U32`**

Description SCC = (S0.u <= S1.u).

Microcode SOPC Opcode 11 (0xB)

1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	OP	SSRC1	SSRC0	+0

Instruction **S_CMP_LG_I32**

Description SCC = (S0.i != S1.i).

Microcode SOPC Opcode 1 (0x1)

1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	OP	SSRC1	SSRC0	+0

Instruction **S_CMP_LG_U32**

Description SCC = (S0.u != S1.u).

Microcode SOPC Opcode 7 (0x7)

1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	OP	SSRC1	SSRC0	+0

Instruction **S_CMP_LT_I32**

Description SCC = (S0.i < S1.i).

Microcode SOPC Opcode 4 (0x4)

1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	OP	SSRC1	SSRC0	+0

Instruction **S_CMP_LT_U32**

Description SCC = (S0.u < S1.u).

Microcode SOPC Opcode 10 (0xA)

1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	OP	SSRC1	SSRC0	+0

Instruction **`S_SETVSKIP`**

Description VSKIP = S0.u[S1.u[4:0]].

Extract one bit from the SSRC0 SGPR, and use that bit to enable or disable VSKIP mode. In some cases, VSKIP mode can be used to skip over sections of code more quickly than branching. When VSKIP is enabled, the following instruction types are not executed: Vector ALU, Vector Memory, LDS, GDS, and Export.

Microcode SOPC Opcode 16 (0x10)

1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	OP	SSRC1	SSRC0	+0

12.5 SOPP Instructions

Instruction	**S_BARRIER**
Description	Sync waves within a work-group.
Microcode	SOPP Opcode 10 (0xA)

1	0	1	1	1	1	1	1	1	OP	SIMM	+0

Instruction	**S_BRANCH**
Description	PC = PC + signext(SIMM16 * 4) + 4.
Microcode	SOPP Opcode 2 (0x2)

1	0	1	1	1	1	1	1	1	OP	SIMM	+0

Instruction	**S_CBRANCH_CDBGSYS**
Description	Conditional branch when the SYStem debug bit is set. if(conditional_debug_system != 0) then PC = PC + signext(SIMM16 * 4) + 4; else NOP.
Microcode	SOPP Opcode 23 (0x17)

1	0	1	1	1	1	1	1	1	OP	SIMM	+0

Instruction	**S_CBRANCH_CDBGSYS_AND_USER**
Description	Conditional branch when both the SYStem and USER debug bits are set. if(conditional_debug_system && conditional_debug_user) then PC = PC + signext(SIMM16 * 4) + 4; else NOP.
Microcode	SOPP Opcode 26 (0x1A)

1	0	1	1	1	1	1	1	1	OP	SIMM	+0

Instruction **S_CBRANCH_CDBGSYS_OR_USER**

Description Conditional branch when either the SYStem or USER debug bits are set.

if(conditional_debug_system || conditional_debug_user) then PC = PC + signext(SIMM16 * 4) + 4; else NOP.

Microcode SOPP Opcode 25 (0x19)

1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	OP	SIMM	+0

Instruction **S_CBRANCH_CDBGUSER**

Description Conditional branch when the USER debug bit is set.

if(conditional_debug_user != 0) then PC = PC + signext(SIMM16 * 4) + 4; else NOP.

Microcode SOPP Opcode 24 (0x18)

1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	OP	SIMM	+0

Instruction **S_CBRANCH_EXECNZ**

Description if(EXEC != 0) then PC = PC + signext(SIMM16 * 4) + 4; else nop.

Microcode SOPP Opcode 9 (0x9)

1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	OP	SIMM	+0

Instruction **S_CBRANCH_EXECZ**

Description if(EXEC == 0) then PC = PC + signext(SIMM16 * 4) + 4; else nop.

Microcode SOPP Opcode 8 (0x8)

1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	OP	SIMM	+0

Instruction **S_CBRANCH_SCC0**

Description if(SCC == 0) then PC = PC + signext(SIMM16 * 4) + 4; else nop.

Microcode SOPP Opcode 4 (0x4)

1	0	1	1	1	1	1	1	1	OP	SIMM	+0

Instruction **S_CBRANCH_SCC1**

Description if(SCC == 1) then PC = PC + signext(SIMM16 * 4) + 4; else nop.

Microcode SOPP Opcode 5 (0x5)

1	0	1	1	1	1	1	1	1	OP	SIMM	+0

Instruction **S_CBRANCH_VCCNZ**

Description if(VCC != 0) then PC = PC + signext(SIMM16 * 4) + 4; else nop.

Microcode SOPP Opcode 7 (0x7)

1	0	1	1	1	1	1	1	1	OP	SIMM	+0

Instruction **S_CBRANCH_VCCZ**

Description if(VCC == 0) then PC = PC + signext(SIMM16 * 4) + 4; else nop.

Microcode SOPP Opcode 6 (0x6)

1	0	1	1	1	1	1	1	1	OP	SIMM	+0

Instruction	**S_DECPERFLEVEL**
Description	Decrement performance counter specified in SIMM16[3:0] by 1.

Microcode SOPP Opcode 21 (0x15)

1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	OP	SIMM	+0

Instruction	**S_ENDPGM**
Description	End of program; terminate wavefront.

Microcode SOPP Opcode 1 (0x1)

1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	OP	SIMM	+0

Instruction	**S_ICACHE_INV**
Description	Invalidate entire L1 instruction cache.

Microcode SOPP Opcode 19 (0x13)

1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	OP	SIMM	+0

Instruction	**S_INCPERFLEVEL**
Description	Increment performance counter specified in SIMM16[3:0] by 1.

Microcode SOPP Opcode 20 (0x14)

1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	OP	SIMM	+0

Instruction **`S_NOP`**

Description Do nothing. Repeat NOP 1..8 times based on SIMM16[2:0]. 0 = 1 time, 7 = 8 times.

Microcode SOPP Opcode 0 (0x0)

1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	OP	SIMM	+0

Instruction **`S_SENDMSG`**

Description Send a message.

SIMM[3:0]	Messsage	Payload
1	interrupt	M0[7:0] carries user data. IDs are also sent (wave_id, cu_id, …).
2	Gs	SIMM[5:4] defines GS_OP.
3	Gs_done	
4-14	unused	
15	System	Hardware internal use only.
SIMM[5:4]	**GS OP**	**Payload**
0	nop	Use for gs-done only. M0[7:0] = gs-waveID
1	cut	SIMM[9:8] = stream_id EXEC is also sent. M0[7:0] = gs-waveID
2	emit	
3	emit-cut	

Microcode SOPP Opcode 16 (0x10)

1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	OP	SIMM	+0

Instruction	**S_SENDMSGHALT**
Description	Send a message and then HALT.
Microcode	SOPP Opcode 17 (0x11)

1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	OP	SIMM	+0

Instruction	**S_SETHALT**
Description	set HALT bit to value of SIMM16[0]. 1=halt, 0=resume. Halt is ignored while priv=1.
Microcode	SOPP Opcode 13 (0xD)

1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	OP	SIMM	+0

Instruction	**S_SETKILL**
Description	Set KILL bit to value of SIMM16[0].
Microcode	SOPP Opcode 11 (0xB)

1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	OP	SIMM	+0

Instruction	**S_SETPRIO**
Description	User-settable wave priority. The priority value is indicated in the two LSBs of the SIMM field. 0 = lowest, 3 = highest.
Microcode	SOPP Opcode 15 (0xF)

1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	OP	SIMM	+0

Instruction **S_SLEEP**

Description Cause a wave to sleep for approximately 64*SIMM16[2:0] clocks.

Microcode SOPP Opcode 14 (0xE)

1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	OP	SIMM	+0

Instruction **S_TRAP**

Description Enter the trap handler. TrapID = SIMM16[7:0]. Wait for all instructions to complete, save {pc_rewind,trapID,pc} into ttmp0,1; load TBA into PC, set PRIV=1 and continue. A trapID of zero is not allowed.

Microcode SOPP Opcode 18 (0x12)

1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	OP	SIMM	+0

Instruction **S_TTRACEDATA**

Description Send M0 as user data to thread-trace.

Microcode SOPP Opcode 22 (0x16)

1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	OP	SIMM	+0

Instruction **S_WAITCNT**

Description Wait for count of outstanding lds, vector-memory and export/vmem-write-data to be at or below the specified levels. simm16[3:0] = vmcount, simm16[6:4] = export/mem-write-data count, simm16[12:8] = LGKM_cnt (scalar-mem/GDS/LDS count). See Section 4.4, on page 4-2.

Microcode SOPP Opcode 12 (0xC)

1	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	OP	SIMM	+0

12.6 SMRD Instructions

Instruction **S_BUFFER_LOAD_DWORD**

Description Read one Dword from read-only memory describe by a buffer a constant (V#) through the constant cache (kcache).

```
m_offset = IMM ? OFFSET : SGPR[OFFSET]
m_base = { SGPR[SBASE * 2 +1][15:0], SGPR[SBASE] }
m_stride = SGPR[SBASE * 2 +1][31:16]
m_num_records = SGPR[SBASE * 2 + 2]
m_size = (m_stride == 0) ? 1 : m_num_records
m_addr = (SGPR[SBASE * 2] + m_offset) & ~0x3
SGPR[SDST] = read_dword_from_kcache(m_base, m_offset, m_size)
```

Microcode SMRD Opcode 8 (0x8)

1	**1**	**0**	**0**	**0**	OP	SDST	SBASE	IMM	OFFSET	+0

Instruction **S_BUFFER_LOAD_DWORDX2**

Description Read two Dwords from read-only memory describe by a buffer a constant (V#) through the constant cache (kcache).

```
m_offset = IMM ? OFFSET : SGPR[OFFSET]
m_base = { SGPR[SBASE * 2 +1][15:0], SGPR[SBASE * 2] }
m_stride = SGPR[SBASE * 2 +1][31:16]
m_num_records = SGPR[SBASE * 2 + 2]
m_size = (m_stride == 0) ? 1 : m_num_records
m_addr = (SGPR[SBASE * 2] + m_offset) & ~0x3
SGPR[SDST] = read_dword_from_kcache(m_base, m_offset, m_size)
SGPR[SDST + 1] = read_dword_from_kcache(m_base, m_offset + 4, m_size)
```

Microcode SMRD Opcode 9 (0x9)

1	**1**	**0**	**0**	**0**	OP	SDST	SBASE	IMM	OFFSET	+0

Instruction **S_BUFFER_LOAD_DWORDX4**

Description Read four Dwords from read-only memory describe by a buffer a constant (V#) through the constant cache (kcache).

```
m_offset = IMM ? OFFSET : SGPR[OFFSET]
m_base = { SGPR[SBASE * 2 +1][15:0], SGPR[SBASE * 2] }
m_stride = SGPR[SBASE * 2 +1][31:16]
m_num_records = SGPR[SBASE * 2 + 2]
m_size = (m_stride == 0) ? 1 : m_num_records
m_addr = (SGPR[SBASE * 2] + m_offset) & ~0x3
SGPR[SDST] = read_dword_from_kcache(m_base, m_offset, m_size)
SGPR[SDST + 1] = read_dword_from_kcache(m_base, m_offset + 4, m_size)
SGPR[SDST + 2] = read_dword_from_kcache(m_base, m_offset + 8, m_size)
SGPR[SDST + 3] = read_dword_from_kcache(m_base, m_offset + 12, m_size)
```

Microcode SMRD Opcode 10 (0xA)

1	**1**	**0**	**0**	**0**	OP	SDST	SBASE	IMM	OFFSET	+0

Instruction **S_BUFFER_LOAD_DWORDX8**

Description Read eight Dwords from read-only memory describe by a buffer a constant (V#) through the constant cache (kcache).

```
m_offset = IMM ? OFFSET : SGPR[OFFSET]
m_base = { SGPR[SBASE * 2 +1][15:0], SGPR[SBASE * 2] }
m_stride = SGPR[SBASE * 2 +1][31:16]
m_num_records = SGPR[SBASE * 2 + 2]
m_size = (m_stride == 0) ? 1 : m_num_records
m_addr = (SGPR[SBASE * 2] + m_offset) & ~0x3
SGPR[SDST] = read_dword_from_kcache(m_base, m_offset, m_size)
SGPR[SDST + 1] = read_dword_from_kcache(m_base, m_offset + 4, m_size)
SGPR[SDST + 2] = read_dword_from_kcache(m_base, m_offset + 8, m_size)
. . .
SGPR[SDST + 7] = read_dword_from_kcache(m_base, m_offset + 28, m_size)
```

Microcode SMRD Opcode 11 (0xB)

1	**1**	**0**	**0**	**0**	OP	SDST	SBASE	IMM	OFFSET	+0

Instruction **S_BUFFER_LOAD_DWORDX16**

Description Read 16 Dwords from read-only memory describe by a buffer a constant (V#) through the constant cache (kcache).

```
m_offset = IMM ? OFFSET : SGPR[OFFSET]
m_base = { SGPR[SBASE * 2 +1][15:0], SGPR[SBASE * 2] }
m_stride = SGPR[SBASE * 2 +1][31:16]
m_num_records = SGPR[SBASE * 2 + 2]
m_size = (m_stride == 0) ? 1 : m_num_records
m_addr = (SGPR[SBASE * 2] + m_offset) & ~0x3
SGPR[SDST] = read_dword_from_kcache(m_base, m_offset, m_size)
SGPR[SDST + 1] = read_dword_from_kcache(m_base, m_offset + 4, m_size)
SGPR[SDST + 2] = read_dword_from_kcache(m_base, m_offset + 8, m_size)
. . .
SGPR[SDST + 15] = read_dword_from_kcache(m_base, m_offset + 60, m_size)
```

Microcode SMRD Opcode 12 (0xC)

1	**1**	**0**	**0**	**0**	OP	SDST	SBASE	IMM	OFFSET	+0

Instruction **S_DCACHE_INV**

Description Invalidate entire L1 constant cache.

Microcode SMRD Opcode 31 (0x1F)

1	**1**	**0**	**0**	**0**	OP	SDST	SBASE	IMM	OFFSET	+0

Instruction **S_DCACHE_INV_VOL**

Description Invalidate all volatile lines in L1 constant cache.

Microcode SMRD Opcode 29 (0x1D)

1	**1**	**0**	**0**	**0**	OP	SDST	SBASE	IMM	OFFSET	+0

Instruction **S_LOAD_DWORD**

Description Read one Dword from read-only constant memory through the constant cache (kcache).

```
m_offset = IMM ? OFFSET : SGPR[OFFSET]
m_addr = (SGPR[SBASE] + m_offset) & ~0x3
SGPR[SDST] = read_dword_from_kcache(m_addr)
```

Microcode SMRD Opcode 0 (0x0)

1	**1**	**0**	**0**	**0**	OP	SDST	SBASE	IMM	OFFSET	+0

Instruction **S_LOAD_DWORDX2**

Description Read two Dwords from read-only constant memory through the constant cache (kcache).

```
m_offset = IMM ? OFFSET : SGPR[OFFSET]
m_addr = (SGPR[SBASE * 2] + m_offset) & ~0x3
SGPR[SDST] = read_dword_from_kcache(m_addr)
SGPR[SDST+1] = read_dword_from_kcache(m_addr+4)
```

Microcode SMRD Opcode 1 (0x1)

1	**1**	**0**	**0**	**0**	OP	SDST	SBASE	IMM	OFFSET	+0

Instruction **S_LOAD_DWORDX4**

Description Read four Dwords from read-only constant memory through the constant cache (kcache).

```
m_offset = IMM ? OFFSET : SGPR[OFFSET]
m_addr = (SGPR[SBASE * 2] + m_offset) & ~0x3
SGPR[SDST] = read_dword_from_kcache(m_addr)
SGPR[SDST+1] = read_dword_from_kcache(m_addr+4)
SGPR[SDST+2] = read_dword_from_kcache(m_addr+8)
SGPR[SDST+3] = read_dword_from_kcache(m_addr+12)
```

Microcode SMRD Opcode 2 (0x2)

1	**1**	**0**	**0**	**0**	OP	SDST	SBASE	IMM	OFFSET	+0

Instruction **S_LOAD_DWORDX8**

Description Read eight Dwords from read-only constant memory through the constant cache (kcache).

```
m_offset = IMM ? OFFSET : SGPR[OFFSET]
m_addr = (SGPR[SBASE * 2] + m_offset) & ~0x3
SGPR[SDST] = read_dword_from_kcache(m_addr)
SGPR[SDST+1] = read_dword_from_kcache(m_addr+4)
SGPR[SDST+2] = read_dword_from_kcache(m_addr+8)
. . .
SGPR[SDST+7] = read_dword_from_kcache(m_addr+28)
```

Microcode SMRD Opcode 3 (0x3)

1	**1**	**0**	**0**	**0**	OP	SDST	SBASE	IMM	OFFSET	+0

Instruction **S_LOAD_DWORDX16**

Description Read 16 Dwords from read-only constant memory through the constant cache (kcache).

```
m_offset = IMM ? OFFSET : SGPR[OFFSET]
m_addr = (SGPR[SBASE * 2] + m_offset) & ~0x3
SGPR[SDST] = read_dword_from_kcache(m_addr)
SGPR[SDST+1] = read_dword_from_kcache(m_addr+4)
SGPR[SDST+2] = read_dword_from_kcache(m_addr+8)
. . .
SGPR[SDST+15] = read_dword_from_kcache(m_addr+60)
```

Microcode SMRD Opcode 4 (0x4)

1	**1**	**0**	**0**	**0**	OP	SDST	SBASE	IMM	OFFSET	+0

Instruction **S_MEMTIME**

Description Return current 64-bit timestamp.This “time” is a free-running clock counter based on the shader core clock.

Microcode SMRD Opcode 30 (0x1E)

1	**1**	**0**	**0**	**0**	OP	SDST	SBASE	IMM	OFFSET	+0

12.7 VOP2 Instructions

Instruction **`V_ADD_F32`**

Description Floating-point add.

D.f = S0.f + S1.f.

Microcode VOP2 Opcode 3 (0x3)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 259 (0x103)

<table>
<tr><td colspan="3">NEG</td><td colspan="2">OMOD</td><td colspan="2">SRC2</td><td colspan="3">SRC1</td><td colspan="2">SRC0</td><td>+4</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>OP</td><td>r</td><td>CLMP</td><td>ABS</td><td colspan="2">VDST</td><td>+0</td></tr>
</table>

Instruction **`V_ADD_I32`**

Description Unsigned integer add based on signed or unsigned integer components. Produces an unsigned carry out in VCC or a scalar register.

D.u = S0.u + S1.u; VCC=carry-out (VOP3:sgpr=carry-out).

Microcode VOP2 Opcode 37 (0x25)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3b Opcode 293 (0x125)

<table>
<tr><td colspan="3">NEG</td><td colspan="2">OMOD</td><td colspan="2">SRC2</td><td colspan="2">SRC1</td><td colspan="2">SRC0</td><td>+4</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>OP</td><td>r</td><td>SDST</td><td colspan="2">VDST</td><td>+0</td></tr>
</table>

Instruction **`V_ADDC_U32`**

Description Integer add based on unsigned integer components, with carry in. Produces a carry out in VCC or a scalar register.

Output carry bit of unsigned integer ADD.

D.u = S0.u + S1.u + VCC; VCC=carry-out (VOP3:sgpr=carry-out, S2.u=carry-in).

Microcode VOP2 Opcode 40 (0x28)

<table>
<tr><td>0</td><td>OP</td><td>VDST</td><td>VSRC1</td><td>SRC0</td><td>+0</td></tr>
</table>

Microcode VOP3b Opcode 296 (0x128)

<table>
<tr><td colspan="3">NEG</td><td colspan="2">OMOD</td><td colspan="2">SRC2</td><td colspan="2">SRC1</td><td>SRC0</td><td>+4</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>OP</td><td>r</td><td>SDST</td><td>VDST</td><td>+0</td></tr>
</table>

Instruction **`V_AND_B32`**

Description Logical bit-wise AND.

D.u = S0.u & S1.u.

Microcode VOP2 Opcode 27 (0x1B)

<table>
<tr><td>0</td><td>OP</td><td>VDST</td><td>VSRC1</td><td>SRC0</td><td>+0</td></tr>
</table>

Microcode VOP3a Opcode 283 (0x11B)

<table>
<tr><td colspan="3">NEG</td><td colspan="2">OMOD</td><td colspan="2">SRC2</td><td colspan="3">SRC1</td><td>SRC0</td><td>+4</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>OP</td><td>r</td><td>CLMP</td><td>ABS</td><td>VDST</td><td>+0</td></tr>
</table>

Instruction **`V_ASHR_I32`**

Description Scalar Arithmetic Shift Right. The sign bit is shifted into the vacated locations.

D.i = S0.i >> S1.i[4:0].

Microcode VOP2 Opcode 23 (0x17)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 279 (0x117)

NEG		OMOD	SRC2			SRC1			SRC0			+4
1	**1**	**0**	**1**	**0**	**0**	OP		r	CLMP	ABS	VDST	+0

Instruction **`V_ASHRREV_I32`**

Description D.i = S1.i >> S0.i[4:0]. The sign bit is shifted into the vacated bits.

Microcode VOP2 Opcode 24 (0x18)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 280 (0x118)

NEG		OMOD	SRC2			SRC1			SRC0			+4
1	**1**	**0**	**1**	**0**	**0**	OP		r	CLMP	ABS	VDST	+0

Instruction **V_BCNT_U32_B32**

Description Bit count.

D.u = CountOneBits(S0.u) + S1.u.

Microcode VOP2 Opcode 34 (0x22)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 290 (0x122)

NEG			OMOD		SRC2		SRC1		SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_BFM_B32**

Description Bitfield mask. Used before BFI to implement DX11 bitfield insert.

D.u = ((1<<S0.u[4:0])-1) << S1.u[4:0]; S0=bitfield_width, S1=bitfield_offset.

Microcode VOP2 Opcode 30 (0x1E)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 286 (0x11E)

NEG			OMOD		SRC2		SRC1		SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_CNDMASK_B32**

Description Boolean conditional based on bit mask from SGPRs or VCC.

D.u = VCC[i] ? S1.u : S0.u (i = threadID in wave); VOP3: specify VCC as a scalar GPR in S2.

Microcode VOP2 Opcode 0 (0x0)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 256 (0x100)

NEG		OMOD	SRC2			SRC1				SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST		+0

Instruction **V_CVT_PK_I16_I32**

Description DX signed 32-bit integer to signed 16-bit integer.

Overflow clamped to 0x7FFF. Underflow clamped to 0x8000.

D = {(i32→i16)S1.i, (i32→i16)S0.i}.

Microcode VOP2 Opcode 49 (0x31)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 305 (0x131)

NEG		OMOD	SRC2			SRC1				SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST		+0

Instruction **V_CVT_PK_U16_U32**

Description DX11 unsigned 32-bit integer to unsigned 16-bit integer.
Overflow clamped to 0xFFFF.
D = {(u32→u16)S1.u, (u32→u16)S0.u}.

Microcode VOP2 Opcode 48 (0x30)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 304 (0x130)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_CVT_PKACCUM_U8_F32**

Description f32→u8(s0.f), pack into byte(s1.u), of dst.

Microcode VOP2 Opcode 44 (0x2C)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 300 (0x12C)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_CVT_PKNORM_I16_F32**

Description DX Float32 to SNORM16, a signed, normalized 16-bit value.

D = {(snorm)S1.f, (snorm)S0.f}.

Microcode VOP2 Opcode 45 (0x2D)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 301 (0x12D)

NEG			OMOD		SRC2		SRC1			SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST		+0

Instruction **V_CVT_PKNORM_U16_F32**

Description DX Float32 to UNORM16, an unsigned, normalized 16-bit value.

D = {(unorm)S1.f, (unorm)S0.f}.

Microcode VOP2 Opcode 46 (0x2E)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 302 (0x12E)

NEG			OMOD		SRC2		SRC1			SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST		+0

Instruction **`V_CVT_PKRTZ_F16_F32`**

Description Convert two float 32 numbers into a single register holding two packed 16-bit floats.
D = {flt32_to_flt16(S1.f),flt32_to_flt16(S0.f)}, with round-toward-zero.

Microcode VOP2 Opcode 47 (0x2F)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 303 (0x12F)

NEG	OMOD	SRC2	SRC1	SRC0	+4

1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **`V_LDEXP_F32`**

Description C math library ldexp function.
Result = S0.f * (2 ^ S1.i)
So = float 32
S1 = signed integer

Microcode VOP2 Opcode 43 (0x2B)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 299 (0x12B)

NEG	OMOD	SRC2	SRC1	SRC0	+4

1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_LSHL_B32**

Description Scalar Logical Shift Left. Zero is shifted into the vacated locations.

D.u = S0.u << S1.u[4:0].

Microcode VOP2 Opcode 25 (0x19)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 281 (0x119)

NEG		OMOD	SRC2				SRC1			SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST		+0

Instruction **V_LSHLREV_B32**

Description D.u = S1.u << S0.u[4:0].

Microcode VOP2 Opcode 26 (0x1A)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 282 (0x11A)

NEG		OMOD	SRC2				SRC1			SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST		+0

Instruction **V_LSHR_B32**

Description Scalar Logical Shift Right. Zero is shifted into the vacated locations.

D.u = S0.u >> S1.u[4:0].

Microcode VOP2 Opcode 21 (0x15)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 277 (0x115)

NEG		OMOD		SRC2			SRC1			SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST		+0

Instruction **V_LSHRREV_B32**

Description D.u = S1.u >> S0.u[4:0].

Microcode VOP2 Opcode 22 (0x16)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 278 (0x116)

NEG		OMOD		SRC2			SRC1			SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST		+0

Instruction **V_MAC_F32**

Description D.f = S0.f * S1.f + D.f.

Microcode VOP2 Opcode 31 (0x1F)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 287 (0x11F)

NEG		OMOD		SRC2		SRC1			SRC0			+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST		+0

Instruction **V_MAC_LEGACY_F32**

Description D.f = S0.F * S1.f + D.f. (Note that "legacy" means that, unlike IEEE rules, 0 * anything = 0.)

Microcode VOP2 Opcode 6 (0x6)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 262 (0x106)

NEG		OMOD		SRC2		SRC1			SRC0			+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST		+0

Instruction **V_MADAK_F32**

Description D.f = S0.f * S1.f + K; K is a 32-bit literal constant.

Microcode VOP2 Opcode 33 (0x21)

LITERAL					+4
0	OP	VDST	VSRC1	SRC0	+0

Instruction **V_MADMK_F32**

Description D.f = S0.f * K + S1.f; K is a 32-bit literal constant.

Microcode VOP2 Opcode 32 (0x20)

LITERAL					+4
0	OP	VDST	VSRC1	SRC0	+0

Instruction **V_MAX_F32**

Description

```
if (ieee_mode)
   if (S0.f==sNaN          result = quiet(S0.f);
   else if (S1.f==sNaN   result = quiet(S1.f);
   else if (S0.f==NaN)   result = S1.f;
   else if (S1.f==NaN)   result = S0.f;
   else if (S0.f>S1.f)    result = S0.f;
   else                             result = S1.f;
else
   else if (S0.f==NaN)   result = S1.f;
   else if (S1.f==NaN)   result = S0.f;
   else if (S0.f>=S1.f)  result = S0.f;
   else                             result = S1.f;
```

Microcode VOP2 Opcode 16 (0x10)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 272 (0x110)

NEG		OMOD		SRC2			SRC1			SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST		+0

Instruction **V_MAX_I32**

Description Integer maximum based on signed integer components.

D.i = max(S0.i, S1.i).

Microcode VOP2 Opcode 18 (0x12)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 274 (0x112)

NEG		OMOD		SRC2		SRC1				SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST		+0

Instruction **V_MAX_LEGACY_F32**

Description Floating-point maximum.

```
If (S0.f >= S1.f)
    D.f = S0.f;
Else
    D.f = S1.f;
```

D.f = max(S0.f, S1.f) (DX9 rules for NaN).

Microcode VOP2 Opcode 14 (0xE)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 270 (0x10E)

NEG		OMOD		SRC2		SRC1				SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST		+0

Instruction **V_MAX_U32**

Description Integer maximum based on unsigned integer components.

```
If (S0.u >= S1.u)
    D.u = S0.u;
Else
    D.u = S1.u;
```

Microcode VOP2 Opcode 20 (0x14)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 276 (0x114)

NEG		OMOD		SRC2			SRC1				SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP		r	CLMP	ABS		VDST	+0

Instruction **V_MBCNT_HI_U32_B32**

Description Masked bit count of the upper 32 threads (threads 32-63). For each thread, this instruction returns the number of active threads which come before it.

ThreadMask = (1 << ThreadPosition) - 1;

D.u = CountOneBits(S0.u & ThreadMask[63:32]) + S1.u.

Microcode VOP2 Opcode 36 (0x24)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 292 (0x124)

NEG		OMOD		SRC2			SRC1				SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP		r	CLMP	ABS		VDST	+0

Instruction **V_MBCNT_LO_U32_B32**

Description Masked bit count set 32 low. ThreadPosition is the position of this thread in the wavefront (in 0..63).

ThreadMask = (1 << ThreadPosition) - 1;
D.u = CountOneBits(S0.u & ThreadMask[31:0]) + S1.u.

Microcode VOP2 Opcode 35 (0x23)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 291 (0x123)

NEG	OMOD	SRC2	SRC1	SRC0	+4

1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_MIN_F32**

Description

```
if (ieee_mode)
   if (S0.f==sNaN        result = quiet(S0.f);
   else if (S1.f==sNaN  result = quiet(S1.f);
   else if (S0.f==NaN)  result = S1.f;
   else if (S1.f==NaN)   result = S0.f;
   else if (S0.f<S1.f)    result = S0.f;
   else                              result = S1.f;
else
   else if (S0.f==NaN)    result = S1.f;
   else if (S1.f==NaN)    result = S0.f;
   else if (S0.f<S1.f)    result = S0.f;
   else                   result = S1.f;
```

Microcode VOP2 Opcode 15 (0xF)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 271 (0x10F)

NEG	OMOD	SRC2	SRC1	SRC0	+4

1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_MIN_I32**

Description Integer minimum based on signed integer components.

```
If (S0.i < S1.i)
    D.i = S0.i;
Else
    D.i = S1.i;
```

Microcode VOP2 Opcode 17 (0x11)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 273 (0x111)

NEG	OMOD	SRC2	SRC1	SRC0	+4

1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_MIN_LEGACY_F32**

Description Floating-point minimum.

```
If (S0.f < S1.f)
    D.f = S0.f;
Else
    D.f = S1.f;
```

D.f = min(S0.f, S1.f) (DX9 rules for NaN).

Microcode VOP2 Opcode 13 (0xD)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 269 (0x10D)

NEG	OMOD	SRC2	SRC1	SRC0	+4

1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_MIN_U32**

Description Integer minimum based on signed unsigned integer components.

```
If (S0.u < S1.u)
    D.u = S0.u;
Else
    D.u = S1.u;
```

Microcode VOP2 Opcode 19 (0x13)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 275 (0x113)

NEG	OMOD	SRC2	SRC1	SRC0	+4

1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_MUL_F32**

Description Floating point multiply. Uses IEEE rules for 0*anything.

D.f = S0.f * S1.f.

Microcode VOP2 Opcode 8 (0x8)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 264 (0x108)

NEG	OMOD	SRC2	SRC1	SRC0	+4

1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_MUL_HI_I32_I24**

Description 24-bit signed integer multiply.

S0 and S1 are treated as 24-bit signed integers. Bits [31:24] are ignored. The result represents the high-order 16 bits of the 48-bit multiply result, sign extended to 32 bits:

D.i = (S0.i[23:0] * S1.i[23:0])>>32.

Microcode VOP2 Opcode 10 (0xA)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 266 (0x10A)

NEG			OMOD		SRC2		SRC1			SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST		+0

Instruction **V_MUL_HI_U32_U24**

Description 24-bit unsigned integer multiply.

S0 and S1 are treated as 24-bit unsigned integers. Bits [31:24]are ignored. The result represents the high-order 16 bits of the 48-bit multiply result: {16'b0, mul_result[47:32]}.

D.i = (S0.u[23:0] * S1.u[23:0])>>32.

Microcode VOP2 Opcode 12 (0xC)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 268 (0x10C)

NEG			OMOD		SRC2		SRC1			SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST		+0

Instruction **V_MUL_I32_I24**

Description 24 bit signed integer multiply

Src a and b treated as 24 bit signed integers. Bits [31:24] ignored. The result represents the low-order 32 bits of the 48 bit multiply result: mul_result[31:0].

D.i = S0.i[23:0] * S1.i[23:0].

Microcode VOP2 Opcode 9 (0x9)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 265 (0x109)

NEG		OMOD		SRC2			SRC1			SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST		+0

Instruction **V_MUL_LEGACY_F32**

Description Floating-point multiply.

D.f = S0.f * S1.f (DX9 rules, 0.0*x = 0.0).

Microcode VOP2 Opcode 7 (0x7)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 263 (0x107)

NEG		OMOD		SRC2			SRC1			SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST		+0

Instruction **V_MUL_U32_U24**

Description 24-bit unsigned integer multiply.

S0 and S1 are treated as 24-bit unsigned integers. Bits [31:24] are ignored. The result represents the low-order 32 bits of the 48-bit multiply result: mul_result[31:0].

D.u = S0.u[23:0] * S1.u[23:0].

Microcode VOP2 Opcode 11 (0xB)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 267 (0x10B)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_OR_B32**

Description Logical bit-wise OR.

D.u = S0.u | S1.u.

Microcode VOP2 Opcode 28 (0x1C)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 284 (0x11C)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_READLANE_B32**

Description Copy one VGPR value to one SGPR. Dst = SGPR-dest, Src0 = Source Data (VGPR# or M0(lds-direct)), Src1 = Lane Select (SGPR or M0). Ignores exec mask. A lane corresponds to one thread in a wavefront.

The VOP3a version does not apply input or output modifiers.

Microcode VOP2 Opcode 1 (0x1)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 257 (0x101)

<table>
<tr><td colspan="3">NEG</td><td colspan="2">OMOD</td><td colspan="2">SRC2</td><td colspan="4">SRC1</td><td colspan="2">SRC0</td><td>+4</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td colspan="2">OP</td><td>r</td><td>CLMP</td><td colspan="2">ABS</td><td>VDST</td><td>+0</td></tr>
</table>

Instruction **V_SUB_F32**

Description D.f = S0.f - S1.f.

