"AMD Instinct MI300" Instruction Set Architecture

Reference Guide

15-July-2024

Specification Agreement

This Specification Agreement ("Agreement") is a legal agreement between Advanced Micro Devices, Inc. ("AMD") and "You" as the recipient of the attached AMD Specification ("Specification"). If you are accessing the Specification as part of your performance of work for another party, you acknowledge that you have authority to bind such party to the terms and conditions of this Agreement. If you accessed the Specification by any means or otherwise use or provide Feedback (defined below) on the Specification, You agree to the terms and conditions set forth in this Agreement. If You do not agree to the terms and conditions set forth in this Agreement, you are not licensed to use the Specification; do not use, access, or provide Feedback about the Specification. In consideration of Your use or access of the Specification (in whole or in part), the receipt and sufficiency of which are acknowledged, You agree as follows:

1. You may review the Specification only (a) as a reference to assist You in planning and designing Your product, service or technology ("Product") to interface with an AMD product in compliance with the requirements as set forth in the Specification and (b) to provide Feedback about the information disclosed in the Specification to AMD.

2. Except as expressly set forth in Paragraph 1, all rights in and to the Specification are retained by AMD. This Agreement does not give You any rights under any AMD patents, copyrights, trademarks, or other intellectual property rights. You may not (i) duplicate any part of the Specification; (ii) remove this Agreement or any notices from the Specification, or (iii) give any part of the Specification, or assign or otherwise provide Your rights under this Agreement, to anyone else.

3. You agree that You shall not use nor procure others to use the contents of this Agreement for (i) modifying any existing patent or patent application or creating any continuation, continuation in part or other extension of any patent or patent application, nor (ii) analyzing, assessing any patent or patent application (which shall include the creation or modification of any patent claim charts or infringement analyses).

4. The Specification may contain preliminary information, errors, or inaccuracies, or may not include certain necessary information. Additionally, AMD reserves the right to discontinue or make changes to the Specification and its products at any time without notice. The Specification is provided entirely "AS IS." AMD MAKES NO WARRANTY OF ANY KIND AND DISCLAIMS ALL EXPRESS, IMPLIED AND STATUTORY WARRANTIES, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, TITLE OR THOSE WARRANTIES ARISING AS A COURSE OF DEALING OR CUSTOM OF TRADE. AMD SHALL NOT BE LIABLE FOR DIRECT, INDIRECT, CONSEQUENTIAL, SPECIAL, INCIDENTAL, PUNITIVE OR EXEMPLARY DAMAGES OF ANY KIND (INCLUDING LOSS OF BUSINESS, LOSS OF INFORMATION OR DATA, LOST PROFITS, LOSS OF CAPITAL, LOSS OF GOODWILL) REGARDLESS OF THE FORM OF ACTION WHETHER IN CONTRACT, TORT (INCLUDING NEGLIGENCE) AND STRICT PRODUCT LIABILITY OR OTHERWISE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

5. Furthermore, AMD’s products are not designed, intended, authorized or warranted for use as components in systems intended for surgical implant into the body, or in other applications intended to support or sustain life, or in any other application in which the failure of AMD’s product could create a situation where personal injury, death, or severe property or environmental damage may occur.

6. You have no obligation to give AMD any suggestions, comments, or feedback ("Feedback") relating to the Specification. However, any Feedback You voluntarily provide may be used by AMD without restriction, fee, or obligation of confidentiality. Accordingly, if You do give AMD Feedback on any version of the Specification, You agree AMD may freely use, reproduce, license, distribute, and otherwise commercialize Your Feedback in any product, as well as has the right to sublicense third parties to do the same. Further, You will not give AMD any Feedback that You may have reason to believe is (i) subject to any patent, copyright or other intellectual property claim or right of any third party; or (ii) subject to license terms which seek to require any product or intellectual property incorporating or derived from Feedback or any Product or other AMD intellectual property to be licensed to or otherwise provided to any third party.

7. You shall adhere to all applicable U.S., European, and other export laws, including but not limited to the U.S. Export Administration Regulations ("EAR"), (15 C.F.R. Sections 730 through 774), and E.U. Council Regulation (EC) No 428/2009 of 5 May 2009. Further, pursuant to Section 740.6 of the EAR, You hereby certifies that, except pursuant to a license granted by the United States Department of Commerce Bureau of Industry and Security or as otherwise permitted pursuant to a License Exception under the U.S. Export Administration Regulations ("EAR"), You will not (1) export, re-export or release to a national of a country in Country Groups D:1, E:1 or E:2 any restricted technology, software, or source code You receive hereunder, or (2) export to Country Groups D:1, E:1 or E:2 the direct product of such technology or software, if such foreign produced direct product is subject to national security controls as identified on the Commerce Control List (currently found in Supplement 1 to Part 774 of EAR). For the most current Country Group listings, or for additional information about the EAR or Your obligations under those regulations, please refer to the U.S. Bureau of Industry and Security’s website at http://www.bis.doc.gov/. This Section 7 is applicable solely to Specifications shall not apply to any Specifications that are released publicly.

8. If You are a part of the U.S. Government, then the Specification is provided with "RESTRICTED RIGHTS" as set forth in subparagraphs (c) (1) and (2) of the Commercial Computer Software-Restricted Rights clause at FAR 52.227-14 or subparagraph (c) (1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.277-7013, as applicable.

9. This Agreement is governed by the laws of the State of California without regard to its choice of law principles. Any dispute involving it must be brought in a court having jurisdiction of such dispute in Santa Clara County, California, and You waive any defenses and rights allowing the dispute to be litigated elsewhere. If any part of this agreement is unenforceable, it will be considered modified to the extent necessary to make it

enforceable, and the remainder shall continue in effect. The failure of AMD to enforce any rights granted hereunder or to take action against You in the event of any breach hereunder shall not be deemed a waiver by AMD as to subsequent enforcement of rights or subsequent actions in the event of future breaches. This Agreement is the entire agreement between You and AMD concerning the Specification; it may be changed only by a written document signed by both You and an authorized representative of AMD.

DISCLAIMER

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes. THIS INFORMATION IS PROVIDED "AS IS." AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Advanced Micro Devices, Inc.
2485 Augustine Drive
Santa Clara, CA, 95054
www.amd.com

Contents

Preface

About This Document

This document describes the current environment, organization and program state of AMD CDNA "Instinct MI300" 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 CDNA processors' hardware and programming environment (Chapter 1).
Chapter 2 describes the organization of CDNA 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 vector Matrix ALU operations.
Chapter 8 describes the scalar memory operations.
Chapter 9 describes the vector memory operations.
Chapter 10 provides information about the flat memory instructions.
Chapter 11 describes the data share 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.
{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 bit 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.

Contact Information

For information concerning AMD Accelerated Parallel Processing development, please see:
http://developer.amd.com/

Chapter 1. Introduction

AMD CDNA processors implement a parallel micro-architecture that is designed to provide an excellent platform for general-purpose data parallel applications. Data-intensive applications that require high bandwidth or are computationally intensive are a candidate for running on an AMD CDNA processor.

The figure below shows a block diagram of the AMD CDNA Generation series processors

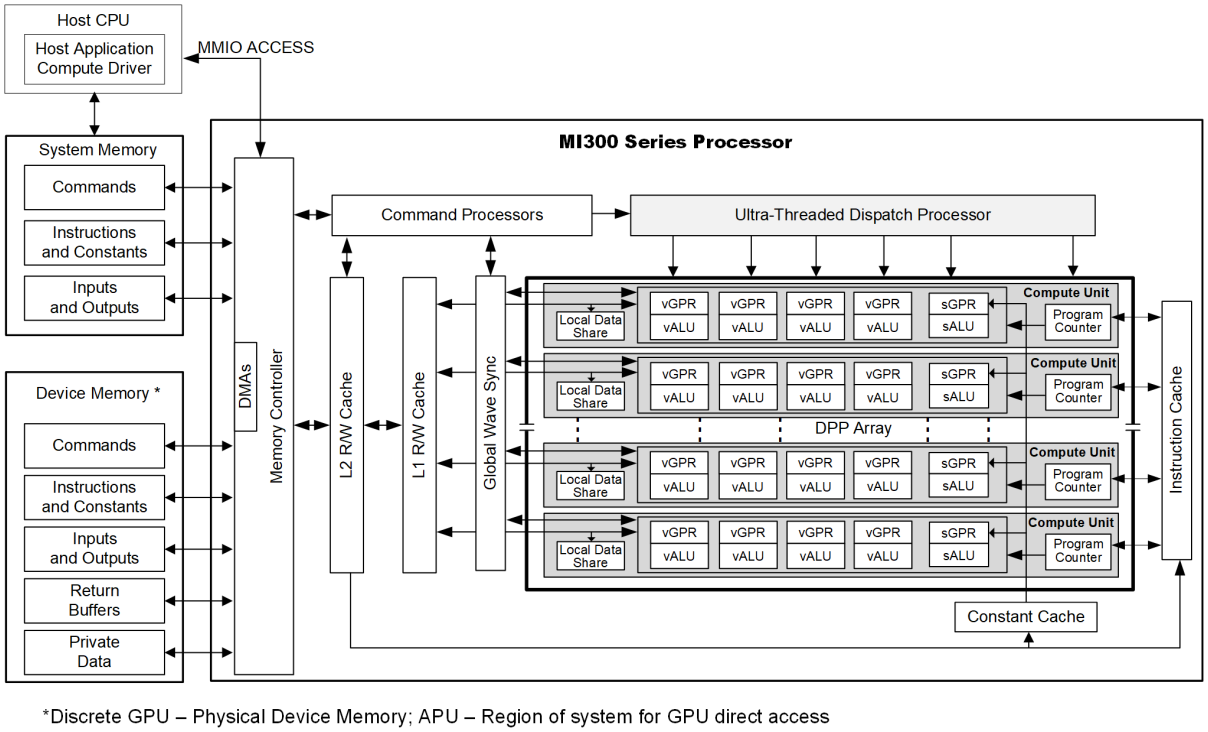


Figure 1. AMD CDNA Generation Series Block Diagram

The CDNA device includes a data-parallel processor (DPP) array, a command processor, a memory controller, and other logic (not shown). The CDNA command processor reads commands that the host has written to memory-mapped CDNA registers in the system-memory address space. The command processor sends hardware-generated interrupts to the host when the command is completed. The CDNA memory controller has direct access to all CDNA 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. In the CDNA environment, a complete application includes two parts:

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

The CDNA programs are controlled by host commands that

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

The CDNA driver program runs on the host.

The DPP array is the heart of the CDNA processor. The array is organized as a set of compute unit pipelines, each independent from the others, that are designed to 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.

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 CDNA hardware automatically fetches instructions from memory into on-chip caches; CDNA software plays no role in this. CDNA kernels can load data from off-chip memory into on-chip general-purpose registers (GPRs) and caches.

The AMD CDNA devices can detect floating point exceptions and can generate interrupts. In particular, they can detect IEEE floating-point exceptions in hardware; these can be recorded for post-execution analysis. The software interrupts shown in the previous figure from the command processor to the host represent hardware-generated interrupts for signaling command-completion and related management functions.

The CDNA processor is designed to hide 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.

1.1. Terminology

Table 1. Basic Terms

Term	Description
CDNA Processor	The shader processor is a scalar and vector ALU capable of running complex programs on behalf of a wavefront.
Dispatch	A dispatch launches a 1D, 2D, or 3D grid of work to the CDNA processor array.
Workgroup	A workgroup is a collection of wavefronts that have the ability to synchronize with each other quickly; they also can share data through the Local Data Share.
Wavefront	A collection of 64 work-items that execute in parallel on a single CDNA processor.
Work-item	A single element of work: one element from the dispatch grid, or in graphics a pixel or vertex.
Literal Constant	A 32-bit integer or float constant that is placed in the instruction stream.
Scalar ALU (SALU)	The scalar ALU operates on one value per wavefront and manages all control flow.
Vector ALU (VALU)	The vector ALU maintains Vector GPRs that are unique for each work item and execute arithmetic operations uniquely on each work-item.
Microcode format	The microcode format describes the bit patterns used to encode instructions. Each instruction is either 32 or 64 bits.
Instruction	An instruction is the basic unit of the kernel. Instructions include: vector ALU, scalar ALU, memory transfer, and control flow operations.
Texture Sampler (S#)	A texture sampler is a 128-bit entity that describes how the vector memory system reads and samples (filters) a texture map.
Texture Resource (T#)	A texture resource descriptor describes an image in memory: address, data format, stride, etc.
Buffer Resource (V#)	A buffer resource descriptor describes a buffer in memory: address, data format, stride, etc.

Chapter 2. Program Organization

CDNA kernels are programs executed by the CDNA processor. Conceptually, the kernel is executed independently on every work-item, but in reality the CDNA processor groups 64 work-items into a wavefront, which executes the kernel on all 64 work-items in one pass.

The CDNA processor consists of:

- A scalar ALU, which operates on one value per wavefront (common to all work items).
- A vector ALU, which operates on unique values per work-item.
- Local data storage, which allows work-items within a workgroup to communicate and share data.
- Scalar memory, which can transfer data between SGPRs and memory through a cache.
- Vector memory, which can transfer data between VGPRs and memory, including sampling texture maps.

All kernel control flow is handled using scalar ALU instructions. This includes if/else, branches and looping. Scalar ALU (SALU) and memory instructions work on an entire wavefront and operate on up to two SGPRs, as well as literal constants.

Vector memory and ALU instructions operate on all work-items in the wavefront at one time. In order to support branching and conditional execute, every wavefront has an EXECute mask that determines which work-items are active at that moment, and which are dormant. Active work-items execute the vector instruction, and dormant ones treat the instruction as a NOP. The EXEC mask can be changed at any time by Scalar ALU instructions.

Vector ALU instructions can take up to three arguments, which can come from VGPRs, SGPRs, or literal constants that are part of the instruction stream. They operate on all work-items enabled by the EXEC mask. Vector compare and add with- carryout return a bit-per-work-item mask back to the SGPRs to indicate, per work-item, which had a "true" result from the compare or generated a carry-out.

Vector memory instructions transfer data between VGPRs and memory. Each work-item supplies its own memory address and supplies or receives unique data. These instructions are also subject to the EXEC mask.

2.1. Compute Shaders

Compute kernels (shaders) are generic programs that can run on the CDNA processor, taking data from memory, processing it, and writing results back to memory. Compute kernels are created by a dispatch, which causes the CDNA processors to run the kernel over all of the work-items in a 1D, 2D, or 3D grid of data. The CDNA processor walks through this grid and generates wavefronts, which then run the compute kernel. Each work-item is initialized with its unique address (index) within the grid. Based on this index, the work-item computes the address of the data it is required to work on and what to do with the results.

2.2. Data Sharing

The AMD CDNA stream processors can share data between different work-items. Data sharing can significantly boost performance. The figure below shows the memory hierarchy that is available to each work-item.

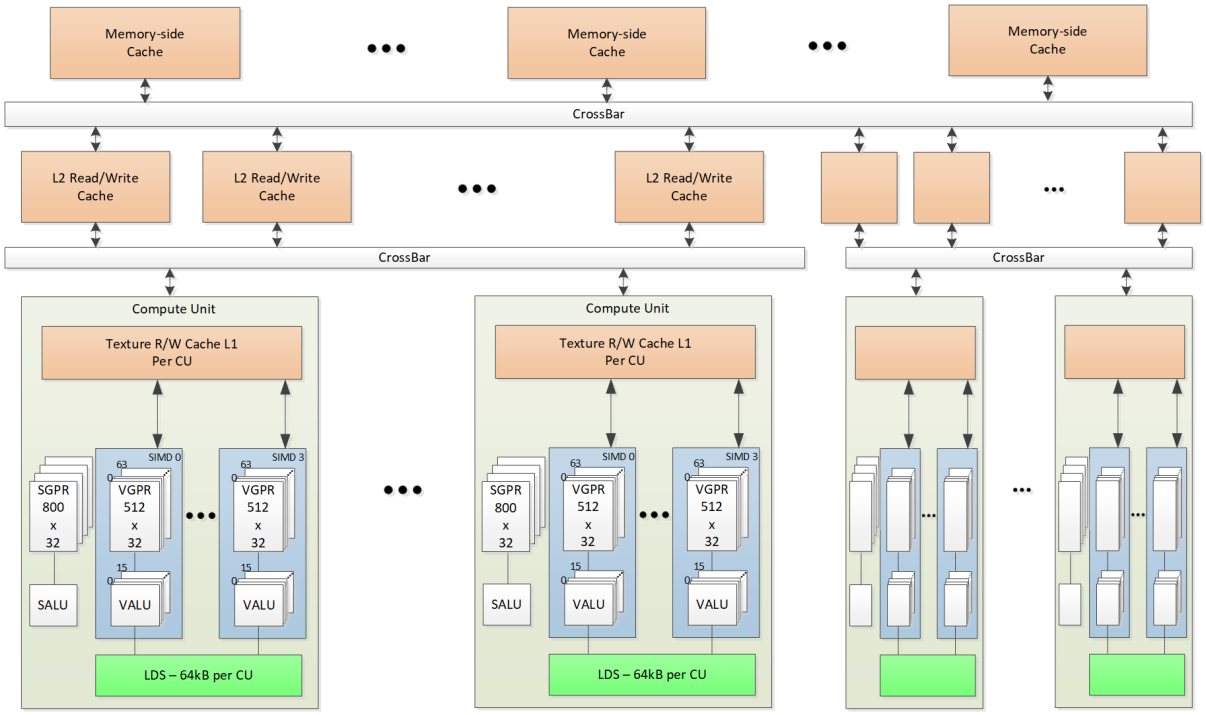


Figure 2. Shared Memory Hierarchy

2.2.1. Local Data Share (LDS)

Each compute unit has a 64 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 512 entries of 4 bytes. The shared memory contains 32 integer atomic units designed 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 efficient access to off-chip memory.

2.2.2. Global Wave Sync (GWS)

The AMD CDNA devices contain a global synchronization unit (GWS) capable of synchronizing workgroups across the device. Instructions which use this capability are sometimes referred to as "GDS".

2.3. Device Memory

The AMD CDNA devices offer 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-write cache that provides data to an L1 cache for each compute unit. Specific 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 acknowledgment to the requesting PE upon write confirmation of the return value (pre-atomic op value at destination) being stored to device memory.

This acknowledgment 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 acknowledgment, and
- enabling the system to maintain a relaxed consistency model.

Each scatter write from a given PE to a given memory channel maintains order. The acknowledgment 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

The table below shows all of the hardware states readable or writable by a shader program.

Table 2. Readable and Writable Hardware States

Abbrev.	Name	Size (bits)	Description
PC	Program Counter	48	Points to the memory address of the next shader instruction to execute.
V0-V255	VGPR	32	Vector general-purpose register ("architectural VGPRs").
AV0-AV255	AccVGPR	32	Matrix Accumulation Vector general-purpose register.
S0-S103	SGPR	32	Vector general-purpose register.
LDS	Local Data Share	64kB	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	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	A single bit flag indicating that the EXEC mask is all zeros.
VCC	Vector Condition Code	64	A bit mask with one bit per thread; it holds the result of a vector compare operation.
VCCZ	VCC is zero	1	A single bit-flag indicating that the VCC mask is all zeros.
SCC	Scalar Condition Code	1	Result from a scalar ALU comparison instruction.
FLAT_SCRATCH	Flat scratch address	64	The 64-bit base address of scratch memory, in NumSGPRs-5 and -6. Read Only.
XNACK_MASK	Address translation failure.	64	Bit mask of threads that have failed their address translation.
STATUS	Status	32	Read-only shader status bits.
MODE	Mode	32	Writable shader mode bits.
M0	Memory Reg	32	A temporary register that has various uses, including GPR indexing and bounds checking.
HW_ID	Hardware ID	32	Read-only status register that has various wave ID state.
XCC_ID	Compute ID	32	Read-only status register that contains the compute device ID.
TRAPSTS	Trap Status	32	Holds information about exceptions and pending traps.
TBA	Trap Base Address	64	Holds the pointer to the current trap handler program.
TMA	Trap Memory Address	64	Temporary register for shader operations. For example, can hold a pointer to memory used by the trap handler.
TTMP0-TTMP15	Trap Temporary SGPRs	32	16 SGPRs available only to the Trap Handler for temporary storage.
VMCNT	Vector memory instruction count	6	Counts the number of VMEM instructions issued but not yet completed.
EXPCNT	Export Count	3	Counts the number of GDS instructions issued but not yet completed.