Microcode VOP2 Opcode 4 (0x4)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 260 (0x104)

<table>
<tr><td colspan="3">NEG</td><td colspan="2">OMOD</td><td colspan="2">SRC2</td><td colspan="4">SRC1</td><td colspan="2">SRC0</td><td>+4</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td colspan="2">OP</td><td>r</td><td>CLMP</td><td colspan="2">ABS</td><td>VDST</td><td>+0</td></tr>
</table>

Instruction	**V_SUB_I32**
Description	Unsigned integer subtract based on unsigned integer components. Produces an unsigned borrow out in VCC or a scalar register. D.u = S0.u - S1.u; VCC=carry-out (VOP3:sgpr=carry-out).

Microcode VOP2 Opcode 38 (0x26)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3b Opcode 294 (0x126)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	SDST		VDST	+0

Instruction	**V_SUBB_U32**
Description	Integer subtract based on signed or unsigned integer components, with borrow in. Produces a borrow out in VCC or a scalar register. D.u = S0.u - S1.u - VCC; VCC=carry-out (VOP3:sgpr=carry-out, S2.u=carry-in).

Microcode VOP2 Opcode 41 (0x29)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3b Opcode 297 (0x129)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	SDST		VDST	+0

Instruction **`V_SUBBREV_U32`**

Description D.u = S1.u - S0.u - VCC; VCC=carry-out (VOP3:sgpr=carry-out, S2.u=carry-in).

Microcode VOP2 Opcode 42 (0x2A)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3b Opcode 298 (0x12A)

NEG		OMOD		SRC2			SRC1		SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	SDST		VDST	+0

Instruction **`V_SUBREV_F32`**

Description D.f = S1.f - S0.f.

Microcode VOP2 Opcode 5 (0x5)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 261 (0x105)

NEG		OMOD		SRC2			SRC1			SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS		VDST	+0

Instruction **V_SUBREV_I32**

Description D.u = S1.u - S0.u; VCC=carry-out (VOP3:sgpr=carry-out).

Microcode VOP2 Opcode 39 (0x27)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3b Opcode 295 (0x127)

NEG			OMOD		SRC2		SRC1		SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	SDST		VDST	+0

Instruction **V_WRITELANE_B32**

Description Write value into one VGPR one one lane. Dst = VGPR-dest, Src0 = Source Data (SGPR, M0, exec, or constants), Src1 = Lane Select (SGPR or M0). Ignores exec mask.

Microcode VOP2 Opcode 2 (0x2)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 258 (0x102)

NEG			OMOD		SRC2		SRC1			SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS		VDST	+0

Instruction **`V_XOR_B32`**

Description Logical bit-wise XOR.

D.u = S0.u ^ S1.u.

Microcode VOP2 Opcode 29 (0x1D)

0	OP	VDST	VSRC1	SRC0	+0

Microcode VOP3a Opcode 285 (0x11D)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

12.8 VOP1 Instructions

Instruction **V_BFREV_B32**

Description Bitfield reverse.

D.u[31:0] = S0.u[0:31].

Microcode VOP1 Opcode 56 (0x38)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 440 (0x1B8)

NEG		OMOD			SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_CEIL_F32**

Description Floating point ceiling function.

D.f = ceil(S0.f). Implemented as: D.f = trunc(S0.f);

if (S0 > 0.0 && S0 != D), D += 1.0.

Microcode VOP1 Opcode 34 (0x22)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 418 (0x1A2)

NEG		OMOD			SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_CEIL_F64**

Description 64-bit floating-point ceiling.

D.d = trunc(S0.d); if (S0.d > 0.0 && S0.d != D.d), D.d += 1.0.

Microcode VOP1 Opcode 24 (0x18)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Instruction **V_CLREXCP**

Description Clear wave's exception state in SIMD.

Microcode VOP1 Opcode 65 (0x41)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 449 (0x1C1)

NEG	OMOD	SRC2	SRC1	SRC0	+4

1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction	**`V_COS_F32`**
Description	Cos function. Input must be normalized from radians by dividing by 2*PI. Valid input domain [-256, +256], which corresponds to an un-normalized input domain [-512*PI, +512*PI]. Out-of-range input results in float 1. D.f = cos(S0.f).

Microcode VOP1 Opcode 54 (0x36)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 438 (0x1B6)

NEG			OMOD		SRC2		SRC1			SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP		r	CLMP	ABS	VDST	+0

Instruction	**`V_CVT_F16_F32`**
Description	Float32 to Float16. D.f16 = flt32_to_flt16(S0.f).

Microcode VOP1 Opcode 10 (0xA)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 394 (0x18A)

NEG			OMOD		SRC2		SRC1			SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP		r	CLMP	ABS	VDST	+0

Instruction	**`V_CVT_F32_F16`**
Description	DX11 Float16 to Float32. D.f = flt16_to_flt32(S0.f16).

Microcode VOP1 Opcode 11 (0xB)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 395 (0x18B)

NEG			OMOD		SRC2		SRC1			SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP		r	CLMP	ABS	VDST	+0

Instruction	**`V_CVT_F32_F64`**
Description	Convert Double Precision Float to Single Precision Float. Overflows obey round mode rules. Infinity is exact. D.f = (float)S0.d.

Microcode VOP1 Opcode 15 (0xF)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 399 (0x18F)

NEG			OMOD		SRC2		SRC1			SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP		r	CLMP	ABS	VDST	+0

Instruction **`V_CVT_F32_I32`**

Description The input is interpreted as a signed integer value and converted to a float.

D.f = (float)S0.i.

Microcode VOP1 Opcode 5 (0x5)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 389 (0x185)

<table>
<tr><td colspan="3">NEG</td><td colspan="2">OMOD</td><td colspan="2">SRC2</td><td colspan="4">SRC1</td><td colspan="2">SRC0</td><td>+4</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td colspan="2">OP</td><td>r</td><td>CLMP</td><td colspan="2">ABS</td><td>VDST</td><td>+0</td></tr>
</table>

Instruction **`V_CVT_F32_U32`**

Description The input is interpreted as an unsigned integer value and converted to a float.

D.f = (float)S0.u.

Microcode VOP1 Opcode 6 (0x6)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 390 (0x186)

<table>
<tr><td colspan="3">NEG</td><td colspan="2">OMOD</td><td colspan="2">SRC2</td><td colspan="4">SRC1</td><td colspan="2">SRC0</td><td>+4</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td colspan="2">OP</td><td>r</td><td>CLMP</td><td colspan="2">ABS</td><td>VDST</td><td>+0</td></tr>
</table>

Instruction **`V_CVT_F32_UBYTE0`**

Description Byte 0 to float. Perform unsigned int to float conversion on byte 0 of S0.
D.f = UINT2FLT(S0.u[7:0]).

Microcode VOP1 Opcode 17 (0x11)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 401 (0x191)

NEG	OMOD	SRC2	SRC1	SRC0	+4						
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **`V_CVT_F32_UBYTE1`**

Description Byte 1 to float. Perform unsigned int to float conversion on byte 1 of S0.
D.f = UINT2FLT(S0.u[15:8]).

Microcode VOP1 Opcode 18 (0x12)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 402 (0x192)

NEG	OMOD	SRC2	SRC1	SRC0	+4						
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_CVT_F32_UBYTE2**

Description Byte 2 to float. Perform unsigned int to float conversion on byte 2 of S0.
D.f = UINT2FLT(S0.u[23:16]).

Microcode VOP1 Opcode 19 (0x13)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 403 (0x193)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_CVT_F32_UBYTE3**

Description Byte 3 to float. Perform unsigned int to float conversion on byte 3 of S0.
D.f = UINT2FLT(S0.u[31:24]).

Microcode VOP1 Opcode 20 (0x14)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 404 (0x194)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction	**`V_CVT_F64_F32`**
Description	Convert Single Precision Float to Double Precision Float. D.d = (double)S0.f.

Microcode VOP1 Opcode 16 (0x10)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 400 (0x190)

NEG		OMOD		SRC2		SRC1			SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction	**`V_CVT_F64_I32`**
Description	Convert Signed Integer to Double Precision Float. D.f = (float)S0.i.

Microcode VOP1 Opcode 4 (0x4)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 388 (0x184)

NEG		OMOD		SRC2		SRC1			SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_CVT_F64_U32**

Description Convert Unsigned Integer to Double Precision Float.

D.d = (double)S0.u.

Microcode VOP1 Opcode 22 (0x16)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 406 (0x196)

NEG		OMOD		SRC2		SRC1			SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_CVT_FLR_I32_F32**

Description Float input is converted to a signed integer value using floor function. Float magnitudes too great to be represented by an integer float (unbiased exponent > 30) saturate to max_int or -max_int.

D.i = (int)floor(S0.f).

Microcode VOP1 Opcode 13 (0xD)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 397 (0x18D)

NEG		OMOD		SRC2		SRC1			SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction	**V_CVT_I32_F32**
Description	Float input is converted to a signed integer using truncation. Float magnitudes too great to be represented by an integer float (unbiased exponent > 30) saturate to max_int or -max_int. Special case number handling: inf → max_int -inf → -max_int NaN & –Nan & 0 & -0 → 0 D.i = (int)S0.f.

Microcode VOP1 Opcode 8 (0x8)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 392 (0x188)

NEG		OMOD	SRC2			SRC1			SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction	**V_CVT_I32_F64**
Description	Covert Double Precision Float to Signed Integer. Truncate (round-to-zero) only. Other round modes require a rne_f64, ceil_f64 or floor_f64 pre-op. Float magnitudes too great to be represented by an integer float (unbiased exponent > 30) saturate to max_int or -max_int. Special case number handling: inf → max_int -inf → -max_int NaN & –Nan & 0 & -0 → 0 D.i = (int)S0.d.

Microcode VOP1 Opcode 3 (0x3)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 387 (0x183)

NEG		OMOD		SRC2		SRC1			SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_CVT_OFF_F32_I4**

Description 4-bit signed int to 32-bit float. For interpolation in shader.

S0	Result
1000	-0.5f
1001	-0.4375f
1010	-0.375f
1011	-0.3125f
1100	-0.25f
1101	-0.1875f
1110	-0.125f
1111	-0.0625f
0000	0.0f
0001	0.0625f
0010	0.125f
0011	0.1875f
0100	0.25f
0101	0.3125f
0110	0.375f
0111	0.4375f

Microcode VOP1 Opcode 14 (0xE)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 398 (0x18E)

<table>
<tr><td colspan="3">NEG</td><td colspan="2">OMOD</td><td colspan="2">SRC2</td><td colspan="3">SRC1</td><td>SRC0</td><td>+4</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>OP</td><td>r</td><td>CLMP</td><td>ABS</td><td>VDST</td><td>+0</td></tr>
</table>

Instruction **V_CVT_RPI_I32_F32**

Description Float input is converted to a signed integer value using round to positive infinity tiebreaker for 0.5. Float magnitudes too great to be represented by an integer float (unbiased exponent > 30) saturate to max_int or -max_int.

D.i = (int)floor(S0.f + 0.5).

Microcode VOP1 Opcode 12 (0xC)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 396 (0x18C)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_CVT_U32_F32**

Description Input is converted to an unsigned integer value using truncation. Positive float magnitudes too great to be represented by an unsigned integer float (unbiased exponent > 31) saturate to max_uint.

Special number handling:

-inf & NaN & 0 & -0 → 0

Inf → max_uint

D.u = (unsigned)S0.f.

Microcode VOP1 Opcode 7 (0x7)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 391 (0x187)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_CVT_U32_F64**

Description Covert Double Precision Float to Unsigned Integer

Truncate (round-to-zero) only. Other round modes require a rne_f64, ceil_f64 or floor_f64 pre-op. Positive float magnitudes too great to be represented by an unsigned integer float (unbiased exponent > 31) saturate to max_uint.

Special number handling:

-inf & NaN & 0 & -0 → 0

Inf → max_uint

D.u = (uint)S0.d.

Microcode VOP1 Opcode 21 (0x15)

0	1	1	1	1	1	1	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 405 (0x195)

NEG		OMOD		SRC2		SRC1			SRC0		+4
1	1	0	1	0	0	OP	r	CLMP	ABS	VDST	+0

Instruction **V_EXP_F32**

Description Base2 exponent function.

```
If (Arg1 == 0.0f) {
  Result = 1.0f;
}
Else {
  Result = Approximate2ToX(Arg1);
}
```

Microcode VOP1 Opcode 37 (0x25)

0	1	1	1	1	1	1	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 421 (0x1A5)

NEG		OMOD		SRC2		SRC1			SRC0		+4
1	1	0	1	0	0	OP	r	CLMP	ABS	VDST	+0

Instruction **`V_EXP_LEGACY_F32`**

Description Return 2^(argument) floating-point value, using the same precision as Sea Islands.

D.f = pow(2.0, S0.f). Same as Sea Islands.

Microcode VOP1 Opcode 70 (0x46)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Instruction **`V_FFBH_I32`**

Description Find first bit signed high.

Find first bit set in a positive integer from MSB, or find first bit clear in a negative integer from MSB

D.u = position of first bit different from sign bit in S0 from MSB; D=0xFFFFFFFF if S0==0 or 0xFFFFFFFF.

Microcode VOP1 Opcode 59 (0x3B)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 443 (0x1BB)

NEG		OMOD		SRC2			SRC1		SRC0			+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST		+0

Instruction	**`V_FFBH_U32`**
Description	Find first bit high. D.u = position of first 1 in S0 from MSB; D=0xFFFFFFFF if S0==0.

Microcode VOP1 Opcode 57 (0x39)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 441 (0x1B9)

NEG			OMOD		SRC2		SRC1			SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST		+0

Instruction	**`V_FFBL_B32`**
Description	Find first bit low. D.u = position of first 1 in S0 from LSB; D=0xFFFFFFFF if S0==0.

Microcode VOP1 Opcode 58 (0x3A)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 442 (0x1BA)

NEG			OMOD		SRC2		SRC1			SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST		+0

Instruction **`V_FLOOR_F32`**

Description Floating-point floor function.

D.f = trunc(S0); if ((S0 < 0.0) && (S0 != D)) D += -1.0.

Microcode VOP1 Opcode 36 (0x24)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 420 (0x1A4)

NEG	OMOD	SRC2	SRC1	SRC0	+4

1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **`V_FLOOR_F64`**

Description 64-bit floating-point floor.

D.d = trunc(S0.d); if (S0.d < 0.0 && S0.d != D.d), D.d += -1.0.

Microcode VOP1 Opcode 26 (0x1A)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Instruction **V_FRACT_F32**

Description Floating point 'fractional' part of S0.f.

D.f = S0.f - floor(S0.f).

Microcode VOP1 Opcode 32 (0x20)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 416 (0x1A0)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_FRACT_F64**

Description Double-precision fractional part of Arg1.

Double result written to two consecutive GPRs; the instruction Dest specifies the lesser of the two.

D.d = FRAC64(S0.d);

Return fractional part of input as double [0.0 - 1.0).

Microcode VOP1 Opcode 62 (0x3E)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 446 (0x1BE)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction	**`V_FREXP_EXP_I32_F32`**
Description	C math library frexp function. Returns the exponent of a single precision float input, such that: original single float = significand * $2^{exponent}$. D.f = 2’s complement (exponent(S0.f) – 127 +1).

Microcode VOP1 Opcode 63 (0x3F)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 447 (0x1BF)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction	**`V_FREXP_EXP_I32_F64`**
Description	C++ FREXP math function. Returns exponent of double precision float input, such that: original double float = significand * $2^{exponent}$. D.i = 2’s complement (exponent(S0.d) – 1023 +1).

Microcode VOP1 Opcode 60 (0x3C)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 444 (0x1BC)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction	**V_FREXP_MANT_F32**
Description	C math library frexp function. Returns binary significand of single precision float input, such that: original single float = significand * $2^{exponent}$. D.f =Mantissa(S0.f). D.f range(-1.0,-0.5] or [0.5,1.0).

Microcode VOP1 Opcode 64 (0x40)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 448 (0x1C0)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction	**V_FREXP_MANT_F64**
Description	C++ FREXP math function. Returns binary significand of double precision float input, such that original double float = significand * $2^{exponent}$. D.d =Mantissa(S0.d). D.d range(-1.0,-0.5] or [0.5,1.0).

Microcode VOP1 Opcode 61 (0x3D)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 445 (0x1BD)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_LOG_CLAMP_F32**

Description Base2 log function.

The clamp prevents infinite results, clamping infinities to max_float.

```
If (Arg1 == 1.0f) {
  Result = 0.0f;
}
Else {
  Result = LOG_IEEE(Arg1)
// clamp result
if (Result == -INFINITY) {
  Result = -MAX_FLOAT;
}
```

Microcode VOP1 Opcode 38 (0x26)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 422 (0x1A6)

NEG		OMOD		SRC2			SRC1			SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST		+0

Instruction **V_LOG_F32**

Description Base2 log function.

D.f = log2(S0.f).

Microcode VOP1 Opcode 39 (0x27)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 423 (0x1A7)

NEG		OMOD		SRC2			SRC1			SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST		+0

Instruction **V_LOG_F32**

Description Base2 log function.

D.f = log2(S0.f).

Microcode VOP1 Opcode 39 (0x27)

0	1	1	1	1	1	1	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 423 (0x1A7)

NEG			OMOD		SRC2		SRC1			SRC0		+4
1	1	0	1	0	0	OP		r	CLMP	ABS	VDST	+0

Instruction **V_LOG_LEGACY_F32**

Description Return the algorithm of a 32-bit floating point value, using the same precision as Sea Islands.

D.f = log2(S0.f). Base 2 logarithm. Same as Sea Islands.

Microcode VOP1 Opcode 69 (0x45)

0	1	1	1	1	1	1	VDST	OP	SRC0	+0

Instruction **`V_MOV_B32`**

Description Single operand move instruction. Allows denorms in and out, regardless of denorm mode, in both single and double precision designs.

D.u = S0.u.

Microcode VOP1 Opcode 1 (0x1)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 385 (0x181)

NEG	OMOD	SRC2	SRC1	SRC0	+4

1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **`V_MOVRELD_B32`**

Description VGPR[D.u + M0.u] = VGPR[S0.u].

Microcode VOP1 Opcode 66 (0x42)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 450 (0x1C2)

NEG	OMOD	SRC2	SRC1	SRC0	+4

1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **`V_MOVRELSD_B32`**

Description VGPR[D.u + M0.u] = VGPR[S0.u + M0.u].

Microcode VOP1 Opcode 68 (0x44)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 452 (0x1C4)

NEG		OMOD		SRC2		SRC1			SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **`V_NOP`**

Description Do nothing.

Microcode VOP1 Opcode 0 (0x0)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 384 (0x180)

NEG		OMOD		SRC2		SRC1			SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_NOT_B32**

Description Logical bit-wise NOT.

D.u = ~S0.u.

Microcode VOP1 Opcode 55 (0x37)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 439 (0x1B7)

NEG	OMOD	SRC2	SRC1	SRC0	+4

1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_RCP_CLAMP_F32**

Description Reciprocal, < 1 ulp error.

The clamp prevents infinite results, clamping infinities to max_float.

This reciprocal approximation converges to < 0.5 ulp error with one newton rhapson performed with two fused multiple adds (FMAs).

D.f = 1.0 / S0.f, result clamped to +-max_float.

Microcode VOP1 Opcode 40 (0x28)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 424 (0x1A8)

NEG	OMOD	SRC2	SRC1	SRC0	+4

1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction	**`V_RCP_CLAMP_F64`**
Description	Double reciprocal. The clamp prevents infinite results, clamping infinities to max_float. Inputs from two consecutive GPRs, instruction source specifies less of the two. Double result are written to two consecutive GPRs, instruction Dest specifies the lesser of the two. D.f = 1.0 / (S0.f), result clamped to +-max_float.

Microcode VOP1 Opcode 48 (0x30)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 432 (0x1B0)

NEG		OMOD		SRC2			SRC1			SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST		+0

Instruction	**`V_RCP_F32`**
Description	Reciprocal, < 1 ulp error. This reciprocal approximation converges to < 0.5 ulp error with one newton rhapson performed with two fused multiple adds (FMAs). D.f = 1.0 / S0.f.

Microcode VOP1 Opcode 42 (0x2A)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 426 (0x1AA)

NEG		OMOD		SRC2			SRC1			SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST		+0

Instruction **V_RCP_F64**

Description Double reciprocal.

Inputs from two consecutive GPRs, the instruction source specifies less of the two. Double result written to two consecutive GPRs; the instruction Dest specifies the lesser of the two.

D.d = 1.0 / (S0.d).

Microcode VOP1 Opcode 47 (0x2F)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 431 (0x1AF)

NEG			OMOD			SRC2	SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_RCP_IFLAG_F32**

Description Reciprocal.

Signals exceptions using integer divide by zero flag only; does not trigger any floating point exceptions. To be used in an integer reciprocal macro by the compiler.

D.f = 1.0 / S0.f, only integer div_by_zero flag can be raised.

Microcode VOP1 Opcode 43 (0x2B)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 427 (0x1AB)

NEG			OMOD			SRC2	SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction	**V_RCP_LEGACY_F32**
Description	Reciprocal, < 1 ulp error.

Legacy refers to the behavior that rcp_legacy(+/-0)=+0.

This reciprocal approximation converges to < 0.5 ulp error with one newton rhapson performed with two fused multiple adds (FMAs).

```
If (Arg1 == 1.0f) {
  Result = 1.0f;
}
Else If (Arg1 == 0.0f) {
  Result = 0.0f;
}
Else {
  Result = RECIP_IEEE(Arg1);
}
// clamp result
if (Result == -INFINITY) {
  Result = -ZERO;
}
if (Result == +INFINITY) {
  Result = +ZERO;
}
```

Microcode VOP1 Opcode 41 (0x29)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 425 (0x1A9)

NEG		OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_READFIRSTLANE_B32**

Description copy one VGPR value to one SGPR. Dst = SGPR-dest, Src0 = Source Data (VGPR# or M0(lds-direct)), Lane# = FindFirst1fromLSB(exec) (lane = 0 if exec is zero). Ignores exec mask.

Microcode VOP1 Opcode 2 (0x2)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 386 (0x182)

NEG	OMOD	SRC2	SRC1	SRC0	+4

1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_RNDNE_F32**

Description Floating-point Round-to-Nearest-Even Integer.

D.f = round_nearest_even(S0.f).

Microcode VOP1 Opcode 35 (0x23)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 419 (0x1A3)

NEG	OMOD	SRC2	SRC1	SRC0	+4

1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_RNDNE_F64**

Description 64-bit floating-point round-to-nearest-even.

D.d = round_nearest_even(S0.d).

Microcode VOP1 Opcode 25 (0x19)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Instruction **V_RSQ_CLAMP_F32**

Description Reciprocal square root.

The clamp prevents infinite results, clamping infinities to max_float.

D.f = 1.0 / sqrt(S0.f), result clamped to +-max_float.

Microcode VOP1 Opcode 44 (0x2C)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 428 (0x1AC)

NEG	OMOD	SRC2	SRC1	SRC0	+4						
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction	**`V_RSQ_CLAMP_F64`**
Description	Double reciprocal square root. The clamp prevents infinite results, clamping infinities to max_float. Inputs from two consecutive GPRs, the instruction source specifies the lesser of the two. Double result written to two consecutive GPRs, the instruction Dest specifies the lesser of the two. D.d = 1.0 / sqrt(S0.d), result clamped to +-max_float.

Microcode VOP1 Opcode 50 (0x32)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 434 (0x1B2)

NEG		OMOD	SRC2		SRC1				SRC0			+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST		+0

Instruction	**`V_RSQ_F32`**
Description	Reciprocal square roots. D.f = 1.0 / sqrt(S0.f).

Microcode VOP1 Opcode 46 (0x2E)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 430 (0x1AE)

NEG		OMOD	SRC2		SRC1				SRC0			+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST		+0

Instruction **`V_RSQ_F64`**

Description Double reciprocal square root.

Inputs from two consecutive GPRs; the instruction source specifies the lesser of the two. The double result is written to two consecutive GPRs; the instruction Dest specifies the lesser of the two.

D.f = 1.0 / sqrt(S0.f).

Microcode VOP1 Opcode 49 (0x31)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 433 (0x1B1)

NEG	OMOD	SRC2	SRC1	SRC0	+4

1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **`V_RSQ_LEGACY_F32`**

Description Reciprocal square root.

Legacy refers to the behavior that rsq_legacy(+/-0)=+0.

The clamp prevents infinite results, clamping infinities to max_float.

D.f = 1.0 / sqrt(S0.f).

Microcode VOP1 Opcode 45 (0x2D)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 429 (0x1AD)

NEG	OMOD	SRC2	SRC1	SRC0	+4

1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_SIN_F32**

Description Sin function.

Input must be normalized from radians by dividing by 2*PI.

Valid input domain [-256, +256], which corresponds to an un-normalized input domain [-512*PI, +512*PI].

Out of range input results in float 0.

D.f = sin(S0.f).

Microcode VOP1 Opcode 53 (0x35)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 437 (0x1B5)

NEG			OMOD		SRC2		SRC1			SRC0			+4
1	**1**	**0**	**1**	**0**	**0**	OP		r	CLMP	ABS	VDST		+0

Instruction **V_SQRT_F32**

Description Square root. Useful for normal compression.

D.f = sqrt(S0.f).

Microcode VOP1 Opcode 51 (0x33)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 435 (0x1B3)

NEG			OMOD		SRC2		SRC1			SRC0			+4
1	**1**	**0**	**1**	**0**	**0**	OP		r	CLMP	ABS	VDST		+0

Instruction **`V_SQRT_F64`**

Description D.d = sqrt(S0.d).

Microcode VOP1 Opcode 52 (0x34)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 436 (0x1B4)

NEG	OMOD	SRC2	SRC1	SRC0	+4					
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **`V_TRUNC_F32`**

Description Floating point ‘integer’ part of S0.f.
D.f = trunc(S0.f), return integer part of S0.

Microcode VOP1 Opcode 33 (0x21)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

Microcode VOP3a Opcode 417 (0x1A1)

NEG	OMOD	SRC2	SRC1	SRC0	+4						
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction	**`V_TRUNC_F64`**
Description	Truncate a 64-bit floating-point value, and return the resulting integer value. D.d = trunc(S0.d), return integer part of S0.d.

Microcode VOP1 Opcode 23 (0x17)

0	**1**	**1**	**1**	**1**	**1**	**1**	VDST	OP	SRC0	+0

12.9 VOPC Instructions

The bitfield map for VOPC is:

0	1	1	1	1	1	0	OP	VSRC1	SRC0	+0

where:

SRC0 = First operand for instruction.
VSRC1 = Second operand for instruction.
OP = Instructions.

All VOPC instructions are also part of VOP3b microcode format, for which the bitfield is:

NEG	OMOD	SRC2	SRC1	SRC0	+4

1	1	0	1	0	0	OP	r	SDST	VDST	+0

where:

VDST = Destination for instruction in the VGPR.
SDST = Scalar general-purpose register.
OP = Instructions.
SRC0 = First operand for instruction.
SRC1 = Second operand for instruction.
SRC2 = Third operand for instruction. Unused in VOPC instructions.
OMOD = Output modifier for instruction. Unused in VOPC instructions.
NEG = Floating-point negation.

The first eight VOPC instructions have `{OP16}` embedded in them. This refers to each of the compare operations listed below.

Compare Operation	Opcode Offset	Description
F	0	D.u = 0
LT	1	D.u = (S0 < S1)
EQ	2	D.u = (S0 == S1)
LE	3	D.u = (S0 <= S1)
GT	4	D.u = (S0 > S1)
LG	5	D.u = (S0 <> S1)
GE	6	D.u = (S0 >= S1)
O	7	D.u = (!isNaN(S0) && !isNaN(S1))
U	8	D.u = (!isNaN(S0) || !isNaN(S1))
NGE	9	D.u = !(S0 >= S1)
NLG	10	D.u = !(S0 <> S1)

NGT	11	D.u = !(S0 > S1)
NLE	12	D.u = !(S0 <= S1)
NEQ	13	D.u = !(S0 == S1)
NLT	14	D.u = !(S0 < S1)
TRU	15	D.u = 1

Table 12.1 VOPC Instructions with 16 Compare Operations

Instruction	Description	Hex Range
`V_CMP_{OP16}_F32`	Signal on sNaN input only.	0x00 to 0x0F
`V_CMPX_{OP16}_F32`	Signal on sNaN input only. Also write EXEC.	0x10 to 0x1F
`V_CMP_{OP16}_F64`	Signal on sNaN input only.	0x20 to 0x2F
`V_CMPX_{OP16}_F64`	Signal on sNaN input only. Also write EXEC.	0x30 to 0x3F
`V_CMPS_{OP16}_F32`	Signal on any NaN.	0x40 to 0x4F
`V_CMPSX_{OP16}_F32`	Signal on any NaN. Also write EXEC.	0x50 to 0x5F
`V_CMPS_{OP16}_F64`	Signal on any NaN.	0x60 to 0x6F
`V_CMPSX_{OP16}_F64`	Signal on any NaN. Also write EXEC.	0x70 to 0x7F

The second eight VOPC instructions have `{OP8}` embedded in them. This refers to each of the compare operations listed below.

Compare Operation	Opcode Offset	Description
F	0	D.u = 0
LT	1	D.u = (S0 < S1)
EQ	2	D.u = (S0 == S1)
LE	3	D.u = (S0 <= S1)
GT	4	D.u = (S0 > S1)
LG	5	D.u = (S0 <> S1)
GE	6	D.u = (S0 >= S1)
TRU	7	D.u = 1

Table 12.2 VOPC Instructions with Eight Compare Operations

Instruction	Description	Hex Range
`V_CMP_{OP8}_I32`	On 32-bit integers.	0x80 to 0x87
`V_CMPX_{OP8}_I32`	Also write EXEC.	0x90 to 0x97
`V_CMP_{OP8}_I64`	On 64-bit integers.	0xA0 to 0xA7
`V_CMPX_{OP8}_I64`	Also write EXEC.	0xB0 to 0xB7
`V_CMP_{OP8}_U32`	On unsigned 32-bit intergers.	0xC0 to 0xC7
`V_CMPX_{OP8}_U32`	Also write EXEC.	0xD0 to 0xD7
`V_CMP_{OP8}_U64`	On unsigned 64-bit integers.	0xE0 to 0xE7
`V_CMPX_{OP8}_U64`	Also write EXEC.	0xF0 to 0xF7

The final instructions for VOPC are four CLASS instructions.

Table 12.3 VOPC CLASS Instructions

Instruction	Description	Hex
V_CMP_CLASS_F32	D = IEEE numeric class function specified in S1.u, performed on S0.f.	0x88
V_CMPX_CLASS_F32	D = IEEE numeric class function specified in S1.u, performed on S0.f. Also write EXEC.	0x98
V_CMP_CLASS_F64	D = IEEE numeric class function specified in S1.u, performed on S0.d. Result is single bit Boolean for each thread, aggregrated across wavefront and returned to SQ. Result is true if Arg1 is a member of any of the classes indicated by the mask (Arg2). mask[0] - signalingNaN mask[1] - quietNaN mask[2] - negativeInfinity mask[3] - negativeNormal mask[4] - negativeSubnormal mask[5] - negativeZero mask[6] - positiveZero mask[7] - positiveSubnormal mask[8] - positiveNormal mask[9] - positiveInfinity There is no vector result written to a gpr, and no vector feedback path for this opcode. This opcode does not raise exceptions under any circumstances.	0xA8
V_CMPX_CLASS_F64	D = IEEE numeric class function specified in S1.u, performed on S0.d. Also write EXEC. Result is single bit Boolean for each thread, aggregrated across wavefront and returned to SQ. Result is true if Arg1 is a member of any of the classes indicated by the mask (Arg2). mask[0] - signalingNaN mask[1] - quietNaN mask[2] - negativeInfinity mask[3] - negativeNormal mask[4] - negativeSubnormal mask[5] - negativeZero mask[6] - positiveZero mask[7] - positiveSubnormal mask[8] - positiveNormal mask[9] - positiveInfinity There is no vector result written to a gpr, and no vector feedback path for this opcode. This opcode does not raise exceptions under any circumstances.	0xB8

12.10 VOP3 3 in, 1 out Instructions (VOP3a)

Add Floating-Point, 64-Bit

Instruction **V_ADD_F64**

Description Double-precision floating-point add.

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. Double result written to 2 consecutive gpr registers, instruction dest specifies lesser of the two.

D.d = S0.d + S1.d.

Table 12.4 Result of V_ADD_F64 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.

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

These properties hold true for this instruction:

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

Microcode VOP3a Opcode 356 (0x164)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_ALIGNBIT_B32**

Description Bit align. Arbitrarily align 32 bits within 64 into a GPR.

D.u = ({S0,S1} >> S2.u[4:0]) & 0xFFFFFFFF.

Microcode VOP3a Opcode 334 (0x14E)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_ALIGNBYTE_B32**

Description Byte align.

```
dst = ({src0, src1} >> (8 * src2[1:0])) & 0xFFFFFFFF;
```

D.u = ({S0,S1} >> (8*S2.u[4:0])) & 0xFFFFFFFF.

Microcode VOP3a Opcode 335 (0x14F)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_ASHR_I64**

Description D = S0.u >> S1.u[4:0].

Microcode VOP3a Opcode 355 (0x163)

<table>
<tr><td colspan="3">NEG</td><td colspan="2">OMOD</td><td colspan="2">SRC2</td><td colspan="4">SRC1</td><td colspan="2">SRC0</td><td>+4</td></tr>
<tr><td>**1**</td><td>**1**</td><td>**0**</td><td>**1**</td><td>**0**</td><td>**0**</td><td colspan="2">OP</td><td>r</td><td>CLMP</td><td colspan="2">ABS</td><td>VDST</td><td>+0</td></tr>
</table>

Instruction **V_BFE_I32**

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

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

D.i = (S0.i>>S1.u[4:0]) & ((1<<S2.u[4:0])-1); bitfield extract, S0=data, S1=field_offset, S2=field_width.

Microcode VOP3a Opcode 329 (0x149)

<table>
<tr><td colspan="3">NEG</td><td colspan="2">OMOD</td><td colspan="2">SRC2</td><td colspan="4">SRC1</td><td colspan="2">SRC0</td><td>+4</td></tr>
<tr><td>**1**</td><td>**1**</td><td>**0**</td><td>**1**</td><td>**0**</td><td>**0**</td><td colspan="2">OP</td><td>r</td><td>CLMP</td><td colspan="2">ABS</td><td>VDST</td><td>+0</td></tr>
</table>

Instruction **V_BFE_U32**

Description DX11 unsigned bitfield extract. Src0 = input data, scr1 = offset, and src2 = width. Bit position offset is extracted through offset + width from input data.

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

D.u = (S0.u>>S1.u[4:0]) & ((1<<S2.u[4:0])-1); bitfield extract, S0=data, S1=field_offset, S2=field_width.