Abbrev.	Name	Size (bits)	Description
LGKMCNT	LDS, GDS, Constant and Message count	4	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, and LDS 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.

This Accelerator does no optimization when EXEC = 0. The shader hardware executes every instruction, wasting instruction issue bandwidth. Use CBRANCH or VSKIP to 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. The table below lists and briefly describes the status register fields. *The status register fields may be written when PRIV=1. Some fields are set as a result of shader instructions.*

Table 3. 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.

Field	Bit Position	Description
SPI_PRIO	2:1	Wavefront priority set by the shader processor interpolator (SPI) when the wavefront is created. See the S_SETPRIO instruction (page 12-49) 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 12-49) 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 not taken.
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.
VALID	16	Wavefront is active (has been created and not yet ended).
ECC_ERR	17	An ECC error has occurred.
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.
ALLOW_REPLAY	22	Indicates that ATC replay is enabled. terminating.

3.5. Mode register

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

Table 4. Mode Register Fields

Field	Bit Position	Description
FP_ROUND	3:0	[1:0] Single precision round mode. [3:2] Double/Half precision round mode. Round Modes: 0=nearest even, 1= +infinity, 2= -infinity, 3= toward zero.
FP_DENORM	7:4	[1:0] Single precision denormal mode. [3:2] Double/Half precision 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.

Field	Bit Position	Description
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. [19] : address watch [20] : memory violation [20] : trap on wave end
FP16_OVFL	23	If set, an overflowed FP16 result is clamped to +/- MAX_FP16, regardless of round mode, while still preserving true INF values.
DISABLE_PERF	26	1 = disable performance counting for this wave
GPR_IDX_EN	27	GPR index enable.
VSKIP	28	0 = normal operation. 1 = skip (do not execute) any vector instructions: valu, vmem, 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.

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

This section defines the behavior when a source or destination GPR or memory address is outside the legal range for a wavefront.

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_DWORDx4).
    - 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 16 to 102 SGPRs, in units of 16 GPRs (Dwords). These are logically viewed as SGPRs 0-101. The VCC is physically stored as part of the wavefront's SGPRs in the highest numbered two SGPRs (SGPR 106 and 107; 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 typically 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 eight Dwords.

VGPRs are allocated out of two pools: regular VGPRs and accumulation VGPRs. Accumulation VGPRs are used with matrix VALU instructions, and can also be loaded directly from memory. A wave may have up to 512 total

VGPRs, 256 of each type. When a wave has fewer than 512 total VGPRs, the number of each type is flexible - it is not required to be equal numbers of both types.

Instructions which operate on 64-bit data must use aligned (i.e. even) VGPRs. This applies to ALU and memory instructions. GWS instructions must also be even-aligned.

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 512 bytes on 512-byte 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. M0 Memory Descriptor

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

- Local Data Share (LDS)
  - LDS addressing for Memory/Vfetch → LDS: {16'h0, lds_offset[15:0]} // in bytes
- Global Wave Sync (GWS)
  - { base[5:0], 16'h0}
- Indirect GPR addressing for both vector and scalar instructions. M0 is an unsigned index.

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 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 fully written; there are no partial mask updates.

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.

Shader Hazard with VCC 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 registers which control and report kernel exceptions.

All Trap temporary SGPRs (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. TMA and TBA are read-only; they can be accessed through S_GETREG_B32.

When a trap is taken (either user initiated, 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 one for host initiated traps, and 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).

STATUS . TRAP_EN - This bit indicates to 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.

MODE . EXCP_EN[8:0] - Floating point exception enables. Defines which exceptions and events 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	Address Watch - TC (L1) has witnessed a thread access to an 'address of interest'

3.10.1. Trap Status register

The trap status register records previously seen traps or exceptions. It can be read and written by the kernel.

Table 5. Exception Field Bits

Field	Bits	Description
EXCP	8:0	Status bits of 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. These can be read or written without shader privilege. Bit Exception 0 invalid 1 Input Denormal 2 Divide by zero 3 overflow 4 underflow 5 inexact 6 integer divide by zero 7 address watch 8 memory violation
SAVECTX	10	A bit set by the host command indicating that this wave must jump to its trap handler and save its context. This bit must be cleared by the trap handler using S_SETREG. Note - a shader can set this bit to 1 to cause a save-context trap, and due to hardware latency the shader may execute up to 2 additional instructions before taking the trap.
ILLEGAL_INST	11	An illegal instruction has been detected.
ADDR_WATCH1-3	14:12	Indicates that address watch 1, 2, or 3 has been hit. Bit 12 is address watch 1; bit 13 is 2; bit 14 is 3.
EXCP_CYCLE	21:16	When a float exception occurs, this tells the trap handler on which cycle the exception occurred on. 0-3 for normal float operations, 0-7 for double float add, and 0-15 for double float muladd or transcendentals. This register records the cycle number of the first occurrence of an enabled (unmasked) exception. EXCP_CYCLE[1:0] Phase: threads 0-15 are in phase 0, 48-63 in phase 3. EXCP_CYCLE[3:2] Multi-slot pass. EXCP_CYCLE[5:4] Hybrid pass: used for machines running at lower rates.
DP_RATE	31:29	Determines how the shader interprets the TRAP_STS.cycle. Different Vector Shader Processors (VSP) process instructions at different rates.

3.11. Memory Violations

A Memory Violation is reported from:

- LDS alignment error.
- Memory read/write/atomic alignment error.
- Flat access where the address is invalid (does not fall in any aperture).
- Write to a read-only memory address.
- 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.

3.12. Hardware ID Registers

The values below indicate where a wave is currently execution. It is not safe to rely on these values as they may change over the lifetime of a wave.

Table 6. Hardware ID (HW_ID)

Field	Bits	Description
WAVE_ID	3:0	Wave buffer slot number
SIMD_ID	5:4	SIMD which the wave is assigned to within the CU
PIPE_ID	7:6	Pipeline from which the wave was dispatched
CU_ID	11:8	Compute Unit the wave is assigned to
SH_ID	12	Shader Array (within an SE) the wave is assigned to. *Is set to zero.*
SE_ID	15:13	Shader Engine the wave is assigned to
TG_ID	19:16	Thread-group ID
VM_ID	23:20	Virtual Memory ID
QUEUE_ID	26:24	Queue from which this wave was dispatched
STATE_ID	29:27	State ID (UNUSED)
ME_ID	31:30	Micro-engine ID

Table 7. XCC ID (XCC_ID)

Field	Bits	Description
XCC_ID	3:0	ID of this XCC

3.13. GPR Initialization

When a compute shader wave is launched VGPR0 and a number of SGPRs are initialized.

Compute shaders have VGPR0 initialized with the X, Y and Z index within the workgroup: { 2'b00, Z[9:0], Y[9:0], X[9:0] }.

Table 8. **CS SGPR Load**

SGPR Order	Description	Enable
First 0.. 16 of	User data registers	COMPUTE_PGM_RSRC2.user_sgpr
then	work_group_id0[31:0]	COMPUTE_PGM_RSRC2.tgid_x_en
then	work_group_id1[31:0]	COMPUTE_PGM_RSRC2.tgid_y_en
then	work_group_id2[31:0]	COMPUTE_PGM_RSRC2.tgid_z_en
then	{first_wave, 6'h00, wave_id_in_group[4:0], 2'h0, ordered_append_term[11:0], work-group_size_in_waves[5:0]}	COMPUTE_PGM_RSRC2.tg_size_en
TTMP4,5	0	
TTMP6	dispatch packet addr lo	
TTMP7	dispatch packet addr hi	
TTMP8	dispatch grid X[31:0]	
TTMP9	dispatch grid Y[31:0]	
TTMP10	dispatch grid Z[31:0]	
TTMP11	{ 26'b0, wave_id_in_workgroup[5:0] }	

Other TTMPs are not initialized.

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 the table below control the priority and termination of a shader program, as well as provide support for trap handlers.

Table 9. Control Instructions

Instructions	Description
S_ENDPGM	Terminates the wavefront. It can appear anywhere in the kernel and can appear multiple times.
S_ENDPGM_SAVED	Terminates the wavefront due to context save. It can appear anywhere in the kernel and can appear multiple times.
S_NOP	Does nothing; it can be repeated in hardware up to 16 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 - 8128 clock cycles.
S_SENDMSG	Sends a message (typically an interrupt) to the host CPU.
S_WAKEUP	Causes one wave in a work-group to signal all other waves in the same work-group to wake up from S_SLEEP early. If waves are not sleeping, they are not affected by this instruction.

4.2. Branching

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

Table 10. Branch Instructions

Instructions	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.
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_{G,I}_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.

Instructions	Description
S_CALL_B64	Jump to a subroutine, and save return address. SGPR_pair = PC+4; PC = PC+4+SIMM16*4.

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. Workgroups

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 workgroup, 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 GWS 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 (MUBUF, MTBUF, or FLAT 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, GWS, (K)constant, (M)essage) Determines when one of these low-latency instructions have completed.
  - Incremented by 1 for every LDS or GWS instruction issued, as well as by Dword-count for scalar-memory reads (1 for 1-dword loads, 2 for 2-dword or larger loads). S_memtime counts the same as an s_load_dwordx2.
  - Decremented by 1 for LDS/GWS 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/GWS writes when the data has been written to LDS/GWS.
- Decremented by 1 for each Dword returned from the data-cache (SMEM).
  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 GWS, at which time it is safe to overwrite the contents of that VGPR.
  - Incremented when an GWS instruction is issued from the wavefront buffer.
  - Decremented for GWS when the last cycle of the GWS instruction is granted and executed (VGPRs read out).

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 11. Required Software-inserted Wait States

First Instruction	Second Instruction	Wait	Notes
S_SETREG <*>	S_GETREG <same reg>	2	
S_SETREG <*>	S_SETREG <same reg>	2	
SET_VSKIP	S_GETREG MODE	2	Reads VSKIP from MODE.
S_SETREG MODE.vskip	any vector op	2	Requires two nops or non-vector instructions.
VALU that sets VCC or EXEC	VALU that uses EXECZ or VCCZ as a data source	5	
VALU writes SGPR/VCC (readlane, cmp, add/sub, div_scale)	V_{READ,WRITE}LANE using that SGPR/VCC as the lane select	4	
VALU writes VCC (including v_div_scale)	V_DIV_FMAS	4	
FLAT_STORE_X3 FLAT_STORE_X4 FLAT_ATOMIC_{F}CMPSWAP_X2 (and global & scratch stores/atomics) BUFFER_STORE_DWORD_X3 BUFFER_STORE_DWORD_X4 BUFFER_STORE_FORMAT_XYZ BUFFER_STORE_FORMAT_XYZW BUFFER_ATOMIC_{F}CMPSWAP_X2	Write VGPRs holding writedata from those instructions.	1	BUFFER_STORE_* operations that use an SGPR for "offset" do not require any wait states.
FLAT_STORE_X3 FLAT_STORE_X4 (and global & scratch stores/atomics) 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	VALU writes VGPRs holding writedata from those instructions.	2	BUFFER_STORE_* operations that use an SGPR for "offset" do not require any wait states.

First Instruction	Second Instruction	Wait	Notes
VALU writes SGPR	VMEM reads that SGPR	5	Hardware assumes that there is no dependency here. If the VALU writes the SGPR that is used by a VMEM, the user must add five wait states.
SALU writes M0	GDS, S_SENDMSG	1	
VALU writes VGPR	VALU DPP reads that VGPR	2	
VALU writes EXEC	VALU DPP op	5	ALU does not forward EXEC to DPP.
Mixed use of VCC: alias vs SGPR# v_readlane, v_readfirstlane v_cmp v_add**i/u* *v_sub*_i/u* *v_div_scale** (writes vcc)	VALU which reads VCC as a constant (not as a carry-in which is 0 wait states).	1	VCC can be accessed by name or by the logical SGPR which holds VCC. The data dependency check logic does not understand that these are the same register and do not prevent races.
S_SETREG TRAPSTS	RFE, RFE_restore	1	
SALU writes M0	LDS "add-TID" instruction, buffer_store_LDS_dword, scratch or global with LDS = 1	1	
SALU writes M0	S_MOVEREL	1	
VALU writes SGPR/VCC: v_readlane, v_readfirstlane, v_cmp, v_add*_i/u, v_sub*_i/u, v_div_scale*	VALU reads SGPR as constant	2	
	VALU reads SGPR as carry-in	0	
	v_readlane, v_writelane reads SGPR as lane-select	4	
v_cmpx	VALU reads EXEC as constant	2	
	V_readlane, v_readfirstlane, v_writelane	4	
	Other VALU	0	
VALU writes VGPRn	v_readlane vsrc0 reads VGPRn	1	
VALU op which uses OPSEL or SDWA with changes the result's bit position	VALU op consumes result of that op	1	
VALU Trans op	Non-trans VALU op consumes result of that op	1	

Table 12. Trans Ops

V_EXP_F32	V_LOG_F32	V_RCP_F32	V_RCP_IFLAG_F32
V_RSQ_F32	V_RCP_F64	V_RSQ_F64	V_SQRT_F32
V_SQRT_F64	V_SIN_F32	V_COS_F32	V_RCP_F16
V_SQRT_F16	V_RSQ_F16	V_LOG_F16	V_EXP_F16
V_SIN_F16	V_COS_F16	V_EXP_LEGACY_F32	V_LOG_LEGACY_F32

4.6. Arbitrary Divergent Control Flow

In the CDNA 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 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. 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.

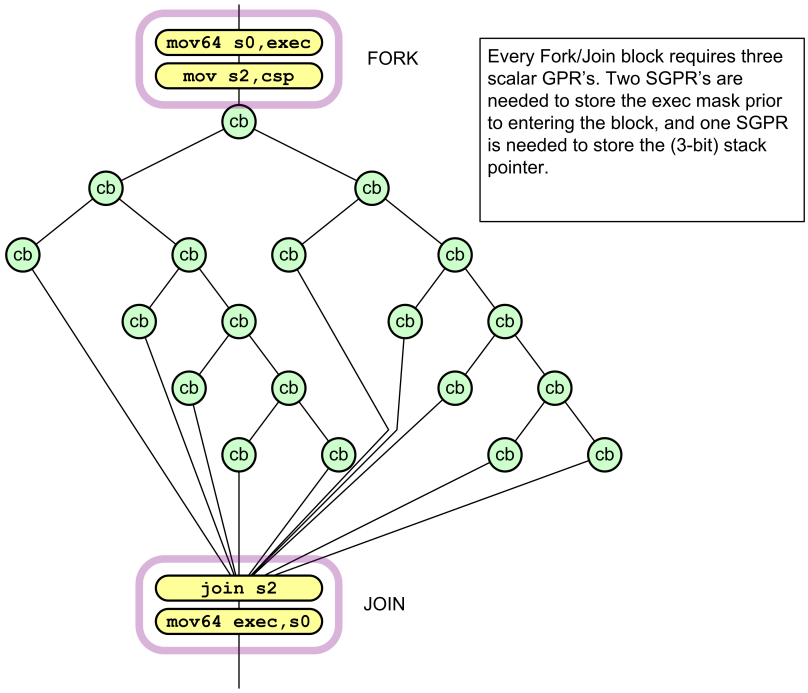


Figure 3. Example of Complex Control Flow Graph

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 $Log_2 64 = 6$.

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: 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)
```

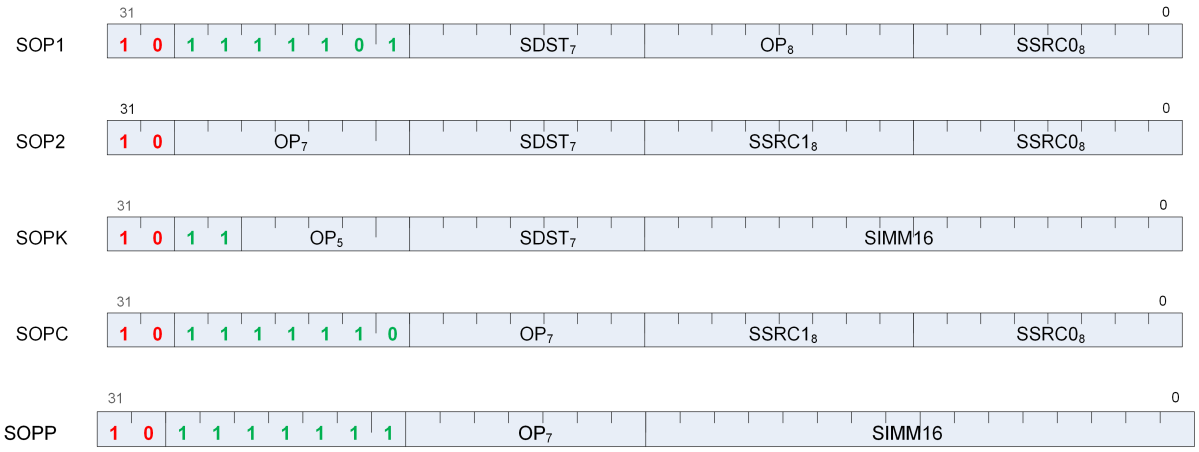
```
    I_FORK : PC += 4 + target_addr_offset
    G_FORK: PC = SGPR[arg1]
else if (mask_fail == exec)
    PC += 4
else if (bitcount(mask_fail) < bitcount(mask_pass))
    exec = mask_fail
    I_FORK : SGPR[CSP*4] = { (pc + 4 + target_addr_offset), mask_pass }
    G_FORK: SGPR[CSP*4] = { SGPR[arg1], mask_pass }
    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 bit-wise 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 five microcode formats, shown below:



Each of these instruction formats uses some of these fields:

Field	Description
OP	Opcode: instruction to be executed.
SDST	Destination SGPR.
SSRC0	First source operand.
SSRC1	Second source operand.
SIMM16	Signed immediate 16-bit integer constant.

The lists of similar instructions sometimes use 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 SCC.
- Hardware registers.
- 32-bit literal constant.

In the table below, 0-127 can be used as scalar sources or destinations; 128-255 can only be used as sources.

Table 13. Scalar Operands

	Code	Meaning	Description
Scalar Dest (7 bits)	0 - 101	SGPR 0 to 101	Scalar GPRs
	102	FLAT_SCRATCH_LO	Holds the low Dword of the flat-scratch memory descriptor
	103	FLAT_SCRATCH_HI	Holds the high Dword of the flat-scratch memory descriptor
	104	XNACK_MASK_LO	Holds the low Dword of the XNACK mask.
	105	XNACK_MASK_HI	Holds the high Dword of the XNACK mask.
	106	VCC_LO	Holds the low Dword of the vector condition code
	107	VCC_HI	Holds the high Dword of the vector condition code
	108-123	TTMP0 to TTMP15	Trap temps (privileged)
	124	M0	Holds the low Dword of the flat-scratch memory descriptor
	125	reserved	reserved
	126	EXEC_LO	Execute mask, low Dword
	127	EXEC_HI	Execute mask, high Dword
	128	0	zero
	129-192	int 1 to 64	Positive integer values.
	193-208	int -1 to -16	Negative integer values.
	209-234	reserved	Unused.
	235	SHARED_BASE	Memory Aperture definition.
	236	SHARED_LIMIT	
	237	PRIVATE_BASE	
	238	PRIVATE_LIMIT	
	239	Reserved	Reserved
	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	1.0 / (2 * PI)	
	249-250	reserved	unused
	251	VCCZ	{ zeros, VCCZ }

	Code	Meaning	Description
	252	EXECZ	{ zeros, EXECZ }
	253	SCC	{ zeros, SCC }
	254	reserved	unused
	255	Literal	constant 32-bit constant from instruction stream.