Microcode VOP3a Opcode 328 (0x148)

NEG		OMOD		SRC2		SRC1			SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_BFI_B32**

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

src0 = bitfield mask (from BFM)

src 1 & src2 = input data

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

D.u = (S0.u & S1.u) | (~S0.u & S2.u).

Microcode VOP3a Opcode 330 (0x14A)

NEG		OMOD		SRC2		SRC1			SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_CUBEID_F32**

Description Cubemap Face ID determination. Result is a floating point face ID.

```
S0.f = x
S1.f = y
S2.f = z
If (Abs(S2.f) >= Abs(S0.f) &&
      Abs(S2.f) >= Abs(S1.f))
   If (S2.f < 0) D.f = 5.0
   Else D.f = 4.0
Else if (Abs(S1.f) >= Abs(S0.f))
   If (S1.f < 0) D.f = 3.0
   Else D.f = 2.0
Else
   If (S0.f < 0) D.f = 1.0
   Else D.f = 0.0
```

Microcode VOP3a Opcode 324 (0x144)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	1	0	1	0	0	OP	r	CLMP	ABS	VDST	+0

Instruction **V_CUBEMA_F32**

Description Cubemap Major Axis determination. Result is 2.0 * Major Axis.

```
S0.f = x
S1.f = y
S2.f = z
If (Abs(S2.f) >= Abs(S0.f) &&
      Abs(S2.f) >= Abs(S1.f))
   D.f = 2.0*S2.f
Else if (Abs(S1.f) >= Abs(S0.f))
   D.f = 2.0 * S1.f
Else
   D.f = 2.0 * S0.f
```

Microcode VOP3a Opcode 327 (0x147)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	1	0	1	0	0	OP	r	CLMP	ABS	VDST	+0

Instruction **V_CUBESC_F32**

Description Cubemap S coordination determination.

```
S0.f = x
S1.f = y
S2.f = z
If (Abs(S2.f) >= Abs(S0.f) &&
      Abs(S2.f) >= Abs(S1.f))
   If (S2.f < 0) D.f = –S0.f
   Else D.f = S0.f
Else if (Abs(S1.f) >= Abs(S0.f))
   D.f = S0.f
Else
   If (S0.f < 0) D.f = S2.f
   Else D.f = –S2.f
```

Microcode VOP3a Opcode 325 (0x145)

<table>
<tr><td colspan="3">NEG</td><td colspan="2">OMOD</td><td colspan="2">SRC2</td><td colspan="3">SRC1</td><td colspan="2">SRC0</td><td>+4</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>OP</td><td>r</td><td>CLMP</td><td>ABS</td><td colspan="2">VDST</td><td>+0</td></tr>
</table>

Instruction **V_CUBETC_F32**

Description Cubemap T coordinate determination.

```
S0.f = x
S1.f = y
S2.f = z
If (Abs(S2.f) >= Abs(S0.f) &&
      Abs(S2.f) >= Abs(S1.f))
   D.f = –S1.f
Else if (Abs(S1.f) >= Abs(S0.f))
   If (S1.f < 0) D.f = –S2.f
   Else D.f = S2.f
Else
   D.f = –S1.f
```

Microcode VOP3a Opcode 326 (0x146)

<table>
<tr><td colspan="3">NEG</td><td colspan="2">OMOD</td><td colspan="2">SRC2</td><td colspan="3">SRC1</td><td colspan="2">SRC0</td><td>+4</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>OP</td><td>r</td><td>CLMP</td><td>ABS</td><td colspan="2">VDST</td><td>+0</td></tr>
</table>

Instruction	**V_CVT_PK_U8_F32**
Description	Float to 8 bit unsigned integer conversion Replacement for 8xx/9xx FLT_TO_UINT4 opcode. Float to 8 bit uint conversion placed into any byte of result, accumulated with S2.f. Four applications of this opcode can accumulate 4 8-bit integers packed into a single dword. D.f = ((flt_to_uint(S0.f) & 0xff) << (8*S1.f[1:0])) || (S2.f & ~(0xff << (8*S1.f[1:0]))); Intended use, ops in any order: op - cvt_pk_u8_f32 r0 foo2, 2, r0 op - cvt_pk_u8_f32 r0 foo1, 1, r0 op - cvt_pk_u8_f32 r0 foo3, 3, r0 op - cvt_pk_u8_f32 r0 foo0, 0, r0 r0 result is 4 bytes packed into a dword: {foo3, foo2, foo1, foo0}

Microcode VOP3a Opcode 350 (0x15E)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **`V_DIV_FIXUP_F32`**

Description Single precision division fixup.

Given a numerator, denominator, and quotient from a divide, this opcode detects and applies special-case numerics, touching up the quotient if necessary. This opcode also generates all exceptions caused by the division. The generation of the inexact exception requires a fused multiple add (FMA), making this opcode a variant of FMA.

S0.f = Quotient

S1.f = Denominator

S2.f = Numerator

If (S1.f==Nan && S2.f!=SNan)

  D.f = Quiet(S1.f);

Else if (S2.f==Nan)

  D.f = Quiet(S2.f);

Else if (S1.f==S2.f==0)

  # 0/0

  D.f = pele_nan(0xffc00000);

Else if (abs(S1.f)==abs(S2.f)==infinity)

  # inf/inf

  D.f = pele_nan(0xffc00000);

Else if (S1.f==0)

  # x/0

  D.f = (sign(S1.f)^sign(S0.f) ? –inf : inf;

Else if (abs(S1.f)==inf)

  # x/inf

  D.f = (sign(S1.f)^sign(S0.f) ? –0 : 0;

Else if (S0.f==Nan)

  # division error correction nan due to N*1/D overflow (result of divide is overflow)

  D.f = (sign(S1.f)^sign(S0.f) ? –inf : inf;

Else

  D.f = S0.f;

Microcode VOP3a Opcode 351 (0x15F)

<table>
<tr><td colspan="3">NEG</td><td colspan="2">OMOD</td><td colspan="2">SRC2</td><td colspan="3">SRC1</td><td colspan="2">SRC0</td><td>+4</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>OP</td><td>r</td><td>CLMP</td><td>ABS</td><td colspan="2">VDST</td><td>+0</td></tr>
</table>

Instruction	**V_DIV_FIXUP_F64**
Description	Double precision division fixup. Given a numerator, denominator, and quotient from a divide, this opcode will detect and apply special case numerics, touching up the quotient if necessary. This opcode also generates all exceptions caused by the division. The generation of the inexact exception requires a fused multiply add (FMA), making this opcode a variant of FMA. D.d = Special case divide fixup and flags(s0.d = Quotient, s1.d = Denominator, s2.d = Numerator).

Microcode VOP3a Opcode 352 (0x160)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction	**V_DIV_FMAS_F32**
Description	D.f = Special case divide FMA with scale and flags(s0.f = Quotient, s1.f = Denominator, s2.f = Numerator).

Microcode VOP3a Opcode 367 (0x16F)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction	**V_DIV_FMAS_F64**
Description	D.d = Special case divide FMA with scale and flags(s0.d = Quotient, s1.d = Denominator, s2.d = Numerator).

Microcode VOP3a Opcode 368 (0x170)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_FMA_F32**

Description Fused single-precision multiply-add. Only for double-precision parts.

D.f = S0.f * S1.f + S2.f.

Microcode VOP3a Opcode 331 (0x14B)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_FMA_F64**

Description Double-precision floating-point fused multiply add (FMA).

Adds the src2 to the product of the src0 and `src1`. A single round is performed on the sum - the product of `src0` and `src1` is not truncated or rounded.

The instruction specifies which one of two data elements in a four-element vector is operated on (the two dwords of a double precision floating point number), and the result can be stored in the wz or yx elements of the destination GPR.

D.d = S0.d * S1.d + S2.d.

Microcode VOP3a Opcode 332 (0x14C)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction	**V_LDEXP_F64**

Description

Double-precision LDEXP from the C math library.

This 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 12.5 Result of LDEXP_F64 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.

Microcode VOP3a Opcode 360 (0x168)

<table>
<tr><td colspan="3">NEG</td><td colspan="2">OMOD</td><td colspan="2">SRC2</td><td colspan="3">SRC1</td><td colspan="2">SRC0</td><td>+4</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>OP</td><td>r</td><td>CLMP</td><td>ABS</td><td colspan="2">VDST</td><td>+0</td></tr>
</table>

Instruction **`V_LERP_U8`**

Description Unsigned eight-bit pixel average on packed unsigned bytes (linear interpolation). S2 acts as a round mode; if set, 0.5 rounds up; otherwise, 0.5 truncates.

D.u = ((S0.u[31:24] + S1.u[31:24] + S2.u[24]) >> 1) << 24 + ((S0.u[23:16] + S1.u[23:16] + S2.u[16]) >> 1) << 16 + ((S0.u[15:8] + S1.u[15:8] + S2.u[8]) >> 1) << 8 + ((S0.u[7:0] + S1.u[7:0] + S2.u[0]) >> 1).

```
dst = ((src0[31:24] + src1[31:24] + src2[24]) >> 1) << 24 +
((src0[23:16] + src1[23:16] + src2[16]) >>1) << 16 +
((src0[15:8] + src1[15:8] + src2[8]) >> 1) << 8 +

((src0[7:0] + src1[7:0] + src2[0]) >> 1)
```

Microcode VOP3a Opcode 333 (0x14D)

<table>
<tr><td colspan="3">NEG</td><td colspan="2">OMOD</td><td colspan="2">SRC2</td><td colspan="3">SRC1</td><td colspan="2">SRC0</td><td>+4</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>OP</td><td>r</td><td>CLMP</td><td>ABS</td><td colspan="2">VDST</td><td>+0</td></tr>
</table>

Instruction **`V_LSHL_B64`**

Description D = S0.u << S1.u[4:0].

Microcode VOP3a Opcode 353 (0x161)

NEG			OMOD		SRC2		SRC1		SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **`V_LSHR_B64`**

Description D = S0.u >> S1.u[4:0].

Microcode VOP3a Opcode 354 (0x162)

NEG			OMOD		SRC2		SRC1		SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **`V_MAD_F32`**

Description Floating point multiply-add (MAD). Gives same result as `ADD` after `MUL_IEEE`. Uses IEEE rules for 0*anything.

D.f = S0.f * S1.f + S2.f.

Microcode VOP3a Opcode 321 (0x141)

NEG			OMOD		SRC2		SRC1		SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **`V_MAD_I32_I24`**

Description 24-bit signed integer muladd.

S0 and S1 are treated as 24-bit signed integers. S2 is treated as a 32-bit signed or unsigned integer. Bits [31:24] are ignored. The result represents the low-order sign extended 32 bits of the multiply add result.

Result = Arg1.i[23:0] * Arg2.i[23:0] + Arg3.i[31:0] (low order bits).

Microcode VOP3a Opcode 322 (0x142)

NEG			OMOD		SRC2		SRC1		SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **`V_MAD_I64_I32`**

Description Multiply add using the product of two 32-bit signed integers, then added to a 64-bit integer.

{vcc_out,D.i64} = S0.i32 * S1.i32 + S2.i64.

Microcode VOP3a Opcode 375 (0x177)

NEG			OMOD		SRC2		SRC1		SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **`V_MAD_LEGACY_F32`**

Description Floating-point multiply-add (MAD). Gives same result as ADD after MUL.

D.f = S0.f * S1.f + S2.f (DX9 rules, 0.0*x = 0.0).

Microcode VOP3a Opcode 320 (0x140)

NEG			OMOD		SRC2		SRC1		SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction	**`V_MAD_U32_U24`**
Description	24 bit unsigned integer muladd Src a and b treated as 24 bit unsigned integers. Src c treated as 32 bit signed or unsigned integer. Bits [31:24] ignored. The result represents the low-order 32 bits of the multiply add result. D.u = S0.u[23:0] * S1.u[23:0] + S2.u[31:0].

Microcode VOP3a Opcode 323 (0x143)

NEG	OMOD	SRC2	SRC1	SRC0	+4

1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction	**`V_MAD_U64_U32`**
Description	Multiply add using the product of two 32-bit unsigned integers, then added to a 64-bit integer. {vcc_out,D.u64} = S0.u32 * S1.u32 + S2.u64.

Microcode VOP3a Opcode 374 (0x176)

NEG	OMOD	SRC2	SRC1	SRC0	+4

1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_MAX_F64**

Description The instruction specifies which one of two data elements in a four-element vector is operated on (the two dwords of a double precision floating point number), and the result can be stored in the wz or yx elements of the destination GPR.

D.d = max(S0.d, S1.d).

```
if (src0 > src1)
  dst = src0;
else
  dst = src1;

max(-0,+0)=max(+0,-0)=+0
```

Microcode VOP3a Opcode 359 (0x167)

NEG			OMOD		SRC2		SRC1			SRC0		+4
1	1	0	1	0	0	OP		r	CLMP	ABS	VDST	+0

Instruction **V_MAX3_F32**

Description Maximum of three numbers. DX10 NaN handland and flag creation.

D.f = max(S0.f, S1.f, S2.f).

Microcode VOP3a Opcode 340 (0x154)

NEG			OMOD		SRC2		SRC1			SRC0		+4
1	1	0	1	0	0	OP		r	CLMP	ABS	VDST	+0

Instruction **V_MAX3_I32**

Description Maximum of three numbers.

D.i = max(S0.i, S1.i, S2.i).

Microcode VOP3a Opcode 341 (0x155)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CL MP	ABS	VDST	+0

Instruction **V_MAX3_U32**

Description Maximum of three numbers.

D.u = max(S0.u, S1.u, S2.u).

Microcode VOP3a Opcode 342 (0x156)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CL MP	ABS	VDST	+0

Instruction **V_MED3_F32**

Description Median of three numbers. DX10 NaN handling and flag creation.

```
If (isNan(S0.f) || isNan(S1.f) || isNan(S2.f))
  D.f = MIN3(S0.f, S1.f, S2.f)
Else if (MAX3(S0.f,S1.f,S2.f) == S0.f)
  D.f = MAX(S1.f, S2.f)
Else if (MAX3(S0.f,S1.f,S2.f) == S1.f)
  D.f = MAX(S0.f, S2.f)
Else
  D.f = MAX(S0.f, S1.f)
```

Microcode VOP3a Opcode 343 (0x157)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CL MP	ABS	VDST	+0

Instruction **V_MED3_I32**

Description Median of three numbers.

```
If (isNan(S0.f) || isNan(S1.f) || isNan(S2.f))
  D.f = MIN3(S0.f, S1.f, S2.f)
Else if (MAX3(S0.f,S1.f,S2.f) == S0.f)
  D.f = MAX(S1.f, S2.f)
Else if (MAX3(S0.f,S1.f,S2.f) == S1.f)
  D.f = MAX(S0.f, S2.f)
Else
  D.f = MAX(S0.f, S1.f)
```

Microcode VOP3a Opcode 344 (0x158)

NEG			OMOD		SRC2		SRC1			SRC0		+4
1	1	0	1	0	0	OP		r	CLMP	ABS	VDST	+0

Instruction **V_MED3_U32**

Description Median of three numbers.

```
If (isNan(S0.f) || isNan(S1.f) || isNan(S2.f))
  D.f = MIN3(S0.f, S1.f, S2.f)
Else if (MAX3(S0.f,S1.f,S2.f) == S0.f)
  D.f = MAX(S1.f, S2.f)
Else if (MAX3(S0.f,S1.f,S2.f) == S1.f)
  D.f = MAX(S0.f, S2.f)
Else
  D.f = MAX(S0.f, S1.f)
```

Microcode VOP3a Opcode 345 (0x159)

NEG			OMOD		SRC2		SRC1			SRC0		+4
1	1	0	1	0	0	OP		r	CLMP	ABS	VDST	+0

Instruction **V_MIN_F64**

Description Double precision floating point minimum.

The instruction specifies which one of two data elements in a four-element vector is operated on (the two dwords of a double precision floating point number), and the result can be stored in the wz or yx elements of the destination GPR.

DX10 implies slightly different handling of Nan's. See the SP Numeric spec for details.

Double result written to two consecutive GPRs; the instruction Dest specifies the lesser of the two.

```
if (src0 < src1)
  dst = src0;
else
  dst = src1;
```

```
min(-0,+0)=min(+0,-0)=-0
```

D.d = min(S0.d, S1.d).

Microcode VOP3a Opcode 358 (0x166)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_MIN3_F32**

Description Minimum of three numbers. DX10 NaN handling and flag creation.

D.f = min(S0.f, S1.f, S2.f).

Microcode VOP3a Opcode 337 (0x151)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_MIN3_I32**

Description Minimum of three numbers.

D.i = min(S0.i, S1.i, S2.i).

Microcode VOP3a Opcode 338 (0x152)

<table>
<tr><td colspan="3">NEG</td><td colspan="2">OMOD</td><td colspan="2">SRC2</td><td colspan="3">SRC1</td><td colspan="2">SRC0</td><td>+4</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>OP</td><td>r</td><td>CLMP</td><td>ABS</td><td colspan="2">VDST</td><td>+0</td></tr>
</table>

Instruction **V_MIN3_U32**

Description Minimum of three numbers.

D.u = min(S0.u, S1.u, S2.u).

Microcode VOP3a Opcode 339 (0x153)

<table>
<tr><td colspan="3">NEG</td><td colspan="2">OMOD</td><td colspan="2">SRC2</td><td colspan="3">SRC1</td><td colspan="2">SRC0</td><td>+4</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>OP</td><td>r</td><td>CLMP</td><td>ABS</td><td colspan="2">VDST</td><td>+0</td></tr>
</table>

Instruction **V_MQSAD_PK_U16_U8**

Description D.u = Masked Quad-Byte SAD with accum_lo/hi(S0.u[63:0], S1.u[31:0], S2.u[63:0]).

Microcode VOP3a Opcode 371 (0x173)

<table>
<tr><td colspan="3">NEG</td><td colspan="2">OMOD</td><td colspan="2">SRC2</td><td colspan="3">SRC1</td><td colspan="2">SRC0</td><td>+4</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>OP</td><td>r</td><td>CLMP</td><td>ABS</td><td colspan="2">VDST</td><td>+0</td></tr>
</table>

Instruction **`V_MQSAD_U32_U8`**

Description Masked quad sum-of-absolute-difference.

D.u128 = Masked Quad-Byte SAD with 32-bit accum_lo/hi(S0.u[63:0], S1.u[31:0], S2.u[127:0])

Microcode VOP3a Opcode 373 (0x175)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **`V_MSAD_U8`**

Description D.u = Masked Byte SAD with accum_lo(S0.u, S1.u, S2.u).

Microcode VOP3a Opcode 369 (0x171)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_MUL_F64**

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`. Inputs are from two consecutive GPRs, with the instruction specifying the lesser of the two; the double result is written to two consecutive GPRs.

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

D.d = S0.d * S1.d.

Table 12.6 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 V_MUL_F64 instruction is a four-slot instruction. Therefore, a single V_MUL_F64 instruction can be issued in slots 0, 1, 2, and 3. Slot 4 can contain any other valid instruction.

Microcode VOP3a Opcode 357 (0x165)

NEG			OMOD			SRC2	SRC1		SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_MUL_HI_I32**

Description Signed integer multiplication. The result represents the high-order 32 bits of the multiply result.

D.i = (S0.i * S1.i)>>32.

Microcode VOP3a Opcode 364 (0x16C)

NEG	OMOD	SRC2	SRC1	SRC0	+4

1	**1**	**0**	**1**	**0**	**0**	OP	r	CL MP	ABS	VDST	+0

Instruction **V_MUL_HI_U32**

Description Unsigned integer multiplication. The result represents the high-order 32 bits of the multiply result.

D.u = (S0.u * S1.u)>>32.

Microcode VOP3a Opcode 362 (0x16A)

NEG	OMOD	SRC2	SRC1	SRC0	+4

1	**1**	**0**	**1**	**0**	**0**	OP	r	CL MP	ABS	VDST	+0

Instruction **V_MUL_LO_I32**

Description Signed integer multiplication. The result represents the low-order 32 bits of the multiply result.

D.i = S0.i * S1.i.

Microcode VOP3a Opcode 363 (0x16B)

NEG	OMOD	SRC2	SRC1	SRC0	+4

1	**1**	**0**	**1**	**0**	**0**	OP	r	CL MP	ABS	VDST	+0

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **`V_MUL_LO_U32`**

Description Unsigned integer multiplication. The result represents the low-order 32 bits of the multiply result.

D.u = S0.u * S1.u.

Microcode VOP3a Opcode 361 (0x169)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **`V_MULLIT_F32`**

Description Floating-point multiply for lighting calculation.

It is used when emulating LIT instruction. 0*anything = 0.

Note this instruction takes three inputs.

D.f = S0.f * S1.f, replicate result into 4 components (0.0 * x = 0.0; special INF, NaN, overflow rules).

Microcode VOP3a Opcode 336 (0x150)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **`V_QSAD_PK_U16_U8`**

Description D.u = Quad-Byte SAD with accum_lo/hiu(S0.u[63:0], S1.u[31:0], S2.u[63:0]).

Microcode VOP3a Opcode 370 (0x172)

NEG			OMOD		SRC2		SRC1			SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_SAD_HI_U8**

Description Sum of absolute differences with accumulation.

Perform 4x1 SAD with S0.u and S1.u, and accumulate result into msb's of S2.u. Overflow is lost.

```
ABS_DIFF (A,B) = (A>B) ? (A-B) : (B-A)

D.u = (ABS_DIFF (S0.u[31:24],S1.u[31:24])+ ABS_DIFF (S0.u[23:16],S1.u[23:16])
+ ABS_DIFF (S0.u[15:8],S1.u[15:8]) + ABS_DIFF (S0.u[7:0],S1.u[7:0]) ) << 16 +
S2.u
```

Microcode VOP3a Opcode 347 (0x15B)

NEG			OMOD		SRC2		SRC1		SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_SAD_U8**

Description Sum of absolute differences with accumulation.

Perform 4x1 SAD with S0.u and S1.u, and accumulate result into lsbs of S2.u. Overflow into S2.u upper bits is allowed.

```
ABS_DIFF (A,B) = (A>B) ? (A-B) : (B-A)

D.u = ABS_DIFF (S0.u[31:24],S1.u[31:24])+ ABS_DIFF (S0.u[23:16],S1.u[23:16])+
ABS_DIFF (S0.u[15:8],S1.u[15:8])+ ABS_DIFF (S0.u[7:0],S1.u[7:0]) + S2.u
```

Microcode VOP3a Opcode 346 (0x15A)

NEG			OMOD		SRC2		SRC1		SRC0		+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	CLMP	ABS	VDST	+0

Instruction **V_SAD_U16**

Description Sum of absolute differences with accumulation.

Perform 2x1 SAD with S0.u and S1.u, and accumulate result with S2.u.

```
ABS_DIFF (A,B) = (A>B) ? (A-B) : (B-A)

D.u = ABS_DIFF (S0.u[31:16],S1.u[31:16]) + ABS_DIFF (S0.u[15:0],S1.u[15:0]) +
S2.u
```

Microcode VOP3a Opcode 348 (0x15C)

NEG	OMOD	SRC2	SRC1	SRC0	+4

1	1	0	1	0	0	OP	r	CLMP	ABS	VDST	+0

Instruction **V_SAD_U32**

Description Sum of absolute differences with accumulation.

Perform a single-element SAD with S0.u and S1.u, and accumulate result into msb's of S2.u. Overflow is lost.

```
ABS_DIFF (A,B) = (A>B) ? (A-B) : (B-A)

D.u = ABS_DIFF (S0.u,S1.u)  + S2.u
```

Microcode VOP3a Opcode 349 (0x15D)

NEG	OMOD	SRC2	SRC1	SRC0	+4

1	1	0	1	0	0	OP	r	CLMP	ABS	VDST	+0

Instruction **`V_TRIG_PREOP_F64`**

Description D.d = Look Up 2/PI (S0.d) with segment select S1.u[4:0].

Microcode VOP3a Opcode 372 (0x174)

<table>
<tr><td colspan="3">NEG</td><td colspan="2">OMOD</td><td colspan="2">SRC2</td><td colspan="3">SRC1</td><td colspan="2">SRC0</td><td>+4</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>OP</td><td>r</td><td>CL
MP</td><td>ABS</td><td colspan="2">VDST</td><td>+0</td></tr>
</table>

12.11 VOP3 Instructions (3 in, 2 out), (VOP3b)

Instruction **V_DIV_SCALE_F32**

Description D.f = Special case divide preop and flags(s0.f = Quotient, s1.f = Denominator, s2.f = Numerator) s0 must equal s1 or s2.

Microcode VOP3b Opcode 365 (0x16D)

NEG			OMOD		SRC2		SRC1		SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	SDST	VDST	+0

Instruction **V_DIV_SCALE_F64**

Description D.d = Special case divide preop and flags(s0.d = Quotient, s1.d = Denominator, s2.d = Numerator) s0 must equal s1 or s2.

Microcode VOP3b Opcode 366 (0x16E)

NEG			OMOD		SRC2		SRC1		SRC0	+4
1	**1**	**0**	**1**	**0**	**0**	OP	r	SDST	VDST	+0

12.12 VINTRP Instructions

Instruction **V_INTERP_MOV_F32**

Description Vertex Parameter Interpolation using parameters stored in LDS and barycentric coordinates in VGPRs.

M0 must contain: `{ 1'b0, new_prim_mask[15:1], lds_param_offset[15:0] }`.

The ATTR field indicates which attribute (0-32) to interpolate.

The ATTRCHAN field indicates which channel: 0=x, 1=y, 2=z and 3=w.

Microcode VINTRP Opcode 2 (0x2)

1	**1**	**0**	**0**	**1**	**0**	VDST	OP	ATTR	ATTR CHAN	VSRC (I, J)	+0

Instruction **V_INTERP_P1_F32**

Description Vertex Parameter Interpolation using parameters stored in LDS and barycentric coordinates in VGPRs.

M0 must contain: `{ 1'b0, new_prim_mask[15:1], lds_param_offset[15:0] }`.

The ATTR field indicates which attribute (0-32) to interpolate.

The ATTRCHAN field indicates which channel: 0=x, 1=y, 2=z and 3=w.

Microcode VINTRP Opcode 0 (0x0)

1	**1**	**0**	**0**	**1**	**0**	VDST	OP	ATTR	ATTR CHAN	VSRC (I, J)	+0

Instruction **V_INTERP_P2_F32**

Description Vertex Parameter Interpolation using parameters stored in LDS and barycentric coordinates in VGPRs.

M0 must contain: `{ 1'b0, new_prim_mask[15:1], lds_param_offset[15:0] }`.

The ATTR field indicates which attribute (0-32) to interpolate.

The ATTRCHAN field indicates which channel: 0=x, 1=y, 2=z and 3=w.

Microcode VINTRP Opcode 1 (0x1)

1	**1**	**0**	**0**	**1**	**0**	VDST	OP	ATTR	ATTR CHAN	VSRC (I, J)	+0

12.13 LDS/GDS Instructions

This suite of instructions operates on data stored within the data share memory. The instructions transfer data between VGPRs and data share memory.

The bitfield map for the the LDS/GDS is:

<table>
<tr><td colspan="4">VDST</td><td colspan="5">DATA1</td><td>DATA0</td><td>ADDR</td><td>+4</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>OP</td><td>GDS</td><td>r</td><td>OFFSET1</td><td>OFFSET0</td><td>+0</td></tr>
</table>

where:

OFFSET0	= Unsigned byte offset added to the address supplied by the ADDR VGPR.
OFFSET1	= Unsigned byte offset added to the address supplied by the ADDR VGPR.
GDS	= Set if GDS, cleared if LDS.
OP	= DS instructions.
ADDR	= Source LDS address VGPR 0 - 255.
DATA0	= Source data0 VGPR 0 - 255.
DATA1	= Source data1 VGPR 0 - 255.
VDST	= Destination VGPR 0- 255.

Table 12.7 DS Instructions for the Opcode Field

Instruction	Description (C-Function Equivalent)	Decimal/Hex
DS_ADD_U32	DS[A] = DS[A] + D0; uint add.	00 (0x0)
DS_SUB_U32	DS[A] = DS[A] - D0; uint subtract.	01 (0x1)
DS_RSUB_U32	DS[A] = D0 - DS[A]; uint reverse subtract.	02 (0x2)
DS_INC_U32	DS[A] = (DS[A] >= D0 ? 0 : DS[A] + 1); uint increment.	03 (0x3)
DS_DEC_U32	DS[A] = (DS[A] == 0 || DS[A] > D0 ? D0 : DS[A] - 1); uint decrement.	04 (0x4)
DS_MIN_I32	DS[A] = min(DS[A], D0); int min.	05 (0x5)
DS_MAX_I32	DS[A] = max(DS[A], D0); int max.	06 (0x6)
DS_MIN_U32	DS[A] = min(DS[A], D0); uint min.	07 (0x7)
DS_MAX_U32	DS[A] = max(DS[A], D0); uint max.	08 (0x8)
DS_AND_B32	DS[A] = DS[A] & D0; Dword AND.	09 (0x9)
DS_OR_B32	DS[A] = DS[A] | D0; Dword OR.	10 (0xA)
DS_XOR_B32	DS[A] = DS[A] ^ D0; Dword XOR.	11 (0xB)
DS_MSKOR_B32	DS[A] = (DS[A] ^ ~D0) | D1; masked Dword OR.	12 (0xC)
DS_WRITE_B32	DS[A] = D0; write a Dword.	13 (0xD)
DS_WRITE2_B32	DS[ADDR+offset0*4] = D0; DS[ADDR+offset1*4] = D1; write 2 Dwords.	14 (0xE)

Table 12.7 DS Instructions for the Opcode Field (Cont.)

Instruction	Description (C-Function Equivalent)	Decimal/Hex
DS_WRITE2ST64_B32	DS[ADDR+offset0*4*64] = D0; DS[ADDR+offset1*4*64] = D1; write 2 Dwords.	15 (0xF)
DS_CMPST_B32	DS[A] = (DS[A] == D0 ? D1 : DS[A]); compare store.	16 (0x10)
DS_CMPST_F32	DS[A] = (DS[A] == D0 ? D1 : DS[A]); compare store with float rules.	17 (0x11)
DS_MIN_F32	DS[A] = (DS[A] < D1) ? D0 : DS[A]; float compare swap (handles NaN/INF/denorm).	18 (0x12)
DS_MAX_F32	DS[A] = (D0 > DS[A]) ? D0 : DS[A]; float, handles NaN/INF/denorm.	19 (0x13)
DS_NOP	Do nothing.	20 (0x14)
DS_GWS_SEMA_RELEASE_ALL	GDS Only. Release all wavefronts waiting on this semaphore. ResourceID is in offset[4:0].	24 (0x18)
DS_GWS_INIT	GDS only.	25 (0x19)
DS_GWS_SEMA_V	GDS only.	26 (0x1A)
DS_GWS_SEMA_BR	GDS only.	27 (0x1B)
DS_GWS_SEMA_P	GDS only.	28 (0x1C)
DS_GWS_BARRIER	GDS only.	29 (0x1D)
DS_WRITE_B8	DS[A] = D0[7:0]; byte write.	30 (0x1E)
DS_WRITE_B16	DS[A] = D0[15:0]; short write.	31 (0x1F)
DS_ADD_RTN_U32	Uint add.	32 (0x20)
DS_SUB_RTN_U32	Uint subtract.	33 (0x21)
DS_RSUB_RTN_U32	Uint reverse subtract.	34 (0x22)
DS_INC_RTN_U32	Uint increment.	35 (0x23)
DS_DEC_RTN_U32	Uint decrement.	36 (0x24)
DS_MIN_RTN_I32	Int min.	37 (0x25)
DS_MAX_RTN_I32	Int max.	38 (0x26)
DS_MIN_RTN_U32	Uint min.	39 (0x27)
DS_MAX_RTN_U32	Uint max.	40 (0x28)
DS_AND_RTN_B32	Dword AND.	41 (0x29)
DS_OR_RTN_B32	Dword OR.	42 (0x2A)
DS_XOR_RTN_B32	Dword XOR.	43 (0x2B)
DS_MSKOR_RTN_B32	Masked Dword OR.	44 (0x2C)
DS_WRXCHG_RTN_B32	Write exchange. Offset = {offset1,offset0}. A = ADDR+offset. D=DS[Addr]. DS[Addr]=D0.	45 (0x2D)
DS_WRXCHG2_RTN_B32	Write exchange 2 separate Dwords.	46 (0x2E)
DS_WRXCHG2ST64_RTN_B32	Write exchange 2 Dwords, stride 64.	47 (0x2F)
DS_CMPST_RTN_B32	Compare store.	48 (0x30)
DS_CMPST_RTN_F32	Compare store with float rules.	49 (0x31)
DS_MIN_RTN_F32	DS[A] = (DS[A] < D1) ? D0 : DS[A]; float compare swap (handles NaN/INF/denorm).	50 (0x32)
DS_MAX_RTN_F32	DS[A] = (D0 > DS[A]) ? D0 : DS[A]; float, handles NaN/INF/denorm.	51 (0x33)
DS_WRAP_RTN_B32	DS[A] = (DS[A] >= D0) ? DS[A] - D0 : DS[A] + D1.	52 (0x34)

Table 12.7 DS Instructions for the Opcode Field (Cont.)

Instruction	Description (C-Function Equivalent)	Decimal/Hex
DS_WRITE2ST64_B32	DS[ADDR+offset0*4*64] = D0; DS[ADDR+offset1*4*64] = D1; write 2 Dwords.	15 (0xF)
DS_CMPST_B32	DS[A] = (DS[A] == D0 ? D1 : DS[A]); compare store.	16 (0x10)
DS_CMPST_F32	DS[A] = (DS[A] == D0 ? D1 : DS[A]); compare store with float rules.	17 (0x11)
DS_MIN_F32	DS[A] = (DS[A] < D1) ? D0 : DS[A]; float compare swap (handles NaN/INF/denorm).	18 (0x12)
DS_MAX_F32	DS[A] = (D0 > DS[A]) ? D0 : DS[A]; float, handles NaN/INF/denorm.	19 (0x13)
DS_NOP	Do nothing.	20 (0x14)
DS_GWS_SEMA_RELEASE_ALL	GDS Only. Release all wavefronts waiting on this semaphore. ResourceID is in offset[4:0].	24 (0x18)
DS_GWS_INIT	GDS only.	25 (0x19)
DS_GWS_SEMA_V	GDS only.	26 (0x1A)
DS_GWS_SEMA_BR	GDS only.	27 (0x1B)
DS_GWS_SEMA_P	GDS only.	28 (0x1C)
DS_GWS_BARRIER	GDS only.	29 (0x1D)
DS_WRITE_B8	DS[A] = D0[7:0]; byte write.	30 (0x1E)
DS_WRITE_B16	DS[A] = D0[15:0]; short write.	31 (0x1F)
DS_ADD_RTN_U32	Uint add.	32 (0x20)
DS_SUB_RTN_U32	Uint subtract.	33 (0x21)
DS_RSUB_RTN_U32	Uint reverse subtract.	34 (0x22)
DS_INC_RTN_U32	Uint increment.	35 (0x23)
DS_DEC_RTN_U32	Uint decrement.	36 (0x24)
DS_MIN_RTN_I32	Int min.	37 (0x25)
DS_MAX_RTN_I32	Int max.	38 (0x26)
DS_MIN_RTN_U32	Uint min.	39 (0x27)
DS_MAX_RTN_U32	Uint max.	40 (0x28)
DS_AND_RTN_B32	Dword AND.	41 (0x29)
DS_OR_RTN_B32	Dword OR.	42 (0x2A)
DS_XOR_RTN_B32	Dword XOR.	43 (0x2B)
DS_MSKOR_RTN_B32	Masked Dword OR.	44 (0x2C)
DS_WRXCHG_RTN_B32	Write exchange. Offset = {offset1,offset0}. A = ADDR+offset. D=DS[Addr]. DS[Addr]=D0.	45 (0x2D)
DS_WRXCHG2_RTN_B32	Write exchange 2 separate Dwords.	46 (0x2E)
DS_WRXCHG2ST64_RTN_B32	Write exchange 2 Dwords, stride 64.	47 (0x2F)
DS_CMPST_RTN_B32	Compare store.	48 (0x30)
DS_CMPST_RTN_F32	Compare store with float rules.	49 (0x31)
DS_MIN_RTN_F32	DS[A] = (DS[A] < D1) ? D0 : DS[A]; float compare swap (handles NaN/INF/denorm).	50 (0x32)
DS_MAX_RTN_F32	DS[A] = (D0 > DS[A]) ? D0 : DS[A]; float, handles NaN/INF/denorm.	51 (0x33)
DS_WRAP_RTN_B32	DS[A] = (DS[A] >= D0) ? DS[A] - D0 : DS[A] + D1.	52 (0x34)

Table 12.7 DS Instructions for the Opcode Field (Cont.)

Instruction	Description (C-Function Equivalent)	Decimal/Hex

DS_SWIZZLE_B32	Swizzles input thread data based on offset mask and returns; note does not read or write the DS memory banks.
	
	`offset = offset1:offset0;`
	`// full data sharing within 4 consecutive threads`
	`if (offset[15]) {`
	`    for (i = 0; i < 32; i+=4) {`
	`        thread_out[i+0] = thread_valid[i+offset[1:0]] ?`
	`                    thread_in[i+offset[1:0]] : 0;`
	`        thread_out[i+1] = thread_valid[i+offset[3:2]] ?`
	`                    thread_in[i+offset[3:2]] : 0;`
	`        thread_out[i+2] = thread_valid[i+offset[5:4]] ?`
	`                    thread_in[i+offset[5:4]] : 0;`
	`        thread_out[i+3] = thread_valid[i+offset[7:6]] ?`
	`                    thread_in[i+offset[7:6]] : 0;`
	`    }`
	`}`
	`// limited data sharing within 32 consecutive threads`
	`else {`
	`    and_mask = offset[4:0];`
	`    or_mask = offset[9:5];`
	`    xor_mask = offset[14:10];`
	`    for (i = 0; i < 32; i++) {`
	`        j = ((i & and_mask) | or_mask) ^ xor_mask;`
	`       thread_out[i] = thread_valid[j] ? thread_in[j] :`
	`0;`
	`    }`
	`}`
		53 (0x35)

DS_READ_B32	R = DS[A]; Dword read.	54 (0x36)
DS_READ2_B32	R = DS[ADDR+offset0*4], R+1 = DS[ADDR+offset1*4]. Read 2 Dwords.	55 (0x37)
DS_READ2ST64_B32	R = DS[ADDR+offset0*4*64], R+1 = DS[ADDR+offset1*4*64]. Read 2 Dwords.	56 (0x38)
DS_READ_I8	R = signext(DS[A][7:0]}; signed byte read.	57 (0x39)
DS_READ_U8	R = {24'h0,DS[A][7:0]}; unsigned byte read.	58 (0x3A)
DS_READ_I16	R = signext(DS[A][15:0]}; signed short read.	59 (0x3B)
DS_READ_U16	R = {16'h0,DS[A][15:0]}; unsigned short read.	60 (0x3C)
DS_CONSUME	Consume entries from a buffer.	61 (0x3D)
DS_APPEND	Append one or more entries to a buffer.	62 (0x3E)
DS_ORDERED_COUNT	Increment an append counter. The operation is done in wavefront-creation order.	63 (0x3F)
DS_ADD_U64	Uint add.	64 (0x40)
DS_SUB_U64	Uint subtract.	65 (0x41)
DS_RSUB_U64	Uint reverse subtract.	66 (0x42)
DS_INC_U64	Uint increment.	67 (0x43)
DS_DEC_U64	Uint decrement.	68 (0x44)
DS_MIN_I64	Int min.	69 (0x45)
DS_MAX_I64	Int max.	70 (0x46)
DS_MIN_U64	Uint min.	71 (0x47)
DS_MAX_U64	Uint max.	72 (0x48)
DS_AND_B64	Dword AND.	73 (0x49)
DS_OR_B64	Dword OR.	74 (0x4A)
DS_XOR_B64	Dword XOR.	75 (0x4B)

Table 12.7 DS Instructions for the Opcode Field (Cont.)

Instruction	Description (C-Function Equivalent)	Decimal/Hex
DS_MSKOR_B64	Masked Dword XOR.	76 (0x4C)
DS_WRITE_B64	Write.	77 (0x4D)
DS_WRITE2_B64	DS[ADDR+offset0*8] = D0; DS[ADDR+offset1*8] = D1; write 2 Dwords.	78 (0x4E)
DS_WRITE2ST64_B64	DS[ADDR+offset0*8*64] = D0; DS[ADDR+offset1*8*64] = D1; write 2 Dwords.	79 (0x4F)
DS_CMPST_B64	Compare store.	80 (0x50)
DS_CMPST_F64	Compare store with float rules.	81 (0x51)
DS_MIN_F64	DS[A] = (D0 < DS[A]) ? D0 : DS[A]; float, handles NaN/INF/denorm.	82 (0x52)
DS_MAX_F64	DS[A] = (D0 > DS[A]) ? D0 : DS[A]; float, handles NaN/INF/denorm.	83 (0x53)
DS_ADD_RTN_U64	Uint add.	96 (0x60)
DS_SUB_RTN_U64	Uint subtract.	97 (0x61)
DS_RSUB_RTN_U64	Uint reverse subtract.	98 (0x62)
DS_INC_RTN_U64	Uint increment.	99 (0x63)
DS_DEC_RTN_U64	Uint decrement.	100 (0x64)
DS_MIN_RTN_I64	Int min.	101 (0x65)
DS_MAX_RTN_I64	Int max.	102 (0x66)
DS_MIN_RTN_U64	Uint min.	103 (0x67)
DS_MAX_RTN_U64	Uint max.	104 (0x68)
DS_AND_RTN_B64	Dword AND.	105 (0x69)
DS_OR_RTN_B64	Dword OR.	106 (0x6A)
DS_XOR_RTN_B64	Dword XOR.	107 (0x6B)
DS_MSKOR_RTN_B64	Masked Dword XOR.	108 (0x6C)
DS_WRXCHG_RTN_B64	Write exchange.	109 (0x6D)
DS_WRXCHG2_RTN_B64	Write exchange relative.	110 (0x6E)
DS_WRXCHG2ST64_RTN_B64	Write echange 2 Dwords.	111 (0x6F)
DS_CMPST_RTN_B64	Compare store.	112 (0x70)
DS_CMPST_RTN_F64	Compare store with float rules.	113 (0x71)
DS_MIN_RTN_F64	DS[A] = (D0 < DS[A]) ? D0 : DS[A]; float, handles NaN/INF/denorm.	114 (0x72)
DS_MAX_RTN_F64	DS[A] = (D0 > DS[A]) ? D0 : DS[A]; float, handles NaN/INF/denorm.	115 (0x73)
DS_READ_B64	Dword read.	118 (0x74)
DS_READ2_B64	R = DS[ADDR+offset0*8], R+1 = DS[ADDR+offset1*8]. Read 2 Dwords	119 (0x75)
DS_READ2ST64_B64	R = DS[ADDR+offset0*8*64], R+1 = DS[ADDR+offset1*8*64]. Read 2 Dwords.	120 (0x76)
DS_CONDXCHG32_RTN_B64	Conditional write exchange.	126 (ox7E)
DS_ADD_SRC2_U32	B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0}). DS[A] = DS[A] + DS[B]; uint add.	128 (0x80)
DS_SUB_SRC2_U32	B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0}). DS[A] = DS[A] - DS[B]; uint subtract.	129 (0x81)

Table 12.7 DS Instructions for the Opcode Field (Cont.)

Instruction	Description (C-Function Equivalent)	Decimal/Hex
DS_RSUB_SRC2_U32	B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0}). DS[A] = DS[B] - DS[A]; uint reverse subtract.	130 (0x82)
DS_INC_SRC2_U32	B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0}). DS[A] = (DS[A] >= DS[B] ? 0 : DS[A] + 1); uint increment.	131 (0x83)
DS_DEC_SRC2_U32	B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0}). DS[A] = (DS[A] == 0 || DS[A] > DS[B] ? DS[B] : DS[A] - 1); uint decrement.	132 (0x84)
DS_MIN_SRC2_I32	B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0}). DS[A] = min(DS[A], DS[B]); int min.	133 (0x85)
DS_MAX_SRC2_I32	B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0}). DS[A] = max(DS[A], DS[B]); int max.	134 (0x86)
DS_MIN_SRC2_U32	B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0}). DS[A] = min(DS[A], DS[B]); uint min.	135 (0x87)
DS_MAX_SRC2_U32	B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0}). DS[A] = max(DS[A], DS[B]); uint maxw	136 (0x88)
DS_AND_SRC2_B32	B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0}). DS[A] = DS[A] & DS[B]; Dword AND.	137 (0x89)
DS_OR_SRC2_B32	B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0}). DS[A] = DS[A] | DS[B]; Dword OR.	138 (0x8A)
DS_XOR_SRC2_B32	B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0}). DS[A] = DS[A] ^ DS[B]; Dword XOR.	139 (0x8B)
DS_WRITE_SRC2_B32	B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0}). DS[A] = DS[B]; write Dword.	140 (0x8C)
DS_MIN_SRC2_F32	B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0}). DS[A] = (DS[B] < DS[A]) ? DS[B] : DS[A]; float, handles NaN/INF/denorm.	146 (0x92)
DS_MAX_SRC2_F32	B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0}). DS[A] = (DS[B] > DS[A]) ? DS[B] : DS[A]; float, handles NaN/INF/denorm.	147 (0x93)
DS_ADD_SRC2_U64	Uint add.	192 (0xC0)
DS_SUB_SRC2_U64	Uint subtract.	193 (0xC1)
DS_RSUB_SRC2_U64	Uint reverse subtract.	194 (0xC2)
DS_INC_SRC2_U64	Uint increment.	195 (0xC3)
DS_DEC_SRC2_U64	Uint decrement.	196 (0xC4)
DS_MIN_SRC2_I64	Int min.	197 (0xC5)
DS_MAX_SRC2_I64	Int max.	198 (0xC6)
DS_MIN_SRC2_U64	Uint min.	199 (0xC7)
DS_MAX_SRC2_U64	Uint max.	200 (0xC8)
DS_AND_SRC2_B64	Dword AND.	201 (0xC9
DS_OR_SRC2_B64	Dword OR.	202 (0xCA)

Table 12.7 DS Instructions for the Opcode Field (Cont.)

Instruction	Description (C-Function Equivalent)	Decimal/Hex
DS_XOR_SRC2_B64	Dword XOR.	203 (0xCB)
DS_WRITE_SRC2_B64	B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0}). DS[A] = DS[B]; write Qword.	204 (0xCC)
DS_MIN_SRC2_F64	B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0}). [A] = (D0 < DS[A]) ? D0 : DS[A]; float, handles NaN/INF/denorm.	210 (0xD2
DS_MAX_SRC2_F64	B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0}). [A] = (D0 > DS[A]) ? D0 : DS[A]; float, handles NaN/INF/denorm.	211 (0xD3)
DS_WRITE_B96	{DS[A+2], DS[A+1], DS[A]} = D0[95:0]; tri-dword write.	222 (0xDE)
DS_WRITE_B128	{DS[A+3], DS[A+2], DS[A+1], DS[A]} = D0[127:0]; qword write.	223 (0xDF)
DS_CONDXCHG32_RTN_B128	Conditional write exchange.	253 (0xFD)
DS_READ_B96	Tri-dword read.	254 (0xFE)
DS_READ_B128	Qword read.	255 (0xFF)

12.14 MUBUF Instructions

The bitfield map of the MUBUF format is:

SOFFSET	TFE	SLC	r	SRSRC	VDATA	VADDR	+4

1	**1**	**1**	**0**	**0**	**0**	r	OP	r	LDS	ADDR64	GLC	IDXEN	OFFEN	OFFSET	+0

where:

OFFSET = Unsigned byte offset.
OFFEN = Send offset either as VADDR or as zero..
IDXEN = Send index either as VADDR or as zero.
GLC = Global coherency.
ADDR64 = Buffer address of 64 bits.
LDS = Data read from/written to LDS or VGPR.
OP = Opcode instructions.
VADDR = VGPR address source.
VDATA = Destination vector GPR.
SRSRC = Scalar GPR that specifies resource constant.
SLC = System level coherent.
TFE = Texture fail enable.
SOFFSET = Byte offset added to the memory address.

Table 12.8 MUBUF Instructions for the Opcode Field

Instruction	Description	Decimal (Hex)
LOAD FORMAT		
BUFFER_LOAD_FORMAT_X	Untyped buffer load 1 Dword with format conversion.	0 (0x0)
BUFFER_LOAD_FORMAT_XY	Untyped buffer load 2 Dwords with format conversion.	1 (0x1)
BUFFER_LOAD_FORMAT_XYZ	Untyped buffer load 3 Dwords with format conversion.	2 (0x2)
BUFFER_LOAD_FORMAT_XYZW	Untyped buffer load 4 Dwords with format conversion.	3 (0x3)
STORE FORMAT		
BUFFER_STORE_FORMAT_X	Untyped buffer store 1 Dword with format conversion.	4 (0x4)
BUFFER_STORE_FORMAT_XY	Untyped buffer store 2 Dwords with format conversion.	5 (0x5)
BUFFER_STORE_FORMAT_XYZ	Untyped buffer store 3 Dwords with format conversion.	6 (0x6)
BUFFER_STORE_FORMAT_XYZW	Untyped buffer store 4 Dwords with format conversion.	7 (0x7)

Table 12.8 MUBUF Instructions for the Opcode Field (Cont.)

Instruction	Description	Decimal (Hex)
	LOAD	
BUFFER_LOAD_UBYTE	Untyped buffer load unsigned byte.	8 (0x8)
BUFFER_LOAD_SBYTE	Untyped buffer load signed byte.	9 (0x9)
BUFFER_LOAD_USHORT	Untyped buffer load unsigned short.	10 (0xA)
BUFFER_LOAD_SSHORT	Untyped buffer load signed short.	11 (0xB)
BUFFER_LOAD_DWORD	Untyped buffer load Dword.	12 (0xC)
BUFFER_LOAD_DWORDX2	Untyped buffer load 2 Dwords.	13 (0xD)
BUFFER_LOAD_DWORDX4	Untyped buffer load 4 Dwords.	14 (0xE)
BUFFER_LOAD_DWORDX3	Untyped buffer load 3 Dwords.	15 (0xF)
	STORE	
BUFFER_STORE_BYTE	Untyped buffer store byte.	24 (0x18)
BUFFER_STORE_SHORT	Untyped buffer store short.	26 (0x1A)
BUFFER_STORE_DWORD	Untyped buffer store Dword.	28 (0x1C)
BUFFER_STORE_DWORDX2	Untyped buffer store 2 Dwords.	29 (0x1D)
BUFFER_STORE_DWORDX4	Untyped buffer store 4 Dwords.	30 (0x1E)
BUFFER_STORE_DWORDX3	Untyped buffer store 3 Dwords.	31 (0x1F)
	ATOMIC	
BUFFER_ATOMIC_SWAP	32b. dst=src, returns previous value if glc==1.	48 (0x30)
BUFFER_ATOMIC_CMPSWAP	32b, dst = (dst==cmp) ? src : dst. Returns previous value if glc==1. src comes from the first data-vgpr, cmp from the second.	49 (0x31)
BUFFER_ATOMIC_ADD	32b, dst += src. Returns previous value if glc==1.	50 (0x32)
BUFFER_ATOMIC_SUB	32b, dst -= src. Returns previous value if glc==1.	51 (0x33)
BUFFER_ATOMIC_SMIN	32b, dst = (src < dst) ? src : dst (signed). Returns previous value if glc==1.	53 (0x35)
BUFFER_ATOMIC_UMIN	32b, dst = (src < dst) ? src : dst (unsigned). Returns previous value if glc==1.	54 (0x36)
BUFFER_ATOMIC_SMAX	32b, dst = (src > dst) ? src : dst (signed). Returns previous value if glc==1.	55 (0x37)
BUFFER_ATOMIC_UMAX	32b, dst = (src > dst) ? src : dst (unsigned). Returns previous value if glc==1.	56 (0x38)
BUFFER_ATOMIC_AND	32b, dst &= src. Returns previous value if glc==1.	57 (0x39)
BUFFER_ATOMIC_OR	32b, dst |= src. Returns previous value if glc==1.	58 (0x3A)
BUFFER_ATOMIC_XOR	32b, dst ^= src. Returns previous value if glc==1.	59 (0x3B)
BUFFER_ATOMIC_INC	32b, dst = (dst >= src) ? 0 : dst+1. Returns previous value if glc==1.	60 (0x3C)
BUFFER_ATOMIC_DEC	32b, dst = ((dst==0 || (dst > src)) ? src : dst-1. Returns previous value if glc==1.	61 (0x3D)
BUFFER_ATOMIC_FCMPSWAP	32b , dst = (dst == cmp) ? src : dst, returns previous value if glc==1. Float compare swap (handles NaN/INF/denorm). src comes from the first data-vgpr; cmp from the second.	62 (0x3E)
BUFFER_ATOMIC_FMIN	32b , dst = (src < dst) ? src : dst,. Returns previous value if glc==1. float, handles NaN/INF/denorm.	63 (0x3F)

Table 12.8 MUBUF Instructions for the Opcode Field (Cont.)

Instruction	Description	Decimal (Hex)
BUFFER_ATOMIC_FMAX	32b , dst = (src > dst) ? src : dst, returns previous value if glc==1. float, handles NaN/INF/denorm.	64 (0x40)
BUFFER_ATOMIC_SWAP_X2	64b. dst=src, returns previous value if glc==1.	80 (0x50)
BUFFER_ATOMIC_CMPSWAP_X2	64b, dst = (dst==cmp) ? src : dst. Returns previous value if glc==1. src comes from the first two data-vgprs, cmp from the second two.	81 (0x51)
BUFFER_ATOMIC_ADD_X2	64b, dst += src. Returns previous value if glc==1.	82 (0x52)
BUFFER_ATOMIC_SUB_X2	64b, dst -= src. Returns previous value if glc==1.	83 (0x53)
BUFFER_ATOMIC_SMIN_X2	64b, dst = (src < dst) ? src : dst (signed). Returns previous value if glc==1.	85 (0x55)
BUFFER_ATOMIC_UMIN_X2	64b, dst = (src < dst) ? src : dst (unsigned). Returns previous value if glc==1.	86 (0x56)
BUFFER_ATOMIC_SMAX_X2	64b, dst = (src > dst) ? src : dst (signed). Returns previous value if glc==1.	87 (0x57)
BUFFER_ATOMIC_UMAX_X2	64b, dst = (src > dst) ? src : dst (unsigned). Returns previous value if glc==1.	88 (0x58)
BUFFER_ATOMIC_AND_X2	64b, dst &= src. Returns previous value if glc==1.	89 (0x59)
BUFFER_ATOMIC_OR_X2	64b, dst |= src. Returns previous value if glc==1.	90 (0x5A)
BUFFER_ATOMIC_XOR_X2	64b, dst ^= src. Returns previous value if glc==1.	91 (0x5B
BUFFER_ATOMIC_INC_X2	64b, dst = (dst >= src) ? 0 : dst+1. Returns previous value if glc==1.	92 (0x5C
BUFFER_ATOMIC_DEC_X2	64b, dst = ((dst==0 || (dst > src)) ? src : dst-1. Returns previous value if glc==1.	93 (0x5D0
BUFFER_ATOMIC_FCMPSWAP_X2	64b , dst = (dst == cmp) ? src : dst, returns previous value if glc==1. Double compare swap (handles NaN/INF/denorm). src comes from the first two data-vgprs, cmp from the second two.	94 (0x5E)
BUFFER_ATOMIC_FMIN_X2	64b , dst = (src < dst) ? src : dst, returns previous value if glc==1. Double, handles NaN/INF/denorm.	95 (0x5F)
BUFFER_ATOMIC_FMAX_X2	64b , dst = (src > dst) ? src : dst, returns previous value if glc==1. Double, handles NaN/INF/denorm.	96 (0x60)
Cache Invalidation		
BUFFER_WBINVL1_VOL	Write back and invalidate the shader L1 only for lines of MTYPE SC and GC. Always returns ACK to shader.	112 (0x70)
BUFFER_WBINVL1	Write back and invalidate the shader L1. Always returns ACK to shader.	113 (0x71)

12.15 MTBUF Instructions

The bitfield map of the MTBUF format is:

<table>
<tr><td colspan="7">SOFFSET</td><td>TFE</td><td>SLC</td><td>r</td><td colspan="2">SRSRC</td><td colspan="5">VDATA</td><td>VADDR</td><td>+4</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td colspan="2">NFMT</td><td colspan="3">DFMT</td><td>OP</td><td>ADDR64</td><td>GLC</td><td>IDXEN</td><td>OFFEN</td><td colspan="2">OFFSET</td><td>+0</td></tr>
</table>

where:

OFFSET = Unsigned byte offset.
OFFEN = Send offset either as VADDR or as zero.
IDXEN = Send index either as VADDR or as zero.
GLC = Global coherency.
ADDR64 = Buffer address of 64 bits.
OP = Opcode instructions.
DFMT = Data format for typed buffer.
NFMT = Number format for typed buffer.
VADDR = VGPR address source.
VDATA = Vector GPR for read/write result.
SRSRC = Scalar GPR that specifies resource constant.

Table 12.9 MTBUF Instructions for the Opcode Field

Instruction	Description	Decimal (Hex)
LOAD		
`TBUFFER_LOAD_FORMAT_X`	Typed buffer load 1 Dword with format conversion.	0 (0x0)
`TBUFFER_LOAD_FORMAT_XY`	Typed buffer load 2 Dwords with format conversion.	1 (0x1)
`TBUFFER_LOAD_FORMAT_XYZ`	Typed buffer load 3 Dwords with format conversion.	2 (0x2)
`TBUFFER_LOAD_FORMAT_XYZW`	Typed buffer load 4 Dwords with format conversion.	3 (0x3)
STORE		
`TBUFFER_STORE_FORMAT_X`	Typed buffer store 1 Dword with format conversion.	4 (0x4)
`TBUFFER_STORE_FORMAT_XY`	Typed buffer store 2 Dwords with format conversion.	5 (0x5)
`TBUFFER_STORE_FORMAT_XYZ`	Typed buffer store 3 Dwords with format conversion.	6 (0x6)
`TBUFFER_STORE_FORMAT_XYZW`	Typed buffer store 4 Dwords with format conversion.	7 (0x7)

Table 12.10 NFMT: Shader Num_Format

Value	Encode	Buffer r	Buffer w
0	unorm	yes	yes
1	snorm	yes	yes
2	uscaled	yes	no
3	sscaled	yes	no
4	uint	yes	yes
5	sint	yes	yes
6	snorm_nz	yes	no
7	float	yes	yes
8	reserved		
9	srgb	no	no
10	ubnorm	no	no
11	ubnorm_nz	no	no
12	ubint	no	no
13	ubscaled	no	no

Table 12.11 DFMT: Data_Format

Value	Encode
0	invalid
1	8
2	16
3	8_8
4	32
5	16_16
6	10_11_11
7	11_11_10

Value	Encode
8	10_10_10_2
9	2_10_10_10
10	8_8_8_8
11	32_32
12	16_16_16_16
13	32_32_32
14	32_32_32_32
15	reserved

12.16 MIMG Instructions

The bitfield map of the MTBUF format is:

r	SSAMP	SRSRC	VDATA	VADDR	+4

1	1	1	1	0	0	SLC	OP	LWE	TFE	R128	DA	GLC	UNRM	DMASK	r	+0

where:

DMASK	= Enable mask for image read/write data components.
UNRM	= Force address to be unnormalized.
GLC	= Global coherency.
DA	= Declare an array.
R128	= Texture resource size.
TFE	= Texture fail enable.
LWE	= LOD warning enable.
OP	= Opcode instructions.
SLC	= System level coherent.
VADDR	= VGPR address source.
VDATA	= Vector GPR for read/write result.
SRSRC	= Scalar GPR that specifies resource constant.
SSAMP	= Scalar GPR that specifies sampler constant.

Table 12.12 MIMG Instructions for the Opcode Field

Instruction	Description	Decimal (Hex)
	LOAD	
IMAGE_LOAD	Image memory load with format conversion specified in T#. No sampler.	0 (0x0)
IMAGE_LOAD_MIP	Image memory load with user-supplied mip level. No sampler.	1 (0x1)
IMAGE_LOAD_PCK	Image memory load with no format conversion. No sampler.	2 (0x2)
IMAGE_LOAD_PCK_SGN	Image memory load with with no format conversion and sign extension. No sampler.	3 (0x3)
IMAGE_LOAD_MIP_PCK	Image memory load with user-supplied mip level, no format conversion. No sampler.	4 (0x4)
IMAGE_LOAD_MIP_PCK_SGN	Image memory load with user-supplied mip level, no format conversion and with sign extension. No sampler.	5 (0x5)
	STORE	
IMAGE_STORE	Image memory store with format conversion specified in T#. No sampler.	8 (0x8)
IMAGE_STORE_MIP	Image memory store with format conversion specified in T# to user specified mip level. No sampler.	9 (0x9)

Table 12.12 MIMG Instructions for the Opcode Field (Cont.)

Instruction	Description	Decimal (Hex)
IMAGE_STORE_PCK	Image memory store of packed data without format conversion. No sampler.	10 (0xA)
IMAGE_STORE_MIP_PCK	Image memory store of packed data without format conversion to user-supplied mip level. No sampler.	11 (0xB)
ATOMIC		
IMAGE_ATOMIC_SWAP	dst=src, returns previous value if glc==1.	15 (0xF)
IMAGE_ATOMIC_CMPSWAP	dst = (dst==cmp) ? src : dst. Returns previous value if glc==1.	16 (0x10)
IMAGE_ATOMIC_ADD	dst += src. Returns previous value if glc==1.	17 (0x11)
IMAGE_ATOMIC_SUB	dst -= src. Returns previous value if glc==1.	18 (0x12)
IMAGE_ATOMIC_SMIN	dst = (src < dst) ? src : dst (signed). Returns previous value if glc==1.	20 (0x14)
IMAGE_ATOMIC_UMIN	dst = (src < dst) ? src : dst (unsigned). Returns previous value if glc==1.	21 (0x15)
IMAGE_ATOMIC_SMAX	dst = (src > dst) ? src : dst (signed). Returns previous value if glc==1.	22 (0x16)
IMAGE_ATOMIC_UMAX	dst = (src > dst) ? src : dst (unsigned). Returns previous value if glc==1.	23 (0x17)
IMAGE_ATOMIC_AND	dst &= src. Returns previous value if glc==1.	24 (0x18)
IMAGE_ATOMIC_OR	dst |= src. Returns previous value if glc==1.	25 (0x19)
IMAGE_ATOMIC_XOR	dst |= src. Returns previous value if glc==1.	26 (0x1A)
IMAGE_ATOMIC_INC	dst = (dst >= src) ? 0 : dst+1. Returns previous value if glc==1.	27 (0x1B)
IMAGE_ATOMIC_DEC	dst = ((dst==0 || (dst > src)) ? src : dst-1. Returns previous value if glc==1.	28 (0x1C)
The following three instructions IMAGE_ATOMIC_FCMPSWAP/FMIN/FMX (0x1D, 0x1E, and 0x1F) can operate on floating-point data (either single or double precision). All other DS instructions operate on integers.		
IMAGE_ATOMIC_FCMPSWAP	dst = (dst == cmp) ? src : dst. Returns previous value of dst if glc==1. Double and float atomic compare swap. Obeys floating point compare rules for special values.	29 (0x1D)
IMAGE_ATOMIC_FMIN	dst = (src < dst) ? src : dst, returns previous value of dst if glc==1 - double and float atomic min (handles NaN/INF/denorm).	30 (0x1E)
IMAGE_ATOMIC_FMAX	dst = (src > dst) ? src : dst, returns previous value of dst if glc==1 - double and float atomic min (handles NaN/INF/denorm).	31 (0x1F)
SAMPLE		
IMAGE_SAMPLE	Sample texture map.	32 (0x20)
IMAGE_SAMPLE_CL	Sample texture map, with LOD clamp specified in shader.	33 (0x21)
IMAGE_SAMPLE_D	Sample texture map, with user derivatives.	34 (0x22)
IMAGE_SAMPLE_D_CL	Sample texture map, with LOD clamp specified in shader, with user derivatives.	35 (0x23)
IMAGE_SAMPLE_L	Sample texture map, with user LOD.	36 (0x24)
IMAGE_SAMPLE_B	Sample texture map, with lod bias.	37 (0x25)
IMAGE_SAMPLE_B_CL	Sample texture map, with LOD clamp specified in shader, with lod bias.	38 (0x26)
IMAGE_SAMPLE_LZ	Sample texture map, from level 0.	39 (0x27)

Table 12.12 MIMG Instructions for the Opcode Field (Cont.)

Instruction	Description	Decimal (Hex)
IMAGE_SAMPLE_C	Sample texture map, with PCF.	40 (0x28)
IMAGE_SAMPLE_C_CL	SAMPLE_C, with LOD clamp specified in shader.	41 (0x29)
IMAGE_SAMPLE_C_D	SAMPLE_C, with user derivatives.	42 (0x2A)
IMAGE_SAMPLE_C_D_CL	SAMPLE_C, with LOD clamp specified in shader, with user derivatives.	43 (0x2B)
IMAGE_SAMPLE_C_L	SAMPLE_C, with user LOD.	44 (0x2C)
IMAGE_SAMPLE_C_B	SAMPLE_C, with lod bias.	45 (0x2D)
IMAGE_SAMPLE_C_B_CL	SAMPLE_C, with LOD clamp specified in shader, with lod bias.	46 (0x2E)
IMAGE_SAMPLE_C_LZ	SAMPLE_C, from level 0.	47 (0x2F)
IMAGE_SAMPLE_O	Sample texture map, with user offsets.	48 (0x30)
IMAGE_SAMPLE_CL_O	SAMPLE_O with LOD clamp specified in shader.	49 (0x31)
IMAGE_SAMPLE_D_O	SAMPLE_O, with user derivatives.	50 (0x32)
IMAGE_SAMPLE_D_CL_O	SAMPLE_O, with LOD clamp specified in shader, with user derivatives.	51 (0x33)
IMAGE_SAMPLE_L_O	SAMPLE_O, with user LOD.	52 (0x34)
IMAGE_SAMPLE_B_O	SAMPLE_O, with lod bias.	53 (0x35)
IMAGE_SAMPLE_B_CL_O	SAMPLE_O, with LOD clamp specified in shader, with lod bias.	54 (0x36)
IMAGE_SAMPLE_LZ_O	SAMPLE_O, from level 0.	55 (0x37)
IMAGE_SAMPLE_C_O	SAMPLE_C with user specified offsets.	56 (0x38)
IMAGE_SAMPLE_C_CL_O	SAMPLE_C_O, with LOD clamp specified in shader.	57 (0x39)
IMAGE_SAMPLE_C_D_O	SAMPLE_C_O, with user derivatives.	58 (0x3A)
IMAGE_SAMPLE_C_D_CL_O	SAMPLE_C_O, with LOD clamp specified in shader, with user derivatives.	59 (0x3B)
IMAGE_SAMPLE_C_L_O	SAMPLE_C_O, with user LOD.	60 (0x3C)
IMAGE_SAMPLE_C_B_O	SAMPLE_C_O, with lod bias.	61 (0x3D)
IMAGE_SAMPLE_C_B_CL_O	SAMPLE_C_O, with LOD clamp specified in shader, with lod bias.	62 (0x3E)
IMAGE_SAMPLE_C_LZ_O	SAMPLE_C_O, from level 0.	63 (0x3F)
IMAGE_SAMPLE_CD	Sample texture map, with user derivatives (LOD per quad).	104 (0x68)
IMAGE_SAMPLE_CD_CL	Sample texture map, with LOD clamp specified in shader, with user derivatives (LOD per quad).	105 (0x69)
IMAGE_SAMPLE_C_CD	SAMPLE_C, with user derivatives (LOD per quad).	106 (0x6A)
IMAGE_SAMPLE_C_CD_CL	SAMPLE_C, with LOD clamp specified in shader, with user derivatives (LOD per quad).	107 (0x6B)
IMAGE_SAMPLE_CD_O	SAMPLE_O, with user derivatives (LOD per quad).	108 (0x6C)
IMAGE_SAMPLE_CD_CL_O	SAMPLE_O, with LOD clamp specified in shader, with user derivatives (LOD per quad).	109 (0x6D)
IMAGE_SAMPLE_C_CD_O	SAMPLE_C_O, with user derivatives (LOD per quad).	110 (0x6E)
IMAGE_SAMPLE_C_CD_CL_O	SAMPLE_C_O, with LOD clamp specified in shader, with user derivatives (LOD per quad).	111 (0x6F)
GATHER4		
IMAGE_GATHER4	gather 4 single component elements (2x2).	64 (0x40)

Table 12.12 MIMG Instructions for the Opcode Field (Cont.)