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. Literal constants are used by setting the source instruction field to "literal" (255), and then the following instruction dword is used as the source value.

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. However, SCC and EXEC (for saveexec) are written.

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. For add, 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 (AB), 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.

5.4. Integer Arithmetic Instructions

This section describes the arithmetic operations supplied by the SALU. The table below shows the scalar integer arithmetic instructions:

Table 14. Integer Arithmetic Instructions

Instruction	Encoding	Sets SCC?	Operation
S_ADD_I32	SOP2	y	D = S0 + S1, SCC = overflow.
S_ADD_U32	SOP2	y	D = S0 + S1, SCC = carry out.
S_ADDC_U32	SOP2	y	D = S0 + S1 + SCC = overflow.
S_SUB_I32	SOP2	y	D = S0 - S1, SCC = overflow.
S_SUB_U32	SOP2	y	D = S0 - S1, SCC = carry out.
S_SUBB_U32	SOP2	y	D = S0 - S1 - SCC = carry out.
S_ABSDIFF_I32	SOP2	y	D = abs (s1 - s2), SCC = result not zero.

Instruction	Encoding	Sets SCC?	Operation
S_MIN_I32 S_MIN_U32	SOP2	y	D = (S0 < S1) ? S0 : S1. SCC = 1 if S0 was min.
S_MAX_I32 S_MAX_U32	SOP2	y	D = (S0 > S1) ? S0 : S1. SCC = 1 if S0 was max.
S_MUL_I32	SOP2	n	D = S0 * S1. Low 32 bits of result.
S_ADDK_I32	SOPK	y	D = D + simm16, SCC = overflow. Sign extended version of simm16.
S_MULK_I32	SOPK	n	D = D * simm16. Return low 32bits. Sign extended version of simm16.
S_ABS_I32	SOP1	y	D.i = abs (S0.i). SCC=result not zero.
S_SEXT_I32_I8	SOP1	n	D = { 24{S0[7]}, S0[7:0] }.
S_SEXT_I32_I16	SOP1	n	D = { 16{S0[15]}, S0[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 15. Conditional Instructions

Instruction	Encoding	Sets SCC?	Operation
S_CSELECT_{B32, B64}	SOP2	n	D = SCC ? S0 : S1.
S_CMOVK_I32	SOPK	n	if (SCC) D = signext(simm16).
S_CMOV_{B32,B64}	SOP1	n	if (SCC) D = S0, 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 16. Conditional Instructions

Instruction	Encoding	Sets SCC?	Operation
S_CMP_EQ_U64, S_CMP_NE_U64	SOPC	y	Compare two 64-bit source values. SCC = S0 <cond> S1.
S_CMP_{EQ,NE,GT,GE,LE,LT}_{I32,U32}	SOPC	y	Compare two source values. SCC = S0 <cond> S1.
S_CMPK_{EQ,NE,GT,GE,LE,LT}_{I32,U32}	SOPK	y	Compare Dest SGPR to a constant. SCC = DST <cond> simm16. simm16 is zero-extended (U32) or sign-extended (I32).
S_BITCMP0_{B32,B64}	SOPC	y	Test for "is a bit zero". SCC = !S0[S1].
S_BITCMP1_{B32,B64}	SOPC	y	Test for "is a bit one". SCC = S0[S1].

5.7. Bit-Wise Instructions

Bit-wise instructions operate on 32- or 64-bit data without interpreting it has having a type. For bit-wise operations if noted in the table below, SCC is set if the result is nonzero.

Table 17. Bit-Wise Instructions

Instruction	Encoding	Sets SCC?	Operation
S_MOV_{B32,B64}	SOP1	n	D = S0
S_MOVK_I32	SOPK	n	D = signext(simm16)
{S_AND,S_OR,S_XOR}_{B32,B64}	SOP2	y	D = S0 & S1, S0 OR S1, S0 XOR S1
{S_ANDN2,S_ORN2}_{B32,B64}	SOP2	y	D = S0 & ~S1, S0 OR ~S1, S0 XOR ~S1,
{S_NAND,S_NOR,S_XNOR}_{B32,B64}	SOP2	y	D = ~(S0 & S1), ~(S0 OR S1), ~(S0 XOR S1)
S_LSHL_{B32,B64}	SOP2	y	D = S0 << S1[4:0], [5:0] for B64.
S_LSHR_{B32,B64}	SOP2	y	D = S0 >> S1[4:0], [5:0] for B64.
S_ASHR_{I32,I64}	SOP2	y	D = sext(S0 >> S1[4:0]) ([5:0] for I64).
S_BFM_{B32,B64}	SOP2	n	Bit field mask. D = ((1 << S0[4:0]) - 1) << S1[4:0].
S_BFE_U32, S_BFE_U64 S_BFE_I32, S_BFE_I64 (signed/unsigned)	SOP2	y	Bit Field Extract, then sign-extend result for I32/64 instructions. S0 = data, S1[5:0] = offset, S1[22:16]= width.
S_NOT_{B32,B64}	SOP1	y	D = ~S0.
S_WQM_{B32,B64}	SOP1	y	D = wholeQuadMode(S0). 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(S0[3:0]), D[1]=OR(S0[7:4]), etc.
S_BREV_{B32,B64}	SOP1	n	D = S0[0:31] are reverse bits.
S_BCNT0_I32_{B32,B64}	SOP1	y	D = CountZeroBits(S0).
S_BCNT1_I32_{B32,B64}	SOP1	y	D = CountOneBits(S0).
S_FF0_I32_{B32,B64}	SOP1	n	D = Bit position of first zero in S0 starting from LSB. -1 if not found.
S_FF1_I32_{B32,B64}	SOP1	n	D = Bit position of first one in S0 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[S0[4:0], [5:0] for B64] = 0
S_BITSET1_{B32,B64}	SOP1	n	D[S0[4:0], [5:0] for B64] = 1
S_{and,or,xor,andn2,orn2,nand, nor,xnor}_SAVEEXEC_B64	SOP1	y	Save the EXEC mask, then apply a bit-wise operation to it. D = EXEC EXEC = S0 <op> EXEC SCC = (exec != 0)
S_{ANDN{1,2}_WREXEC_B64	SOP1	y	N1: EXEC, D = ~S0 & EXEC N2: EXEC, D = S0 & ~EXEC Both D and EXEC get the same result. SCC = (result != 0).

Instruction	Encoding	Sets SCC?	Operation
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[S0+M0] MOVERELD: SGPR[D+M0] = S0 Index must be even for 64. M0 is an unsigned index.

5.8. Access Instructions

These instructions access hardware internal registers.

Table 18. Hardware Internal Registers

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 the table above. Some bits of the DEST specify which register to read/write, but additional bits specify which bits in the register to read/write:

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

Table 19. 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. {lgkm_cnt, exp_cnt, vm_cnt}.
8 - 15		reserved.
16	TBA_LO	Trap base address register [31:0].
17	TBA_HI	Trap base address register [47:32].
18	TMA_LO	Trap memory address register [31:0].
19	TMA_HI	Trap memory address register [47:32].
20	XCC_ID	ID of the XCC this wave is running on
21	PERF_SNAPSHOT_DATA	Stochastic Performance sampling data
22	PERF_SNAPSHOT_DATA1	Stochastic Performance sampling data1
23	PERF_SNAPSHOT_PC_LO	Stochastic Performance sampling program counter
24	PERF_SNAPSHOT_PC_HI	Stochastic Performance sampling program counter

Table 20. IB_STS

Code	Register	Description
VM_CNT	23:22, 3:0	Number of VMEM instructions issued but not yet returned.
EXP_CNT	6:4	Number of GDS issued but have not yet read their data from VGPRs.
LGKM_CNT	11:8	LDS, GDS, Constant-memory and Message instructions issued-but-not-completed count.

Table 21. GPR_ALLOC

Code	Register	Description
VGPR_BASE	5:0	Physical address of first VGPR assigned to this wavefront, as [7:2]
VGPR_SIZE	11:6	Number of VGPRs assigned to this wavefront, as [7:2]. 0=4 VGPRs, 1=8 VGPRs, etc.
ACCV_OFF	17:12	Accumulation VGPR offset from VGPR_BASE, in units of 4 VGPRs.
SGPR_BASE	23:18	Physical address of first SGPR assigned to this wavefront, as [8:3].
SGPR_SIZE	27:24	Number of SGPRs assigned to this wave, as [7:4]. 0=16 SGPRs, 1=32 SGPRs, etc.

Table 22. LDS_ALLOC

Code	Register	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.

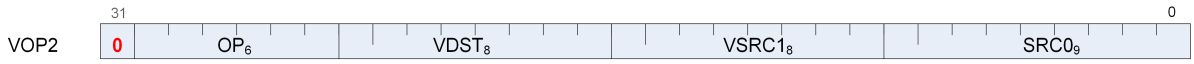
6.1. Microcode Encodings

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.

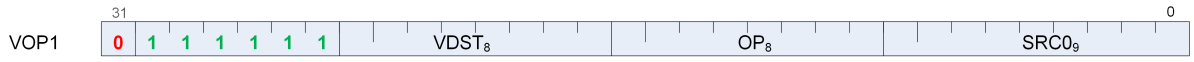
When an instruction is available in two microcode formats, it is up to the user to decide which to use. It is recommended to use the 32-bit encoding whenever possible.

The microcode encodings are shown below.

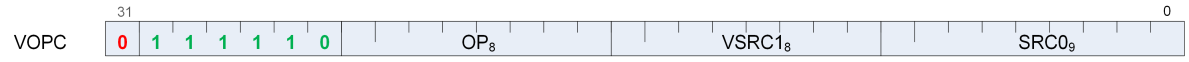
VOP2 is for instructions with two inputs and a single vector destination. Instructions that have a carry-out implicitly write the carry-out to the VCC register.



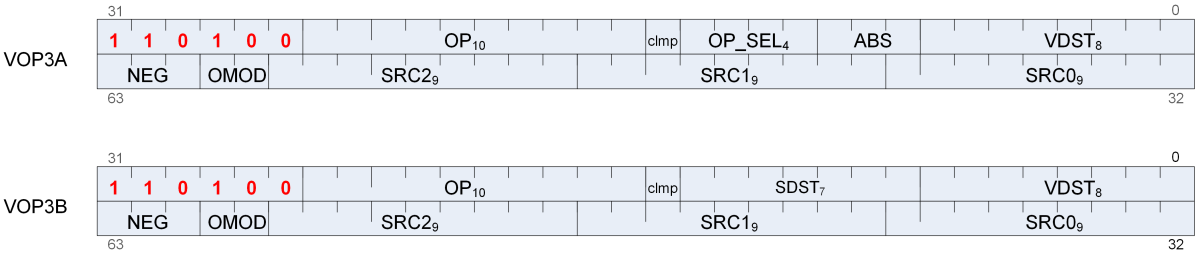
VOP1 is for instructions with no inputs or a single input and one destination.



VOPC is for comparison instructions.

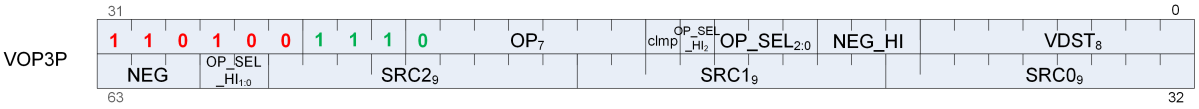


VOP3 is for instructions with up to three inputs, input modifiers (negate and absolute value), and output modifiers. There are two forms of VOP3: one which uses a scalar destination field (used only for div_scale, integer add and subtract); this is designated VOP3b. All other instructions use the common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

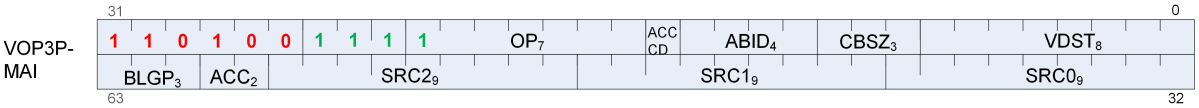


Any of the 32-bit microcode formats may use a 32-bit literal constant, but not VOP3.

VOP3P is for instructions that use "packed math": They perform the operation on a pair of input values that are packed into the high and low 16-bits of each operand; the two 16-bit results are written to a single VGPR as two packed values.



VOP3P-MAI is a variation of the VOP3P format for use with the Matrix Arithmetic Instructions (MAI).



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_MUL_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 - constant selected by a specific VSRC value.
- Literal constant - 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.
- 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.

Limitations for Constants

VALU "ADDC", "SUBB" and CNDMASK all implicitly use an
SGPR value (VCC), so these instructions cannot use an additional SGPR or literal constant.

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.

Limitations for SDWA and OPSEL

DOT instructions must not use SDWA or OPSEL.
VALU ops which use SDWA or OPSEL must not consume the result of that instruction in the next VALU instruction - there must be at least on independent instruction or V_NOP between them.

6.2.1.1. Literal Expansion to 64 bits

Literal constants are 32-bits, but they can be used as sources which normally require 64-bit data:

- 64 bit float: the lower 32-bit are padded with zero.
- 64-bit unsigned integer: zero extended to 64 bits
- 64-bit signed integer: sign extended to 64 bits

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 [0.0, +1.0].

Output modifiers apply only to floating point results and are ignored for integer or bit results. Output modifiers are not compatible with output denormals: if output denormals are enabled, then output modifiers are ignored. If output denormals are disabled, then the output modifier is applied and denormals are flushed to zero. Output modifiers are not IEEE compatible: -0 is flushed to +0. Output modifiers are ignored if the IEEE mode bit is set to 1.

In the table below, 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 23. Instruction Operands

Value	Name	Description
0-101	SGPR	0 .. 101
102	FLAT_SCRATCH_LO	Flat Scratch[31:0].
103	FLAT_SCRATCH_HI	Flat Scratch[63:32].
104	XNACK_MASK_LO	
105	XNACK_MASK_HI	
106	VCC_LO	vcc[31:0].
107	VCC_HI	vcc[63:32].
108-123	TTMP0 to TTMP 15	Trap handler temps (privileged).
124	M0	
125	reserved	
126	EXEC_LO	exec[31:0].
127	EXEC_HI	exec[63:32].
128	0	

Value	Name	Description
129-192	int 1.. 64	Integer inline constants.
193-208	int -1 .. -16	
209-234	reserved	Unused.
235	SHARED_BASE	Memory Aperture definition.
236	SHARED_LIMIT	
237	PRIVATE_BASE	
238	PRIVATE_LIMIT	
239	Reserved	Reserved
240	0.5	Single, double, or half-precision inline floats. 1/(2*PI) is 0.15915494. The exact value used is: half: 0x3118 single: 0x3e22f983 double: 0x3fc45f306dc9c882
241	-0.5	
242	1.0	
243	-1.0	
244	2.0	
245	-2.0	
246	4.0	
247	-4.0	
248	1/(2*PI)	
249	SDWA	Sub Dword Address (only valid as Source-0)
250	DPP	DPP over 16 lanes (only valid as Source-0)
251	VCCZ	{ zeros, VCCZ }
252	EXECZ	{ zeros, EXECZ }
253	SCC	{ zeros, SCC }
254	Reserved	Reserved
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.

6.3. Instructions

The table below lists the complete VALU instruction set by microcode encoding, except for VOP3P instructions which are listed in a later section.

Table 24. VALU Instruction Set

VOP3	VOP3 - 2 operands	VOP2	VOP1
V_ADD3_U32	V_ADD_F64	V_ADDC_CO_U32	V_ACCVGPR_MOV_B32
V_ADD_LSHL_U32	V_ADD_I16	V_ADD_CO_U32	V_BFREV_B32
V_ALIGNBIT_B32	V_ADD_I32	V_ADD_F16	V_CEIL_F16
V_ALIGNBYTE_B32	V_ASHRREV_I64	V_ADD_F32	V_CEIL_F32
V_AND_OR_B32	V_BCNT_U32_B32	V_ADD_U16	V_CEIL_F64
V_BFE_I32	V_BFM_B32	V_ADD_U32	V_CLREXCP
V_BFE_U32	V_CVT_PKACCUM_U8_F32	V_AND_B32	V_COS_F16

VOP3	VOP3 - 2 operands	VOP2	VOP1
V_BFI_B32	V_CVT_PKNORM_I16_F16	V_ASHRREV_I16	V_COS_F32
V_CUBEID_F32	V_CVT_PKNORM_I16_F32	V_ASHRREV_I32	V_CVT_F16_F32
V_CUBEMA_F32	V_CVT_PKNORM_U16_F16	V_CNDMASK_B32	V_CVT_F16_I16
V_CUBESC_F32	V_CVT_PKNORM_U16_F32	V_DOT2C_F32_F16	V_CVT_F16_U16
V_CUBETC_F32	V_CVT_PKRTZ_F16_F32	V_DOT2C_I32_I16	V_CVT_F32_BF8
V_CVT_PK_U8_F32	V_CVT_PK_BF8_F32	V_DOT4C_I32_I8	V_CVT_F32_F16
V_DIV_FIXUP_F16	V_CVT_PK_FP8_F32	V_DOT8C_I32_I4	V_CVT_F32_F64
V_DIV_FIXUP_F32	V_CVT_PK_I16_I32	V_FMAAK_F32	V_CVT_F32_FP8
V_DIV_FIXUP_F64	V_CVT_PK_U16_U32	V_FMAC_F32	V_CVT_F32_I32
V_DIV_FIXUP_LEGACY_F16	V_CVT_SR_BF8_F32	V_FMAC_F64	V_CVT_F32_U32
V_DIV_FMAS_F32	V_CVT_SR_FP8_F32	V_FMAMK_F32	V_CVT_F32_UBYTE0
V_DIV_FMAS_F64	V_LDEXP_F32	V_LDEXP_F16	V_CVT_F32_UBYTE1
V_DIV_SCALE_F32	V_LDEXP_F64	V_LSHLREV_B16	V_CVT_F32_UBYTE2
V_DIV_SCALE_F64	V_LSHLREV_B64	V_LSHLREV_B32	V_CVT_F32_UBYTE3
V_FMA_F16	V_LSHRREV_B64	V_LSHRREV_B16	V_CVT_F64_F32
V_FMA_F32	V_MAX_F64	V_LSHRREV_B32	V_CVT_F64_I32
V_FMA_F64	V_MBCNT_HI_U32_B32	V_MAC_F16	V_CVT_F64_U32
V_FMA_LEGACY_F16	V_MBCNT_LO_U32_B32	V_MADAK_F16	V_CVT_FLR_I32_F32
V_LERP_U8	V_MIN_F64	V_MADMK_F16	V_CVT_I16_F16
V_LSHL_ADD_U32	V_MUL_F64	V_MAX_F16	V_CVT_I32_F32
V_LSHL_ADD_U64	V_MUL_HI_I32	V_MAX_F32	V_CVT_I32_F64
V_LSHL_OR_B32	V_MUL_HI_U32	V_MAX_I16	V_CVT_NORM_I16_F16
V_MAD_F16	V_MUL_LEGACY_F32	V_MAX_I32	V_CVT_NORM_U16_F16
V_MAD_I16	V_MUL_LO_U32	V_MAX_U16	V_CVT_OFF_F32_I4
V_MAD_I32_I16	V_PACK_B32_F16	V_MAX_U32	V_CVT_PK_F32_BF8
V_MAD_I32_I24	V_READLANE_B32	V_MIN_F16	V_CVT_PK_F32_FP8
V_MAD_I64_I32	V_SUB_I16	V_MIN_F32	V_CVT_RPI_I32_F32
V_MAD_LEGACY_F16	V_SUB_I32	V_MIN_I16	V_CVT_U16_F16
V_MAD_LEGACY_I16	V_TRIG_PREOP_F64	V_MIN_I32	V_CVT_U32_F32
V_MAD_LEGACY_U16	V_WRITELANE_B32	V_MIN_U16	V_CVT_U32_F64
V_MAD_U16		V_MIN_U32	V_EXP_F16
V_MAD_U32_U16		V_MUL_F16	V_EXP_F32
V_MAD_U32_U24		V_MUL_F32	V_FFBH_I32
V_MAD_U64_U32		V_MUL_HI_I32_I24	V_FFBH_U32
V_MAX3_F16		V_MUL_HI_U32_U24	V_FFBL_B32
V_MAX3_F32		V_MUL_I32_I24	V_FLOOR_F16
V_MAX3_I16		V_MUL_LO_U16	V_FLOOR_F32
V_MAX3_I32		V_MUL_U32_U24	V_FLOOR_F64
V_MAX3_U16		V_OR_B32	V_FRACT_F16
V_MAX3_U32		V_PK_FMAC_F16	V_FRACT_F32
V_MED3_F16		V_SUBBREV_CO_U32	V_FRACT_F64
V_MED3_F32		V_SUBB_CO_U32	V_FREXP_EXP_I16_F16
V_MED3_I16		V_SUBREV_CO_U32	V_FREXP_EXP_I32_F32
V_MED3_I32		V_SUBREV_F16	V_FREXP_EXP_I32_F64
V_MED3_U16		V_SUBREV_F32	V_FREXP_MANT_F16
V_MED3_U32		V_SUBREV_U16	V_FREXP_MANT_F32
V_MIN3_F16		V_SUBREV_U32	V_FREXP_MANT_F64
V_MIN3_F32		V_SUB_CO_U32	V_LOG_F16
V_MIN3_I16		V_SUB_F16	V_LOG_F32
V_MIN3_I32		V_SUB_F32	V_MOV_B32
V_MIN3_U16		V_SUB_U16	V_MOV_B64
V_MIN3_U32		V_SUB_U32	V_NOP
V_MQSAD_PK_U16_U8		V_XNOR_B32	V_NOT_B32

VOP3	VOP3 - 2 operands	VOP2	VOP1
V_MQSAD_U32_U8		V_XOR_B32	V_RCP_F16
V_MSAD_U8			V_RCP_F32
V_OR3_B32			V_RCP_F64
V_PERM_B32			V_RCP_IFLAG_F32
V_QSAD_PK_U16_U8			V_READFIRSTLANE_B32
V_SAD_HI_U8			V_RNDNE_F16
V_SAD_U16			V_RNDNE_F32
V_SAD_U32			V_RNDNE_F64
V_SAD_U8			V_RSQ_F16
V_XAD_U32			V_RSQ_F32
			V_RSQ_F64
			V_SAT_PK_U8_I16
			V_SIN_F16
			V_SIN_F32
			V_SQRT_F16
			V_SQRT_F32
			V_SQRT_F64
			V_SWAP_B32
			V_TRUNC_F16
			V_TRUNC_F32
			V_TRUNC_F64

The next table lists the compare instructions.