Instruction	Description	Decimal (Hex)
IMAGE_GATHER4_CL	gather 4 single component elements (2x2) with user LOD clamp.	65 (0x41)
IMAGE_GATHER4_L	gather 4 single component elements (2x2) with user LOD.	66 (0x42)
IMAGE_GATHER4_B	gather 4 single component elements (2x2) with user bias.	67 (0x43)
IMAGE_GATHER4_B_CL	gather 4 single component elements (2x2) with user bias and clamp.	68 (0x44)
IMAGE_GATHER4_LZ	gather 4 single component elements (2x2) at level 0.	69 (0x45)
IMAGE_GATHER4_C	gather 4 single component elements (2x2) with PCF.	70 (0x46)
IMAGE_GATHER4_C_CL	gather 4 single component elements (2x2) with user LOD clamp and PCF.	71 (0x47)
IMAGE_GATHER4_C_L	gather 4 single component elements (2x2) with user LOD and PCF.	76 (0x4C)
IMAGE_GATHER4_C_B	gather 4 single component elements (2x2) with user bias and PCF.	77 (0x4D)
IMAGE_GATHER4_C_B_CL	gather 4 single component elements (2x2) with user bias, clamp and PCF.	78 (0x4E)
IMAGE_GATHER4_C_LZ	gather 4 single component elements (2x2) at level 0, with PCF.	79 (0x4F)
IMAGE_GATHER4_O	GATHER4, with user offsets.	80 (0x50)
IMAGE_GATHER4_CL_O	GATHER4_CL, with user offsets.	81 (0x51)
IMAGE_GATHER4_L_O	GATHER4_L, with user offsets.	84 (0x54)
IMAGE_GATHER4_B_O	GATHER4_B, with user offsets.	85 (0x55)
IMAGE_GATHER4_B_CL_O	GATHER4_B_CL, with user offsets.	86 (0x56)
IMAGE_GATHER4_LZ_O	GATHER4_LZ, with user offsets.	87 (0x57)
IMAGE_GATHER4_C_O	GATHER4_C, with user offsets.	88 (0x58)
IMAGE_GATHER4_C_CL_O	GATHER4_C_CL, with user offsets.	89 (0x59)
IMAGE_GATHER4_C_L_O	GATHER4_C_L, with user offsets.	92 (0x5C)
IMAGE_GATHER4_C_B_O	GATHER4_B, with user offsets.	93 (0x5D)
IMAGE_GATHER4_C_B_CL_O	GATHER4_B_CL, with user offsets.	94 (0x5E)
IMAGE_GATHER4_C_LZ_O	GATHER4_C_LZ, with user offsets.	95 (0x5F)
Miscellaneous		
IMAGE_GET_RESINFO	No sampler. Returns resource info into four VGPRs for the specified MIP level. These are 32-bit integer values: Vdata3-0 = { #mipLevels, depth, height, width } For cubemaps, depth = 6 * Number_of_array_slices.	14 (0xE)
IMAGE_GET_LOD	Return calculated LOD.	96 (0x60)

12.17 EXP Instructions

Instruction **`EXPORT`**

Description Transfer vertex position, vertex parameter, pixel color, or pixel depth information to the output buffer.

Every pixel shader must do at least one export to a color, depth or NULL target with the VM bit set to 1. This communicates the pixel-valid mask to the color and depth buffers. Every pixel does only one of the above export types with the DONE bit set to 1.

Vertex shaders must do one or more position exports, and at least one parameter export. The final position export must have the DONE bit set to 1.

Microcode EXP

VSRC3	VSRC2	VSRC1	VSRC0	+4

1	**1**	**1**	**1**	**1**	**0**	r	VM	DONE	COMPR	TARGET	EN	+0

12.18 FLAT Instructions

The bitfield map of the FLAT format is:

<table>
<tr><td colspan="8">VDST</td><td>T
F
E</td><td colspan="2">r</td><td>DATA</td><td>ADDR</td><td>+4</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td colspan="2">OP</td><td>S
L
C</td><td>G
L
C</td><td colspan="2">r</td><td>+0</td></tr>
</table>

where:

GLC	= Global coherency.
SLC	= System level coherency.
OP	= Opcode instructions.
ADDR	= Source of flat address VGPR.
DATA	= Source data.
TFE	= Texture fail enable.
VDST	= Destination VGPR.

Table 12.13 FLAT Instructions for the Opcode Field

Instruction	Description	Decimal (Hex)
	LOAD	
FLAT_LOAD_UBYTE	Flat load unsigned byte. Zero extend to VGPR destination.	8 (0x8)
FLAT_LOAD_SBYTE	Flat load signed byte. Sign extend to VGPR destination.	9 (0x9)
FLAT_LOAD_USHORT	Flat load unsigned short. Zero extebd to VGPR destination.	10 (0xA)
FLAT_LOAD_SSHORT	Flat load signed short. Sign extend to VGPR destination.	11 (0xB)
FLAT_LOAD_DWORD	Flat load Dword.	12 (0xC)
FLAT_LOAD_DWORDX2	Flat load 2 Dwords.	13 (0xD)
FLAT_LOAD_DWORDX4	Flat load 4 Dwords.	14 (0xE)
FLAT_LOAD_DWORDX3	Flat load 3 Dwords.	15 (0xF)
	STORE	
FLAT_STORE_BYTE	Flat store byte.	24 (0x18)
FLAT_STORE_SHORT	Flat store short.	26 (0x1A)
FLAT_STORE_DWORD	Flat store Dword.	28 (0x1C)
FLAT_STORE_DWORDX2	Flat store 2 Dwords.	29 (0x1D)
FLAT_STORE_DWORDX4	Flat store 4 Dwords.	30 (0x1E)
FLAT_STORE_DWORDX3	Flat store 3 Dwords.	31 (0x1F)
	ATOMIC	
FLAT_ATOMIC_SWAP	32b. dst=src, returns previous value if rtn==1.	48 (0x30)

Table 12.13 FLAT Instructions for the Opcode Field (Cont.)

Instruction	Description	Decimal (Hex)
FLAT_ATOMIC_CMPSWAP	32b, dst = (dst==cmp) ? src : dst. Returns previous value if rtn==1. src comes from the first data-VGPR, cmp from the second.	49 (0x31)
FLAT_ATOMIC_ADD	32b, dst += src. Returns previous value if rtn==1.	50 (0x32)
FLAT_ATOMIC_SUB	32b, dst -= src. Returns previous value if rtn==1.	51 (0x33)
FLAT_ATOMIC_SMIN	32b, dst = (src < dst) ? src : dst (signed comparison). Returns previous value if rtn==1.	53 (0x35)
FLAT_ATOMIC_UMIN	32b, dst = (src < dst) ? src : dst (unsigned comparison). Returns previous value if rtn==1.	54 (0x36)
FLAT_ATOMIC_SMAX	32b, dst = (src > dst) ? src : dst (signed comparison). Returns previous value if rtn==1.	55 (0x37)
FLAT_ATOMIC_UMAX	32b, dst = (src > dst) ? src : dst (unsigned comparison). Returns previous value if rtn==1.	56 (0x38)
FLAT_ATOMIC_AND	32b, dst &= src. Returns previous value if rtn==1.	57 (0x39)
FLAT_ATOMIC_OR	32b, dst |= src. Returns previous value if rtn==1.	58 (0x3A)
FLAT_ATOMIC_XOR	32b, dst ^= src. Returns previous value if rtn==1.	59 (0x3B)
FLAT_ATOMIC_INC	32b, dst = (dst >= src) ? 0 : dst+1 (unsigned comparison). Returns previous value if rtn==1.	60 (0x3C)
FLAT_ATOMIC_DEC	32b, dst = ((dst==0 || (dst > src)) ? src : dst-1 (unsigned comparison). Returns previous value if rtn==1.	61 (0x3D)
FLAT_ATOMIC_FCMPSWAP	32b , dst = (dst == cmp) ? src : dst, returns previous value if rtn==1. Floating point compare-swap handles NaN/INF/denorm. src comes from the first data-VGPR; cmp from the second.	62 (0x3E)
FLAT_ATOMIC_FMIN	32b , dst = (src < dst) ? src : dst. Returns previous value if rtn==1. float, handles NaN/INF/denorm.	63 (0x3F)
FLAT_ATOMIC_FMAX	32b , dst = (src > dst) ? src : dst, returns previous value if rtn==1. Floating point compare handles NaN/INF/denorm.	64 (0x40)
FLAT_ATOMIC_SWAP_X2	64b. dst=src, returns previous value if rtn==1.	80 (0x50)
FLAT_ATOMIC_CMPSWAP_X2	64b, dst = (dst==cmp) ? src : dst. Returns previous value if rtn==1. src comes from the first two data-VGPRs, cmp from the second two.	81 (0x51)
FLAT_ATOMIC_ADD_X2	64b, dst += src. Returns previous value if rtn==1.	82 (0x52)
FLAT_ATOMIC_SUB_X2	64b, dst -= src. Returns previous value if rtn==1.	83 (0x53)
FLAT_ATOMIC_SMIN_X2	64b, dst = (src < dst) ? src : dst (signed comparison). Returns previous value if rtn==1.	85 (0x55)
FLAT_ATOMIC_UMIN_X2	64b, dst = (src < dst) ? src : dst (unsigned comparison). Returns previous value if rtn==1.	86 (0x56)
FLAT_ATOMIC_SMAX_X2	64b, dst = (src > dst) ? src : dst (signed comparison). Returns previous value if rtn==1.	87 (0x57)
FLAT_ATOMIC_UMAX_X2	64b, dst = (src > dst) ? src : dst (unsigned comparison). Returns previous value if rtn==1.	88 (0x58)
FLAT_ATOMIC_AND_X2	64b, dst &= src. Returns previous value if rtn==1.	89 (0x59)
FLAT_ATOMIC_OR_X2	64b, dst |= src. Returns previous value if rtn==1.	90 (0x5A)
FLAT_ATOMIC_XOR_X2	64b, dst ^= src. Returns previous value if rtn==1.	91 (0x5B)
FLAT_ATOMIC_INC_X2	64b, dst = (dst >= src) ? 0 : dst+1. Returns previous value if rtn==1.	92 (0x5C)

Table 12.13 FLAT Instructions for the Opcode Field (Cont.)

Instruction	Description	Decimal (Hex)
FLAT_ATOMIC_DEC_X2	64b, dst = ((dst==0 || (dst > src)) ? src : dst - 1. Returns previous value if rtn==1.	93 (0x5D)
FLAT_ATOMIC_FCMPSWAP_X2	64b , dst = (dst == cmp) ? src : dst, returns previous value if rtn==1. Double compare swap (handles NaN/INF/denorm). src comes from the first two data-VGPRs, cmp from the second two.	94 (0x5E)
FLAT_ATOMIC_FMIN_X2	64b , dst = (src < dst) ? src : dst, returns previous value if rtn==1. Double, handles NaN/INF/denorm.	95 (0x5F)
FLAT_ATOMIC_FMAX_X2	64b , dst = (src > dst) ? src : dst, returns previous value if rtn==1. Double, handles NaN/INF/denorm.	96 (0x60)

Chapter 13 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.

Endian Order – The Sea Islands series 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.

Table 13.1 summarizes the microcode formats and their widths. The sections that follow provide details.

Table 13.1 Summary of Microcode Formats

Microcode Formats	Reference	Width (bits)
Scalar ALU and Control Formats		
SOP2 SOPK SOP1 SOPC SOPP	page 13-3 page 13-5 page 13-8 page 13-11 page 13-13	32[1]
Scalar Memory Format		
SMRD	page 13-15	32
Vector ALU Formats		
VOP2 VOP1 VOPC VOP3 (3 input, one output) VOP3 (3 input, two output)	page 13-16 page 13-19 page 13-22 page 13-25 page 13-31	32[1] 32[1] 32[1] 64 64
Vector Parameter Interpolation Format		
VINTRP	page 13-35	32
LDS/GDS Format		
DS	page 13-36	64

Table 13.1 Summary of Microcode Formats (Cont.)

Microcode Formats	Reference	Width (bits)
Vector Memory Buffer Formats		
MUBUF MTBUF	page 13-41 page 13-45	64
Vector Memory Image Format		
MIMG	page 13-48	64
Export Formats		
EXP	page 13-52	64
Flat Formats		
FLAT	page 13-53	64

1. This can be 64-bit with a literal constant.

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 unsigned 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.

The default value of all fields is zero. Any bitfield not identified is assumed to be reserved.

13.1 Scalar ALU and Control Formats

Scalar Format Two Inputs, One Output

Format **SOP2**

Description This is a scalar instruction with two inputs and one output. Can be followed by a 32-bit literal constant.

Opcode

Field Name	Bits	Format
SSRC0	[7:0]	enum(8)
	Source 0. First operand for the instruction. 0 – 103 SGPR0 to SGPR103: Scalar general-purpose registers. 104 – 105 reserved. 106 VCC_LO: vcc[31:0]. 107 VCC_HI: vcc[63:32]. 108 TBA_LO: Trap handler base address [31:0]. 109 TBA_HI: Trap handler base address [63:32]. 110 TMA_LO: Pointer to data in memory used by trap handler. 111 TMA_HI: Pointer to data in memory used by trap handler. 112 – 123 TTMP0 to TTMP11: Trap handler temporary registers (privileged). 124 M0. Memory register 0. 125 reserved. 126 EXEC_LO: exec[31:0]. 127 EXEC_HI: exec[63:32]. 128 0. 129 – 192 Signed integer 1 to 64. 193 – 208 Signed integer -1 to -16. 209 – 239 reserved. 240 0.5. 241 -0.5. 242 1.0. 243 -1.0. 244 2.0. 245 -2.0. 246 4.0. 247 -4.0. 248 – 250 reserved. 251 VCCZ. 252 EXECZ. 253 SCC. 254 reserved. 255 Literal constant.	
SSRC1	[15:8]	enum(8)
	Source 1. Second operand for instruction. Same codes as for SSRC0, above.	
SDST	[22:16]	enum(7)
	Scalar destination for instruction. Same codes as for SSRC0, above, except that this can use only codes 0 to 127.	

Scalar Format Two Inputs, One Output

OP	[29:23]	enum(7)

Opcode.

where the suffix of the instruction specifies the type and size of the result:
D = destination
U = unsigned integer
S = source
SCC = scalar condition code
I = signed integer

B = bitfield

0 `S_ADD_U32`: D.u = S0.u + S1.u. SCC = carry out.
1 `S_SUB_U32`: D.u = S0.u - S1.u. SCC = carry out.
2 `S_ADD_I32`: D.u = S0.i + S1.i. SCC = overflow.
3 `S_SUB_I32`: D.u = S0.i - S1.i. SCC = overflow.
4 `S_ADDC_U32`: D.u = S0.u + S1.u + SCC. SCC = carry-out.
5 `S_SUBB_U32`: D.u = S0.u - S1.u - SCC. SCC = carry-out.
6 `S_MIN_I32`: D.i = (S0.i < S1.i) ? S0.i : S1.i. SCC = 1 if S0 is min.
7 `S_MIN_U32`: D.u = (S0.u < S1.u) ? S0.u : S1.u. SCC = 1 if S0 is min.
8 `S_MAX_I32`: D.i = (S0.i > S1.i) ? S0.i : S1.i. SCC = 1 if S0 is max.
9 `S_MAX_U32`: D.u = (S0.u > S1.u) ? S0.u : S1.u. SCC = 1 if S0 is max.
10 `S_CSELECT_B32`: D.u = SCC ? S0.u : S1.u.
11 `S_CSELECT_B64`: D.u = SCC ? S0.u : S1.u.
12 – 13 reserved.
14 `S_AND_B32`: D.u = S0.u & S1.u. SCC = 1 if result is non-zero.
15 `S_AND_B64`: D.u = S0.u & S1.u. SCC = 1 if result is non-zero.
16 `S_OR_B32`: D.u = S0.u | S1.u. SCC = 1 if result is non-zero.
17 `S_OR_B64`: D.u = S0.u | S1.u. SCC = 1 if result is non-zero.
18 `S_XOR_B32`: D.u = S0.u ^ S1.u. SCC = 1 if result is non-zero.
19 `S_XOR_B64`: D.u = S0.u ^ S1.u. SCC = 1 if result is non-zero.
20 `S_ANDN2_B32`: D.u = S0.u & ~S1.u. SCC = 1 if result is non-zero.
21 `S_ANDN2_B64`: D.u = S0.u & ~S1.u. SCC = 1 if result is non-zero.
22 `S_ORN2_B32`: D.u = S0.u | ~S1.u. SCC = 1 if result is non-zero.
23 `S_ORN2_B64`: D.u = S0.u | ~S1.u. SCC = 1 if result is non-zero.
24 `S_NAND_B32`: D.u = ~(S0.u & S1.u). SCC = 1 if result is non-zero.
25 `S_NAND_B64`: D.u = ~(S0.u & S1.u). SCC = 1 if result is non-zero.
26 `S_NOR_B32`: D.u = ~(S0.u | S1.u). SCC = 1 if result is non-zero.
27 `S_NOR_B64`: D.u = ~(S0.u | S1.u). SCC = 1 if result is non-zero.
28 `S_XNOR_B32`: D.u = ~(S0.u ^ S1.u). SCC = 1 if result is non-zero.
29 `S_XNOR_B64`: D.u = ~(S0.u ^ S1.u). SCC = 1 if result is non-zero.
30 `S_LSHL_B32`: D.u = S0.u << S1.u[4:0]. SCC = 1 if result is non-zero.
31 `S_LSHL_B64`: D.u = S0.u << S1.u[5:0]. SCC = 1 if result is non-zero.
32 `S_LSHR_B32`: D.u = S0.u >> S1.u[4:0]. SCC = 1 if result is non-zero.
33 `S_LSHR_B64`: D.u = S0.u >> S1.u[5:0]. SCC = 1 if result is non-zero.
34 `S_ASHR_I32`: D.i = signtext(S0.i) >> S1.i[4:0]. SCC = 1 if result is non-zero.
35 `S_ASHR_I64`: D.i = signtext(S0.i) >> S1.i[5:0]. SCC = 1 if result is non-zero.
36 `S_BFM_B32`: D.u = ((1 << S0.u[4:0]) - 1) << S1.u[4:0]; bitfield mask.
37 `S_BFM_B64`: D.u = ((1 << S0.u[5:0]) - 1) << S1.u[5:0]; bitfield mask.
38 `S_MUL_I32`: D.i = S0.i * S1.i.

Scalar Format Two Inputs, One Output

	39	S_BFE_U32: Bit field extract. S0 is data, S1[4:0] is field offset, S1[22:16] is field width. D.u = (S0.u >> S1.u[4:0]) & ((1 << S1.u[22:16]) - 1). SCC = 1 if result is non-zero.
	40	S_BFE_I32: Bit field extract. S0 is data, S1[4:0] is field offset, S1[22:16] is field width. D.i = (S0.u >> S1.u[4:0]) & ((1 << S1.u[22:16]) - 1). SCC = 1 if result is non-zero. Test sign-extended result.
	41	S_BFE_U64: Bit field extract. S0 is data, S1[4:0] is field offset, S1[22:16] is field width. D.u = (S0.u >> S1.u[5:0]) & ((1 << S1.u[22:16]) - 1). SCC = 1 if result is non-zero.
	42	S_BFE_I64: Bit field extract. S0 is data, S1[5:0] is field offset, S1[22:16] is field width. D.i = (S0.u >> S1.u[5:0]) & ((1 << S1.u[22:16]) - 1). SCC = 1 if result is non-zero. Test sign-extended result.
	43	S_CBRANCH_G_FORK: Conditional branch using branch stack. Arg0 = compare mask (VCC or any SGPR), Arg1 = 64-bit byte address of target instruction.
	44	S_ABSDIFF_I32: D.i = abs(S0.i >> S1.i). SCC = 1 if result is non-zero.
	All other values are reserved.	
ENCODING	[31:30]	enum(2)
	Must be 1 0 .	

Scalar Instruction One Inline Constant Input, One Output

Format **SOPK**

Description This is a scalar instruction with one inline constant input and one output.

Opcode

Field Name	Bits	Format
SIMM16	[15:0]	int(16)

Scalar Instruction One Inline Constant Input, One Output

`SDST`	[22:16]	enum(7)
	Scalar destination for instruction. Same codes as for SIMM16, above, except that this can use only codes 0 to 127. 16-bit integer input for opcode. Signedness is determined by opcode. 0 – 103 `SGPR0` to `SGPR103`: Scalar general-purpose registers. 104 – 105 reserved. 106 `VCC_LO`: vcc[31:0]. 107 `VCC_HI`: vcc[63:32]. 108 `TBA_LO`: Trap handler base address [31:0]. 109 `TBA_HI`: Trap handler base address [63:32]. 110 `TMA_LO`: Pointer to data in memory used by trap handler. 111 `TMA_HI`: Pointer to data in memory used by trap handler. 112 - 123 `TTMP0` to `TTMP11`: Trap handler temporary registers (privileged). 124 `M0`. Memory register 0. 125 reserved. 126 `EXEC_LO`: exec[31:0]. 127 `EXEC_HI`: exec[63:32]. 128 0. 129 – 192 Signed integer 1 to 64. 193 – 208 Signed integer -1 to -16. 209 – 239 reserved. 240 0.5. 241 -0.5. 242 1.0. 243 -1.0. 244 2.0. 245 -2.0. 246 4.0. 247 -4.0. 248 – 250 reserved. 251 VCCZ. 252 EXECZ. 253 SCC. 254 reserved. 255 Literal constant.	

Scalar Instruction One Inline Constant Input, One Output

OP	[27:23]	enum(5)

Opcode.

where the suffix of the instruction specifies the type and size of the result:
D = destination
U = unsigned integer
S = source
SCC = scalar condition code
I = signed integer

B = bitfield

0 `S_MOVK_I32`: D.i = signext(SIMM16).
1 reserved.
2 `S_CMOVK_I32`: if (SCC) D.i = signext(SIMM16); else NOP.
3 `S_CMPK_EQ_I32`: SCC = (D.i == signext(SIMM16).
4 `S_CMPK_LG_I32`: SCC = (D.i != signext(SIMM16).
5 `S_CMPK_GT_I32`: SCC = (D.i != signext(SIMM16)).
6 `S_CMPK_GE_I32`: SCC = (D.i >= signext(SIMM16)).
7 `S_CMPK_LT_I32`: SCC = (D.i < signext(SIMM16)).
8 `S_CMPK_LE_I32`: SCC = (D.i <= signext(SIMM16)).
9 `S_CMPK_EQ_U32`: SCC = (D.u == SIMM16).
10 `S_CMPK_LG_U32`: SCC = (D.u != SIMM16).
11 `S_CMPK_GT_U32`: SCC = (D.u > SIMM16).
12 `S_CMPK_GE_U32`: SCC = (D.u >= SIMM16).
13 `S_CMPK_LT_U32`: SCC = (D.u < SIMM16).
14 `S_CMPK_LE_U32`: D.u = SCC = (D.u <= SIMM16).
15 `S_ADDK_I32`: D.i = D.i + signext(SIMM16). SCC = overflow.
16 `S_MULK_I32`: D.i = D.i * signext(SIMM16). SCC = overflow.
17 `S_CBRANCH_I_FORK`: Conditional branch using branch-stack.
Arg0(sdst) = compare mask (VCC or any SGPR), SIMM16 = signed DWORD branch offset relative to next instruction.
18 `S_GETREG_B32`: D.u = hardware register. Read some or all of a hardware register into the LSBs of D. SIMM16 = {size[4:0], offset[4:0], hwRegId[5:0]}; offset is 0–31, size is 1–32.
19 `S_SETREG_B32`: hardware register = D.u. Write some or all of the LSBs of D into a hardware register (note that D is a source SGPR).
SIMM16 = {size[4:0], offset[4:0], hwRegId[5:0]}; offset is 0–31, size is 1–32.
20 reserved.
21 `S_SETREG_IMM32_B32`: This instruction uses a 32-bit literal constant. Write some or all of the LSBs of IMM32 into a hardware register.
SIMM16 = {size[4:0], offset[4:0], hwRegId[5:0]}; offset is 0–31, size is 1–32.

All other values are reserved.

ENCODING	[31:28]	enum(4)

Must be `1 0 1 1`.

Scalar Instruction One Input, One Output

Format **SOP1**

Description This is a scalar instruction with one input and one output. Can be followed by a 32-bit literal constant.

Opcode

Field Name	Bits	Format
SSRC0	[7:0]	enum(8)

Source 0. First operand for the instruction.

0 – 103 SGPR0 to SGPR103: Scalar general-purpose registers.
104 – 105 reserved.
106 VCC_LO: vcc[31:0]
107 VCC_HI: vcc[63:32]
108 TBA_LO: Trap handler base address [31:0].
109 TBA_HI: Trap handler base address [63:32].
110 TMA_LO: Pointer to data in memory used by trap handler.
111 TMA_HI: Pointer to data in memory used by trap handler.
112 – 123 TTMP0 to TTMP11: Trap handler temporary registers (privileged).
124 M0. Memory register 0.
125 reserved.
126 EXEC_LO: exec[31:0].
127 EXEC_HI: exec[63:32].
128 0.
129 – 192 Signed integer 1 to 64.
193 – 208 Signed integer -1 to -16.
209 – 239 reserved.
240 0.5.
241 -0.5.
242 1.0.
243 -1.0.
244 2.0.
245 -2.0.
246 4.0.
247 -4.0.
248 – 250 reserved.
251 VCCZ.
252 EXECZ.
253 SCC.
254 reserved.
255 Literal constant.

Scalar Instruction One Input, One Output

OP	[15:8]	enum(8)

0 – 2 reserved.
3 S_MOV_B32: D.u = S0.u.
4 S_MOV_B64: D/u = S0.u.
5 S_CMOV_B32: if(SCC) D.u = S0.u; else NOP.
6 S_CMOV_B64: if(SCC) D.u = S0.u; else NOP.
7 S_NOT_B32: D.u = ~S0.u SCC = 1 if result non-zero.
8 S_NOT_B64: D.u = ~S0.u SCC = 1 if result non-zero.
9 S_WQM_B32: D.u = WholeQuadMode(S0.u). SCC = 1 if result is non-zero.
10 S_WQM_B64 : D.u = WholeQuadMode(S0.u). SCC = 1 if result is non-zero.
11 S_BREV_B32: D.u = S0.u[0:31] (reverse bits).
12 S_BREV_B64: D.u = S0.u[0:63] (reverse bits).
13 S_BCNT0_I32_B32: D.i = CountZeroBits(S0.u). SCC = 1 if result is non-zero.
14 S_BCNT0_I32_B64: D.i = CountZeroBits(S0.u). SCC = 1 if result is non-zero.
15 S_BCNT1_I32_B32: D.i = CountOneBits(S0.u). SCC = 1 if result is non-zero.
16 S_BCNT1_I32_B64: D.i = CountOneBits(S0.u). SCC = 1 if result is non-zero.
17 S_FF0_I32_B32: D.i = FindFirstZero(S0.u) from LSB; if no zeros, return -1.
18 S_FF0_I32_B64: D.i = FindFirstZero(S0.u) from LSB; if no zeros, return -1.
19 S_FF1_I32_B32: D.i = FindFirstOne(S0.u) from LSB; if no ones, return -1.
20 S_FF1_I32_B64: D.i = FindFirstOne(S0.u) from LSB; if no ones, return -1.
21 S_FLBIT_I32_B32: D.i = FindFirstOne(S0.u) from MSB; if no ones, return -1.
22 S_FLBIT_I32_B64: D.i = FindFirstOne(S0.u) from MSB; if no ones, return -1.
23 S_FLBIT_I32: D.i = Find first bit opposite of sign bit from MSB. If S0 == -1, return -1.
24 S_FLBIT_I32_I64: D.i = Find first bit opposite of sign bit from MSB. If S0 == -1, return -1.
25 S_SEXT_I32_I8: D.i = signext(S0.i[7:0]).
26 S_SEXT_I32_I16: D.i = signext(S0.i[15:0]).
27 S_BITSET0_B32: D.u[S0.u[4:0]] = 0.
28 S_BITSET0_B64: D.u[S0.u[5:0]] = 0.
29 S_BITSET1_B32: D.u[S0.u[4:0]] = 1.
30 S_BITSET1_B64: D.u[S0.u[5:0]] = 1.
31 S_GETPC_B64: D.u = PC + 4; destination receives the byte address of the next instruction.
32 S_SETPC_B64: PC = S0.u; S0.u is a byte address of the instruction to jump to.
33 S_SWAPPC_B64: D.u = PC + 4; PC = S0.u.
34 S_RFE_B64: Return from Exception; PC = TTMP1,0.
35 reserved.
36 S_AND_SAVEEXEC_B64: D.u = EXEC, EXEC = S0.u & EXEC. SCC = 1 if the new value of EXEC is non-zero.
37 S_OR_SAVEEXEC_B64: D.u = EXEC, EXEC = S0.u | EXEC. SCC = 1 if the new value of EXEC is non-zero.
38 S_XOR_SAVEEXEC_B64: D.u = EXEC, EXEC = S0.u ^ EXEC. SCC = 1 if the new value of EXEC is non-zero.
39 S_ANDN2_SAVEEXEC_B64: D.u = EXEC, EXEC = S0.u & ~EXEC. SCC = 1 if the new value of EXEC is non-zero.
40 S_ORN2_SAVEEXEC_B64: D.u = EXEC, EXEC = S0.u | ~EXEC. SCC = 1 if the new value of EXEC is non-zero.
41 S_NAND_SAVEEXEC_B64: D.u = EXEC, EXEC = ~(S0.u & EXEC). SCC = 1 if the new value of EXEC is non-zero.

Scalar Instruction One Input, One Output

	42	S_NOR_SAVEEXEC_B64: D.u = EXEC, EXEC = ~(S0.u | EXEC). SCC = 1 if the new value of EXEC is non-zero.	
	43	S_XNOR_SAVEEXEC_B64: D.u = EXEC, EXEC = ~(S0.u ^ EXEC). SCC = 1 if the new value of EXEC is non-zero.	
	44	S_QUADMASK_B32: D.u = QuadMask(S0.u). D[0] = OR(S0[3:0]), D[1] = OR(S0[7:4]) .... SCC = 1 if result is non-zero.	
	45	S_QUADMASK_B64: D.u = QuadMask(S0.u). D[0] = OR(S0[3:0]), D[1] = OR(S0[7:4]) .... SCC = 1 if result is non-zero. Returns a 64-bit result even though the upper 48 bits are zero.	
	46	S_MOVRELS_B32: SGPR[D.u] = SGPR[S0.u + M0.u].	
	47	S_MOVRELS_B64: SGPR[D.u] = SGPR[S0.u + M0.u].	
	48	S_MOVRELD_B32: SGPR[D.u + M0.u] = SGPR[S0.u].	
	49	S_MOVRELD_B64: SGPR[D.u + M0.u] = SGPR[S0.u].	
	50	S_CBRANCH_JOIN: Conditional branch join point. Arg0 = saved CSP value. No dest.	
	51	reserved.	
	52	S_ABS_I32: D.i = abs(S0.i). SCC=1 if result is non-zero.	
	All other values are reserved.		
SDST	[22:16]	enum(7)	
	Scalar destination for instruction. Same codes as for SSRC0, above, except that this can use only codes 0 to 127.		
ENCODING	[31:23]	enum(9)	
	Must be 1 0 1 1 1 1 1 0 1.		

Scalar Instruction Two Inputs, One Comparison

Format **SOPC**

Description Scalar instruction taking two inputs and producing a comparison result. Can be followed by a 32-bit literal constant.

Opcode

Field Name	Bits	Format
SSRC0	[7:0]	enum(8)
	Source 0. First operand for the instruction. 0 – 103 SGPR0 to SGPR103: Scalar general-purpose registers. 104 – 105 reserved. 106 VCC_LO: vcc[31:0] 107 VCC_HI: vcc[63:32] 108 TBA_LO: Trap handler base address [31:0]. 109 TBA_HI: Trap handler base address [63:32]. 110 TMA_LO: Pointer to data in memory used by trap handler. 111 TMA_HI: Pointer to data in memory used by trap handler. 112 – 123 TTMP0 to TTMP11: Trap handler temporary registers (privileged). 124 M0. Memory register 0. 125 reserved. 126 EXEC_LO: exec[31:0]. 127 EXEC_HI: exec[63:32]. 128 0. 129 – 192 Signed integer 1 to 64. 193 – 208 Signed integer -1 to -16. 209 – 239 reserved. 240 0.5. 241 -0.5. 242 1.0. 243 -1.0. 244 2.0. 245 -2.0. 246 4.0. 247 -4.0. 248 – 250 reserved. 251 VCCZ. 252 EXECZ. 253 SCC. 254 reserved. 255 Literal constant.	
SSRC1	[15:8]	enum(8)
	Source 1. Second operand for instruction. Same codes as for SSRC0, above.	

Scalar Instruction Two Inputs, One Comparison

OP	[22:16]	enum(8)
	0	S_CMP_EQ_I32: SCC = (S0.i == S1.i).
	1	S_CMP_LG_I32: SCC = (S0.i != S1.i).
	2	S_CMP_GT_I32: SCC = (S0.i > S1.i).
	3	S_CMP_GE_I32: SCC = (S0.i >= S1.i).
	4	S_CMP_LT_I32: SCC = (S0.i < S1.i).
	5	S_CMP_LE_I32: SCC = (S0.i <= S1.i).
	6	S_CMP_EQ_U32: SCC = (S0.u == S1.u).
	7	S_CMP_LG_U32: SCC = (S0.u != S1.u).
	8	S_CMP_GT_U32: SCC = (S0.u > S1.u).
	9	S_CMP_GE_U32: SCC = (S0.u >= S1.u).
	10	S_CMP_LT_U32: SCC = (S0.u < S1.u).
	11	S_CMP_LE_U32: SCC = (S0.u <= S1.u).
	12	S_BITCMP0_B32: SCC = (S0.u[S1.u[4:0]] == 0).
	13	S_BITCMP1_B32: SCC = (S0.u[S1.u[4:0]] == 1).
	14	S_BITCMP0_B64: SCC = (S0.u[S1.u[5:0]] == 0).
	15	S_BITCMP1_B64: SCC = (S0.u[S1.u[5:0]] == 1).
	16	S_SETVSKIP: VSKIP = S0.u[S1.u[4:0]].
ENCODING	[31:23]	enum(9)
	Must be 1 0 1 1 1 1 1 1 0.	

Scalar Instruction One Input, One Special Operation

Format **SOPP**

Description Scalar instruction taking one inline constant input and performing a special operation (for example: jump).

Opcode

Field Name	Bits	Format
SIMM16	[15:0]	int(16)
	16-bit integer input for opcode. Signedness is determined by opcode.	

Scalar Instruction One Input, One Special Operation

Field	Bits	Description
`OP`	[22:16]	enum(8)
	0	`S_NOP`: do nothing. Repeat NOP 1..8 times based on SIMM16[2:0]. 0 = 1 time, 7 = 8 times.
	1	`S_ENDPGM`: end of program; terminate wavefront.
	2	`S_BRANCH`: PC = PC + signext(SIMM16 * 4) + 4.
	3	reserved.
	4	`S_CBRANCH_SCC0`: if(SCC == 0) then PC = PC + signext(SIMM16 * 4) + 4; else nop.
	5	`S_CBRANCH_SCC1`: if(SCC == 1) then PC = PC + signext(SIMM16 * 4) + 4; else nop.
	6	`S_CBRANCH_VCCZ`: if(VCC == 0) then PC = PC + signext(SIMM16 * 4) + 4; else nop.
	7	`S_CBRANCH_VCCNZ`: if(VCC != 0) then PC = PC + signext(SIMM16 * 4) + 4; else nop.
	8	`S_CBRANCH_EXECZ`: if(EXEC == 0) then PC = PC + signext(SIMM16 * 4) + 4; else nop.
	9	`S_CBRANCH_EXECNZ`: if(EXEC != 0) then PC = PC + signext(SIMM16 * 4) + 4; else nop.
	10	`S_BARRIER`: Sync waves within a thread group.
	11	`S_SETKILL`: Set KILL bit to value of SIMM16[0]. Used primarily for debugging kill wave cmd behavior.
	12	`S_WAITCNT`: Wait for count of outstanding lds, vector-memory and export/vmem-write-data to be at or below the specified levels. simm16[3:0] = vmcount, simm16[6:4] = export/mem-write-data count, simm16[12:8] = LGKM_cnt (scalar-mem/GDS/LDS count).
	13	`S_SETHALT`: set HALT bit to value of SIMM16[0]. 1=halt, 0=resume. Halt is ignored while priv=1.
	14	`S_SLEEP`: Cause a wave to sleep for approximately 64*SIMM16[2:0] clocks.
	15	`S_SETPRIO`: User-settable wave priority. 0 = lowest, 3 = highest.
	16	`S_SENDMSG`: Send a message.
	17	`S_SENDMSGHALT`: Send a message and then HALT.
	18	`S_TRAP`: Enter the trap handler. TrapID = SIMM16[7:0]. Wait for all instructions to complete, save {pc_rewind,trapID,pc} into ttmp0,1; load TBA into PC, set PRIV=1 and continue.
	19	`S_ICACHE_INV`: Invalidate entire L1 I cache.
	20	`S_INCPERFLEVEL`: Increment performance counter specified in SIMM16[3:0] by 1.
	21	`S_DECPERFLEVEL`: Decrement performance counter specified in SIMM16[3:0] by 1.
	22	`S_TTRACEDATA`: Send M0 as user data to thread-trace.
	23	`S_CBRANCH_CDBGSYS`: If (conditional_debug_system != 0) then PC = PC + signext(SIMM16 * 4) + 4; else NOP.
	24	`S_CBRANCH_CDBGUSER`: If (conditional_debug_user != 0) then PC = PC + signext(SIMM16 * 4) + 4; else NOP.
	25	`S_CBRANCH_CDBGSYS_OR_USER`: If (conditional_debug_system || conditional_debug_user) then PC = PC + signext(SIMM16 * 4) + 4; else NOP.
	26	`S_CBRANCH_CDBGSYS_AND_USER`: If (conditional_debug_system && conditional_debug_user) then PC = PC + signext(SIMM16 * 4) + 4; else NOP.
`ENCODING`	[31:23]	enum(9)
		Must be `1 0 1 1 1 1 1 1 1`.

13.2 Scalar Memory Instruction

Scalar Instruction Memory Read

Format **SMRD**

Description Scalar instruction performing a memory read from L1 (constant) memory.

Opcode

Field Name	Bits	Format
OFFSET	[7:0]	int(8)
	Unsigned eight-bit Dword offset to the address specified in SBASE.	
IMM	8	enum(1)
	Boolean. IMM = 0: Specifies an SGPR address that supplies a Dword offset for the memory operation (see enumeration). IMM = 1: Specifies an 8-bit unsigned Dword offset.	
SBASE	[14:9]	enum(6)
	Bits [6:1] of an aligned pair of SGPRs specifying {size[15:0], base[47:0]}, where base and size are in Dword units. The low-order bits are in the first SGPR.	
SDST	[21:15]	enum(7)
	Destination for instruction. 0 – 103 SGPR0 to SGPR103: Scalar general-purpose registers. 104 – 105 reserved. 106 VCC_LO: vcc[31:0]. 107 VCC_HI: vcc[63:32]. 108 TBA_LO: Trap handler base address [31:0]. 109 TBA_HI: Trap handler base address [63:32]. 110 TMA_LO: Pointer to data in memory used by trap handler. 111 TMA_HI: Pointer to data in memory used by trap handler. 112 – 123 TTMP0 to TTMP11: Trap handler temporary registers (privileged). 124 M0. Memory register 0. 125 reserved. 126 EXEC_LO: exec[31:0]. 127 EXEC_HI: exec[63:32].	
OP	[26:21]	enum(8)
	0 S_LOAD_DWORD: Read from read-only constant memory. 1 S_LOAD_DWORDX2: Read from read-only constant memory. 2 S_LOAD_DWORDX4: Read from read-only constant memory. 3 S_LOAD_DWORDX8: Read from read-only constant memory. 4 S_LOAD_DWORDX16: Read from read-only constant memory. 8 S_BUFFER_LOAD_DWORD: Read from read-only constant memory. 9 S_BUFFER_LOAD_DWORDX2: Read from read-only constant memory. 10 S_BUFFER_LOAD_DWORDX4: Read from read-only constant memory. 11 S_BUFFER_LOAD_DWORDX8: Read from read-only constant memory. 12 S_BUFFER_LOAD_DWORDX16: Read from read-only constant memory. 29 S_DCACHE_INV_VOL: Invalidate all volatile lines in L1 constant cache. 30 S_MEMTIME: Return current 64-bit timestamp. 31 S_DCACHE_INV: Invalidate entire L1 K cache. All other values are reserved.	
ENCODING	[31:27]	enum(5)
	Must be 1 1 0 0 0.	

13.3 Vector ALU instructions

Vector Instruction Two Inputs, One Output

Format **VOP2**

Description Vector instruction taking two inputs and producing one output. Can be followed by a 32-bit literal constant.

Opcode

Field Name	Bits	Format
SRC0	[8:0]	enum(9)
	First operand for instruction. Source 0. First operand for the instruction. 0 – 103 SGPR0 to SGPR103: Scalar general-purpose registers. 104 – 105 reserved. 106 VCC_LO: vcc[31:0]. 107 VCC_HI: vcc[63:32]. 108 TBA_LO: Trap handler base address [31:0]. 109 TBA_HI: Trap handler base address [63:32]. 110 TMA_LO: Pointer to data in memory used by trap handler. 111 TMA_HI: Pointer to data in memory used by trap handler. 112 – 123 TTMP0 to TTMP11: Trap handler temporary registers (privileged). 124 M0. Memory register 0. 125 reserved. 126 EXEC_LO: exec[31:0]. 127 EXEC_HI: exec[63:32]. 128 0. 129 – 192 Signed integer 1 to 64. 193 – 208 Signed integer -1 to -16. 209 – 239 reserved. 240 0.5 241 -0.5. 242 1.0. 243 -1.0. 244 2.0. 245 -2.0. 246 4.0. 247 -4.0. 248 – 250 reserved. 251 VCCZ. 252 EXECZ. 253 SCC. 254 LDS direct. 255 Literal constant. 256 – 511 Vector General-Purpose Registers (VGPRs) 0 – 255.	
VSRC1	[16:9]	enum(8)
	Second operand for instruction. 0 - 255 Vector General-Purpose Registers (VGPRs) 0 - 255.	
VDST	[24:17]	enum(8)
	Destination for instruction. 0 - 255 Vector General-Purpose Registers (VGPRs) 0 - 255.	

Vector Instruction Two Inputs, One Output

OP [30:25] enum(7)

0 V_CNDMASK_B32: D.u = VCC[i] ? S1.u : S0.u (i = threadID in wave).
1 V_READLANE_B32: copy one VGPR value to one SGPR. Dst = SGPR-dest, Src0 = Source Data (VGPR# or M0(lds-direct)), Src1 = Lane Select (SGPR or M0). Ignores exec mask.
2 V_WRITELANE_B32: Write value into one VGPR lane. Dst = VGPR-dest, Src0 = Source Data (sgpr, m0, exec or constants), Src1 = Lane Select (SGPR or M0). Ignores exec mask.
3 V_ADD_F32: D.f = S0.f + S1.f.
4 V_SUB_F32: D.f = S0.f - S1.f.
5 V_SUBREV_F32: D.f = S1.f - S0.f.
6 V_MAC_LEGACY_F32: D.f = S0.F * S1.f + D.f.
7 V_MUL_LEGACY_F32: D.f = S0.f * S1.f (DX9 rules, 0.0*x = 0.0).
8 V_MUL_F32: D.f = S0.f * S1.f.
9 V_MUL_I32_I24: D.i = S0.i[23:0] * S1.i[23:0].
10 V_MUL_HI_I32_I24: D.i = (S0.i[23:0] * S1.i[23:0])>>32.
11 V_MUL_U32_U24: D.u = S0.u[23:0] * S1.u[23:0].
12 V_MUL_HI_U32_U24: D.i = (S0.u[23:0] * S1.u[23:0])>>32.
13 V_MIN_LEGACY_F32: D.f = min(S0.f, S1.f) (DX9 rules for NaN).
14 V_MAX_LEGACY_F32: D.f = max(S0.f, S1.f) (DX9 rules for NaN).
15 V_MIN_F32: D.f = min(S0.f, S1.f).
16 V_MAX_F32: D.f = max(S0.f, S1.f).
17 V_MIN_I32: D.i = min(S0.i, S1.i).
18 V_MAX_I32: D.i = max(S0.i, S1.i).
19 V_MIN_U32: D.u = min(S0.u, S1.u).
20 V_MAX_U32: D.u = max(S0.u, S1.u).
21 V_LSHR_B32: D.u = S0.u >> S1.u[4:0].
22 V_LSHRREV_B32: D.u = S1.u >> S0.u[4:0].
23 V_ASHR_I32: D.i = S0.i >> S1.i[4:0].
24 V_ASHRREV_I32: D.i = S1.i >> S0.i[4:0].
25 V_LSHL_B32: D.u = S0.u << S1.u[4:0].
26 V_LSHLREV_B32: D.u = S1.u << S0.u[4:0].
27 V_AND_B32: D.u = S0.u & S1.u.
28 V_OR_B32: D.u = S0.u | S1.u.
29 V_XOR_B32: D.u = S0.u ^ S1.u.
30 V_BFM_B32: D.u = ((1<<S0.u[4:0])-1) << S1.u[4:0]; S0=bitfield_width, S1=bitfield_offset.
31 V_MAC_F32: D.f = S0.f * S1.f + D.f.
32 V_MADMK_F32: D.f = S0.f * K + S1.f; K is a 32-bit inline constant.
33 V_MADAK_F32: D.f = S0.f * S1.f + K; K is a 32-bit inline constant.
34 V_BCNT_U32_B32: D.u = CountOneBits(S0.u) + S1.u. Bit count.
35 V_MBCNT_LO_U32_B32: ThreadMask = (1 << ThreadPosition) - 1; D.u = CountOneBits(S0.u & ThreadMask[31:0]) + S1.u. Masked bit count, ThreadPosition is the position of this thread in the wavefront (in 0 63).
36 V_MBCNT_HI_U32_B32: ThreadMask = (1 << ThreadPosition) - 1; D.u = CountOneBits(S0.u & ThreadMask[63:32]) + S1.u. Masked bit count, ThreadPosition is the position of this thread in the wavefront (in 0..63).

Vector Instruction Two Inputs, One Output

		37 V_ADD_I32: D.u = S0.u + S1.u; VCC=carry-out (VOP3:sgpr=carry-out). 38 V_SUB_I32: D.u = S0.u - S1.u; VCC=carry-out (VOP3:sgpr=carry-out). 39 V_SUBREV_I32: D.u = S1.u - S0.u; VCC=carry-out (VOP3:sgpr=carry-out). 40 V_ADDC_U32: D.u = S0.u + S1.u + VCC; VCC=carry-out (VOP3:sgpr=carry-out, S2.u=carry-in). 41 V_SUBB_U32: D.u = S0.u - S1.u - VCC; VCC=carry-out (VOP3:sgpr=carry-out, S2.u=carry-in). 42 V_SUBBREV_U32: D.u = S1.u - S0.u - VCC; VCC=carry-out (VOP3:sgpr=carry-out, S2.u=carry-in). 43 V_LDEXP_F32: D.d = pow(S0.f, S1.i). 44 V_CVT_PKACCUM_U8_F32: f32→u8(s0.f), pack into byte(s1.u), of dst. 45 V_CVT_PKNORM_I16_F32: D = {(snorm)S1.f, (snorm)S0.f}. 46 V_CVT_PKNORM_U16_F32: D = {(unorm)S1.f, (unorm)S0.f}. 47 V_CVT_PKRTZ_F16_F32: D = {flt32_to_flt16(S1.f),flt32_to_flt16(S0.f)}, with round-toward-zero. 48 V_CVT_PK_U16_U32: D = {(u32→u16)S1.u, (u32→u16)S0.u}. 49 V_CVT_PK_I16_I32: D = {(i32→i16)S1.i, (i32→i16)S0.i}. All other values are reserved.
	Encode	31 enum(1) Must be 0.

Vector Instruction One Input, One Output

Format **VOP1**

Description Vector instruction taking one input and producing one output. Can be followed by a 32-bit literal constant.

Opcode

Field Name	Bits	Format
SRC0	[8:0]	enum(9)
	First operand for instruction. Source 0. First operand for the instruction. 0 – 103 SGPR0 to SGPR103: Scalar general-purpose registers. 104 – 105 reserved. 106 VCC_LO: vcc[31:0]. 107 VCC_HI: vcc[63:32]. 108 TBA_LO: Trap handler base address [31:0]. 109 TBA_HI: Trap handler base address [63:32]. 110 TMA_LO: Pointer to data in memory used by trap handler. 111 TMA_HI: Pointer to data in memory used by trap handler. 112 – 123 TTMP0 to TTMP11: Trap handler temporary registers (privileged). 124 M0. Memory register 0. 125 reserved. 126 EXEC_LO: exec[31:0]. 127 EXEC_HI: exec[63:32]. 128 0. 129 – 192 Signed integer 1 to 64. 193 – 208 Signed integer -1 to -16. 209 – 239 reserved. 240 0.5. 241 -0.5. 242 1.0. 243 -1.0. 244 2.0. 245 -2.0. 246 4.0. 247 -4.0. 248 – 250 reserved. 251 VCCZ. 252 EXECZ. 253 SCC. 254 LDS direct. 255 Literal constant. 256 – 511 Vector General-Purpose Registers (VGPRs) 0 – 255.	

Vector Instruction One Input, One Output (Cont.)

OP	[16:9]	enum(8)
	0	V_NOP: do nothing.
	1	V_MOV_B32: D.u = S0.u.
	2	V_READFIRSTLANE_B32: copy one VGPR value to one SGPR. Dst = SGPR-dest, Src0 = Source Data (VGPR# or M0(lds-direct)), Lane# = FindFirst1fromLSB(exec) (lane = 0 if exec is zero). Ignores exec mask.
	3	V_CVT_I32_F64: D.i = (int)S0.d.
	4	V_CVT_F64_I32: D.f = (float)S0.i.
	5	V_CVT_F32_I32: D.f = (float)S0.i.
	6	V_CVT_F32_U32: D.f = (float)S0.u.
	7	V_CVT_U32_F32: D.u = (unsigned)S0.f.
	8	V_CVT_I32_F32: D.i = (int)S0.f.
	9	reserved.
	10	V_CVT_F16_F32: D.f16 = flt32_to_flt16(S0.f).
	11	V_CVT_F32_F16: D.f = flt16_to_flt32(S0.f16).
	12	V_CVT_RPI_I32_F32: D.i = (int)floor(S0.f + 0.5).
	13	V_CVT_FLR_I32_F32: D.i = (int)floor(S0.f).
	14	V_CVT_OFF_F32_I4: 4-bit signed int to 32-bit float. For interpolation in shader.
	15	V_CVT_F32_F64: D.f = (float)S0.d.
	16	V_CVT_F64_F32: D.d = (double)S0.f.
	17	V_CVT_F32_UBYTE0: D.f = UINT2FLT(S0.u[7:0]).
	18	V_CVT_F32_UBYTE1: D.f = UINT2FLT(S0.u[15:8]).
	19	V_CVT_F32_UBYTE2: D.f = UINT2FLT(S0.u[23:16]).
	20	V_CVT_F32_UBYTE3: D.f = UINT2FLT(S0.u[31:24]).
	21	V_CVT_U32_F64: D.u = (uint)S0.d.
	22	V_CVT_F64_U32: D.d = (double)S0.u.
	23	V_TRUNC_F64: D.d = trunc(S0.d), return integer part of S0.d.
	24	V_CEIL_F64: D.d = trunc(S0.d); if (S0.d > 0.0 && S0.d != D.d), D.d += 1.0.
	25	V_RNDNE_F64: D.d = round_nearest_even(S0.d).
	26	V_FLOOR_F64: D.d = trunc(S0.d); if (S0.d < 0.0 && S0.d != D.d), D.d += -1.0.
	27 – 31	reserved.
	32	V_FRACT_F32: D.f = S0.f - floor(S0.f).
	33	V_TRUNC_F32: D.f = trunc(S0.f), return integer part of S0.
	34	V_CEIL_F32: D.f = ceil(S0.f). Implemented as: D.f = trunc(S0.f); if (S0 > 0.0 && S0 != D), D += 1.0.
	35	V_RNDNE_F32: D.f = round_nearest_even(S0.f).
	36	V_FLOOR_F32: D.f = trunc(S0); if ((S0 < 0.0) && (S0 != D)) D += -1.0.
	37	V_EXP_F32: D.f = pow(2.0, S0.f).
	38	V_LOG_CLAMP_F32: D.f = log2(S0.f), clamp -infinity to -max_float.
	39	V_LOG_F32: D.f = log2(S0.f).
	40	V_RCP_CLAMP_F32: D.f = 1.0 / S0.f, result clamped to +-max_float.
	41	V_RCP_LEGACY_F32: D.f = 1.0 / S0.f, +-infinity result clamped to +-0.0.
	42	V_RCP_F32: D.f = 1.0 / S0.f.
	43	V_RCP_IFLAG_F32: D.f = 1.0 / S0.f, only integer div_by_zero flag can be raised.
	44	V_RSQ_CLAMP_F32: D.f = 1.0 / sqrt(S0.f), result clamped to +-max_float.
	45	V_RSQ_LEGACY_F32: D.f = 1.0 / sqrt(S0.f).
	46	V_RSQ_F32: D.f = 1.0 / sqrt(S0.f).
	47	V_RCP_F64: D.d = 1.0 / (S0.d).
	48	V_RCP_CLAMP_F64: D.f = 1.0 / (S0.f), result clamped to +-max_float.

Vector Instruction One Input, One Output (Cont.)

	49	V_RSQ_F64: D.f = 1.0 / sqrt(S0.f).
	50	V_RSQ_CLAMP_F64: D.d = 1.0 / sqrt(S0.d), result clamped to +-max_float.
	51	V_SQRT_F32: D.f = sqrt(S0.f).
	52	V_SQRT_F64: D.d = sqrt(S0.d).
	53	V_SIN_F32: D.f = sin(S0.f).
	54	V_COS_F32: D.f = cos(S0.f).
	55	V_NOT_B32: D.u = ~S0.u.
	56	V_BFREV_B32: D.u[31:0] = S0.u[0:31], bitfield reverse.
	57	V_FFBH_U32: D.u = position of first 1 in S0 from MSB; D=0xFFFFFFFF if S0==0.
	58	V_FFBL_B32: D.u = position of first 1 in S0 from LSB; D=0xFFFFFFFF if S0==0.
	59	V_FFBH_I32: D.u = position of first bit different from sign bit in S0 from MSB; D=0xFFFFFFFF if S0==0 or 0xFFFFFFFF.
	60	V_FREXP_EXP_I32_F64: See V_FREXP_EXP_I32_F32.
	61	V_FREXP_MANT_F64: See V_FREXP_MANT_F32.
	62	V_FRACT_F64: S0.d - floor(S0.d).
	63	V_FREXP_EXP_I32_F32: If (S0.f == INF || S0.f == NAN), then D.i = 0; else D.i = TwosComplement(Exponent(S0.f) - 127 + 1). Returns exponent of single precision float input, such that S0.f = significand * (2 ** exponent). See also FREXP_MANT_F32, which returns the significand.
	64	V_FREXP_MANT_F32: if (S0.f == INF || S0.f == NAN) then D.f = S0.f; else D.f = Mantissa(S0.f). Result range is in (-1.0,-0.5][0.5,1.0) in normal cases. Returns binary significand of single precision float input, such that S0.f = significand * (2 ** exponent). See also FREXP_EXP_I32_F32, which returns integer exponent.
	65	V_CLREXCP: Clear wave's exception state in SIMD(SP).
	66	V_MOVRELD_B32: VGPR[D.u + M0.u] = VGPR[S0.u].
	67	V_MOVRELS_B32: VGPR[D.u] = VGPR[S0.u + M0.u].
	68	V_MOVRELSD_B32: VGPR[D.u + M0.u] = VGPR[S0.u + M0.u].
	69	V_LOG_LEGACY_F32: D.f = log2(S0.f). Base 2 logarithm. Same as Sea Islands.
	70	V_EXP_LEGACY_F32: D.f = pow(2.0, S0.f). Same as Southern Islands.
	All other values are reserved.	
VDST	[24:17	enum(8)
	Destination for instruction.	
	0 – 255	Vector General-Purpose Registers (VGPRs) 0 – 255.
ENCODE	[31:25]	enum(7)
	Must be 0 1 1 1 1 1 1.	

Vector Instruction Two Inputs, One Comparison Result

Format **VOPC**

Description Vector instruction taking two inputs and producing a comparison result. Can be followed by a 32-bit literal constant.

Vector Comparison operations are divided into three groups:

- those which can use any one of 16 comparison operations,
- those which can use any one of 8, and
- those which have only a single comparison operation.

The final opcode number is determined by adding the base for the opcode family plus the offset from the compare op.

Every compare instruction writes a result to VCC (for VOPC) or an SGPR (for VOP3). Additionally, every compare instruction has a variant that also writes to the EXEC mask.

The destination of the compare result is always VCC when encoded using the VOPC format, and can be an arbitrary SGPR when encoded in the VOP3 format.

Opcode

Field Name	Bits	Format
SRC0	[8:0]	enum(9)

First operand for instruction.

Source 0. First operand for the instruction.

0 – 103 SGPR0 to SGPR103: Scalar general-purpose registers.
104 – 105 reserved.
106 VCC_LO: vcc[31:0].
107 VCC_HI: vcc[63:32].
108 TBA_LO: Trap handler base address [31:0].
109 TBA_HI: Trap handler base address [63:32].
110 TMA_LO: Pointer to data in memory used by trap handler.
111 TMA_HI: Pointer to data in memory used by trap handler.
112 – 123 TTMP0 to TTMP11: Trap handler temporary registers (privileged).
124 M0. Memory register 0.
125 reserved.
126 EXEC_LO: exec[31:0].
127 EXEC_HI: exec[63:32].
128 0.
129 – 192: Signed integer 1 to 64.
193 – 208: Signed integer -1 to -16.
209 – 239: reserved.
240 0.5.
241 -0.5.
242 1.0.
243 -1.0.
244 2.0.
245 -2.0.
246 4.0.
247 -4.0.
248 – 250 reserved.
251 VCCZ.
252 EXECZ.
253 SCC.
254 LDS direct.
255 Literal constant.
256 - 511 Vector General-Purpose Registers (VGPRs) 0 - 255.

Vector Instruction Two Inputs, One Comparison Result

VSRC1	[16:9]	enum(8)	Second operand for instruction. 0 – 255 Vector General-Purpose Registers (VGPRs) 0 - 255.
OP	[24:17]	enum(8)	See compare operations below.

Sixteen Compare Operations (OP16)

Compare Operation	Opcode Offset	Description
F	0	D.u = 0
LT	1	D.u = (S0 < S1)
EQ	2	D.u = (S0 == S1)
LE	3	D.u = (S0 <= S1)
GT	4	D.u = (S0 > S1)
LG	5	D.u = (S0 <> S1)
GE	6	D.u = (S0 >= S1)
O	7	D.u = (!isNaN(S0) && !isNaN(S1))
U	8	D.u = (!isNaN(S0) || !isNaN(S1))
NGE	9	D.u = !(S0 >= S1)
NLG	10	D.u = !(S0 <> S1)
NGT	11	D.u = !(S0 > S1)
NLE	12	D.u = !(S0 <= S1)
NEQ	13	D.u = !(S0 == S1)
NLT	14	D.u = !(S0 < S1)
TRU	15	D.u = 1

Eight Compare Operations (OP8)

Compare Operation	Opcode Offset	Description
F	0	D.u = 0
LT	1	D.u = (S0 < S1)
EQ	2	D.u = (S0 == S1)
LE	3	D.u = (S0 <= S1)
GT	4	D.u = (S0 > S1)
LG	5	D.u = (S0 <> S1)
GE	6	D.u = (S0 >= S1)
TRU	7	D.u = 1

Vector Instruction Two Inputs, One Comparison Result

	Single Vector Compare Operations		
	Opcode Family	Opcode Base	Description
	V_CMP_{OP16}_F32	0x00	Signal on sNaN input only.
	V_CMPX_{OP16}_F32	0x10	Signal on sNaN input only. Also write EXEC.
	V_CMP_{OP16}_F64	0x20	Signal on sNaN input only.
	V_CMPX_{OP16}_F64	0x30	Signal on sNaN input only. Also write EXEC.
	V_CMPS_{OP16}_F32	0x40	Signal on any NaN.
	V_CMPSX_{OP16}_F32	0x50	Signal on any NaN. Also write EXEC.
	V_CMPS_{OP16}_F64	0x60	Signal on any NaN.
	V_CMPSX_{OP16}_F64	0x70	Signal on any NaN. Also write EXEC.
	V_CMP_{OP8}_I32	0x80	On 32-bit integers.
	V_CMPX_{OP8}_I32	0x90	Also write EXEC.
	V_CMP_{OP8}_I64	0xA0	On 64-bit integers.
	V_CMPX_{OP8}_I64	0xB0	Also write EXEC.
	V_CMP_{OP8}_U32	0xC0	On unsigned 32-bit integers.
	V_CMPX_{OP8}_U32	0xD0	Also write EXEC.
	V_CMP_{OP8}_U64	0xE0	On unsigned 64-bit integers.
	V_CMPX_{OP8}_U64	0xF0	Also write EXEC.
	V_CMP_CLASS_F32	0x88	D = IEEE numeric class function specified in S1.u, performed on S0.f.
	V_CMPX_CLASS_F32	0x98	D = IEEE numeric class function specified in S1.u, performed on S0.f. Also write EXEC.
	V_CMP_CLASS_F64	0xA8	D = IEEE numeric class function specified in S1.u, performed on S0.d.
	V_CMPX_CLASS_F64	0xB8	D = IEEE numeric class function specified in S1.u, performed on S0.d. Also write EXEC.
ENCODE	[31:25] enum(7) Must be `0 1 1 1 1 1 0`.		

Vector Instruction, Three Inputs, One Output (VOP3a)

Format **VOP3**

Description Vector instruction taking three inputs and producing one output.

Opcode

Field Name	Bits	Format
VDST	[7:0]	enum(8)
	Destination for instruction in the Vector General-Purpose Registers (VGPR [255:0]). For V_CMP instructions, this field specifies the SGPR or VCC that receives the result of the comparison.	
ABS	[10:8]	enum(3)
	If ABS[N] is set, take the floating-point absolute value of the N'th input operand. This is applied before negation.	
CLAMP	11	enum(1)
	If set, clamp output to [0.0, 1.0]. Applied after output modifier.	
reserved	[16:12]	
	Reserved.	
OP	[25:17]	enum(9)
	0 – 255 Are the VOPC opcodes when VOP3 encoding is required. For example, 0 + V_CMP_F_F32 generates the VOP3 version of V_CMP_F_F32. 256 V_CNDMASK_B32: D.u = S2[i] ? S1.u : S0.u (i = threadID in wave). 257 V_READLANE_B32: copy one VGPR value to one SGPR. Dst = SGPR-dest, Src0 = Source Data (VGPR# or M0(lds-direct)), Src1 = Lane Select (SGPR or M0). Ignores exec mask. 258 V_WRITELANE_B32: Write value into one VGPR lane. Dst = VGPR-dest, Src0 = Source Data (sgpr, m0, exec or constants), Src1 = Lane Select (SGPR or M0). Ignores exec mask. 259 V_ADD_F32: D.f = S0.f + S1.f. 260 V_SUB_F32: D.f = S0.f - S1.f. 261 V_SUBREV_F32: D.f = S1.f - S0.f. 262 V_MAC_LEGACY_F32: D.f = S0.F * S1.f + D.f. 263 V_MUL_LEGACY_F32: D.f = S0.f * S1.f (DX9 rules, 0.0*x = 0.0). 264 V_MUL_F32: D.f = S0.f * S1.f. 265 V_MUL_I32_I24: D.i = S0.i[23:0] * S1.i[23:0]. 266 V_MUL_HI_I32_I24: D.i = (S0.i[23:0] * S1.i[23:0])>>32. 267 V_MUL_U32_U24: D.u = S0.u[23:0] * S1.u[23:0]. 268 V_MUL_HI_U32_U24: D.i = (S0.u[23:0] * S1.u[23:0])>>32. 269 V_MIN_LEGACY_F32: D.f = min(S0.f, S1.f) (DX9 rules for NaN). 270 V_MAX_LEGACY_F32: D.f = max(S0.f, S1.f) (DX9 rules for NaN). 271 V_MIN_F32: D.f = min(S0.f, S1.f). 272 V_MAX_F32: D.f = max(S0.f, S1.f). 273 V_MIN_I32: D.i = min(S0.i, S1.i). 274 V_MAX_I32: D.i = max(S0.i, S1.i). 275 V_MIN_U32: D.u = min(S0.u, S1.u). 276 V_MAX_U32: D.u = max(S0.u, S1.u). 277 V_LSHR_B32: D.u = S0.u >> S1.u[4:0]. 278 V_LSHRREV_B32: D.u = S1.u >> S0.u[4:0]. 279 V_ASHR_I32: D.i = S0.i >> S1.i[4:0]. 280 V_ASHRREV_I32: D.i = S1.i >> S0.i[4:0]. 281 V_LSHL_B32: D.u = S0.u << S1.u[4:0]. 282 V_LSHLREV_B32: D.u = S1.u << S0.u[4:0].	

Vector Instruction, Three Inputs, One Output (VOP3a)

283 `V_AND_B32`: D.u = S0.u & S1.u.
284 `V_OR_B32`: D.u = S0.u | S1.u.
285 `V_XOR_B32`: D.u = S0.u ^ S1.u.
286 `V_BFM_B32`: D.u = ((1<<S0.u[4:0])-1) << S1.u[4:0]; S0=bitfield_width, S1=bitfield_offset.
287 `V_MAC_F32`: D.f = S0.f * S1.f + D.f.
288 - 289 reserved.
290 `V_BCNT_U32_B32`: D.u = CountOneBits(S0.u) + S1.u. Bit count.
291 `V_MBCNT_LO_U32_B32`: ThreadMask = (1 << ThreadPosition) - 1; D.u = CountOneBits(S0.u & ThreadMask[31:0]) + S1.u. Masked bit count, ThreadPosition is the position of this thread in the wavefront (in 0..63).
292 `V_MBCNT_HI_U32_B32`: ThreadMask = (1 << ThreadPosition) - 1; D.u = CountOneBits(S0.u & ThreadMask[63:32]) + S1.u. Masked bit count, ThreadPosition is the position of this thread in the wavefront (in 0..63).
293 – 298 See corresponding opcode numbers in VOP3 (3 in, 2 out).
299 `V_LDEXP_F32`: D.d = pow(S0.f, S1.i).
300 `V_CVT_PKACCUM_U8_F32`: f32→u8(s0.f), pack into byte(s1.u), of dst.
301 `V_CVT_PKNORM_I16_F32`: D = {(snorm)S1.f, (snorm)S0.f}.
302 `V_CVT_PKNORM_U16_F32`: D = {(unorm)S1.f, (unorm)S0.f}.
303 `V_CVT_PKRTZ_F16_F32`: D = {flt32_to_flt16(S1.f),flt32_to_flt16(S0.f)}, with round-toward-zero.
304 `V_CVT_PK_U16_U32`: D = {(u32→u16)S1.u, (u32→u16)S0.u}.
305 `V_CVT_PK_I16_I32`: D = {(i32→i16)S1.i, (i32→i16)S0.i}.
318 – 319 Do not use (maps to VOP1 and VOPC).
320 – 372 Are VOP3a-only opcodes.
320 `V_MAD_LEGACY_F32` = D.f = S0.f * S1.f + S2.f (DX9 rules, 0.0*x = 0.0).
321 `V_MAD_F32` = D.f = S0.f * S1.f + S2.f.
322 `V_MAD_I32_I24` = D.i = S0.i * S1.i + S2.iD.i = S0.i * S1.i + S2.i.
323 `V_MAD_U32_U24` = D.u = S0.u * S1.u + S2.u.
324 `V_CUBEID_F32` = Rm.w <- Rn,x, Rn,y, Rn.z.
325 `V_CUBESC_F32` = Rm.y <- Rn,x, Rn,y, Rn.z.
326 `V_CUBETC_F32` = Rm.x <- Rn,x, Rn,y, Rn.z.
327 `V_CUBEMA_F32` = Rm.z <- Rn,x, Rn,y, Rn.z
328 `V_BFE_U32` = D.u = (S0.u>>S1.u[4:0]) & ((1<<S2.u[4:0])-1); bitfield extract, S0=data, S1=field_offset, S2=field_width.
329 `V_BFE_I32` = D.i = (S0.i>>S1.u[4:0]) & ((1<<S2.u[4:0])-1); bitfield extract, S0=data, S1=field_offset, S2=field_width.
330 `V_BFI_B32` = D.u = (S0.u & S1.u) | (~S0.u & S2.u); bitfield insert.
331 `V_FMA_F32` = D.f = S0.f * S1.f + S2.f
332 `V_FMA_F64` = D.d = S0.d * S1.d + S2.d.
333 `V_LERP_U8` = D.u = ((S0.u[31:24] + S1.u[31:24] + S2.u[24]) >> 1) << 24 + ((S0.u[23:16] + S1.u[23:16] + S2.u[16]) >> 1) << 16 + ((S0.u[15:8] + S1.u[15:8] + S2.u[8]) >> 1) << 8 + ((S0.u[7:0] + S1.u[7:0] + S2.u[0]) >> 1). Unsigned eight-bit pixel average on packed unsigned bytes (linear interpolation). S2 acts as a round mode; if set, 0.5 rounds up; otherwise, 0.5 truncates.
334 `V_ALIGNBIT_B32` = D.u = ({S0,S1} >> S2.u[4:0]) & 0xFFFFFFFF.
335 `V_ALIGNBYTE_B32` = D.u = ({S0,S1} >> (8*S2.u[4:0])) & 0xFFFFFFFF.
336 `V_MULLIT_F32` = D.f = S0.f * S1.f (0.0 * x = 0.0; special INF, NaN, overflow rules).
337 `V_MIN3_F32` = D.f = min(S0.f, S1.f, S2.f).
338 `V_MIN3_I32` = D.i = min(S0.i, S1.i, S2.i).