Table 25. VALU Instruction Set

Op	Formats	Functions	Result
V_CMP	I16, I32, I64, U16, U32, U64	F, LT, EQ, LE, GT, LG, GE, T	Write VCC..
V_CMPX			Write VCC and exec.
V_CMP	F16, 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)	Write VCC.
V_CMPX			Write VCC and exec.
V_CMP_CLASS	F16, F32, F64	Test for one of: signaling-NaN, quiet-NaN, positive or negative: infinity, normal, subnormal, zero.	Write VCC.
V_CMPX_CLASS			Write VCC and exec.

6.4. Denormalized 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.

Note: that V_DOT2 instructions operating on floating point data do not support denormal and rounding modes. They flush input and output denorms.

Table 26. Round and Denormal Modes

Field	Bit Position	Description
FP_ROUND	3:0	[1:0] Single-precision round mode. [3:2] Double/Half-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/Half-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.

6.5. ALU Clamp Bit Usage

When using V_CMP instructions, setting the clamp bit to 1 indicates that the compare signals if a floating point exception occurs. For integer operations, it clamps the result to the largest and smallest representable value. For floating point operations, it clamps the result to the range: [0.0, 1.0].

6.6. VGPR Indexing

VGPR Indexing allows a value stored in the M0 register to act as an index into the VGPRs either for the source or destination registers in VALU instructions.

6.6.1. Indexing Instructions

The table below describes the instructions which enable, disable and control VGPR indexing.

Table 27. VGPR Indexing Instructions

Instruction	Encoding	Sets SCC?	Operation
S_SET_GPR_IDX_OFF	SOPP	N	Disable VGPR indexing mode. Sets: mode.gpr_idx_en = 0.
S_SET_GPR_IDX_ON	SOPC	N	Enable VGPR indexing, and set the index value and mode from an SGPR. mode.gpr_idx_en = 1 M0[7:0] = S0.u[7:0] M0[15:12] = SIMM4
S_SET_GPR_IDX_IDX	SOP1	N	Set the VGPR index value: M0[7:0] = S0.u[7:0]
S_SET_GPR_IDX_MODE	SOPP	N	Change the VGPR indexing mode, which is stored in M0[15:12]. M0[15:12] = SIMM4

Indexing is enabled and disabled by a bit in the MODE register: gpr_idx_en. When enabled, two fields from M0 are used to determine the index value and what it applies to:

- M0[7:0] holds the unsigned index value, added to selected source or destination VGPR addresses.
- M0[15:12] holds a four-bit mask indicating to which source or destination the index is applied.
  - M0[15] = dest_enable.
  - M0[14] = src2_enable.

- M0[13] = src1_enable.
- M0[12] = src0_enable.

Indexing only works on VGPR source and destinations, not on inline constants or SGPRs. It is illegal for the index attempt to address VGPRs that are out of range.

6.6.2. VGPR Indexing Details

This section describes how VGPR indexing is applied to instructions that use source and destination registers in unusual ways. The table below shows which M0 bits control indexing of the sources and destination registers for these specific instructions.

Instruction	Microcode Encodes	VALU Receives	M0[15] (dst)	M0[15] (s2)	M0[15] (s1)	M0[12] (s0)
v_readlane	sdst = src0, SS1		x	x	x	src0
v_readfirstlane	sdst = func(src0)		x	x	x	src0
v_writelane	dst = func(ss0, ss1)		dst	x	x	x
v_mac_*	dst = src0 * src1 + dst	mad: dst, src0, src1, src2	dst, s2	x	src1	src0
v_madak	dst = src0 * src1 + imm	mad: dst, src0, src1, src2	dst	x	src1	src0
v_madmk	dst = S0 * imm + src1	mad: dst, src0, src1, src2	dst	src2	x	src0
v_*sh*_rev	dst = S1 << S0	<shift> (src1, src0)	dst	x	src1	src0
v_cvt_pkaccum	uses dst as src2		dst, s2	x	src1	src0
SDWA (dest preserve, sub-Dword mask)	uses dst as src2 for read-mod-write			dst, s2		

where:
src= vector source
SS = scalar source
dst = vector destination
sdst = scalar destination

6.7. Packed Math

CDNA supports **packed math**, which performs operations on two 16-bit values within a Dword as if they were separate elements. For example, a packed add of V0=V1+V2 is really two separate adds: adding the low 16 bits of each Dword and storing the result in the low 16 bits of V0, and adding the high halves.

Packed math uses the instructions below and the microcode format "VOP3P". This format adds op_sel and neg fields for both the low and high operands, and removes ABS and OMOD.

Packed Math Opcodes:

V_PK_MAD_I16	V_PK_MUL_LO_U16	V_PK_ADD_I16	V_PK_SUB_I16
V_PK_LSHLREV_B16	V_PK_LSHRREV_B16	V_PK_ASHRREV_I16	V_PK_MAX_I16
V_PK_MIN_I16	V_PK_MAD_U16	V_PK_ADD_U16	V_PK_SUB_U16
V_PK_MAX_U16	V_PK_MIN_U16	V_PK_FMA_F16	V_PK_ADD_F16
V_PK_MUL_F16	V_PK_MIN_F16	V_PK_MAX_F16	
V_MAD_MIX_F32	V_MAD_MIXLO_F16	V_MAD_MIXHI_F16	

V_PK_FMA_F32	V_PK_MUL_F32	V_PK_ADD_F32	V_PK_MOV_B32

V_MAD_MIX_* are not packed math, but perform a single Multiply-Add operation on a mixture of 16- and 32-bit inputs. The Multiply-add is performed as an FMA - fused multiply-add. They are listed here because they use the VOP3P encoding.

Packed 32-bit instructions operate on 2 dwords at a time and those operands must be two-dword aligned (i.e. an even VGPR address). Output modifiers are not supported for these instructions. OPSEL and OPSEL_HI work to select the first or second DWORD for each source.

Chapter 7. Matrix Arithmetic Instructions

Matrix core is an extension to CDNA architecture shader instruction set supporting the Machine Intelligence SIMD. The matrix core has its own VGPR file: the Accumulation ("Acc") GPRs. This is separate from the normal (Architectural, or "Arch") VGPRs in the original SIMD. Shader I/O can only use both types of VGPRs. Instructions have an ACC bit to indicate if data is transferred to/from architectural or accumulation VGPRs. Data can be moved between the ACC and ARCH VGPRs via the V_ACCVGPR_READ and V_ACCVGPR_WRITE instructions.

The core operation implemented inside the matrix core is the 4×1 times 1×4 *outer* matrix product, yielding 16 output values. The outer product can be performed both on dense inputs and on 2,4 sparse ones (where two of each set of four values is zero). The matrix core unit uses combinations of these operations, both in parallel and in series, to implement the dense matrix-fused-multiply-add (*MFMA*) instructions described in Subsection Matrix fused-multiply-add (MFMA) and their 2,4-sparse variants described in Subsection Sparse Matrices.

Because these matrix instructions do not produce their output in a single cycle, and since their partially-written results may be observable, a certain amount of independent instructions must sometimes be present between the issuance of a matrix core instruction and accesses to its results or modification of the registers that hold its inputs, as described in Subsection Dependency Resolution: Required Independent Instructions.

> Additional information can be found on the GPUOpen blog: https://gpuopen.com/learn/amd-lab-notes/amd-lab-notes-matrix-cores-README/
> This blog post relates to CDNA2 but may be helpful with understanding CDNA3.
>
> The AMD Matrix Instruction Calculator (https://github.com/RadeonOpenCompute/amd_matrix_instruction_calculator) contains a helper tool that allows developers to view detailed information about the MFMA instructions in the CDNA architecture. It allows users to query instruction-level information such as computational throughput and register usage. It also allows users to generate mappings between matrix element and hardware registers for each MFMA instruction and their modifiers.

7.1. Matrix fused-multiply-add (MFMA)

The matrix fused-multiply-add (MFMA) instructions use the matrix core to perform one or more matrix multiplications. Note that the matrix core unit, which executes these instructions, has the 4×1 by 1×4 outer product as its fundamental computational primitive, and so the MFMA instructions implement outer-product-like operations.

These instructions all have names of the form `V_MFMA_[output type]_[M]X[N]X[K][_[B]B]_[input type]` where B (which is 1 if not specified) is the number of matrices (or *blocks*) that are multiplied, and M, N, and K, are the multiplication dimensions for each block. For example, the instruction V_MFMA_F32_32x32x1_2B_F32 perform the operations

```
D[0,:,:] = C[0,:,:] + A[0,:,:] * B[0,:,:]
D[1,:,:] = C[1,:,:] + A[1,:,:] * B[1,:,:]
```

where the `D[`b`,:,:]` and `C[`b`,:,:]` are 32×32 matrices of 32-bit floats, the `A[`b`,:,:]` are 32×1 matrices of floats,

and the `B[b,:,:]` are 1 × 32 matrices of floats.

The input and output values for an MFMA can be stored either in the standard architectural vector registers (VGPRs) or accumulation VGPRs (AccVGPRs), which are additional registers exclusive to the matrix core unit. The register file that the registers holding matrices A and B are controlled by the low and high bits, respectively, of the ACC field of a MFMA instruction (0 for VGPRs, 1 for AccVGPRs), while the ACC_CD bit determines if the C and D matrices are stored in VGPRs (0) or AccVGPRs (1). Data can be moved to and from AGPRs using the `V_ACCVGPR_*` instructions.

Note that the registers holding input or output data for a MFMA instruction must be contiguous, and that the first register must be aligned to the number of registers required as the input or output. For instance, if an instruction requires four input registers for matrix A, registers 4 through 7 may be used (by setting SRC0 to 4) but not registers 5 through 8.

7.1.1. Notation

When indexing values that have multiple blocks, `M[b,i,j]` is the value in block *b*, row *i*, and column *j* of the value *M*, where matrices are zero-indexed.

When describing the inputs and outputs of an MFMA operation, they are written as matrices with each column representing a different lane in a wavefront and each row representing a different register (or logical item) that a lane has.

For example, consider the following pair of matrices `A`

A[0,0,0]	A[0,0,1]
A[0,1,0]	A[0,1,1]
...	...
A[0,31,0]	A[0,31,1]
----	----
A[1,0,0]	A[0,0,1]
...	...
A[1,31,0]	A[1,31,1]

When the value is written as:

	Lane 0	Lane 1	...	Lane 31	Lane 32	...	Lane 63
Register 0	A[0,0,0]	A[0,1,0]	...	A[0,31,0]	A[1,0,0]	...	A[1,31,0]
Register 1	A[0,0,1]	A[0,1,1]	...	A[0,31,1]	A[1,0,1]	...	A[1,31,1]

this means that each lane of a wavefront holds two values across two contiguous registers, which are the two values of a row of one of the blocks of A, with the first 32 lanes holding a different row from block 0 and the second 32 lanes holding successive rows of block 1.

This specification writes matrices in their storage layout.

When showing register layouts, this spec assumes the first register is 0.

Unless otherwise specified, the division operator rounds down (takes the floor).

7.1.2. List of Dense MFMA instructions

Table 28. MFMA VALU Opcodes:

Instruction	Variants	Blocks	Cycles	Description
V_MFMA_F32_{*}_F32	32x32x1_2B	2	64	Matrix multiply, using FMA with F32 A & B matrices.
	16x16x1_4B	4	32	
	4x4x1_16B	16	8	
	32x32x2	1	64	
	16x16x4	1	32	
V_MFMA_F32_{*}_F16	32x32x4_2B	2	64	Matrix multiply, using FMA with F16 A & B matrices.
	16x16x4_4B	4	32	
	4x4x4_16B	16	8	
	32x32x8	1	32	
	16x16x16	1	16	
V_MFMA_F32_{*}_BF16	32x32x4_2B	2	64	Matrix multiply, using FMA with BF16 A & B matrices.
	16x16x4_4B	4	32	
	4x4x4_16B	16	8	
	32x32x8	1	32	
	16x16x16	1	16	
V_MFMA_I32_{*}_I8	32x32x4_2B	2	64	Matrix multiply, using FMA with I8 A & B matrices
	16x16x4_4B	4	32	
	4x4x4_16B	16	8	
	32x32x16	1	32	
	16x16x32	1	16	
V_MFMA_F32_{*}_XF32	16x16x8	1	16	Matrix Multiply on F32 data with reduced multiplication precision.
	32x32x4	1	32	
V_MFMA_F64_{*}_F64	16x16x4	1	32	Matrix Multiply on F64 data.
	4x4x4_4B	4	16	
V_MFMA_F32_{*}_BF8_BF8 V_MFMA_F32_{*}_BF8_FP8	16x16x32	1	16	Matrix Multiply on FP8 or BF8 data.
V_MFMA_F32_{*}_FP8_BF8 V_MFMA_F32_{*}_FP8_FP8	32x32x16	1	32	

Control	Behavior
Denorm Control	Ignores Denorm Control from MODE and keep Input/Output Denorms.
Clamp	Supported. uses the FP16_OVFL bit from MODE If set, F32 Result on overflow is clamped to +/- MAX, otherwise the overflow result is normalized to +/-INF. If set, I32 Result is clamped to +/-MAX on overflow/underflow, otherwise the carry out bits are dropped.
Round Mode	ignores Round Mode from MODE and forces it to RNE.
Exceptions	Not Supported
Execution Mask	ignores exec mask from MODE and forces it to 1 for all threads
Sources	Src0/1/2/VDST if VGPR need to be even aligned. Src0/1 can be only VGPR, SRC2 can be inline/constant

XF32

The XF32 instructions take 32-bit floats but round the mantissa to 10 bits in order to perform reduced-precision multiplication

7.1.3. Usage examples

7.1.3.1. V_MFMA_F32_32X32X1_2B_F32

The first examples show MFMA usage in order to build an intuition for the general semantics of these instructions.

Suppose the user wants do two matrix multiplications of 32 × 1 matrices `A[b, :, :]` by 1 × 32 matrices `B[b, :, :]`, accumulating the results into 32 × 32 matrices `D[b, :, :]`.

The input register for A stores columns of A across successive lanes (that is, the `i` coordinate is the fastest-moving) and has the form

	Lane 0	Lane 1	...	Lane 31	Lane 32	...	Lane 63
Register 0	A[0,0,0]	A[0,1,0]	...	A[0,31,0]	A[1,0,0]	...	A[1,31,0]

that is, lane `l` holds the value

```
A[l / 32, l % 32, 0]
```

The layout for B holds rows of B in the same way that the A layout stores its columns. That is, B is stored with lane `l` holding

```
B[l / 32, 0, l % 32]
```

	Lane 0	Lane 1	...	Lane 31	Lane 32	...	Lane 63
Register 0	B[0,0,0]	B[0,0,1]	...	B[0,0,31]	B[1,0,0]	...	B[1,0,31]

The core component of the output layout is the $4 \times N$ (where N is 32 here) tile of values. (The use of $4 \times N$ tiles, as opposed to a simpler layout, is a consequence of the matrix core's internal structure). As many of these tiles as possible (here 2 of them) are packed into the lanes of each group of registers, going by row and then by block.

That is, the layout of D (and the corresponding layout of C) is:

	Lane 0	Lane 1	...	Lane 31	Lane 32	...	Lane 63
Register 0	D[0,0,0]	D[0,0,1]	...	D[0,0,31]	D[0,4,0]	...	D[0,4,31]
Register 1	D[0,1,0]	D[0,1,1]	...	D[0,1,31]	D[0,5,0]	...	D[0,5,31]
...	...	...	...	...	...	...	...
Register 3	D[0,3,0]	D[0,3,1]	...	D[0,3,31]	D[0,7,0]	...	D[0,7,31]
Register 4	D[0,8,0]	D[0,8,1]	...	D[0,8,31]	D[0,12,0]	...	D[0,12,31]
...	...	...	...	...	...	...	...

	Lane 0	Lane 1	...	Lane 31	Lane 32	...	Lane 63
Register 15	D[0,27,0]	D[0,27,1]	...	D[0,27,31]	D[0,31,0]	...	D[0,31,31]
Register 16	D[1,0,0]	D[1,0,1]	...	D[1,0,31]	D[1,4,0]	...	D[1,4,31]
...	...	...	...	...	...	...	...
Register 31	D[1,27,0]	D[1,27,1]	...	D[1,27,31]	D[1,31,0]	...	D[1,31,31]

In other words, the output value D[b, i, j] is located in lane

```
l = j + 32 * ((i/4) % 2)
```

of output register

```
r = 16b + 4(i / 8) + (i % 4)
```

In order to produce these results, the broadcast fields (CBSZ, ABID, and BLGP) must all be set to 0. The usage of these fields is shown in Subsection Broadcasting values.

7.1.3.2. V_MFMA_F32_32X32X2_F32

As another example, consider the instruction V_MFMA_F32_32X32X2_F32. For this instruction, there is only one block being multiplied, and the matrices A[0,:,:] and B[0,:,:] are 32 × 2 and 2 × 32 respectively.

This instruction takes one input register for each of A and B, which the same format as above, except that lanes 32-63 contain the second column of A (row of B) instead of the second block. The output layout is the same as above, except that there is only one block and so there are only 16 output registers. More concretely, the input and output layouts for V_MFMA_F32_32X32X2_F32 are

	Lane 0	Lane 1	...	Lane 31	Lane 32	...	Lane 63
Register 0	A[0,0,0]	A[0,1,0]	...	A[0,31,0]	A[0,0,1]	...	A[0,31,1]

	Lane 0	Lane 1	...	Lane 31	Lane 32	...	Lane 63
Register 0	B[0,0,0]	B[0,0,1]	...	B[0,0,31]	B[0,1,0]	...	B[0,1,31]

	Lane 0	Lane 1	...	Lane 31	Lane 32	...	Lane 63
Register 0	D[0,0,0]	D[0,0,1]	...	D[0,0,31]	D[0,4,0]	...	D[0,4,31]
Register 1	D[0,1,0]	D[0,1,1]	...	D[0,1,31]	D[0,5,0]	...	D[0,5,31]
...	...	...	...	...	...	...	...
Register 3	D[0,3,0]	D[0,3,1]	...	D[0,3,31]	D[0,7,0]	...	D[0,7,31]
Register 4	D[0,8,0]	D[0,8,1]	...	D[0,8,31]	D[0,12,0]	...	D[0,12,31]
...	...	...	...	...	...	...	...
Register 15	D[0,27,0]	D[0,27,1]	...	D[0,27,31]	D[0,31,0]	...	D[0,31,31]

7.1.3.3. V_MFMA_F32_4X4X4_16B_F16

This example demonstrates how values that are not 32 bits long are packed into registers and how the output format changes in the case where an entire matrix cannot fill all lanes in an output register group.

The V_MFMA_F32_4X4X4_16B_F16 instruction performs 16 block multiplications of the form

```
D[b,:,:] = C[b,:,:] + A[b,:,:] * B[b,:,:]
```

where each block of A and B is a 4 × 4 block of half-precision floating point values and each block of C and D holds 4 × 4 single-precision floats.

The instruction uses 2 registers to hold each of A and B, even though, following the input format principles from the previous section, each lane needs to hold four values. This is because each input register holds two half-precision values, with the second of those values in the upper bits (16—31) of the register.

That is, the input layout of A is

	Lane 0	Lane 1	...	Lane 3	Lane 4	...	Lane 63
Register 0[15:0]	A[0,0,0]	A[0,1,0]	...	A[0,3,0]	A[1,0,0]	...	A[15,3,0]
Register 0[31:16]	A[0,0,1]	A[0,1,1]	...	A[0,3,1]	A[1,0,1]	...	A[15,3,1]
Register 1[15:0]	A[0,0,2]	A[0,1,2]	...	A[0,3,2]	A[1,0,2]	...	A[15,3,2]
Register 1[31:16]	A[0,0,3]	A[0,1,3]	...	A[0,3,3]	A[1,0,3]	...	A[15,3,3]

and for B is

	Lane 0	Lane 1	...	Lane 3	Lane 4	...	Lane 63
Register 0[15:0]	B[0,0,0]	B[0,0,1]	...	B[0,0,3]	B[1,0,0]	...	B[15,0,3]
Register 0[31:16]	B[0,1,0]	B[0,1,1]	...	B[0,1,3]	B[1,1,0]	...	B[15,1,3]
Register 1[15:0]	B[0,2,0]	B[0,2,1]	...	B[0,2,3]	B[1,2,0]	...	B[15,2,3]
Register 1[31:16]	B[0,3,0]	B[0,3,1]	...	B[0,3,3]	B[1,3,0]	...	B[15,3,3]

The 16 4 × 4 output blocks of this instruction are arranged into four output registers as follows.

	Lane 0	Lane 1	...	Lane 3	Lane 4	...	Lane 63
Register 0	D[0,0,0]	D[0,0,1]	...	D[0,0,3]	D[1,0,0]	...	D[15,0,3]
...	...	...	...	...	...	...	...
Register 3	D[0,3,0]	D[0,3,1]	...	D[0,3,3]	D[1,3,0]	...	D[15,3,3]

That is, because there are not enough groups of 4 rows available in a block to fill 64 lanes of output in each register, successive blocks are used instead. Note that these outputs are 32-bit floats and so are not packed into registers.

7.1.3.4. V_MFMA_F64_16X16X4_F64

This demonstrates how double-precision values are handled using the example of V_MFMA_F64_16X16X4_F64. This instruction follows the same input layout patterns as the previous examples and operates most similarly to V_MFMA_F32_32X32X2_F32. However, each input is spread across multiple registers in order to accommodate the full 64-bit value.

The output of this instruction, and the other double-precision MFMA instructions, does **not** follow the 4 × *N* block layout of other MFMA instructions. Instead, the output rows are packed contiguously across the lanes of each wavefront, and then packed into pairs (to account for the 64 bits needed to store the output) of registers, as shown below.

The input and output formats for V_MFMA_F64_16X16X4_F64 are

	Lane 0	Lane 1	...	Lane 15	Lane 16	...	Lane 63
Reg. 0	A[0,0,0][31:0]	A[0,1,0][31:0]	...	A[0,15,0][31:0]	A[0,0,1][31:0]	...	A[0,15,3][31:0]
Reg. 1	A[0,0,0][63:32]	A[0,1,0][63:32]	...	A[0,15,0][63:0]	A[0,0,1][63:32]	...	A[0,15,3][63:32]

	Lane 0	Lane 1	...	Lane 15	Lane 16	...	Lane 63
Reg. 0	B[0,0,0][31:0]	B[0,0,1][31:0]	...	B[0,0,15][31:0]	B[0,1,0][31:0]	...	B[0,3,15][31:0]
Reg. 1	B[0,0,0][63:32]	B[0,0,1][63:32]	...	B[0,0,15][63:32]	B[0,1,0][63:32]	...	B[0,3,15][63:32]

	Lane 0	Lane 1	...	Lane 15	Lane 16	...	Lane 63
Reg. 0	D[0,0,0][31:0]	D[0,0,1][31:0]	...	D[0,0,15][31:0]	D[0,1,0][31:0]	...	D[0,3,15][31:0]
Reg. 1	D[0,0,0][63:32]	D[0,0,1][63:32]	...	D[0,0,15][63:32]	D[0,1,0][63:32]	...	D[0,3,15][63:32]
Reg. 2	D[0,4,0][31:0]	D[0,4,1][31:0]	...	D[0,4,15][31:0]	D[0,5,0][31:0]	...	D[0,7,15][31:0]
...	...	...	...	...	...	...	...
Reg. 7	D[0,12,0][63:32]	D[0,12,1][63:32]	...	D[0,12,15][63:32]	D[0,13,0][63:32]	...	D[0,15,15][63:32]

7.1.4. General input and output layout

In general, an MFMA instruction is parameterized by its input and output datatypes, the sizes *M*, *N*, and *K* of each matrix block and the number of blocks it operates on *B*.

Semantically, for each `0 <= b < B`, `0 <= i < M`, and `0 <= j < N`, it computes

```
D[b,i,j] = C[b,i,j] + sum_{0 <= k < K} A[b,i,k] * B[b,k,j]
```

where each `A[b,:,:]` is $M \times K$, each `B[b,:,:]` is $K \times N$, and each `D[b,:,:]` and corresponding `C[b,:,:]` is $M \times N$.

The values of the inputs and outputs are placed into the arguments to the instruction according to a fixed layout. For simplicity, this layout is defined in terms of the lanes of a wavefront and of the sequence of *items* for each lane: these items are arranged into the 32-bit registers that are the true arguments to an MFMA instruction in little-endian form.

More specifically,

- For 64-bit quantities, each item corresponds to a *pair* of registers, with the low bits of the quantity in the first of those registers and high bits in the second one
- For 32-bit quantities, each item corresponds to a distinct register
- For 16-bit quantities, an item is half of a register, with the odd-numbered items taking up bits 31-16 and the even ones in bits 0-15
- For 8-bit quantities, four items are packed into a register, analogously to the 16-bit case

7.1.4.1. Input layout

To define the input layout for the matrix A, first define the auxiliary constant

```
K_L = K / (64 / (M * B))
```

which is the number of consecutive values of *K* that each lane holds in its registers.

For example, for the instruction V_MFMA_F32_32X32X1_2B_F32:

```
K_L = 1 / (64 / (32 * 2)) = 1 / 1 = 1
```

and for V_MFMA_F32_32X32X2_F32:

```
K_L = 2 / (64 / (32 * 1)) = 2 / 2 = 1
```

These both show that the one input register holds one value in the `K` dimension, but for V_MFMA_F32_4X4X4_16B_F16:

```
K_L = 4 / (64 / (4 * 16)) = 4 / 1 = 4
```

representing the fact that, for each lane, the two input registers hold four values in the *K* dimension. These input values are packed little-endian. For example, the third value in each row (which has `k = 2` zero-indexed), is in bits 15:0 of the second input register for both *A* and *B* across all lanes. That 16-bit region is "item 2" in the layout computed below for V_MFMA_F32_4X4X4_16B_F16.

Note that, in all MFMA instructions, the products `M * B` and `N * B` are less than 64, that is, the values of a single column of A or row of B, considered over all blocks, fit within a single input item.

With this layout defined, a given input value `A[b,i,k]` is placed in the item

```
k % K_L
```

of lane

```
i + M * (b + B * (k / K_L))
```

The layout for B is the analogous function that places `B[b,k,j]` in item

```
k % K_L
```

of lane

```
j + N * (b + B * (k / K_L))
```

7.1.4.2. Output layout

The output values `D[b,i,j]` of an MFMA instruction, as well as the corresponding values `C[b,i,j]` of the matrix to add to the result, are stored in a fixed layout that is a function of the MFMA instruction being used.

To define this layout, first define the following constants:

- `H`, the *group height*, which indicates how many consecutive rows of output are placed in each *row group* and which are therefore stored in consecutive items on a single lane. For **f64** instructions, `H = 1`, but for **all other MFMA instructions**, `H = 4`.
- `B_I = ceil(64 / (N * M / H))`, the number of blocks stored in each *output item* (an item within the storage for *D* or *C*) across all lanes
- `M_I = (64 / B_I) / N`, the number of rows of D stored in each output item across all lanes
- `G = M / (H * M_I)`, the number of row groups needed to store B_I blocks of output.

For example, using the instruction V_MFMA_F32_32X32X1_2B_F32 gives

```
H = 4
B_I = ceil(64 / (32 * 32 / H)) = ceil(64 / 256) = 1
M_I = (64 / 1) / 32 = 2
G = 32 / (4 * 2) = 4
```

while the instruction V_MFMA_F32_4X4X4_16B_F16 yields the values

```
H = 4
B_I = ceil(64 / (H * 4 / 4)) = ceil(64 / 4) = 16
M_I = (64 / 16) / 4 = 4 / 4 = 1
G = 4 / (H * 1) = 4 / 4 = 1
```

With these constants defined, the value D[b,i,j] of matrix D is located in item

```
(i % H) + H * (i/(H * M_I) + G * (b / B_I))
```

on lane

```
j + N * ((i / H) % M_I + M_I * (b % B_I))
```

7.1.5. Broadcasting values

While the operation of multiplying a 32 × 1 matrix of floats A by a 1 × 64 matrix B is not available natively, one can emulate this multiplication using the broadcast controls *Control Broadcast SiZe* (CBSZ), *A Block ID* (ABID), and *B Lane-Group Permutation* (BLGP).

These controls impact the retrieval of values from lanes: after the lane `l_a` in which a particular element of A would reside is computed, that value is permuted as defined by the CBSZ and ABID fields in order to determine

the lane that is accessed during the computation. Similarly, `l_b`, the lane to be used when retrieving any particular value of B, is permuted in the manner specified by the BLGP field.

7.1.5.1. CBSZ and ABID

Together, the CBSZ and ABID fields control the broadcasting of the blocks of matrix A.

When the 3-bit CBSZ field is non-zero, one block of lanes broadcasts the values it holds for matrix A to the other blocks of lanes, superseding the values those other lanes hold for the A matrix. Setting CBSZ such that (1 << CBSZ) exceeds the number of blocks the MFMA instruction processes is **undefined**.

The broadcast block size is

```
S = 64 / (1 << CBSZ)
```

For example, if CBSZ is 1, then one block of 32 lanes provides the inputs to both groups of 32 lanes in the wavefront, while CBSZ being 3 means that a the values from a block of 8 lanes are replicated.

The largest legal value of CBSZ is 4.

The 4-bit ABID field controls which block of S lanes is used as the broadcast source. The possible blocks are numbered in order, with lanes `S-1:0` being selected by ABID=0, `2S-1:S` corresponding to ABID=1, and so on. For example, if CBSZ=2, then ABID=1 means the values from lanes 16 to 31 are broadcast, to the three other blocks of lanes, while ABID=3 means that lanes 48 to 63 serve as the source of their inputs.

It is not legal to set ABID such that `ABID >= (1 << CBSZ)`, as such values do not refer to a potential source block.

Put differently, the CBSZ and ABID bits cause lane `l_a` to read thir inputs from the lane given by the permutation

```
p_a(l_a) = (l_a % S) + (S * ABID)
```

As a full example, if CBSZ=1 and ABID=1 when using the instruction V_MFMA_F32_32X32X1_2B_F32, both 1 × 32 blocks of B are multiplied by the values in the second 32 × 1 block of A, which is stored by the first 32 lanes. That is, the operation becomes:

```
D[b,i,j] = C[b,i,j] + A[1,i,0] * B[b,0,j]
```

which is a 32 × 1 by 1 × 64 matrix multiplication if the two blocks of B are treated as one matrix with 64 rows.

7.1.5.2. BLGP

The 3-bit BLGP field selects how the lane from which values in matrix B are read is permuted. Once it is determined that some value `B[b,k,j]` is in item `r` on lane `l_b`, using the defined input layout, the lane to be accessed `l_b` is permuted depending on the BLGP field as shown in Table Permutations corresponding to BLGP

values.

Table 29. Permutations corresponding to BLGP values

Value	Description	Expression
0	No broadcast	`l_b`
1	Broadcast first 32 lanes	`l_b % 32`
2	Broadcast second 32 lanes	`l_b % 32 + 32`
3	Rotate 16 lanes left	`(l_b + 16) % 64`
4	Broadcast first 16 lanes	`l_b % 16`
5	Broadcast second 16 lanes	`l_b %16 + 16`
6	Broadcast third 16 lanes	`l_b % 16 + 32`
7	Broadcast fourth 16 lanes	`l_b % 16 + 48`

7.1.5.3. Alternate meaning of broadcast fields for F64 instructions

The MFMA instructions that operate on double-precision floats (f64) do not support the broadcasting methods described above.

These instructions ignore CBSZ and ABID.

The BLGP field is repurposed for signaling the negation of the matrices A, B, and C.

- BLGP[0] causes values from matrix A to be implicitly negated if set
- BLGP[1] causes values from matrix B to be implicitly negated if set
- BLGP[2] causes values from matrix C to be implicitly negated if set

7.2. BF8 and FP8 Formats and Conversions

Table 30. Small Float Data Formats

Fmt	Sign-Exp-Mant	Bias	+0 -0	INF, -INF	NaN, -NaN	Max	Min (norm)	Min (denorm)
FP16	E5M10	15	0x0000 0x8000	0x7C00 0xFC00	(normal)	65504	6.10352E-05	5.96046E-08
FP8	E4M3	8	+/-: 0x00	0x80	0x80	448	+/-2.0^(-6)	+/-2.0^(-9)
BF8	E5M2	16	+/-: 0x00	0x80	0x80	57344	+/-2^(-14)	2.0^(-16)

Table 31. Small Float Data Format Conversion ops

Instruction	Dst	Src0	Src1	Encoding	Control	Notes
CVT_PK_FP8_F32	FP8	FP32	FP32	VOP3	Op_Sel[3] ignores: clamp, omod supports: neg, abs	RNE
CVT_PK_BF8_F32	BF8	FP32	FP32	VOP3	Op_Sel[3] ignores: clamp, omod supports: neg, abs	RNE
CVT_SR_FP8_F32	FP8	FP32	U32	VOP3	Op_Sel[3:2] ignores: clamp, omod supports: neg, abs	Stochastic Rounding

Instruction	Dst	Src0	Src1	Encoding	Control	Notes
CVT_SR_BF8_F32	BF8	FP32	U32	VOP3	Op_Sel[3:2] ignores: clamp, omod supports: neg, abs	Stochastic Rounding
CVT_PK_F32_FP8	F32	FP8	-	VOP1	SDWA, Op_Sel[0], dst,dst+1 ignores: abs, neg, sext	dst must be even
CVT_PK_F32_BF8	F32	BF8	-	VOP1	SDWA, Op_Sel[0], dst,dst+1 ignores: abs, neg, sext	dst must be even
CVT_F32_FP8	F32	FP8	-	VOP1	SDWA, Op_Sel ignores: abs, neg, sext	-
CVT_F32_BF8	F32	BF8	-	VOP1	SDWA, Op_Sel ignores: abs, neg, sext	-

Note: VOP3 instructions may not use SDWA.

In the above table, the CVT_*_F32 instructions do not support 4-cycle forwarding on these operations. The user must insert a NOP or instruction writing some other destination VREG between the conversions writing the low/high half or bytes of the same destination register.

Convert instructions come in two types:

- Packed – convert two 8-bit values into 32-bit values per instruction
- Stochastic Round – one source has the number to convert and the other has a random number used in rounding
  - These ops add a random value from the specified VGPR and then truncate to the smaller result data type

Converts from 8-bit formats and SDWA in VOP1:

- SRC0 must be set to “SDWA”, and the SRC0 VGPR is specified in the SDWA word as is the SRC0_SELECT which specifies which bytes to be converted. The other SDWA fields are ignored.

CVT_SR_FP8_F32 and CVT_SR_BF8_F32 OP_SEL usage:

```
Op_sel[3:2] == 00: dest_vgpr[31:0] = {prev_dst_vgpr[31:8], result[7:0]}
Op_sel[3:2] == 01: dest_vgpr[31:0] = {prev_dst_vgpr[31:16], result[7:0], prev_dst_vgpr[7:0]}
Op_sel[3:2] == 10: dest_vgpr[31:0] = {prev_dst_vgpr[31:24], result[7:0], prev_dst_vgpr[15:0]}
Op_sel[3:2] == 11: dest_vgpr[31:0] = {result[7:0], prev_dst_vgpr[23:0]}
```

CVT_SR_FP8_F32 OP_SEL usage:

> Uses Src0 and Src1 as inputs supplied by the VOP3 encoding, it adds the two operands with attention to not use msbs of src1 mantissa based on opcode and dependent on the F8 data type for the stochastic round before converting to F8 type. Then OP_SEL bits 3 and 2 are repurposed for this 8b write op and used to steer the resulting 8 bits into the correct byte lane of the 32b output preserving the remaining 24b of the destination.

CVT_*_FP8_F32 and CVT*_BF8_F32 FP16_OVFL rule

The FP16_OVFL flag is applied to data conversions from F32 to FP8/BF8 formats. The overflow behaviour is specified in the table below:

CVT_SR_* and CVT_PK_* support only VGPRs as inputs, not SGPRs, literal or inline constants.

Source Value	Destination Value			
	FP8		BF8	
	FP16_OVFL=1	FP16_OVFL=0	FP16_OVFL=1	FP16_OVFL=0
NaN	NaN	NaN	NaN	NaN
±Inf	±max_E4M3	NaN	±max_E5M2	±Inf
Greater than max FP8 magnitude	±max_E4M3	NaN	±max_E5M2	±Inf

The register SH_MEM_CONFIG, bit[8] must be set to 1 to produce the correct results for BF8 and FP8 operations.

7.3. Floating-point handling details and formats

The handling of denormal numbers varies depending on the datatypes the instruction takes and, in some cases, the MODE flags.