Vector Instruction, Three Inputs, One Output (VOP3a)

339 V_MIN3_U32 = 0x153 D.u = min(S0.u, S1.u, S2.u).
340 V_MAX3_F32 = D.f = max(S0.f, S1.f, S2.f).
341 V_MAX3_I32 = D.i = max(S0.i, S1.i, S2.i).
342 V_MAX3_U32 = D.u = max(S0.u, S1.u, S2.u).
343 V_MED3_F32 = D.f = median(S0.f, S1.f, S2.f).
344 V_MED3_I32 = D.i = median(S0.i, S1.i, S2.i).
345 V_MED3_U32 = D.u = median(S0.u, S1.u, S2.u).
346 V_SAD_U8 = D.u = Byte SAD with accum_lo(S0.u, S1.u, S2.u).
347 V_SAD_HI_U8 = D.u = Byte SAD with accum_hi(S0.u, S1.u, S2.u).
348 V_SAD_U16 = D.u = Word SAD with accum(S0.u, S1.u, S2.u).
349 V_SAD_U32 = D.u = Dword SAD with accum(S0.u, S1.u, S2.u).
350 V_CVT_PK_U8_F32 = f32→u8(s0.f), pack into byte(s1.u), of dword(s2).
351 V_DIV_FIXUP_F32 = D.f = Special case divide fixup and flags(s0.f = Quotient, s1.f = Denominator, s2.f = Numerator).
352 V_DIV_FIXUP_F64 = D.d = Special case divide fixup and flags(s0.d = Quotient, s1.d = Denominator, s2.d = Numerator).
353 V_LSHL_B64 = D = S0.u << S1.u[5:0].
354 V_LSHR_B64 = D = S0.u >> S1.u[5:0].
355 V_ASHR_I64 = D = S0.u >> S1.u[5:0].
356 V_ADD_F64 = D.d = S0.d + S1.d.
357 V_MUL_F64 = D.d = S0.d * S1.d.
358 V_MIN_F64 = D.d = min(S0.d, S1.d).
359 V_MAX_F64 = D.d = max(S0.d, S1.d).
360 V_LDEXP_F64 = D.d = pow(S0.d, S1.i[31:0]).
361 V_MUL_LO_U32 = D.u = S0.u * S1.u.
362 V_MUL_HI_U32 = D.u = (S0.u * S1.u)>>32.
363 V_MUL_LO_I32 = D.i = S0.i * S1.i.
364 V_MUL_HI_I32 = D.i = (S0.i * S1.i)>>32.
365 – 366 See corresponding opcode numbers in VOP3 (3 in, 2 out), (VOP3b).
367 V_DIV_FMAS_F32 = D.f = Special case divide FMA with scale and flags(s0.f = Quotient, s1.f = Denominator, s2.f = Numerator).
368 V_DIV_FMAS_F64 = D.d = Special case divide FMA with scale and flags(s0.d = Quotient, s1.d = Denominator, s2.d = Numerator).
369 V_MSAD_U8 = D.u = Masked Byte SAD with accum_lo(S0.u, S1.u, S2.u).
370 V_QSAD_PK_U16_U8 = D.u = Quad-Byte SAD with accum_lo/hiu(S0.u[63:0], S1.u[31:0], S2.u[63:0]).
371 V_MQSAD_PK_U16_U8 = D.u = Masked Quad-Byte SAD with accum_lo/hi(S0.u[63:0], S1.u[31:0], S2.u[63:0]).
372 V_TRIG_PREOP_F64 = D.d = Look Up 2/PI (S0.d) with segment select S1.u[4:0].
373 V_MQSAD_U32_U8: D.u128 = Masked Quad-Byte SAD with 32-bit accum_lo/hi(S0.u[63:0], S1.u[31:0], S2.u[127:0]).
374 V_MAD_U64_U32: {vcc_out,D.u64} = S0.u32 * S1.u32 + S2.u64.
375 V_MAD_I64_I32: {vcc_out,D.i64} = S0.i32 * S1.i32 + S2.i64.
384 – 452 Are the VOP1 opcodes when VOP3 encoding is required. For example, 384 + V_MOV_B32 generates the VOP3 version of MOV.
384 V_NOP: do nothing.
385 V_MOV_B32: D.u = S0.u.
386 V_READFIRSTLANE_B32: copy one VGPR value to one SGPR. Dst = SGPR-dest, Src0 = Source Data (VGPR# or M0(lds-direct)), Lane# = FindFirst1fromLSB(exec) (lane = 0 if exec is zero). Ignores exec mask.

Vector Instruction, Three Inputs, One Output (VOP3a)

387 `V_CVT_I32_F64`: D.i = (int)S0.d.
388 `V_CVT_F64_I32`: D.f = (float)S0.i.
389 `V_CVT_F32_I32`: D.f = (float)S0.i.
390 `V_CVT_F32_U32`: D.f = (float)S0.u.
391 `V_CVT_U32_F32`: D.u = (unsigned)S0.f.
392 `V_CVT_I32_F32`: D.i = (int)S0.f.
393 reserved.
394 `V_CVT_F16_F32`: D.f16 = flt32_to_flt16(S0.f).
395 `V_CVT_F32_F16`: D.f = flt16_to_flt32(S0.f16).
396 `V_CVT_RPI_I32_F32`: D.i = (int)floor(S0.f + 0.5).
397 `V_CVT_FLR_I32_F32`: D.i = (int)floor(S0.f).
398 `V_CVT_OFF_F32_I4`: 4-bit signed int to 32-bit float. For interpolation in shader.
399 `V_CVT_F32_F64`: D.f = (float)S0.d.
400 `V_CVT_F64_F32`: D.d = (double)S0.f.
401 `V_CVT_F32_UBYTE0`: D.f = UINT2FLT(S0.u[7:0]).
402 `V_CVT_F32_UBYTE1`: D.f = UINT2FLT(S0.u[15:8]).
403 `V_CVT_F32_UBYTE2`: D.f = UINT2FLT(S0.u[23:16]).
404 `V_CVT_F32_UBYTE3`: D.f = UINT2FLT(S0.u[31:24]).
405 `V_CVT_U32_F64`: D.u = (uint)S0.d.
406 `V_CVT_F64_U32`: D.d = (double)S0.u.
407 – 415 reserved.
416 `V_FRACT_F32`: D.f = S0.f - floor(S0.f).
417 `V_TRUNC_F32`: D.f = trunc(S0.f), return integer part of S0.
418 `V_CEIL_F32`: D.f = ceil(S0.f). Implemented as: D.f = trunc(S0.f); if (S0 > 0.0 && S0 != D), D += 1.0.
419 `V_RNDNE_F32`: D.f = round_nearest_even(S0.f).
420 `V_FLOOR_F32`: D.f = trunc(S0); if ((S0 < 0.0) && (S0 != D)) D += -1.0.
421 `V_EXP_F32`: D.f = pow(2.0, S0.f).
422 `V_LOG_CLAMP_F32`: D.f = log2(S0.f), clamp -infinity to -max_float.
423 `V_LOG_F32`: D.f = log2(S0.f).
424 `V_RCP_CLAMP_F32`: D.f = 1.0 / S0.f, result clamped to +-max_float.
425 `V_RCP_LEGACY_F32`: D.f = 1.0 / S0.f, +-infinity result clamped to +-0.0.
426 `V_RCP_F32`: D.f = 1.0 / S0.f.
427 `V_RCP_IFLAG_F32`: D.f = 1.0 / S0.f, only integer div_by_zero flag can be raised.
428 `V_RSQ_CLAMP_F32`: D.f = 1.0 / sqrt(S0.f), result clamped to +-max_float.
429 `V_RSQ_LEGACY_F32`: D.f = 1.0 / sqrt(S0.f).
430 `V_RSQ_F32`: D.f = 1.0 / sqrt(S0.f).
431 `V_RCP_F64`: D.d = 1.0 / (S0.d).
432 `V_RCP_CLAMP_F64`: D.f = 1.0 / (S0.f), result clamped to +-max_float.
433 `V_RSQ_F64`: D.f = 1.0 / sqrt(S0.f).
434 `V_RSQ_CLAMP_F64`: D.d = 1.0 / sqrt(S0.d), result clamped to +-max_float.
435 `V_SQRT_F32`: D.f = sqrt(S0.f).
436 `V_SQRT_F64`: D.d = sqrt(S0.d).
437 `V_SIN_F32`: D.f = sin(S0.f).
438 `V_COS_F32`: D.f = cos(S0.f).
439 `V_NOT_B32`: D.u = ~S0.u.
440 `V_BFREV_B32`: D.u[31:0] = S0.u[0:31], bitfield reverse.
441 `V_FFBH_U32`: D.u = position of first 1 in S0 from MSB; D=0xFFFFFFFF if S0==0.

Vector Instruction, Three Inputs, One Output (VOP3a)

	442 V_FFBL_B32: D.u = position of first 1 in S0 from LSB; D=0xFFFFFFFF if S0==0. 443 V_FFBH_I32: D.u = position of first bit different from sign bit in S0 from MSB; D=0xFFFFFFFF if S0==0 or 0xFFFFFFFF. 444 V_FREXP_EXP_I32_F64: See V_FREXP_EXP_I32_F32. 445 V_FREXP_MANT_F64: See V_FREXP_MANT_F32. 446 V_FRACT_F64: S0.d - floor(S0.d). 447 V_FREXP_EXP_I32_F32: If (S0.f == INF || S0.f == NAN), then D.i = 0; else D.i = TwosComplement(Exponent(S0.f) - 127 + 1). Returns exponent of single precision float input, such that S0.f = significand * (2 ** exponent). See also FREXP_MANT_F32, which returns the significand. 448 V_FREXP_MANT_F32: if (S0.f == INF || S0.f == NAN) then D.f = S0.f; else D.f = Mantissa(S0.f). Result range is in (-1.0,-0.5][0.5,1.0) in normal cases. Returns binary significand of single precision float input, such that S0.f = significand * (2 ** exponent). See also FREXP_EXP_I32_F32, which returns integer exponent. 449 V_CLREXCP: Clear wave's exception state in shader processor SIMD. 450 V_MOVRELD_B32: VGPR[D.u + M0.u] = VGPR[S0.u]. 451 V_MOVRELS_B32: VGPR[D.u] = VGPR[S0.u + M0.u]. 452 V_MOVRELSD_B32: VGPR[D.u + M0.u] = VGPR[S0.u + M0.u]. All other values are reserved.
ENCODING	[31:26] enum(6) Must be 1 1 0 1 0 0.

Vector Instruction, Three Inputs, One Output (VOP3a)

`SRC0`	[39:32]	enum(9)
	First operand for instruction. 0 – 103 32-bit Scalar General-Purpose Register (SGPR) 106 `VCC_LO` (vcc[31:0]). 107 `VCC_HI` (vcc[63:32]). 108 `TBA_LO` Trap handler base address [31:0] 109 `TBA_HI` Trap handler base address [63:32]). 110 `TMA_LO` Pointer to data in memory used by trap handler. 111 `TMA_HI` Pointer to data in memory used by trap handler. 112 `TTMP[11:0]` Trap handler temporary SGPR [11:0]. 124 `M0`. Memory register 0. 126 `EXEC_LO` exec[31:0]. 127 `EXEC_HI` exec[63:32]. [191 – 128] `SRC_[63:0]` = 63:0 integer. 192 `SRC_64_INT` = 64 (integer). [208 – 193] `SRC_M_[16:1]_INT` = [-16:-1] (integer). 240 `SRC_0_5` = 0.5. 241 `SRC_M_0_5` = -0.5. 242 `SRC_1` = 1.0. 243 `SRC_M_1` = -1.0. 244 `SRC_2` = 2.0. 245 `SRC_M_2` = -2.0. 246 `SRC_4` = 4.0. 247 `SRC_M_4` = -4.0. 251 `SRC_VCCZ` = vector-condition-code-is-zero. 252 `SRC_EXECZ` = execute-mask-is-zero. 253 `SRC_SCC` = scalar condition code. 254 `SRC_LDS_DIRECT` = use LDS direct to supply 32-bit value (address from `M0` register). 256 - 511 VGPR 0 to 255. All other values are reserved.	
`SRC1`	[48:40]	enum(9)
	Second operand for instruction. Same format as SRC0.	
`SRC2`	[57:49]	enum(9)
	Third operand for instruction. Same format as SRC0.	
`OMOD`	[59:58]	enum(2)
	Output modifier for instruction. Applied before clamping. 0 : No modification. 1 : Multiply output by 2.0. 2 : Multiply output by 4.0. 3 : Divide output by 2.0.	
`NEG`	[63:60]	enum(3)
	If NEG[N] is set, take the floating-point negation of the N'th input operand. This is applied after absolute value.	

Vector Instruction, Three Inputs, One Vector Output and One Scalar Output (VOP3b)

Format **VOP3**

Description Vector instruction taking three inputs and producing one output.

Opcode

Field Name	Bits	Format
VDST	[7:0]	enum(8)
	Destination for instruction in the Vector General-Purpose Registers (VGPR [255:0]).	
SDST	[14:8]	enum(7)
	0 – 103 32-bit Scalar General-Purpose Register (SGPR) 106 VCC_LO (vcc[31:0]). 107 VCC_HI (vcc[63:32]). 108 TBA_LO Trap handler base address [31:0] 109 TBA_HI Trap handler base address [63:32]). 110 TMA_LO Pointer to data in memory used by trap handler. 111 TMA_HI Pointer to data in memory used by trap handler. 113 – 112 TTMP[11:0] Trap handler temporary SGPR [11:0]. All other values are reserved.	
reserved	[16:15]	
	Reserved.	
OP	[25:17]	enum(9)
	Instructions that use this format: 293 V_ADD_I32: D.u = S0.u + S1.u; VCC=carry-out (VOP3:sgpr=carry-out). 294 V_SUB_I32: D.u = S0.u - S1.u; VCC=carry-out (VOP3:sgpr=carry-out). 295 V_SUBREV_I32: D.u = S1.u - S0.u; VCC=carry-out (VOP3:sgpr=carry-out). 296 V_ADDC_U32: D.u = S0.u + S1.u + VCC; VCC=carry-out (VOP3:sgpr=carry-out, S2.u=carry-in). 297 V_SUBB_U32: D.u = S0.u - S1.u - VCC; VCC=carry-out (VOP3:sgpr=carry-out, S2.u=carry-in). 298 V_SUBBREV_U32: D.u = S1.u - S0.u - VCC; VCC=carry-out (VOP3:sgpr=carry-out, S2.u=carry-in). 365 V_DIV_SCALE_F32 = D.f = Special case divide preop and flags(s0.f = Quotient, s1.f = Denominator, s2.f = Numerator) s0 must equal s1 or s2. 366 V_DIV_SCALE_F64 = D.d = Special case divide preop and flags(s0.d = Quotient, s1.d = Denominator, s2.d = Numerator) s0 must equal s1 or s2.	

Vector Instruction, Three Inputs, One Vector Output and One Scalar Output (VOP3b)

Group 1 and Group 2, below are the VOPC opcodes when VOP3 encoding is required.

Sixteen Compare Operations (OP16)

Compare Operation	Opcode Offset	Description
F	0	D.u = 0
LT	1	D.u = (S0 < S1)
EQ	2	D.u = (S0 == S1)
LE	3	D.u = (S0 <= S1)
GT	4	D.u = (S0 > S1)
LG	5	D.u = (S0 <> S1)
GE	6	D.u = (S0 >= S1)
O	7	D.u = (!isNaN(S0) && !isNaN(S1))
U	8	D.u = (!isNaN(S0) || !isNaN(S1))
NGE	9	D.u = !(S0 >= S1)
NLG	10	D.u = !(S0 <> S1)
NGT	11	D.u = !(S0 > S1)
NLE	12	D.u = !(S0 <= S1)
NEQ	13	D.u = !(S0 == S1)
NLT	14	D.u = !(S0 < S1)
TRU	15	D.u = 1

Eight Compare Operations (OP8)

Compare Operation	Opcode Offset	Description
F	0	D.u = 0
LT	1	D.u = (S0 < S1)
EQ	2	D.u = (S0 == S1)
LE	3	D.u = (S0 <= S1)
GT	4	D.u = (S0 > S1)
LG	5	D.u = (S0 <> S1)
GE	6	D.u = (S0 >= S1)
TRU	7	D.u = 1

Vector Instruction, Three Inputs, One Vector Output and One Scalar Output (VOP3b)

Single Vector Compare Operations

Opcode Family	Opcode Base	Description
V_CMP_{OP16}_F32	0x00	Signal on sNaN input only.
V_CMPX_{OP16}_F32	0x10	Signal on sNaN input only. Also write EXEC.
V_CMP_{OP16}_F64	0x20	Signal on sNaN input only.
V_CMPX_{OP16}_F64	0x30	Signal on sNaN input only. Also write EXEC.
V_CMPS_{OP16}_F32	0x40	Signal on any NaN.
V_CMPSX_{OP16}_F32	0x50	Signal on any NaN. Also write EXEC.
V_CMPS_{OP16}_F64	0x60	Signal on any NaN.
V_CMPSX_{OP16}_F64	0x70	Signal on any NaN. Also write EXEC.
V_CMP_{OP8}_I32	0x80	On 32-bit integers.
V_CMPX_{OP8}_I32	0x90	Also write EXEC.
V_CMP_{OP8}_I64	0xA0	On 64-bit integers.
V_CMPX_{OP8}_I64	0xB0	Also write EXEC.
V_CMP_{OP8}_U32	0xC0	On unsigned 32-bit integers.
V_CMPX_{OP8}_U32	0xD0	Also write EXEC.
V_CMP_{OP8}_U64	0xE0	On unsigned 64-bit integers.
V_CMPX_{OP8}_U64	0xF0	Also write EXEC.
V_CMP_CLASS_F32	0x88	D = IEEE numeric class function specified in S1.u, performed on S0.f.
V_CMPX_CLASS_F32	0x98	D = IEEE numeric class function specified in S1.u, performed on S0.f. Also write EXEC.
V_CMP_CLASS_F64	0xA8	D = IEEE numeric class function specified in S1.u, performed on S0.d.
V_CMPX_CLASS_F64	0xB8	D = IEEE numeric class function specified in S1.u, performed on S0.d. Also write EXEC.

Field	Bits	Format / Description
ENCODING	[31:26]	enum(6) Must be 1 1 0 1 0 0.
SRC0	[39:32]	enum(9) First operand for instruction. 0 – 103 32-bit Scalar General-Purpose Register (SGPR) 106 VCC_LO (vcc[31:0]). 107 VCC_HI (vcc[63:32]). 108 TBA_LO Trap handler base address [31:0] 109 TBA_HI Trap handler base address [63:32]). 110 TMA_LO Pointer to data in memory used by trap handler. 111 TMA_HI Pointer to data in memory used by trap handler. 113 – 112 TTMP[11:0] Trap handler temporary SGPR [11:0]. 124 M0. Memory register 0. 126 EXEC_LO exec[31:0]. 127 EXEC_HI exec[63:32]. [191 – 128] SRC_[63:0] = 63:0 integer. 192 SRC_64_INT = 64 (integer). [208 – 193] SRC_M_[16:1]_INT = [-16:-1] (integer). 240 SRC_0_5 = 0.5. 241 SRC_M_0_5 = -0.5. 242 SRC_1 = 1.0. 243 SRC_M_1 = -1.0.

Vector Instruction, Three Inputs, One Vector Output and One Scalar Output (VOP3b)

		244 SRC_2 = 2.0. 245 SRC_M_2 = -2.0. 246 SRC_4 = 4.0. 247 SRC_M_4 = -4.0. 251 SRC_VCCZ = vector-condition-code-is-zero. 252 SRC_EXECZ = execute-mask-is-zero. 253 SRC_SCC = scalar condition code. 254 SRC_LDS_DIRECT = use LDS direct to supply 32-bit value (address from M0 register). 256 – 511 VGPR 0 to 255. All other values are reserved.
SRC1	[48:40]	enum(9) Second operand for instruction.
SRC2	[57:49]	enum(9) Third operand for instruction.
OMOD	[59:58]	enum(2) Output modifier for instruction. Applied before clamping.
NEG	[63:60]	enum(3) If NEG[N] is set, take the floating-point negation of the N'th input operand. This is applied after absolute value.

13.4 Vector Parameter Interpolation Instruction

Interpolation Instruction

Format **VINTRP**

Description Interpolate data for the pixel shader.

Opcode

Field Name	Bits	Format
VSRC	[7:0]	enum(8)
	Vector General-Purpose Registers (VGPR) containing the i/j coordinate by which to multiply one of the parameter components.	
ATTRCHAN	[9:8]	enum(2)
	Attribute component to interpolate. See Section 10.1 on page 10-1.	
ATTR	[15:10]	int(6)
	Attribute to interpolate.	
OP	[17:16]	enum(2)
	0 V_INTERP_P1_F32: D = P10 * S + P0; parameter interpolation. 1 V_INTERP_P2_F32: D = P20 * S + D; parameter interpolation. 2 V_INTERP_MOV_F32: D = {P10,P20,P0}[S]; parameter load. 3 reserved.	
VDST	[25:18]	enum(8)
	Vector General-Purpose Registers VGPR [255:0] to which results are written, and, optionally, from which they are read when accumulating results.	
ENCODING	[31:26]	enum(6)
	Must be 1 1 0 0 1 0.	

13.5 LDS/GDS Instruction

Data Share Instruction

Format **DS**

Description .Local and global data share instructions.

Opcode

Field Name	Bits	Format
OFFSET0	[7:0]	int(8)
	Unsigned byte offset added to the address supplied by the ADDR VGPR.	
OFFSET1	[15:8]	int(8)
	Unsigned byte offset added to the address supplied by the ADDR VGPR.	
reserved	16	
	Reserved.	
GDS	17	enum(1)
	0 = LDS; 1 = GDS.	
OP	[25:18]	enum(8)
	uint = unsigned integer; int = signed integer.	

00	DS_ADD_U32	DS[A] = DS[A] + D0; uint add.
01	DS_SUB_U32	DS[A] = DS[A] - D0; uint subtract.
02	DS_RSUB_U32	DS[A] = D0 - DS[A]; uint reverse subtract.
03	DS_INC_U32	DS[A] = (DS[A] >= D0 ? 0 : DS[A] + 1); uint increment.
04	DS_DEC_U32	DS[A] = (DS[A] == 0 || DS[A] > D0 ? D0 : DS[A] - 1); uint decrement.
05	DS_MIN_I32	DS[A] = min(DS[A], D0); int min.
06	DS_MAX_I32	DS[A] = max(DS[A], D0); int max.
07	DS_MIN_U32	DS[A] = min(DS[A], D0); uint min.
08	DS_MAX_U32	DS[A] = max(DS[A], D0); uint max.
09	DS_AND_B32	DS[A] = DS[A] & D0; Dword AND.
10	DS_OR_B32	DS[A] = DS[A] | D0; Dword OR.
11	DS_XOR_B32	DS[A] = DS[A] ^ D0; Dword XOR.
12	DS_MSKOR_B32	DS[A] = (DS[A] & ~D0) | D1; masked Dword OR.
13	DS_WRITE_B32	DS[A] = D0; write a Dword.
14	DS_WRITE2_B32	DS[ADDR+offset0*4] = D0; DS[ADDR+offset1*4] = D1; write 2 Dwords.
15	DS_WRITE2ST64_B32	DS[ADDR+offset0*4*64] = D0; DS[ADDR+offset1*4*64] = D1; write 2 Dwords.
16	DS_CMPST_B32	DS[A] = (DS[A] == D0 ? D1 : DS[A]); compare store.
17	DS_CMPST_F32	DS[A] = (DS[A] == D0 ? D1 : DS[A]); compare store with float rules.
18	DS_MIN_F32	DS[A] = (DS[A] < D1) ? D0 : DS[A]; float compare swap (handles NaN/INF/denorm).
19	DS_MAX_F32	DS[A] = (D0 > DS[A]) ? D0 : DS[A]; float, handles NaN/INF/denorm.
20	DS_NOP	Do nothing.
24	DS_GWS_SEMA_RELEASE_ALL	GDS Only. Release all wavefronts waiting on this semaphore. ResourceID is in offset[4:0].
25	DS_GWS_INIT	GDS only.
26	DS_GWS_SEMA_V	GDS only.

Data Share Instruction

27	DS_GWS_SEMA_BR	GDS only.
28	DS_GWS_SEMA_P	GDS only.
29	DS_GWS_BARRIER	GDS only.
30	DS_WRITE_B8	DS[A] = D0[7:0]; byte write.
31	DS_WRITE_B16	DS[A] = D0[15:0]; short write.
32	DS_ADD_RTN_U32	Uint add.
33	DS_SUB_RTN_U32	Uint subtract.
34	DS_RSUB_RTN_U32	Uint reverse subtract.
35	DS_INC_RTN_U32	Uint increment.
36	DS_DEC_RTN_U32	Uint decrement.
37	DS_MIN_RTN_I32	Int min.
38	DS_MAX_RTN_I32	Int max.
39	DS_MIN_RTN_U32	Uint min.
40	DS_MAX_RTN_U32	Uint max.
41	DS_AND_RTN_B32	Dword AND.
42	DS_OR_RTN_B32	Dword OR.
43	DS_XOR_RTN_B32	Dword XOR.
44	DS_MSKOR_RTN_B32	Masked Dword OR.
45	DS_WRXCHG_RTN_B32	Write exchange. Offset = {offset1,offset0}. A = ADDR+offset. D=DS[Addr]. DS[Addr]=D0.
46	DS_WRXCHG2_RTN_B32	Write exchange 2 separate Dwords.
47	DS_WRXCHG2ST64_RTN_B32	Write exchange 2 Dwords, stride 64.
48	DS_CMPST_RTN_B32	Compare store.
49	DS_CMPST_RTN_F32	Compare store with float rules.
50	DS_MIN_RTN_F32	DS[A] = (D0 < DS[A]) ? D0 : DS[A]; float compare swap (handles NaN/INF/denorm).
51	DS_MAX_RTN_F32	DS[A] = (D0 > DS[A]) ? D0 : DS[A]; float, handles NaN/INF/denorm .
52	DS_WRAP_RTN_B32	DS[A] = (DS[A] >= D0) ? DS[A] - D0 : DS[A] + D1.
53	DS_SWIZZLE_B32	R = swizzle(Data(VGPR), offset1:offset0). Dword swizzle. no data is written to LDS. see ds_opcodes.docx for details.
54	DS_READ_B32	R = DS[A]; Dword read.
55	DS_READ2_B32	R = DS[ADDR+offset0*4], R+1 = DS[ADDR+offset1*4]. Read 2 Dwords.
56	DS_READ2ST64_B32	R = DS[ADDR+offset0*4*64], R+1 = DS[ADDR+offset1*4*64]. Read 2 Dwords.
57	DS_READ_I8	R = signext(DS[A][7:0]}; signed byte read.
58	DS_READ_U8	R = {24'h0,DS[A][7:0]}; unsigned byte read.
59	DS_READ_I16	R = signext(DS[A][15:0]}; signed short read.
60	DS_READ_U16	R = {16'h0,DS[A][15:0]}; unsigned short read.
61	DS_CONSUME	Consume entries from a buffer.
62	DS_APPEND	Append one or more entries to a buffer.
63	DS_ORDERED_COUNT	Increment an append counter. Operation is done in order of wavefront creation.
64	DS_ADD_U64	Uint add.
65	DS_SUB_U64	Uint subtract.
66	DS_RSUB_U64	Uint reverse subtract.

Data Share Instruction

67	DS_INC_U64	Uint increment.
68	DS_DEC_U64	Uint decrement.
69	DS_MIN_I64	Int min.
70	DS_MAX_I64	Int max.
71	DS_MIN_U64	Uint min.
72	DS_MAX_U64	Uint max.
73	DS_AND_B64	Dword AND.
74	DS_OR_B64	Dword OR.
75	DS_XOR_B64	Dword XOR.
76	DS_MSKOR_B64	Masked Dword XOR.
77	DS_WRITE_B64	Write.
78	DS_WRITE2_B64	DS[ADDR+offset0*8] = D0; DS[ADDR+offset1*8] = D1; write 2 Dwords.
79	DS_WRITE2ST64_B64	DS[ADDR+offset0*8*64] = D0; DS[ADDR+offset1*8*64] = D1; write 2 Dwords.
80	DS_CMPST_B64	Compare store.
81	DS_CMPST_F64	Compare store with float rules.
82	DS_MIN_F64	DS[A] = (D0 < DS[A]) ? D0 : DS[A]; float, handles NaN/INF/denorm.
83	DS_MAX_F64	DS[A] = (D0 > DS[A]) ? D0 : DS[A]; float, handles NaN/INF/denorm.
96	DS_ADD_RTN_U64	Uint add.
97	DS_SUB_RTN_U64	Uint subtract.
98	DS_RSUB_RTN_U64	Uint reverse subtract.
99	DS_INC_RTN_U64	Uint increment.
100	DS_DEC_RTN_U64	Uint decrement.
101	DS_MIN_RTN_I64	Int min.
102	DS_MAX_RTN_I64	Int max.
103	DS_MIN_RTN_U64	Uint min.
104	DS_MAX_RTN_U64	Uint max.
105	DS_AND_RTN_B64	Dword AND.
106	DS_OR_RTN_B64	Dword OR.
107	DS_XOR_RTN_B64	Dword XOR.
108	DS_MSKOR_RTN_B64	Masked Dword XOR.
109	DS_WRXCHG_RTN_B64	Write exchange.
110	DS_WRXCHG2_RTN_B64	Write exchange relative.
111	DS_WRXCHG2ST64_RTN_B64	Write exchange 2 Dwords.
112	DS_CMPST_RTN_B64	Compare store.
113	DS_CMPST_RTN_F64	Compare store with float rules.
114	DS_MIN_RTN_F64	DS[A] = (D0 < DS[A]) ? D0 : DS[A]; float, handles NaN/INF/denorm.
115	DS_MAX_RTN_F64	DS[A] = (D0 > DS[A]) ? D0 : DS[A]; float, handles NaN/INF/denorm.
118	DS_READ_B64	Dword read.
119	DS_READ2_B64	R = DS[ADDR+offset0*8], R+1 = DS[ADDR+offset1*8]. Read 2 Dwords
120	DS_READ2ST64_B64	R = DS[ADDR+offset0*8*64], R+1 = DS[ADDR+offset1*8*64]. Read 2 Dwords.
126	DS_CONDXCHG32_RTN_B64	Conditional write exchange.
128	DS_ADD_SRC2_U32	B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0}). DS[A] = DS[A] + DS[B]; uint add.

Data Share Instruction			
	129	DS_SUB_SRC2_U32	B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0}). DS[A] = DS[A] - DS[B]; uint subtract.
	130	DS_RSUB_SRC2_U32	B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0}). DS[A] = DS[B] - DS[A]; uint reverse subtract.
	131	DS_INC_SRC2_U32	B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0}). DS[A] = (DS[A] >= DS[B] ? 0 : DS[A] + 1); uint increment.
	132	DS_DEC_SRC2_U32	B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0}). DS[A] = (DS[A] == 0 || DS[A] > DS[B] ? DS[B] : DS[A] - 1); uint decrement.
	133	DS_MIN_SRC2_I32	B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0}). DS[A] = min(DS[A], DS[B]); int min.
	134	DS_MAX_SRC2_I32	B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0}). DS[A] = max(DS[A], DS[B]); int max.
	135	DS_MIN_SRC2_U32	B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0}). DS[A] = min(DS[A], DS[B]); uint min.
	136	DS_MAX_SRC2_U32	B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0}). DS[A] = max(DS[A], DS[B]); uint maxw.
	137	DS_AND_SRC2_B32	B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0}). DS[A] = DS[A] & DS[B]; Dword AND.
	138	DS_OR_SRC2_B32	B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0}). DS[A] = DS[A] | DS[B]; Dword OR.
	139	DS_XOR_SRC2_B32	B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0}). DS[A] = DS[A] ^ DS[B]; Dword XOR.
	140	DS_WRITE_SRC2_B32	B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0}). DS[A] = DS[B]; write Dword.
	146	DS_MIN_SRC2_F32	B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0}). DS[A] = (DS[B] < DS[A]) ? DS[B] : DS[A]; float, handles NaN/INF/denorm.
	147	DS_MAX_SRC2_F32	B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0}). DS[A] = (DS[B] > DS[A]) ? DS[B] : DS[A]; float, handles NaN/INF/denorm.
	192	DS_ADD_SRC2_U64	Uint add.
	193	DS_SUB_SRC2_U64	Uint subtract.
	194	DS_RSUB_SRC2_U64	Uint reverse subtract.
	195	DS_INC_SRC2_U64	Uint increment.
	196	DS_DEC_SRC2_U64	Uint decrement.
	197	DS_MIN_SRC2_I64	Int min.

Data Share Instruction

	198 DS_MAX_SRC2_I64	Int max.
	199 DS_MIN_SRC2_U64	Uint min.
	200 DS_MAX_SRC2_U64	Uint max.
	201 DS_AND_SRC2_B64	Dword AND.
	202 DS_OR_SRC2_B64	Dword OR.
	203 DS_XOR_SRC2_B64	Dword XOR.
	204 DS_WRITE_SRC2_B64	B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0}). DS[A] = DS[B]; write Qword.
	210 DS_MIN_SRC2_F64	B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0}). [A] = (D0 < DS[A]) ? D0 : DS[A]; float, handles NaN/INF/denorm.
	211 DS_MAX_SRC2_F64	B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0}). [A] = (D0 > DS[A]) ? D0 : DS[A]; float, handles NaN/INF/denorm.
	222 DS_WRITE_B96	{DS[A+2], DS[A+1], DS[A]} = D0[95:0]; tri-dword write.
	223 DS_WRITE_B128	{DS[A+3], DS[A+2], DS[A+1], DS[A]} = D0[127:0]; qword write.
	253 DS_CONDXCHG32_RTN_B128	Conditional write exchange.
	254 DS_READ_B96	Tri-Dword read.
	255 DS_READ_B128	Qword read.
	All other values are reserved.	
ENCODING	[31:26] enum(6) Must be 1 1 0 1 1 0.	
ADDR	[39:32] enum(8) Source LDS address VGPR 0 - 255.	
DATA0	[47:40] enum(8) Source data0 VGPR 0 - 255.	
DATA1	[[55:48] enum(8) Source data1 VGPR 0 - 255.	
VDST	[63:56] enum(8) Destination VGPR 0 - 255.	