- V_MFMA_F32_*_F32 instructions, which take 32-bit inputs, honor the denormal-handling flags in MODE
- However, the V_MFMA_F32_*_XF32 instructions, which also take 32-bit inputs but only use 10 bits of mantissa, ignore MODE.denorm and do not flush denormals
- Matrix-C input and result-matrix output ignore MODE.denorm and do not flush denormals
- All instructions that take floats whose size is less than 32-bits (F16, BF16, BF8, FP8) ignore MODE.denorm and do not flush denormals
- The V_MFMA_F64_*_F64 instructions, which take 64-bit inputs and outputs ignores MODE and rounds to nearest even and allows denorms in the input and output
- The V_MFMA_I32_*_I8 perform integer multiply-add and thus do not respect the MODE bits. The 16-bit results of multiplying the I8 input values are sign-extended to 32 bits before multiplication, and the 16-bit result of the multiplication is sign extended to 32 bits prior to being added to the 32-bit result

The matrix core unit does not support arithmetic exceptions, except for DGEMM matrix operation which does support exceptions.

7.4. Sparse Matrices

The V_SMFMAC family of instructions perform matrix multiply-accumulate operations on a 4:2 structurally-sparse matrix A and dense matrices B, C, and D: D = C + A × B.

The A matrix is represented using 4:2 structured sparsity which means that two out of every 4 elements along the matrix K-dimension are zero. These zeros are not stored directly but instead are described in a separate VGPR which holds pairs of 2-bit index values. The index values indicate which two values out of each group of 4 are non-zero and are used to reconstruct full A-matrix. Non-zero samples are tightly packed resulting in 2:1 compression. Only the A-matrix may be sparse.

These SMFMAC instructions are all "accumulate" ops, where the C and D matrices are identical, referenced by the instruction’s VDST field (D-matrix). The C operand input is repurposed to hold the index data offset.

Table 32. SMFMA VALU Opcodes:

Instruction	Variants	Blocks	Cycles	Description
V_SMFMAC_F32_{*}_F16	16x16x32	1	16	Sparse Matrix multiply of F16 data
	32x32x16	1	32	
V_SMFMAC_F32_{*}_BF16	16x16x32	1	16	Sparse Matrix multiply of BF16 data
	32x32x16	1	32	
V_SMFMAC_I32_{*}_I8	16x16x64	1	16	Sparse Matrix multiply of I8 data
	32x32x32	1	32	
V_SMFMAC_F32_{*}_BF8_BF8 V_SMFMAC_F32_{*}_BF8_FP8	16x16x64	1	16	Sparse Matrix multiply of BF8 or FP8 data
V_SMFMAC_F32_{*}_FP8_BF8 V_SMFMAC_F32_{*}_FP8_FP8	32x32x32	1	32	

Matrix A is structurally sparse and occupies two VGPRs per lane at srcA offset. Matrix B is dense and occupies four VGPRs per lane at srcB offset. Matrix C shares VGPR offset with destination argument and occupies 16 VGPRs.

16-bit source data

If CBSZ[1:0] =0, ABID[1:0] selects one of four 8-bit sets of sparse-indices within a VGPR starting at srcC containing 8-bits of index information for a lane. If CBSZ[1:0] !=0; the very first is selected (VGPR[srcC][7..0]).

8-bit source data

If CBSZ[1:0] =0, ABID[0] selects one of two 16-bit sets of sparse-indices within a VGPR starting at srcC containing 16-bits of index information for a lane. If CBSZ[1:0] !=0; the very first is selected (VGPR[srcC][15..0]).

All SMFMAC instructions must follow these restrictions:

1. The Matrix A is sparse matrix and matrix B is the dense matrix. The ALU loads twice more data from VGPR for matrix B comparing with matrix A.
2. Matrix C is the same as the result Matrix. The ALU uses the VDST VGPR to load matrix C. All instructions are encoded as accumulation opcodes.
3. Src2 has the index encoded (all of the indexes are in one VGPR) and it can only be VGPR. Index Data provides information about which 2 out 4 SRCA are non-zero. For this index pair, index 0 < index 1 & index0 != index1.
4. The VGPR address of Src0, src1 and VDST must be even aligned.
5. CBSZ and ABID controls are ONLY used to pick the index from the VGPR read and don't affect SRCA matrix broadcast etc. as defined for other MFMA opcodes that use CBSZ and ABID controls.

SMFMAC instructions interpret the ACC_CD differently from other instructions: For SMFMAC the ACC_CD bit control only the DEST vgpr (arch vs accum), not the SRC2 location. The SRC2 argument provides the index data for sparse data supplied by the SRC0 argument which must reside in the ARCH-vgprs along with the A and B matrix data. In other words SRC2 acts as if ACC_CD==0.

7.4.1. Details of Sparsity Structure

Every index for the matrix B selection is a 2-bit number which identifies one of K=4 is selected. Depending on the matrix B layout, SRC2 may hold multiple sets of indices.

7.4.1.1. 16-bit A/B Matrix

When the A and B matrices consist of 16-bit data (FP16, BF16), the rules below apply.

Table 33. Matrix B Layout

16x16x32	Row0	Row1	Row2	Row3
v0	k=0,1	k=8,9	k=16,17	k=24,25
v1	k=2,3	k=10,11	k=18,19	k=26,27
v2	k=4,5	k=12,13	k=20,21	k=28,29
v3	k=6,7	k=14,15	k=22,23	k=30,31
32x32x16	Row0	Row1	Row2	Row3
v0	k=0,1	k=0,1	k=8,9	k=8,9
v1	k=2,3	k=2,3	k=10,11	k=10,11
v2	k=4,5	k=4,5	k=12,13	k=12,13
v3	k=6,7	k=6,7	k=14,15	k=14,15

Each lane has K=8 values which requires 4 indices per lane (8 bits), so each SRC2 VGPR holds 4 sets of indices.

Table 34. Index Layout

Lane ID	0	1	...	3	4	...	31	32	...	63
Vn[31:30]	set3, V1[31:16]									
Vn[29:28]	set3, V1[15:0]									
Vn[27:26]	set3, V0[31:16]									
Vn[25:24]	set3, V0[15:0]									
...	...									
Vn[9:8]	set1, V0[15:0]									
Vn[7:6]	set0, V1[31:16]									
Vn[5:4]	set0, V1[15:0]									
Vn[3:2]	set0, V0[31:16]									
Vn[1:0]	set0, V0[15:0]									

"Vn" is the SRC-C VGPR holding the index values.

7.4.1.2. 8-bit A/B Matrix

When the A and B matrices consist of 8-bit data (I8, FP8, BF8), the rules below apply.

Table 35. Matrix B Layout

16x16x64	Row0	Row1	Row2	Row3
v0	k=0,1,2,3	k=16,17,18,19	k=32,33,34,35	k=48,49,50,51
v1	k=4,5,6,7	k=20,21,22,23	k=36,37,38,39	k=52,53,54,55
v2	k=8,9.10,11	k=24,25,26,27	k=40,41,42,43	k=56,57,58,59
v3	k=12,13,14,15	k=28,29,30,31	k=44,45,46,47	k=60,61,62,63
32x32x32	Row0	Row1	Row2	Row3
v0	k=0,1,2,3	k=0,1,2,3	k=16,17,18,19	k=16,17,18,19
v1	k=4,5,6,7	k=4,5,6,7	k=20,21,23,23	k=20,21,23,23
v2	k=8,9,10,11	k=8,9,10,11	k=24,25,26,27	k=24,25,26,27

16x16x64	Row0	Row1	Row2	Row3
v3	k=12,13,14,15	k=12,13,14,15	k=28,29,30,31	k=28,29,30,31

Each lane has K=16 values which requires 8 indices per lane (16 bits), so each SRC2 VGPR holds 2 sets of indices.

Table 36. Index Layout

Lane ID	0	1	...	3	4	...	31	32	...	63
Vn[31:30]	set1, V1[31:24]									
Vn[29:28]	set1, V1[23:16]									
Vn[27:26]	set1, V1[15:8]									
Vn[25:24]	set1, V1[7:0]									
Vn[23:22]	set1, V0[31:24]									
Vn[21:20]	set1, V0[23:16]									
Vn[19:19]	set1, V0[15:8]									
Vn[17:16]	set1, V0[7:0]									
Vn[15:14]	set0, V1[31:24]									
...	...									
Vn[3:2]	set0, V0[15:8]									
Vn[1:0]	set0, V0[7:0]									

7.5. Dependency Resolution: Required Independent Instructions

The table below indicates timing conditions which require the user to insert NOPs (or independent VALU instructions).

DLop Dot products

XDLOP Matrix math on {I8, F16, BF16, F32}

DGEMM V_MFMA...F64

PASS 4 clock cycles

Table 37. VOP3P-Matrix Opcodes Required NOPs

First Instruction	Second Instruction	Required Waits	Comments
Non-DLops VALU Write VGPR	V_MFMA* read VGPR OR V_SMFMA* read VGPR	2	No internal 4 & 8 cycle forwarding path.
DL ops Write VGPR	DLops read VGPR as SrcC, and the opcode is exactly the same as 1st DLops	0	supports same opcode of DLops back-to-back SrcC forwarding which is used for accumulation.
	DLops read VGPR as SrcA/B, and the opcode is exactly the same as 1st DLops	3	does not support SrcA/B forwarding in DLops
	Any opcode read/write VGPR that is not the same as 1st DLops opcode (RAW + WAW)	3	Disable all of the forwarding path from DL ops to normal VALU/VM/LDS/FLAT ops

First Instruction	Second Instruction	Required Waits	Comments
XDL Write VGPR or V_SMFMA* Write VGPR	XDL read VGPR as Source C exactly same with 1st vDst OR V_SMFMA* read VGPR for Matrix C exactly same with 1st vDst	2 if 1st V_MFMA is 2 passes	the two V_MFMA must be the same number passes and vDst and vSrc start from the same offset and same VGPR size. V_MFMA & V_SMFMA must be the same number passes and both vDst start from the same offset and same VGPR size. Note: V_SMFMA reads vdst for Matrix C.
		0 if 1st V_MFMA is 4 passes	
		0 if 1st V_MFMA is 8 passes	
		0 if 1st V_MFMA is 16 passes	
XDL Write VGPR or V_SMFMA* Write VGPR	XDL read VGPR as Source C overlapped with 1st vDst OR V_SMFMA* read VGPR for Matrix C overlapped with 1st vDst	3 if 1st V_MFMA is 2 passes	overlapped with XDL Note: V_SMFMA reads vdst for Matrix C.
		5 if 1st V_MFMA is 4 passes	
		9 if 1st V_MFMA is 8 passes	
		17 if 1st V_MFMA is 16 passes	
XDL Write VGPR or V_SMFMA* Write VGPR	S/DGEMM read VGPR as Source C	3 if 1st V_MFMA is 2 passes	Overlapped with S/DGEMM
		5 if 1st V_MFMA is 4 passes	
		9 if 1st V_MFMA is 8 passes	
		17 if 1st V_MFMA is 16 passes	
XDL Write VGPR or V_SMFMA* Write VGPR	V_MFMA read VGPR as SrcA or SrcB OR V_SMFMA* read VGPR as SrcA or SrcB or Index SrcC	5 if 1st V_MFMA is 2 passes	No internal forwarding path waits for previous V_MFMA/V_SMFMA* commit result to VGPR V_SMFMA uses srcC address for extra Index C Reads
		7 if 1st V_MFMA is 4 passes	
		11 if 1st V_MFMA is 8 passes	
		19 if 1st V_MFMA is 16 passes	
XDL Write VGPR or V_SMFMA* Write VGPR	1) VM, L/GDS, FLAT, Export Read VGPR overlapped with 1st vDst 2) VALU read/write VGPR (RAW + WAW)	5 if 1st V_MFMA is 2 passes	V_MFMA_F32_4X4X4F16
		7 if 1st V_MFMA is 4 passes	V_MFMA_F32_16X16X16F16
		11 if 1st V_MFMA is 8 passes	V_MFMA_F32_32X32X8F16
		19 if 1st V_MFMA is 16 passes	V_MFMA_F32_32X32X4F16
SGEMM Write VGPR	XDL read VGPR as Source C exactly same with 1st vDst OR V_SMFMA* read VGPR for Matrix C exactly same with 1st vDst	0	the two V_MFMA must be the same number passes and vDst and vSrc start from the same offset and same VGPR size. V_MFMA & V_SMFMA must be the same number passes and both vDst start from the same offset and same VGPR size. Note: V_SMFMA reads vdst for Matrix C.

First Instruction	Second Instruction	Required Waits	Comments
SGEMM Write VGPR	XDL read VGPR as Source C overlapped with 1st vDst OR V_SMFMA* read VGPR for Matrix C overlapped with 1st vDst	2 if 1st V_MFMA is 2 passes	overlapped with XDL Note: V_SMFMA reads vdst for Matrix C.
		4 if 1st V_MFMA is 4 passes	
		8 if 1st V_MFMA is 8 passes	
		16 if 1st V_MFMA is 16 passes	
SGEMM Write VGPR	S/DGEMM read VGPR as Source C	2 if 1st V_MFMA is 2 passes	Overlapped with S/DGEMM
		4 if 1st V_MFMA is 4 passes	
		8 if 1st V_MFMA is 8 passes	
		16 if 1st V_MFMA is 16 passes	
SGEMM Write VGPR	V_MFMA read VGPR as SrcA or SrcB OR V_SMFMA* read VGPR as SrcA or SrcB or Index SrcC	4 if 1st V_MFMA is 2 passes	No internal forwarding path, need to wait previous V_MFMA commit result to VGPR. Note: V_SMFMA used SrcC for Index reads.
		6 if 1st V_MFMA is 4 passes	
		10 if 1st V_MFMA is 8 passes	
		18 if 1st V_MFMA is 16 passes	
SGEMM Write VGPR	1) VM, L/GDS, FLAT, Export Read VGPR overlapped with 1st vDst 2) VALU read/write VGPR (RAW + WAW)	4 if 1st V_MFMA is 2 passes	V_MFMA_F32_4X4X4F16
		6 if 1st V_MFMA is 4 passes	V_MFMA_F32_16X16X16F16
		10 if 1st V_MFMA is 8 passes	V_MFMA_F32_32X32X8F16
		18 if 1st V_MFMA is 16 passes	V_MFMA_F32_32X32X4F16

First Instruction	Second Instruction	Required Waits	Comments
V_MFMA_16x16x4_F64 Write VGPR	V_MFMA_16x16x4_F64 read VGPR as Source C exactly same with 1st vDst	0	the two V_MFMA must be the same number passes and vDst and vSrc start from the same offset.
	S/DGEMM read VGPR as Source C overlapped with 1st vDst	9	overlapped, different VGPR access sequence
	XDL read VGPR as Source C overlapped with 1st vDst	0	
	V_SMFMA* read VGPR for Matrix C overlapped with 1st vDst	0	V_SMFMA reads vdst for Matrix C.
	S/DGEMM read VGPR as SrcA or SrcB	11	No internal forwarding path, need to wait previous V_MFMA commit result to VGPR
	XDL read VGPR as SrcA or SrcB	11	
	V_SMFMA* read VGPR as SrcA or SrcB or Index SrcC	11	V_SMFMA uses srcC address for extra Index C Reads
	VALU read/write VGPR (RAW + WAW)	11	
	VM, L/GDS, FLAT and Export Read VGPR overlapped with 1st vDst	18	No internal forwarding path, need to wait previous V_MFMA commit result to VGPR
V_MFMA_4x4x4_F64 Write VGPR	V_MFMA_4x4x4_F64, read VGPR as Source C exactly same with 1st vDst	4	4x4x4 needs to do accumulation, so the write VGPR is later than normal XDL 4x4x4, so needs extra wait
	S/DGEMM read VGPR as Source C overlapped with 1st vDst	4	overlapped, different VGPR access sequence
	XDL read VGPR as Source C overlapped with 1st vDst	0	
	V_SMFMA read VGPR for Matrix C overlapped with 1st vDst	0	V_SMFMA reads vdst for Matrix C.
	S/DGEMM read VGPR as SrcA or SrcB	6	No internal forwarding path, need to wait previous V_MFMA commit result to VGPR
	XDL read VGPR as SrcA or SrcB	6	
	V_SMFMA* read VGPR as SrcA or SrcB or Index SrcC	6	V_SMFMA uses srcC address for extra Index C Reads
	VALU read/write VGPR (RAW + WAW)	6	
	VM, L/GDS, FLAT and Export Read VGPR overlapped with 1st vDst	9	No internal forwarding path, need to wait previous V_MFMA commit result to VGPR
V_CMPX* write EXEC MASK	V_MFMA*	4	CDNA doesn't support execution mask forwarding with XDL/DGEMM
XDL/SMFMA Read VGPR SrcC	VALU write VGPR (WAR), Co-Execution Anti-Dependency for over-lapping with 1st SrcC	1	if 1st MFMA is 2 passes
		3	if 1st MFMA is 4 passes
		7	if 1st MFMA is 8 passes
		15	if 1st MFMA is 16 passes

Chapter 8. Scalar Memory Operations

Scalar Memory Read (SMEM) instructions allow a shader program to load data from memory into SGPRs through the Scalar Data Cache, or write data from SGPRs to memory through the Scalar Data Cache. Instructions can read from 1 to 16 Dwords, or write 1 to 4 Dwords at a time. Data is read directly into SGPRs without any format conversion.

The scalar unit reads and writes consecutive Dwords between memory and 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.

8.1. Microcode Encoding

Scalar memory read, write and atomic instructions are encoded using the SMEM microcode format.

The fields are described in the table below:

Table 38. SMEM Encoding Field Descriptions

Field	Size	Description
OP	8	Opcode.
IMM	1	Determines how the OFFSET field is interpreted. IMM=1 : Offset is a 21-bit unsigned byte offset to the address. IMM=0 : Offset[6:0] specifies an SGPR or M0 which provides an unsigned byte offset (for stores, must be M0). STORE and ATOMIC instructions cannot use an SGPR: only imm or M0.
GLC	1	Globally Coherent. For loads, controls L1 cache policy: 0=hit_lru, 1=miss_evict. For stores, controls L1 cache bypass: 0=write-combine, 1=write-thru. For atomics, "1" indicates that the atomic returns the pre-op value.
SDATA	7	SGPRs to return read data to, or to source write-data from. Reads of two Dwords must have an even SDST-sgpr. Reads of four or more Dwords must have their DST-gpr aligned to a multiple of 4. SDATA must be: SGPR or VCC. Not: exec or m0.
SBASE	6	SGPR-pair (SBASE has an implied LSB of zero) which provides a base address, or for BUFFER instructions, a set of 4 SGPRs (4-sgpr aligned) which hold the resource constant. For BUFFER instructions, the only resource fields used are: base, stride, num_records.
OFFSET	21	An unsigned byte offset, or the address of an SGPR holding the offset. Writes and atomics: M0 or immediate only, not SGPR.
NV	1	Non-volatile.
SOE	1	Scalar Offset Enable.

See Memory Scope and Temporal Control for more information on the GLC bit.

8.2. Operations

8.2.1. S_LOAD_DWORD, S_STORE_DWORD

These instructions load 1-16 Dwords or store 1-4 Dwords between SGPRs and memory. The data in SGPRs is specified in SDATA, and the address is composed of the SBASE, OFFSET, and SOFFSET fields.

8.2.1.1. Scalar Memory Addressing

S_LOAD / S_STORE / S_DCACHE_DISCARD:

```
ADDR = SGPR[base] + inst_offset + { M0 or SGPR[offset] or zero }
```

S_SCRATCH_LOAD / S_SCRATCH_STORE:

```
ADDR = SGPR[base] + inst_offset + { M0 or SGPR[offset] or zero } * 64
```

Use of offset fields:

IMM	SOFFSET_EN (SOE)	Address
0	0	SGPR[base] + (SGPR[offset] or M0)
0	1	SGPR[base] + (SGPR[soffset] or M0)
1	0	SGPR[base] + inst_offset
1	1	SGPR[base] + inst_offset + (SGPR[soffset] or M0)

All components of the address (base, offset, inst_offset, M0) are in bytes, but the two LSBs are ignored and treated as if they were zero. S_DCACHE_DISCARD ignores the six LSBs to make the address 64-byte-aligned.

It is illegal and undefined if the inst_offset is negative and the resulting (inst_offset + (M0 or SGPR[offset])) is negative.

Scalar access to private space must either use a buffer constant or manually convert the address:

```
Addr = Addr - private_base + private_base_addr + scratch_baseOffset_for_this_wave
```

"Hidden private base" is not available to the shader through hardware: It must be preloaded into an SGPR or made available through a constant buffer. This is equivalent to what the driver must do to calculate the base address from scratch for buffer constants.

A scalar instruction must not overwrite its own source registers because the possibility of the instruction being replayed due to an ATC XNACK. Similarly, instructions in scalar memory clauses must not overwrite the sources of any of the instructions in the clause. A clause is defined as a string of memory instructions of the same type. A clause is broken by any non-memory instruction. *One exception to this rule is a single SMEM instruction in a clause by itself which loads a single DWORD may legally overwrite its own source SGPRs. (This*

instruction either completely succeeds to execute and continue, or completely fail; it does not overwrite just part of one DWORD).

Atomics are unusual because they are naturally aligned and they must be in a single-instruction clause. By definition, an atomic that returns the pre-op value overwrites its data source, which is acceptable.

Reads/Writes/Atomics using Buffer Constant

Buffer constant fields used: base_address, stride, num_records, NV. Other fields are ignored.

Scalar memory read/write does not support "swizzled" buffers. Stride is used only for memory address bounds checking, not for computing the address to access.

The SMEM supplies only a SBASE address (byte) and an offset (byte or Dword). Any "index * stride" must be calculated manually in shader code and added to the offset prior to the SMEM.

The two LSBs of V#.base and of the final address are ignored to force Dword alignment.

```
"m_*" components come from the buffer constant (V#):
  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] + offset) & ~0x3
  SGPR[SDST] = read_Dword_from_dcache(m_base, offset, m_size)

  If more than 1 dword is being read, it is returned to SDST+1, SDST+2, etc,
  and the offset is incremented by 4 bytes per DWORD.
```

8.2.2. Scalar Atomic Operations

The scalar memory unit supports the same set of memory atomics as the vector memory unit. Addressing is the same as for scalar memory loads and stores. Like the vector memory atomics, scalar atomic operations can return the "pre-operation value" to the SDATA SGPRs. This is enabled by setting the microcode GLC bit to 1.

8.2.3. S_DCACHE_INV, S_DCACHE_WB

This instruction invalidates, or does a "write back" of dirty data, for the entire scalar data cache. It does not return anything to SDST.

8.2.4. S_MEMTIME

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

8.2.5. S_MEMREALTIME

This instruction reads a 64-bit "real time-counter" and returns the value into a pair of SGPRS: SDST and SDST+1. The time value is from a constant 100MHz clock (not affected by power modes or core clock frequency changes).

8.3. Dependency Checking

Scalar memory reads and writes 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 SMEMs before continuing.

8.4. 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 9. 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 these types of VMEM instructions:

- MTBUF: Memory typed-buffer operations.
- MUBUF: Memory untyped-buffer operations.

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#).

9.1. 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 homogeneous and heterogeneous data, but no filtering of read-data. 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 (V#) 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.

9.1.1. Simplified Buffer Addressing

The equation below shows how the hardware calculates the memory address for a buffer access.

```
ADDR =   Base    +  baseOffset   +  Inst_offset  +  Voffset   +  Stride  * (Vindex  +  TID)
           V#          SGPR           Instr          VGPR          V#        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

9.1.2. 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.

The D16 instruction variants convert the results to packed 16-bit values. For example, BUFFER_LOAD_FORMAT_D16_XYZW writes two VGPRs.

Table 39. 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. Globally coherent. Operates on 32-bit or 64-bit values (x2 = 64 bits).

Table 40. Microcode Formats

Field	Bit Size	Description
OP	4 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.

Field	Bit Size	Description
DFMT	4	Data Format of data in memory buffer: 0 invalid 1 8 2 16 3 8_8 4 32 5 16_16 6 10_11_11 7 11_11_10 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
NFMT	3	Numeric format of data in memory: 0 unorm 1 snorm 2 uscaled 3 sscaled 4 uint 5 sint 6 reserved 7 float
OFFSET	12	Unsigned byte offset.
OFFEN	1	1 = Supply an offset from VGPR (VADDR). 0 = Do not (offset = 0).
IDXEN	1	1 = Supply an index from VGPR (VADDR). 0 = Do not (index = 0).
SC0	1	Scope bit 0
NT	1	Non-Temporal
ACC	1	VDATA is Accumulation VGPR
SC1	1	Scope bit 1
LDS	1	MUBUF-ONLY: 0 = Return read-data to VGPRs. 1 = Return read-data to LDS instead of VGPRs.

9.1.3. 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 the table below:

Table 41. 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 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.

9.1.4. 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 42. 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.

The resource's data format must not be INVALID; that format has specific 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.

D16 Instructions : Load-format and store-format instructions also come in a "d16" variant. For stores, each 32-bit VGPR holds two 16-bit data elements that are passed to the texture unit. This texture unit converts them to the texture format before writing to memory. For loads, data returned from the texture unit is converted to 16

bits, and a pair of data are stored in each 32-bit VGPR (LSBs first, then MSBs). Control over int vs. float is controlled by NFMT.

9.1.5. Buffer Addressing

A **buffer** is a data structure in memory that is addressed with an **index** and an **offset**. The index points to a particular record of size **stride** bytes, and the offset is the byte-offset within the record. The **stride** comes from the resource, the index from a VGPR (or zero), and the offset from an SGPR or VGPR and also from the instruction itself.

Table 43. BUFFER Instruction Fields for Addressing

Field	Size	Description
inst_offset	12	Literal byte offset from the instruction.
inst_idxen	1	Boolean: get index from VGPR when true, or no index when false.
inst_offen	1	Boolean: get offset from VGPR when true, or no offset when false. Note that inst_offset is present, regardless of this bit.

The "element size" for a buffer instruction is the amount of data the instruction transfers. It is determined by the DFMT field for MTBUF instructions, or from the opcode for MUBUF instructions. It can be 1, 2, 4, 8, or 16 bytes.

Table 44. V# Buffer Resource Constant Fields for Addressing

Field	Size	Description
const_base	48	Base address, in bytes, of the buffer resource.

const_stride	14 or 18	Stride of the record in bytes (0 to 16,383 bytes, or 0 to 262,143 bytes). Normally 14 bits, but is extended to 18-bits when:
		const_add_tid_enable = true used with MUBUF instructions which are not **format** types (or cache invalidate/WB).
		This is extension intended for use with scratch (private) buffers.
		
		`If (const_add_tid_enable && MUBUF-non-format instr.)`
		`  const_stride [17:0] = { V#.DFMT[3:0],`
		`  V#.const_stride[13:0] }`
		`else`
		`  const_stride is 14 bits: {4'b0, V#.const_stride[13:0]}`

const_num_records	32	Number of records in the buffer. In units of Bytes for raw buffers, units of Stride for structured buffers, and ignored for private (scratch) buffers. In units of: (inst_idxen == 1) ? Bytes : Stride
const_add_tid_enable	1	Boolean. Add thread_ID within the wavefront to the index when true.
const_swizzle_enable	1	Boolean. Indicates that the surface is swizzled when true.
const_element_size	2	Used only when const_swizzle_en = true. Number of contiguous bytes of a record for a given index (2, 4, 8, or 16 bytes). Must be >= the maximum element size in the structure. const_stride must be an integer multiple of const_element_size.
const_index_stride	2	Used only when const_swizzle_en = true. Number of contiguous indices for a single element (of const_element_size) before switching to the next element. There are 8, 16, 32, or 64 indices.

Table 45. Address Components from GPRs

Field	Size	Description
SGPR_offset	32	An unsigned byte-offset to the address. Comes from an SGPR or M0.
VGPR_offset	32	An optional unsigned byte-offset. It is per-thread, and comes from a VGPR.
VGPR_index	32	An optional index value. It is per-thread and comes from a VGPR.

The final buffer memory address is composed of three parts:

- the base address from the buffer resource (V#),
- the offset from the SGPR, and
- a buffer-offset that is calculated differently, depending on whether the buffer is linearly addressed (a simple Array-of-Structures calculation) or is swizzled.

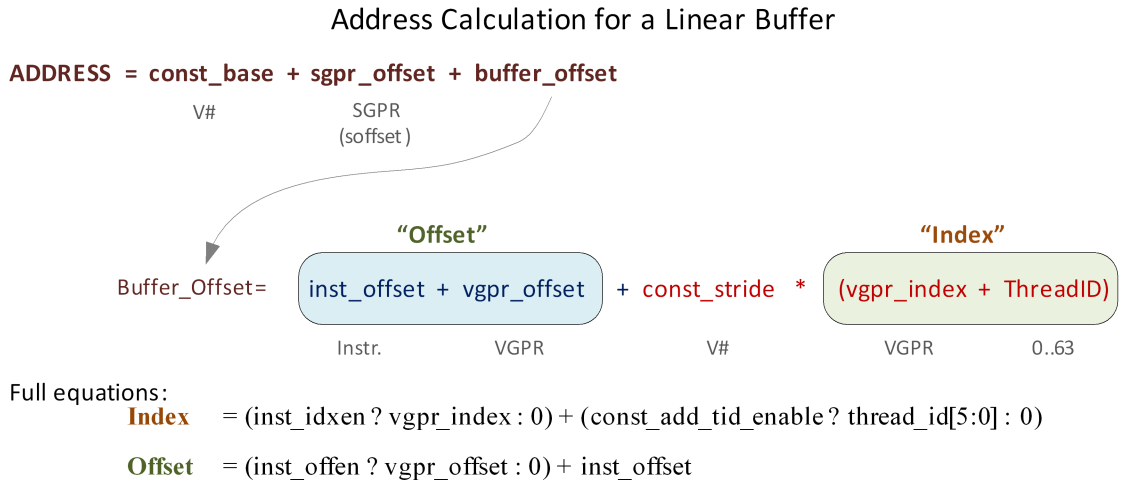


Figure 4. Address Calculation for a Linear Buffer

9.1.5.1. Range Checking

Addresses can be checked to see if they are in or out of range. When an address is out of range, reads return zero, and writes and atomics are dropped. The address range check method depends on the buffer type.

Private (Scratch) Buffer

Used when: AddTID==1 && IdxEn==0
For this buffer, there is no range checking.

Raw Buffer

Used when: AddTID==0 && SWizzleEn==0 && IdxEn==0
Out of Range if: (InstOffset + (OffEN ? vgpr_offset : 0)) >= NumRecords

Structured Buffer

Used when: AddTID==0 && Stride!=0 && IdxEn==1
Out of Range if: Index(vgpr) >= NumRecords

Notes:

1. Reads that go out-of-range return zero (except for components with V#.dst_sel = SEL_1 that return 1).

2. Writes that are out-of-range do not write anything.
3. Load/store-format-* instruction and atomics are range-checked "all or nothing" - either entirely in or out.
4. Load/store-Dword-x{2,3,4} and range-check per component.

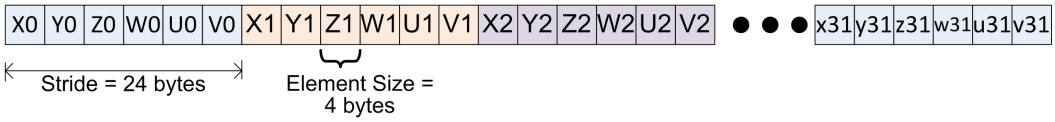
9.1.5.2. Swizzled Buffer Addressing

Swizzled addressing rearranges the data in the buffer to help provide improved cache locality for arrays of structures. Swizzled addressing also requires Dword-aligned accesses. A single fetch instruction cannot attempt to fetch a unit larger than const-element-size. The buffer's STRIDE must be a multiple of element_size.

```
Index = (inst_idxen ? vgpr_index : 0) +
        (const_add_tid_enable ? thread_id[5:0] : 0)