13.6 Vector Memory Buffer Instructions

Untyped Memory Buffer Operation

Format **MUBUF**

Description Untyped memory buffer operation. First word with LDS, second word non-LDS.

Opcode

Field Name	Bits	Format
OFFSET	[11:0]	int(12)
	Unsigned byte offset.	
OFFEN	12	enum(1)
	If set, send VADDR as an offset. If clear, use zero instead of an offset from a VGPR.	
IDXEN	13	enum(1)
	If set, send VADDR as an index. If clear, treat the index as zero.	
GLC	14	enum(1)
	If set, operation is globally coherent.	
ADDR64	15	enum(1)
	If set, buffer address is 64-bits (base and size in resource is ignored).	
LDS	16	enum(1)
	If set, data is read from/written to LDS memory. If unset, data is read from/written to a VGPR.	
reserved	17	
	Reserved.	
OP	[15:8]	enum(8)
	0 BUFFER_LOAD_FORMAT_X: Untyped buffer load 1 Dword with format conversion. 1 BUFFER_LOAD_FORMAT_XY: Untyped buffer load 2 Dwords with format conversion. 2 BUFFER_LOAD_FORMAT_XYZ: Untyped buffer load 3 Dwords with format conversion. 3 BUFFER_LOAD_FORMAT_XYZW: Untyped buffer load 4 Dwords with format conversion. 4 BUFFER_STORE_FORMAT_X: Untyped buffer store 1 Dword with format conversion. 5 BUFFER_STORE_FORMAT_XY: Untyped buffer store 2 Dwords with format conversion. 6 BUFFER_STORE_FORMAT_XYZ: Untyped buffer store 3 Dwords with format conversion. 7 BUFFER_STORE_FORMAT_XYZW: Untyped buffer store 4 Dwords with format conversion. 8 BUFFER_LOAD_UBYTE: Untyped buffer load unsigned byte. 9 BUFFER_LOAD_SBYTE: Untyped buffer load signed byte. 10 BUFFER_LOAD_USHORT: Untyped buffer load unsigned short. 11 BUFFER_LOAD_SSHORT: Untyped buffer load signed short. 12 BUFFER_LOAD_DWORD: Untyped buffer load Dword. 13 BUFFER_LOAD_DWORDX2: Untyped buffer load 2 Dwords. 14 BUFFER_LOAD_DWORDX4: Untyped buffer load 4 Dwords. 15 BUFFER_LOAD_DWORDX3: Untyped buffer load 3 Dwords. 16 – 23 reserved. 24 BUFFER_STORE_BYTE: Untyped buffer store byte.	

Untyped Memory Buffer Operation

25 reserved.
26 BUFFER_STORE_SHORT: Untyped buffer store short.
27 reserved.
28 BUFFER_STORE_DWORD: Untyped buffer store Dword.
29 BUFFER_STORE_DWORDX2: Untyped buffer store 2 Dwords.
30 BUFFER_STORE_DWORDX4: Untyped buffer store 4 Dwords.
31 BUFFER_STORE_DWORDX3: Untyped buffer store 3 Dwords.
31 – 47 reserved.
48 BUFFER_ATOMIC_SWAP: 32b. dst=src, returns previous value if glc==1.
49 BUFFER_ATOMIC_CMPSWAP: 32b, dst = (dst==cmp) ? src : dst. Returns previous value if glc==1. src comes from the first data-VGPR, cmp from the second.
50 BUFFER_ATOMIC_ADD: 32b, dst += src. Returns previous value if glc==1.
51 BUFFER_ATOMIC_SUB: 32b, dst -= src. Returns previous value if glc==1.
52 reserved.
53 BUFFER_ATOMIC_SMIN: 32b, dst = (src < dst) ? src : dst (signed). Returns previous value if glc==1.
54 BUFFER_ATOMIC_UMIN: 32b, dst = (src < dst) ? src : dst (unsigned). Returns previous value if glc==1.
55 BUFFER_ATOMIC_SMAX: 32b, dst = (src > dst) ? src : dst (signed). Returns previous value if glc==1.
56 BUFFER_ATOMIC_UMAX: 32b, dst = (src > dst) ? src : dst (unsigned). Returns previous value if glc==1.
57 BUFFER_ATOMIC_AND: 32b, dst &= src. Returns previous value if glc==1.
58 BUFFER_ATOMIC_OR: 32b, dst |= src. Returns previous value if glc==1.
59 BUFFER_ATOMIC_XOR: 32b, dst ^= src. Returns previous value if glc==1.
60 BUFFER_ATOMIC_INC: 32b, dst = (dst >= src) ? 0 : dst+1. Returns previous value if glc==1.
61 BUFFER_ATOMIC_DEC: 32b, dst = ((dst==0 || (dst > src)) ? src : dst-1. Returns previous value if glc==1.
62 BUFFER_ATOMIC_FCMPSWAP: 32b , dst = (dst == cmp) ? src : dst, returns previous value if glc==1. Float compare swap (handles NaN/INF/denorm). src comes from the first data-VGPR; cmp from the second.
63 BUFFER_ATOMIC_FMIN: 32b , dst = (src < dst) ? src : dst,. Returns previous value if glc==1. float, handles NaN/INF/denorm.
64 BUFFER_ATOMIC_FMAX: 32b , dst = (src > dst) ? src : dst, returns previous value if glc==1. float, handles NaN/INF/denorm.
65 – 79 reserved.
80 BUFFER_ATOMIC_SWAP_X2: 64b. dst=src, returns previous value if glc==1.
81 BUFFER_ATOMIC_CMPSWAP_X2: 64b, dst = (dst==cmp) ? src : dst. Returns previous value if glc==1. src comes from the first two data-VGPRs, cmp from the second two.
82 BUFFER_ATOMIC_ADD_X2: 64b, dst += src. Returns previous value if glc==1.
83 BUFFER_ATOMIC_SUB_X2: 64b, dst -= src. Returns previous value if glc==1.
84 reserved.
85 BUFFER_ATOMIC_SMIN_X2: 64b, dst = (src < dst) ? src : dst (signed). Returns previous value if glc==1.
86 BUFFER_ATOMIC_UMIN_X2: 64b, dst = (src < dst) ? src : dst (unsigned). Returns previous value if glc==1.
87 BUFFER_ATOMIC_SMAX_X2: 64b, dst = (src > dst) ? src : dst (signed). Returns previous value if glc==1.
88 BUFFER_ATOMIC_UMAX_X2: 64b, dst = (src > dst) ? src : dst (unsigned). Returns previous value if glc==1.

Untyped Memory Buffer Operation

	89 BUFFER_ATOMIC_AND_X2: 64b, dst &= src. Returns previous value if glc==1. 90 BUFFER_ATOMIC_OR_X2: 64b, dst |= src. Returns previous value if glc==1. 91 BUFFER_ATOMIC_XOR_X2: 64b, dst ^= src. Returns previous value if glc==1. 92 BUFFER_ATOMIC_INC_X2: 64b, dst = (dst >= src) ? 0 : dst+1. Returns previous value if glc==1. 93 BUFFER_ATOMIC_DEC_X2: 64b, dst = ((dst==0 || (dst > src)) ? src : dst-1. Returns previous value if glc==1. 94 BUFFER_ATOMIC_FCMPSWAP_X2: 64b , dst = (dst == cmp) ? src : dst, returns previous value if glc==1. Double compare swap (handles NaN/INF/denorm). src comes from the first two data-VGPRs, cmp from the second two. 95 BUFFER_ATOMIC_FMIN_X2: 64b , dst = (src < dst) ? src : dst, returns previous value if glc==1. Double, handles NaN/INF/denorm. 96 BUFFER_ATOMIC_FMAX_X2: 64b , dst = (src > dst) ? src : dst, returns previous value if glc==1. Double, handles NaN/INF/denorm. 97 – 111 reserved. 112 BUFFER_WBINVL1_VOL: write back and invalidate the shader L1 only for lines of MTYPE SC and GC. Always returns ACK to shader. 113 BUFFER_WBINVL1: write back and invalidate the shader L1. Always returns ACK to shader. All other values are reserved.	
reserved	25 Reserved.	
ENCODING	[31:26] Must be 1 1 1 0 0 0.	enum(6)
VADDR	[39:32] VGPR address source. Can carry an offset or an index or both (can read two VGPRs).	enum(8)
VDATA	[47:40] Vector GPR to read/write result to.	enum(8)
SRSRC	[52:48] Scalar GPR that specifies the resource constant, in units of four SGPRs.	enum(5)
reserved	53 Reserved.	
SLC	54 System Level Coherent.	enum(1)
TFE	55 Texture Fail Enable (for partially resident textures).	enum(1)

Untyped Memory Buffer Operation

SOFFSET	[63:56]	enum(6)
	Byte offset added to the memory address. Scalar or constant GPR containing the base offset. This is always sent. 0 – 103 SGPR0 to SGPR103: Scalar general-purpose registers. 104 – 105 reserved. 106 VCC_LO: vcc[31:0]. 107 VCC_HI: vcc[63:32]. 108 TBA_LO: Trap handler base address [31:0]. 109 TBA_HI: Trap handler base address [63:32]. 110 TMA_LO: Pointer to data in memory used by trap handler. 111 TMA_HI: Pointer to data in memory used by trap handler. 112 - 123 TTMP0 to TTMP11: Trap handler temporary registers (privileged). 124 M0. Memory register 0. 125 reserved. 126 EXEC_LO: exec[31:0]. 127 EXEC_HI: exec[63:32]. 128 0. 129 – 192: Signed integer 1 to 64. 193 – 208: Signed integer -1 to -16. 209 – 239: reserved. 240 0.5. 241 -0.5. 242 1.0. 243 -1.0. 244 2.0. 245 -2.0. 246 4.0. 247 -4.0. 248 – 250 reserved. 251 VCCZ. 252 EXECZ. 253 SCC.	

Typed Memory Buffer Operation

Format **MTBUF**

Description Typed memory buffer operation. Two words

Opcode

Field Name	Bits	Format
OFFSET	[11:0]	int(12)
	Unsigned byte offset.	
OFFEN	12	enum(1)
	If set, send VADDR as an offset. If clear, use zero instead of an offset from a VGPR.	
IDXEN	13	enum(1)
	If set, send VADDR as an index. If clear, treat the index as zero.	
GLC	14	enum(1)
	If set, operation is globally coherent.	
ADDR64	15	enum(1)
	If set, buffer address is 64-bits (base and size in resource is ignored).	
OP	[18:16]	enum(3)
	0 TBUFFER_LOAD_FORMAT_X: Untyped buffer load 1 Dword with format conversion. 1 TBUFFER_LOAD_FORMAT_XY: Untyped buffer load 2 Dwords with format conversion. 2 TBUFFER_LOAD_FORMAT_XYZ: Untyped buffer load 3 Dwords with format conversion. 3 TBUFFER_LOAD_FORMAT_XYZW: Untyped buffer load 4 Dwords with format conversion. 4 TBUFFER_STORE_FORMAT_X: Untyped buffer store 1 Dword with format conversion. 5 TBUFFER_STORE_FORMAT_XY: Untyped buffer store 2 Dwords with format conversion. 6 TBUFFER_STORE_FORMAT_XYZ: Untyped buffer store 3 Dwords with format conversion. 7 TBUFFER_STORE_FORMAT_XYZW: Untyped buffer store 4 Dwords with format conversion. All other values are reserved.	
DFMT	[22:19]	enum(4)
	Data format for typed buffer. 0 invalid 1 8 2 16 3 8_8 4 32 5 16_16 6 10_11_11 8 10_10_10_2 9 2_10_10_10 10 8_8_8_8 11 32_32 12 16_16_16_16 13 32_32_32 14 32_32_32_32	

Typed Memory Buffer Operation

NFMT	[25:23]	enum(3)
	Number format for typed buffer. 0 unorm 1 snorm 2 uscaled 3 sscaled 4 uint 5 sint 6 snorm_ogl 7 float	
Encoding	[31:26]	enum(7)
	Must be `1 1 1 0 1 0`.	
VADDR	[39:32]	enum(8)
	VGPR address source. Can carry an offset or an index or both (can read two successive VGPRs).	
VDATA	[47:40]	enum(8)
	Vector GPR to read/write result to.	
SRSRC	[52:48]	enum(5)
	Scalar GPR that specifies the resource constant, in units of four SGPRs.	
reserved	53	
	Reserved.	
SLC	54	enum(1)
	System Level Coherent.	
TFE	55	enum(1)

Typed Memory Buffer Operation

	Texture Fail Enable (for partially resident textures).	
SOFFSET	[63:56]	enum(6)
	Byte offset added to the memory address. Scalar or constant GPR containing the base offset. This is always sent. 0 – 103 SGPR0 to SGPR103: Scalar general-purpose registers. 104 – 105 reserved. 106 VCC_LO: vcc[31:0]. 107 VCC_HI: vcc[63:32]. 108 TBA_LO: Trap handler base address [31:0]. 109 TBA_HI: Trap handler base address [63:32]. 110 TMA_LO: Pointer to data in memory used by trap handler. 111 TMA_HI: Pointer to data in memory used by trap handler. 112 – 123 TTMP0 to TTMP11: Trap handler temporary registers (privileged). 124 M0. Memory register 0. 125 reserved. 126 EXEC_LO: exec[31:0]. 127 EXEC_HI: exec[63:32]. 128 0. 129 – 192: Signed integer 1 to 64. 193 – 208: Signed integer -1 to -16. 209 – 239: reserved. 240 0.5. 241 -0.5. 242 1.0. 243 -1.0. 244 2.0. 245 -2.0. 246 4.0. 247 -4.0. 248 – 250 reserved. 251 VCCZ. 252 EXECZ. 253 SCC.	

13.7 Vector Memory Image Instruction

Image Memory Buffer Operations

Format **MIMG**

Description Image memory buffer operations. Two words.

Opcode

Field Name	Bits	Format
reserved	[7:0]	
	Reserved.	
DMASK	[11:8]	enum(4)
	Enable mask for image read/write data components. bit0 = red, 1 = green, 2 = blue, 3 = alpha. At least one bit must be on. Data is assumed to be packed into consecutive VGPRs.	
UNORM	12	enum(1)
	When set to 1, forces the address to be un-normalized, regardless of T#. Must be set to 1 for image stores and atomics	
GLC	13	enum(1)
	If set, operation is globally coherent.	
DA	14	enum(1)
	Declare an Array. 1 Kernel has declared this resource to be an array of texture maps. 0 Kernel has declared this resource to be a single texture map.	
R128	15	enum(1)
	Texture resource size: 1 = 128b, 0 = 256b.	
TFE	16	enum(1)
	Texture Fail Enable (for partially resident textures).	
LWE	17	enum(1)
	LOD Warning Enable (for partially resident textures).	
OP	[24:18]	enum(8)
	0 IMAGE_LOAD: Image memory load with format conversion specified in T#. No sampler. 1 IMAGE_LOAD_MIP: Image memory load with user-supplied mip level. No sampler. 2 IMAGE_LOAD_PCK: Image memory load with no format conversion. No sampler. 3 IMAGE_LOAD_PCK_SGN: Image memory load with no format conversion and sign extension. No sampler. 4 IMAGE_LOAD_MIP_PCK: Image memory load with user-supplied mip level, no format conversion. No sampler. 5 IMAGE_LOAD_MIP_PCK_SGN: Image memory load with user-supplied mip level, no format conversion and with sign extension. No sampler. 6 – 7 reserved. 8 IMAGE_STORE: Image memory store with format conversion specified in T#. No sampler. 9 IMAGE_STORE_MIP: Image memory store with format conversion specified in T# to user specified mip level. No sampler. 10 IMAGE_STORE_PCK: Image memory store of packed data without format conversion. No sampler.	

Image Memory Buffer Operations

11 `IMAGE_STORE_MIP_PCK`: Image memory store of packed data without format conversion to user-supplied mip level. No sampler.
12 – 13 reserved.
14 `IMAGE_GET_RESINFO`: return resource info. No sampler.
15 `IMAGE_ATOMIC_SWAP`: dst=src, returns previous value if glc==1.
16 `IMAGE_ATOMIC_CMPSWAP`: dst = (dst==cmp) ? src : dst. Returns previous value if glc==1.
17 `IMAGE_ATOMIC_ADD`: dst += src. Returns previous value if glc==1.
18 `IMAGE_ATOMIC_SUB`: dst -= src. Returns previous value if glc==1.
19 reserved.
20 `IMAGE_ATOMIC_SMIN`: dst = (src < dst) ? src : dst (signed). Returns previous value if glc==1.
21 `IMAGE_ATOMIC_UMIN`: dst = (src < dst) ? src : dst (unsigned). Returns previous value if glc==1.
22 `IMAGE_ATOMIC_SMAX`: dst = (src > dst) ? src : dst (signed). Returns previous value if glc==1.
23 `IMAGE_ATOMIC_UMAX`: dst = (src > dst) ? src : dst (unsigned). Returns previous value if glc==1.
24 `IMAGE_ATOMIC_AND`: dst &= src. Returns previous value if glc==1.
25 `IMAGE_ATOMIC_OR`: dst |= src. Returns previous value if glc==1.
26 `IMAGE_ATOMIC_XOR`: dst ^= src. Returns previous value if glc==1.
27 `IMAGE_ATOMIC_INC`: dst = (dst >= src) ? 0 : dst+1. Returns previous value if glc==1.
28 `IMAGE_ATOMIC_DEC`: dst = ((dst==0 || (dst > src)) ? src : dst-1. Returns previous value if glc==1.
29 `IMAGE_ATOMIC_FCMPSWAP`: dst = (dst == cmp) ? src : dst, returns previous value of dst if glc==1 - double and float atomic compare swap. Obeys floating point compare rules for special values.
30 `IMAGE_ATOMIC_FMIN`: dst = (src < dst) ? src : dst, returns previous value of dst if glc==1 - double and float atomic min (handles NaN/INF/denorm).
31 `IMAGE_ATOMIC_FMAX`: dst = (src > dst) ? src : dst, returns previous value of dst if glc==1 - double and float atomic min (handles NaN/INF/denorm).
32 `IMAGE_SAMPLE`: sample texture map.
33 `IMAGE_SAMPLE_CL`: sample texture map, with LOD clamp specified in shader.
34 `IMAGE_SAMPLE_D`: sample texture map, with user derivatives.
35 `IMAGE_SAMPLE_D_CL`: sample texture map, with LOD clamp specified in shader, with user derivatives.
36 `IMAGE_SAMPLE_L`: sample texture map, with user LOD.
37 `IMAGE_SAMPLE_B`: sample texture map, with lod bias.
38 `IMAGE_SAMPLE_B_CL`: sample texture map, with LOD clamp specified in shader, with lod bias.
39 `IMAGE_SAMPLE_LZ`: sample texture map, from level 0.
40 `IMAGE_SAMPLE_C`: sample texture map, with PCF.
41 `IMAGE_SAMPLE_C_CL`: SAMPLE_C, with LOD clamp specified in shader.
42 `IMAGE_SAMPLE_C_D`: SAMPLE_C, with user derivatives.
43 `IMAGE_SAMPLE_C_D_CL`: SAMPLE_C, with LOD clamp specified in shader, with user derivatives.
44 `IMAGE_SAMPLE_C_L`: SAMPLE_C, with user LOD.
45 `IMAGE_SAMPLE_C_B`: SAMPLE_C, with lod bias.
46 `IMAGE_SAMPLE_C_B_CL`: SAMPLE_C, with LOD clamp specified in shader, with lod bias.

Image Memory Buffer Operations

47 `IMAGE_SAMPLE_C_LZ`: SAMPLE_C, from level 0.
48 `IMAGE_SAMPLE_O`: sample texture map, with user offsets.
49 `IMAGE_SAMPLE_CL_O`: SAMPLE_O with LOD clamp specified in shader.
50 `IMAGE_SAMPLE_D_O`: SAMPLE_O, with user derivatives.
51 `IMAGE_SAMPLE_D_CL_O`: SAMPLE_O, with LOD clamp specified in shader, with user derivatives.
52 `IMAGE_SAMPLE_L_O`: SAMPLE_O, with user LOD.
53 `IMAGE_SAMPLE_B_O`: SAMPLE_O, with lod bias.
54 `IMAGE_SAMPLE_B_CL_O`: SAMPLE_O, with LOD clamp specified in shader, with lod bias.
55 `IMAGE_SAMPLE_LZ_O`: SAMPLE_O, from level 0.
56 `IMAGE_SAMPLE_C_O`: SAMPLE_C with user specified offsets.
57 `IMAGE_SAMPLE_C_CL_O`: SAMPLE_C_O, with LOD clamp specified in shader.
58 `IMAGE_SAMPLE_C_D_O`: SAMPLE_C_O, with user derivatives.
59 `IMAGE_SAMPLE_C_D_CL_O`: SAMPLE_C_O, with LOD clamp specified in shader, with user derivatives.
60 `IMAGE_SAMPLE_C_L_O`: SAMPLE_C_O, with user LOD.
61 `IMAGE_SAMPLE_C_B_O`: SAMPLE_C_O, with lod bias.
62 `IMAGE_SAMPLE_C_B_CL_O`: SAMPLE_C_O, with LOD clamp specified in shader, with lod bias.
63 `IMAGE_SAMPLE_C_LZ_O`: SAMPLE_C_O, from level 0.
64 `IMAGE_GATHER4`: gather 4 single component elements (2x2).
65 `IMAGE_GATHER4_CL`: gather 4 single component elements (2x2) with user LOD clamp.
66 `IMAGE_GATHER4_L`: gather 4 single component elements (2x2) with user LOD.
67 `IMAGE_GATHER4_B`: gather 4 single component elements (2x2) with user bias.
68 `IMAGE_GATHER4_B_CL`: gather 4 single component elements (2x2) with user bias and clamp.
69 `IMAGE_GATHER4_LZ`: gather 4 single component elements (2x2) at level 0.
70 `IMAGE_GATHER4_C`: gather 4 single component elements (2x2) with PCF.
71 `IMAGE_GATHER4_C_CL`: gather 4 single component elements (2x2) with user LOD clamp and PCF.
72 – 75 reserved.
76 `IMAGE_GATHER4_C_L`: gather 4 single component elements (2x2) with user LOD and PCF.
77 `IMAGE_GATHER4_C_B`: gather 4 single component elements (2x2) with user bias and PCF.
78 `IMAGE_GATHER4_C_B_CL`: gather 4 single component elements (2x2) with user bias, clamp and PCF.
79 `IMAGE_GATHER4_C_LZ`: gather 4 single component elements (2x2) at level 0, with PCF.
80 `IMAGE_GATHER4_O`: GATHER4, with user offsets.
81 `IMAGE_GATHER4_CL_O`: GATHER4_CL, with user offsets.
82 – 83 reserved.
84 `IMAGE_GATHER4_L_O`: GATHER4_L, with user offsets.
85 `IMAGE_GATHER4_B_O`: GATHER4_B, with user offsets.
86 `IMAGE_GATHER4_B_CL_O`: GATHER4_B_CL, with user offsets.
87 `IMAGE_GATHER4_LZ_O`: GATHER4_LZ, with user offsets.

Image Memory Buffer Operations

	88 IMAGE_GATHER4_C_O: GATHER4_C, with user offsets. 89 IMAGE_GATHER4_C_CL_O: GATHER4_C_CL, with user offsets. 90 – 91 reserved. 92 IMAGE_GATHER4_C_L_O: GATHER4_C_L, with user offsets. 93 IMAGE_GATHER4_C_B_O: GATHER4_B, with user offsets. 94 IMAGE_GATHER4_C_B_CL_O: GATHER4_B_CL, with user offsets. 95 IMAGE_GATHER4_C_LZ_O: GATHER4_C_LZ, with user offsets. 96 IMAGE_GET_LOD: Return calculated LOD. 97 – 103 reserved. 104 IMAGE_SAMPLE_CD: sample texture map, with user derivatives (LOD per quad) 105 IMAGE_SAMPLE_CD_CL: sample texture map, with LOD clamp specified in shader, with user derivatives (LOD per quad). 106 IMAGE_SAMPLE_C_CD: SAMPLE_C, with user derivatives (LOD per quad). 107 IMAGE_SAMPLE_C_CD_CL: SAMPLE_C, with LOD clamp specified in shader, with user derivatives (LOD per quad). 108 IMAGE_SAMPLE_CD_O: SAMPLE_O, with user derivatives (LOD per quad). 109 IMAGE_SAMPLE_CD_CL_O: SAMPLE_O, with LOD clamp specified in shader, with user derivatives (LOD per quad). 110 IMAGE_SAMPLE_C_CD_O: SAMPLE_C_O, with user derivatives (LOD per quad). 111 IMAGE_SAMPLE_C_CD_CL_O: SAMPLE_C_O, with LOD clamp specified in shader, with user derivatives (LOD per quad). All other values are reserved.	
SLC	25	enum(1)
	System Level Coherent.	
ENCODING	[31:26]	enum(7)
	Must be `1 1 1 1 0 0`.	
VADDR	[39:32]	enum(8)
	Address source. Can carry an offset or an index. Specifies the VGPR that holds the first of the image address values.	
VDATA	[47:40]	enum(8)
	Vector GPR to which the result is written.	
SRSRC	[52:48]	enum(5)
	Scalar GPR that specifies the resource constant, in units of four SGPRs.	
SSAMP	[57:53]	enum(5)
	Scalar GPR that specifies the sampler constant, in units of four SGPRs.	
reserved	[63:58]	
	Reserved.	

13.8 Export Instruction

Export

Format **EXP**

Description Export (output) pixel color, pixel depth, vertex position, or vertex parameter data. Two words.

Opcode

Field Name	Bits	Format
EN	[3:0]	int(4)
	This bitmask determines which VSRC registers export data. When COMPR is 0: VSRC0 only exports data when en[0] is set to 1; VSRC1 when en[1], VSRC2 when en[2], and VSRC3 when en[3]. When COMPR is 1: VSRC0 contains two 16-bit data and only exports when en[0] is set to 1; VSRC1 only exports when en[2] is set to 1; en[1] and en[3] are ignored when COMPR is 1.	
TGT	[9:4]	enum(6)
	Export target based on the enumeration below. 0–7 EXP_MRT = Output to color MRT 0. Increment from here for additional MRTs. There are EXP_NUM_MRT MRTs in total. 8 EXP_MRTZ = Output to Z. 9 EXP_NULL = Output to NULL. 12–15 EXP_POS = Output to position 0. Increment from here for additional positions. There are EXP_NUM_POS positions in total. 32–63 EXP_PARAM = Output to parameter 0. Increment from here for additional parameters. There are EXP_NUM_PARAM parameters in total. All other values are reserved.	
COMPR	10	enum(1)
	Boolean. If true, data is exported in float16 format; if false, data is 32 bit.	
DONE	11	enum(1)
	If set, this is the last export of a given type. If this is set for a color export (PS only), then the valid mask must be present in the EXEC register.	
VM	12	enum(1)
	Mask contains valid-mask when set; otherwise, mask is just write-mask. Used only for pixel(mrt) exports.	
reserved	[25:13]	
	Reserved.	
ENCODING	[31:26]	enum(7)
	Must be 1 1 1 1 1 0.	
VSRC0	[39:32]	enum(8)
	VGPR of the first data to export.	
VSRC1	[47:40]	enum(8)
	VGPR of the second data to export.	
VSRC2	[55:48]	enum(8)
	VGPR of the third data to export.	
VSRC3	[63:56]	enum(8)
	VGPR of the fourth data to export.	

13.9 FLAT Instruction

Flat

Format **FLAT**

Description Export (output) pixel color, pixel depth, vertex position, or vertex parameter data. Two words.

Opcode

Field Name	Bits	Format
reserved	[15:0] Reserved	
GLC	16 If set, operation is globally coherent.	enum(1)
SLC	17 System Level Coherent. When set, indicates that the operation is "system level coherent". This controls the L2 cache policy.	enum(1)
OP	[24:18] 0 - 7 reserved. 8 FLAT_LOAD_UBYTE: Flat load unsigned byte. Zero extend to VGPR destination. 9 FLAT_LOAD_SBYTE: Flat load signed byte. Sign extend to VGPR destination. 10 FLAT_LOAD_USHORT: Flat load unsigned short. Zero extend to VGPR destination. 11 FLAT_LOAD_SSHORT: Flat load signed short. Sign extend to VGPR destination. 12 FLAT_LOAD_DWORD: Flat load Dword. 13 FLAT_LOAD_DWORDX2: Flat load 2 Dwords. 14 FLAT_LOAD_DWORDX4: Flat load 4 Dwords. 15 FLAT_LOAD_DWORDX3: Flat load 3 Dwords. 16 - 23 reserved. 24 FLAT_STORE_BYTE: Flat store byte. 25 reserved. 26 FLAT_STORE_SHORT: Flat store short. 27 reserved. 28 FLAT_STORE_DWORD: Flat store Dword. 29 FLAT_STORE_DWORDX2: Flat store 2 Dwords. 30 FLAT_STORE_DWORDX4: Flat store 4 Dwords. 31 FLAT_STORE_DWORDX3: Flat store 3 Dwords. 32 – 47 reserved. 48 FLAT_ATOMIC_SWAP: 32b. dst=src, returns previous value if rtn==1. 49 FLAT_ATOMIC_CMPSWAP: 32b, dst = (dst==cmp) ? src : dst. Returns previous value if rtn==1. src comes from the first data-VGPR, cmp from the second. 50 FLAT_ATOMIC_ADD: 32b, dst += src. Returns previous value if rtn==1. 51 FLAT_ATOMIC_SUB: 32b, dst -= src. Returns previous value if rtn==1. 52 reserved. 53 FLAT_ATOMIC_SMIN: 32b, dst = (src < dst) ? src : dst (signed comparison). Returns previous value if rtn==1. 54 FLAT_ATOMIC_UMIN: 32b, dst = (src < dst) ? src : dst (unsigned comparison). Returns previous value if rtn==1. 55 FLAT_ATOMIC_SMAX: 32b, dst = (src > dst) ? src : dst (signed comparison). Returns previous value if rtn==1.	enum(7)

Flat

56 `FLAT_ATOMIC_UMAX`: 32b, dst = (src > dst) ? src : dst (unsigned comparison). Returns previous value if rtn==1.
57 `FLAT_ATOMIC_AND`: 32b, dst &= src. Returns previous value if rtn==1.
58 `FLAT_ATOMIC_OR`: 32b, dst |= src. Returns previous value if rtn==1.
59 `FLAT_ATOMIC_XOR`: 32b, dst ^= src. Returns previous value if rtn==1.
60 `FLAT_ATOMIC_INC`: 32b, dst = (dst >= src) ? 0 : dst+1 (unsigned comparison). Returns previous value if rtn==1.
61 `FLAT_ATOMIC_DEC`: 32b, dst = ((dst==0 || (dst > src)) ? src : dst-1 (unsigned comparison). Returns previous value if rtn==1.
62 `FLAT_ATOMIC_FCMPSWAP`: 32b , dst = (dst == cmp) ? src : dst, returns previous value if rtn==1. Floating point compare-swap handles NaN/INF/denorm. src comes from the first data-VGPR; cmp from the second.
63 `FLAT_ATOMIC_FMIN`: 32b , dst = (src < dst) ? src : dst. Returns previous value if rtn==1. float, handles NaN/INF/denorm.
64 `FLAT_ATOMIC_FMAX`: 32b , dst = (src > dst) ? src : dst, returns previous value if rtn==1. Floating point compare handles NaN/INF/denorm.
65 – 79 reserved.
80 `FLAT_ATOMIC_SWAP_X2`: 64b. dst=src, returns previous value if rtn==1.
81 `FLAT_ATOMIC_CMPSWAP_X2`: 64b, dst = (dst==cmp) ? src : dst. Returns previous value if rtn==1. src comes from the first two data-VGPRs, cmp from the second two.
82 `FLAT_ATOMIC_ADD_X2`: 64b, dst += src. Returns previous value if rtn==1.
83 `FLAT_ATOMIC_SUB_X2`: 64b, dst -= src. Returns previous value if rtn==1.
84 reserved.
85 `FLAT_ATOMIC_SMIN_X2`: 64b, dst = (src < dst) ? src : dst (signed comparison). Returns previous value if rtn==1.
86 `FLAT_ATOMIC_UMIN_X2`: 64b, dst = (src < dst) ? src : dst (unsigned comparison). Returns previous value if rtn==1.
87 `FLAT_ATOMIC_SMAX_X2`: 64b, dst = (src > dst) ? src : dst (signed comparison). Returns previous value if rtn==1.
88 `FLAT_ATOMIC_UMAX_X2`: 64b, dst = (src > dst) ? src : dst (unsigned comparison). Returns previous value if rtn==1.
89 `FLAT_ATOMIC_AND_X2`: 64b, dst &= src. Returns previous value if rtn==1.
90 `FLAT_ATOMIC_OR_X2`: 64b, dst |= src. Returns previous value if rtn==1.
91 `FLAT_ATOMIC_XOR_X2`: 64b, dst ^= src. Returns previous value if rtn==1.
92 `FLAT_ATOMIC_INC_X2`: 64b, dst = (dst >= src) ? 0 : dst+1. Returns previous value if rtn==1.
93 `FLAT_ATOMIC_DEC_X2`: 64b, dst = ((dst==0 || (dst > src)) ? src : dst - 1. Returns previous value if rtn==1.
94 `FLAT_ATOMIC_FCMPSWAP_X2`: 64b , dst = (dst == cmp) ? src : dst, returns previous value if rtn==1. Double compare swap (handles NaN/INF/denorm). src comes from the first two data-VGPRs, cmp from the second two.
95 `FLAT_ATOMIC_FMIN_X2`: 64b , dst = (src < dst) ? src : dst, returns previous value if rtn==1. Double, handles NaN/INF/denorm.
96 `FLAT_ATOMIC_FMAX_X2`: 64b , dst = (src > dst) ? src : dst, returns previous value if rtn==1. Double, handles NaN/INF/denorm.

All other values are reserved.

ENCODING	[31:26]	enum(7) Must be `1 1 0 1 1 1`.
ADDR	[39:32]	enum(8) Source of flat address VGPR.

Flat

`DATA`	[47:40]	enum(8)
	Source data.	
`reserved`	[54:48]	
	Reserved	
`TFE`	55	enum(1)
	Texture Fail Enable. For partially resident textures.	
`VDST`	[63:56]	enum(14)
	Destination VGPR.	