Offset = (inst_offen ? vgpr_offset : 0) + inst_offset

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

Final Address = const_base + sgpr_offset + buffer_offset
```

Remember that the "sgpr_offset" is not a part of the "offset" term in the above equations.

Original Buffer



Swizzled Buffer

const_index_stride = 8 // how many consecutive indices to group together
const_element_size = 4 bytes // the size of a single element, in bytes

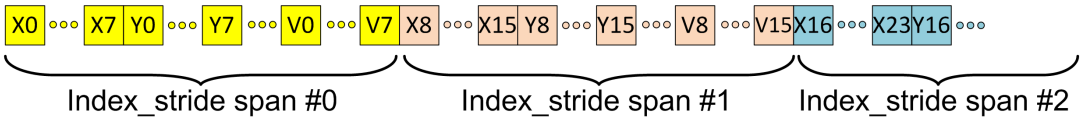
index_msb = index / const_index_stride
index_lsb = index % const_index_stride
offset_msb = offset / const_element_size
offset_lsb = offset % const_element_size

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

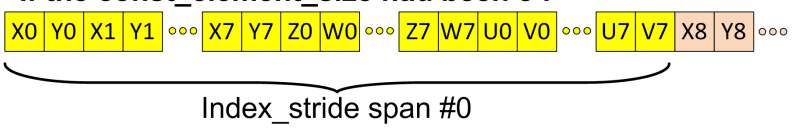
which simplifies to...

Buffer_offset = (index/8 * const_stride + (offset/4)*4) * 8 + index%8 * 4 + offset%4

Note that because we are dealing with dwords, offset%4 is always == 0.



If the const_element_size had been 8 :



An alternate way to visualize Swizzled Buffers

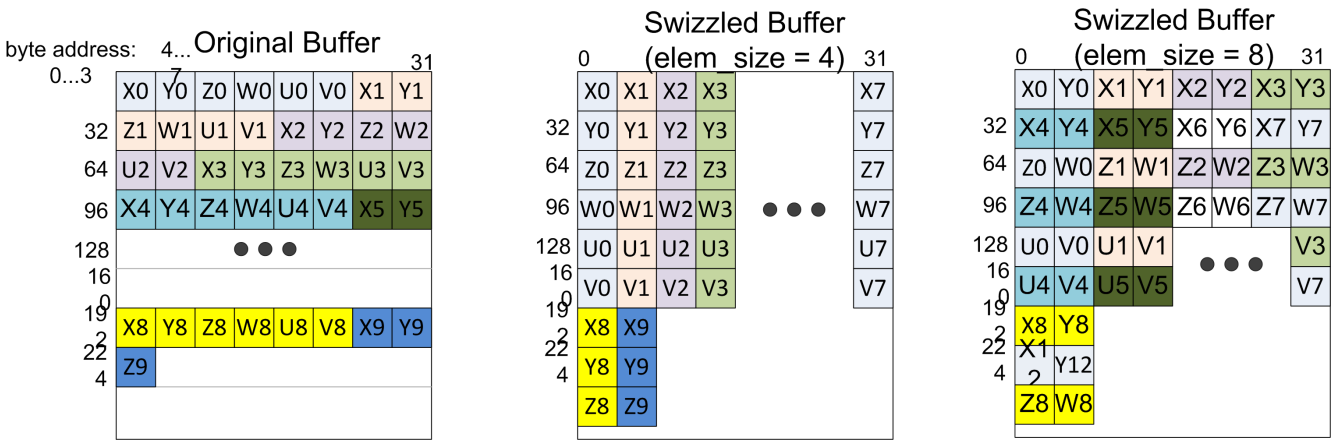


Figure 5. Example of Buffer Swizzling

9.1.5.3. Proposed Use Cases for Swizzled Addressing

Here are few proposed uses of swizzled addressing in common graphics buffers.

Table 46. Swizzled Buffer Use Cases

	DX11 Raw Uav OpenCL Buffer Object	Dx11 Structured (literal offset)	Dx11 Structured (gpr offset)	Scratch	Ring / stream-out	Const Buffer
inst_vgpr_offset_en	T	F	T	T	T	T
inst_vgpr_index_en	F	T	T	F	F	F
const_stride	na	<api>	<api>	scratchSize	na	na
const_add_tid_enable	F	F	F	T	T	F
const_buffer_swizzle	F	T	T	T	F	F
const_elem_size	na	4	4	4 or 16	na	4
const_index_stride	na	16	16	64		

9.1.6. 16-bit Memory Operations

The D16 buffer instructions allow a kernel to load or store just 16 bits per work item between VGPRs and memory. There are two variants of these instructions:

- D16 loads data into or stores data from the lower 16 bits of a VGPR.
- D16_HI loads data into or stores data from the upper 16 bits of a VGPR.

For example, BUFFER_LOAD_UBYTE_D16 reads a byte per work-item from memory, converts it to a 16-bit integer, then loads it into the lower 16 bits of the data VGPR.

When ECC is enabled 16-bit memory loads write the full 32-bit VGPR. Unused bits are set to zero.

9.1.7. Alignment

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

9.1.8. 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.

The table below details the fields that make up the buffer resource descriptor.

Table 47. Buffer Resource Descriptor

Bits	Size	Name	Description
47:0	48	Base address	Byte address.
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 or bytes.
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. For MUBUF instructions with ADD_TID_EN = 1. This field holds Stride [17:14].
115	1	User VM Enable	Resource is mapped via tiled pool / heap.
116	1	User VM mode	Unmapped behavior: 0: null (return 0 / drop write); 1:invalid (results in error)
118:117	2	Index stride	8, 16, 32, or 64. Used for swizzled buffer addressing.
119	1	Add tid enable	Add thread ID to the index for to calculate the address.
122:120	3	RSVD	Reserved. Must be set to zero.
123	1	NV	Non-volatile (0=volatile)
125:124	2	RSVD	Reserved. Must be set to zero.
127:126	2	Type	Value == 0 for buffer. Overlaps upper two bits of four-bit TYPE field in 128-bit T# resource.

A resource set to all zeros acts as an unbound texture or buffer (return 0,0,0,0).

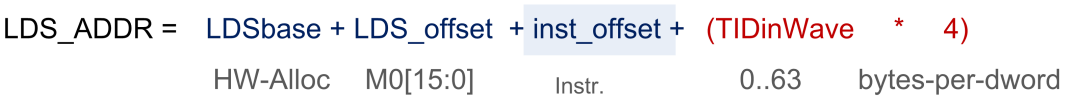
9.1.9. Memory Buffer Load to LDS

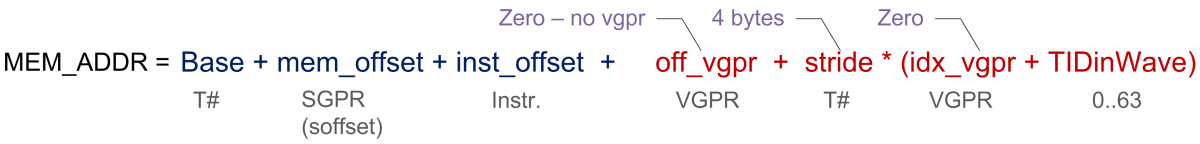
The MUBUF instruction format allows reading data from a memory buffer directly into LDS without passing through VGPRs. This is supported for the following subset of MUBUF instructions.

- BUFFER_LOAD_{ubyte, sbyte, ushort, sshort, dword, format_x}.

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.)

The figure below shows the components of the LDS and memory address calculation:





TIDinWave is only added if the resource (T#) has the ADD_TID_ENABLE field set to 1, whereas LDS adds it. The MEM_ADDR M0 is in the VDATA field; it specifies M0.

9.1.9.1. 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.

9.1.10. Memory Scope and Temporal Control

9.1.10.1. Scalar Memory

Scalar Memory instructions have a single memory control bit: GLC (Globally Coherent). The GLC bit means different things for loads, stores, and atomic ops.

READ
GLC = 0 Reads can hit on the L1 and persist across wavefronts
GLC = 1 Reads miss the L1 and L2 and force fetch to the data fabric. 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.

9.1.10.2. Vector Memory

Vector Memory instructions (Flat, Global, Scratch, and Buffer) have 3 bits to control scope and cacheability:

- SC[1:0] System Cache level: 0=wave, 1=group, 2=device, 3=system
- NT Non-Temporal: 0=expect temporal reuse; 1=do not expect temporal reuse

Loads

Table 48. Load Controls

Scope	SC1	SC0	NT	CU Cache Behavior	L2 Cache Behavior	Last-level Cache Behavior
Wave	0	0	0	Hit LRU	Hit LRU	Hit LRU
			1	Miss Evict	Hit Stream	Hit Evict
Group	0	1	0	Hit LRU	Hit LRU	Hit Evict
			1	Miss Evict	Hit Stream	Hit Evict
Device	1	0	0	Miss Evict	(1 L2 cache): Hit LRU; (>1 L2 cache): Coherent Cache Bypass	Hit LRU
			1	Miss Evict	(1 L2 cache): Hit Stream; (>1 L2 cache): Coherent Cache Bypass	Hit Evict
System	1	1	0	Miss Evict	Coherent Cache Bypass	Hit LRU
			1	Miss Evict	Coherent Cache Bypass	Hit Evict

Note that if TG_SPLIT mode is active, SC1==0, SC0==1, NT==0 case becomes: Miss LRU, Hit LRU, Hit LRU.

Stores

Table 49. Store Controls

Scope	SC1	SC0	NT	CU Cache Behavior	L2 Cache Behavior	Last-level Cache Behavior
Wave	0	0	0	Miss LRU	Hit LRU	Hit LRU
			1	Miss Evict	Hit Stream	Hit Evict
Group	0	1	0	Miss LRU	Hit LRU	Hit LRU
			1	Miss Evict	Hit Stream	Hit Evict
Device	1	0	0	Miss Evict	(1 L2 cache): Hit LRU; (>1 L2 cache): Coherent Cache Bypass	Hit LRU
			1	Miss Evict	(1 L2 cache): Hit Stream; (>1 L2 cache): Coherent Cache Bypass	Hit Evict
System	1	1	0	Miss Evict	Coherent Cache Bypass	Hit LRU
			1	Miss Evict	Coherent Cache Bypass	Hit Evict

Atomics

For atomics, SC0 indicates whether or not to return the "pre-op" memory value (the value in memory before the atomic operation was performed). 0 = no return, 1 = return pre-op value.

SC1 : 0 = device scope atomic, 1 = system scope atomic

NT : 0 = last level cache "allocate" policy; 1 = "no Allocate" policy

Invalidate and Writeback

Table 50. BUFFER_WBL2

SC1	SC0	L2 Cache Behavior
0	any	NOP
1	0	(1 L2 cache): NOP, (>1 L2 cache) Write-back dirty data
1	1	Write back dirty data

Table 51. BUFFER_INV

SC1	SC0	CU Cache Behavior	L2 Cache Behavior
0	0	NOP	NOP
0	1	(if TG Split): Invalidate cache, (If not TG Split): NOP	NOP
1	0	Invalidate cache	(1 L2 cache): NOP, (>1 L2 cache) Invalidate non-coherently cached lines
1	1	Invalidate cache	Invalidate non-coherently cached lines

9.1.11. Data Formats

The table below shows the buffer data formats:

DATA_FORMAT		NUM_FORMAT	
value	enum	value	enum
Buffer formats			
0	INVALID	0	Any
1	8	0	UNORM
1	8	1	SNORM
1	8	2	USCALED
1	8	3	SSCALED
1	8	4	UINT
1	8	5	SINT
1	8	9	SRGB
2	16	0	UNORM
2	16	1	SNORM
2	16	2	USCALED
2	16	3	SSCALED
2	16	4	UINT
2	16	5	SINT
2	16	7	FLOAT
3	8_8	0	UNORM
3	8_8	1	SNORM
3	8_8	2	USCALED
3	8_8	3	SSCALED
3	8_8	4	UINT
3	8_8	5	SINT
3	8_8	9	SRGB
4	32	4	UINT
4	32	5	SINT
4	32	7	FLOAT
5	16_16	0	UNORM
5	16_16	1	SNORM
5	16_16	2	USCALED
5	16_16	3	SSCALED
5	16_16	4	UINT
5	16_16	5	SINT
5	16_16	7	FLOAT
6	10_11_11	7	FLOAT
7	11_11_10	7	FLOAT

DATA_FORMAT		NUM_FORMAT	
value	enum	value	enum
Buffer formats			
8	10_10_10_2	0	UNORM
8	10_10_10_2	1	SNORM
8	10_10_10_2	2	USCALED
8	10_10_10_2	3	SSCALED
8	10_10_10_2	4	UINT
8	10_10_10_2	5	SINT
9	2_10_10_10	0	UNORM
9	2_10_10_10	1	SNORM
9	2_10_10_10	2	USCALED
9	2_10_10_10	3	SSCALED
9	2_10_10_10	4	UINT
9	2_10_10_10	5	SINT
10	8_8_8_8	0	UNORM
10	8_8_8_8	1	SNORM
10	8_8_8_8	2	USCALED
10	8_8_8_8	3	SSCALED
10	8_8_8_8	4	UINT
10	8_8_8_8	5	SINT
10	8_8_8_8	9	SRGB
11	32_32	4	UINT
11	32_32	5	SINT
11	32_32	7	FLOAT
12	16_16_16_16	0	UNORM
12	16_16_16_16	1	SNORM
12	16_16_16_16	2	USCALED
12	16_16_16_16	3	SSCALED
12	16_16_16_16	4	UINT
12	16_16_16_16	5	SINT
12	16_16_16_16	7	FLOAT
13	32_32_32	4	UINT
13	32_32_32	5	SINT
13	32_32_32	7	FLOAT
14	32_32_32_32	4	UINT
14	32_32_32_32	5	SINT
14	32_32_32_32	7	FLOAT

9.2. Float Memory Atomics

Floating point memory atomics are executed in LDS and in the L2 cache. They can be issued as LDS, Buffer, Flat, Global, and Scratch instructions.

This chapter explains the rules for rounding, denormals and NaN for floating point atomics.

9.2.1. Rounding of Float Atomics

All float atomic ADD opcodes use "Round to Nearest-Even" rounding.

9.2.2. Denormal (Subnormal) Handling

When atomics operate on floating point data, there is the possibility of the data containing denormal numbers, or the operation producing a denormal.

Denormals: The floating point atomic instructions have the option of passing denormal values through, or flushing them to zero. This is controlled with the MODE.denorm bits which also control VALU denormal behavior. As with VALU ops, “denorm_single” affects F32 ops and “denorm_double” affects F64 and F16. Some atomics have fixed denormal handling behavior.

LDS instructions allows denormals to be passed through or flushed to zero based on the MODE.denormal wave-state register.

- Float 16 and 32 bit Adder uses both input and output denorm flush controls from MODE
- Float 64 bit adder does not flush denorms
- Float CMP, MIN and MAX use only the “input denormal” flushing control
  - Each input to the comparisons flushes the mantissa of both operands to zero before the compare if the exponent is zero and the flush denorm control is active. For Min and Max the actual result returned is the selected non-flushed input.
  - CompareStore (“compare swap”) flushes the result when input denormal flushing occurs.

Table 52. Denorm Handling Rules for Memory Ops

Atomic type	LDS Handling	L2 Cache Handling
PK_ADD_F16 / BF16	Mode	Do not Flush Denorms
ADD_F32	Mode	Flush Denorms
Min/MAX_F32	Mode	N/A
CMPST_F32	Mode	N/A
MIN/MAX_F64	Mode	Do not Flush Denorms
CMPST_F64	Mode	N/A
ADD_F64	Do not Flush Denorms	Do not Flush Denorms

- “Flush Denorms” = flush all input denorm
- “Do not Flush” = don’t flush input denorm
- “Mode” = denormal flush controlled by bit from shader’s “MODE . fp_denorm” register
- “Mode + reg” = “Mode” from above, but there exists an override register to flush output or not.

Note that MIN and MAX when flushing denormals only do it for the comparison, but the result is an unmodified copy of one of the sources. CompareStore (“compare swap”) flushes the result when input denormal flushing occurs.

9.2.3. NaN Handling

Not A Number (“NaN”) is a IEEE-754 value representing a result which cannot be computed.

There two types of NaN: quiet and signaling

- Quiet NaN Exponent=0xFF, Mantissa MSB=1
- Signaling NaN Exponent=0xFF, Mantissa MSB=0 and at least one other mantissa bit ==1

The LDS does not produce any exception or “signal” due to a signaling NaN.

DS_ADD_F32 can create a quiet NaN, or propagate NaN from its inputs: if either input is a NaN, the output is that same NaN, and if both inputs are NaN, the NaN from the first input is selected as the output. Signaling NaN is converted to Quiet NaN.

Floating point atomics (CMPSWAP, MIN, MAX) flush input denormals only when
MODE (allow_input_denorm)=0, otherwise values are passed through without modification. When flushing, denorms are flushed before the operation (i.e. before the comparison).

FP Max Selection Rules:

```
if (src0 == SNaN) result = QNaN (src0)
else if (src1 == SNaN) result = QNaN (src1)
else result = larger of (src0, src1)
“Larger” order from smallest to largest: QNaN, -inf, -float, -denorm, -0, +0, +denorm, +float, +inf
```

FP Min Selection Rules:

```
if (src0 == SNaN) result = QNaN (src0)
else if (src1 == SNaN) result = QNaN (src1)
else result = smaller of (src0, src1)
“Smaller” order from smallest to largest: -inf, -float, -denorm, -0, +0, +denorm, +float, +inf, QNaN
```

FP Compare Swap: only swap if the compare condition (==) is true, treating +0 and -0 as equal

```
doSwap = (src0 != NaN) && (src1 != NaN) && (src0 == src1) // allow +0 == -0
```

Float Add rules:

1. -INF + INF = QNAN (mantissa is all zeros except MSB)
2. +/-INF + NAN = QNAN (NAN input is copied to output but made quiet NAN)
3. -0 + 0 = +0
4. INF + (float, +0, -0) = INF, with infinity sign preserved
5. NaN + NaN = SRC0’s NaN, converted to QNaN

Chapter 10. 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. 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.

10.1. Flat Memory Instruction

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 the table below:

Table 53. Flat, Global and Scratch Microcode Formats

Field	Bit Size	Description
OP	7	Opcode. Can be Flat, Scratch or Global instruction. See next table.
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 SC[0]=1 (return pre-op value).
SC	2	Memory Scope
NT	1	Non-Temporal
ACC	1	DATA is Accumulation VGPR
SEG	2	Memory Segment: 0=FLAT, 1=SCRATCH, 2=GLOBAL, 3=reserved.
SVE	1	Scratch VGPR Enable - indicates if a VGPR contributes to calculating scratch memory addresses.
NV	1	Non-volatile. When set, the read/write is operating on non-volatile memory.
OFFSET	13	Address offset. Scratch, Global: 13-bit signed byte offset. Flat: 12-bit unsigned offset (MSB is ignored).
SADDR	7	Scalar SGPR that provides an offset address. To disable, set this field to 0x7F. Meaning of this field is different for Scratch and Global: Flat: Unused. Scratch: Use an SGPR (instead of VGPR) for the address. Global: Use the SGPR to provide a base address; the VGPR provides a 32-bit offset.
M0	16	Implied use of M0 for SCRATCH and GLOBAL only when LDS=1. Provides the LDS address-offset.

Table 54. Flat, Global and Scratch Opcodes

Flat Opcodes	Global Opcodes	Scratch Opcodes
FLAT	GLOBAL	SCRATCH
FLAT_LOAD_UBYTE	GLOBAL_LOAD_UBYTE	SCRATCH_LOAD_UBYTE
FLAT_LOAD_UBYTE_D16	GLOBAL_LOAD_UBYTE_D16	SCRATCH_LOAD_UBYTE_D16
FLAT_LOAD_UBYTE_D16_HI	GLOBAL_LOAD_UBYTE_D16_HI	SCRATCH_LOAD_UBYTE_D16_HI
FLAT_LOAD_SBYTE	GLOBAL_LOAD_SBYTE	SCRATCH_LOAD_SBYTE
FLAT_LOAD_SBYTE_D16	GLOBAL_LOAD_SBYTE_D16	SCRATCH_LOAD_SBYTE_D16

Flat Opcodes	Global Opcodes	Scratch Opcodes
FLAT_LOAD_SBYTE_D16_HI	GLOBAL_LOAD_SBYTE_D16_HI	SCRATCH_LOAD_SBYTE_D16_HI
FLAT_LOAD_USHORT	GLOBAL_LOAD_USHORT	SCRATCH_LOAD_USHORT
FLAT_LOAD_SSHORT	GLOBAL_LOAD_SSHORT	SCRATCH_LOAD_SSHORT
FLAT_LOAD_SHORT_D16	GLOBAL_LOAD_SHORT_D16	SCRATCH_LOAD_SHORT_D16
FLAT_LOAD_SHORT_D16_HI	GLOBAL_LOAD_SHORT_D16_HI	SCRATCH_LOAD_SHORT_D16_HI
FLAT_LOAD_DWORD	GLOBAL_LOAD_DWORD	SCRATCH_LOAD_DWORD
FLAT_LOAD_DWORDX2	GLOBAL_LOAD_DWORDX2	SCRATCH_LOAD_DWORDX2
FLAT_LOAD_DWORDX3	GLOBAL_LOAD_DWORDX3	SCRATCH_LOAD_DWORDX3
FLAT_LOAD_DWORDX4	GLOBAL_LOAD_DWORDX4	SCRATCH_LOAD_DWORDX4
FLAT_STORE_BYTE	GLOBAL_STORE_BYTE	SCRATCH_STORE_BYTE
FLAT_STORE_BYTE_D16_HI	GLOBAL_STORE_BYTE_D16_HI	SCRATCH_STORE_BYTE_D16_HI
FLAT_STORE_SHORT	GLOBAL_STORE_SHORT	SCRATCH_STORE_SHORT
FLAT_STORE_SHORT_D16_HI	GLOBAL_STORE_SHORT_D16_HI	SCRATCH_STORE_SHORT_D16_HI
FLAT_STORE_DWORD	GLOBAL_STORE_DWORD	SCRATCH_STORE_DWORD
FLAT_STORE_DWORDX2	GLOBAL_STORE_DWORDX2	SCRATCH_STORE_DWORDX2
FLAT_STORE_DWORDX3	GLOBAL_STORE_DWORDX3	SCRATCH_STORE_DWORDX3
FLAT_STORE_DWORDX4	GLOBAL_STORE_DWORDX4	SCRATCH_STORE_DWORDX4
FLAT_ATOMIC_SWAP	GLOBAL_ATOMIC_SWAP	none
FLAT_ATOMIC_CMPSWAP	GLOBAL_ATOMIC_CMPSWAP	none
FLAT_ATOMIC_ADD	GLOBAL_ATOMIC_ADD	none
FLAT_ATOMIC_SUB	GLOBAL_ATOMIC_SUB	none
FLAT_ATOMIC_SMIN	GLOBAL_ATOMIC_SMIN	none
FLAT_ATOMIC_UMIN	GLOBAL_ATOMIC_UMIN	none
FLAT_ATOMIC_SMAX	GLOBAL_ATOMIC_SMAX	none
FLAT_ATOMIC_UMAX	GLOBAL_ATOMIC_UMAX	none
FLAT_ATOMIC_AND	GLOBAL_ATOMIC_AND	none
FLAT_ATOMIC_OR	GLOBAL_ATOMIC_OR	none
FLAT_ATOMIC_XOR	GLOBAL_ATOMIC_XOR	none
FLAT_ATOMIC_INC	GLOBAL_ATOMIC_INC	none
FLAT_ATOMIC_DEC	GLOBAL_ATOMIC_DEC	none
FLAT_ATOMIC_ADD_F32	GLOBAL_ATOMIC_ADD_F32	none
FLAT_ATOMIC_PK_ADD_F16	GLOBAL_ATOMIC_PK_ADD_F16	none
FLAT_ATOMIC_PK_ADD_BF16	GLOBAL_ATOMIC_PK_ADD_BF16	none
FLAT_ATOMIC_ADD_F64	GLOBAL_ATOMIC_ADD_F64	none
FLAT_ATOMIC_MIN_F64	GLOBAL_ATOMIC_MIN_F64	none
FLAT_ATOMIC_MAX_F64	GLOBAL_ATOMIC_MAX_F64	none
The non-float atomic instructions above are also available in "_X2" versions (64-bit).		

Table 55. SVE Bit

SADDR	SVE	Mode
==EXEC_HI	0	ST : addr = flat_scratch + swizzle(inst.offset, threadID)
!=EXEC_HI	0	SS : addr = flat_scratch + swizzle(sgpr_offset + inst.offset, threadID)
==EXEC_HI	1	SV : addr = flat_scratch + swizzle(vgpr_offset + inst.offset, threadID)
!=EXEC_HI	1	SVS : addr = flat_scratch + swizzle(sgpr_offset + vgpr_offset + inst.offset, threadID)

10.2. 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 specific SGPR-pair to hold scratch-space information in case any threads' address resolves to scratch space. See the Scratch section for details.

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.

10.2.1. 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.

10.2.2. 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.

10.3. 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 texture unit takes the address from the VGPR and does the following.

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

Instructions which return data to LDS address LDS as:

```
DWORDX1:  LDS_ADDR = LDSbase(hw alloc) + LDSoffset(M0[17:2] * 4) + INST.OFFSET + ThreadID*4
DWORDX4:  LDS_ADDR = LDSbase(hw alloc) + LDSoffset(M0[17:2] * 4) + INST.OFFSET + ThreadID*16
```

10.3.1. Atomics

Float atomics must set SC[0]=0 (no return value).

Memory atomics are performed in the data fabric so they are known to be atomic with host memory access.

FP32 atomic operations flush denormals to zero, and both FP64 and FP16 atomic do not flush denormals. The rounding mode is fixed and "round to nearest even".

10.4. Global

Global instructions are similar to Flat instructions, but the programmer must ensure that no threads access LDS space; thus, no LDS bandwidth is used by global instructions.

Global instructions offer two types of addressing:

- Memory_addr = VGPR-address + instruction offset.
- Memory_addr = SGPR-address + VGPR-offset + instruction offset.

The size of the address component is dependent on ADDRESS_MODE: 32-bits or 64-bit pointers. The VGPR-offset is 32 bits.

These instructions also allow direct data movement between LDS and memory without going through VGPRs.

Since these instructions do not access LDS, only VM_CNT is used, not LGKM_CNT. If a global instruction does attempt to access LDS, the instruction returns MEM_VIOL.

10.5. Scratch

Scratch instructions are similar to Flat, but the programmer must ensure that no threads access LDS space, and the memory space is swizzled. Thus, no LDS bandwidth is used by scratch instructions.

Scratch instructions also support multi-Dword access and mis-aligned access (although mis-aligned is slower).

Scratch instructions use the following addressing:

- Memory_addr = flat_scratch.addr + swizzle(V/SGPR_offset + inst_offset, threadID)
- The offset can come from either an SGPR or a VGPR, and is a 32- bit unsigned byte.

The size of the address component is dependent on the ADDRESS_MODE: 32-bits or 64-bit pointers. The VGPR-offset is 32 bits.

These instructions also allow direct data movement between LDS and memory without going through VGPRs.

Since these instructions do not access LDS, only VM_CNT is used, not LGKM_CNT. It is not possible for a Scratch instruction to access LDS; thus, no error or aperture checking is done.

10.6. 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.

10.7. 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 FLAT_SCRATCH address is automatically sent with every FLAT request.

FLAT_SCRATCH is a 64-bit, byte address. The shader composes the value by adding together two separate values: the base address, which can be passed in via an initialized SGPR, or perhaps through a constant buffer, and the per-wave allocation offset (also initialized in an SGPR).

Chapter 11. 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. 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).

11.1. Overview

The figure below shows the conceptual framework of the LDS is integration into the memory of AMD Accelerators using OpenCL.

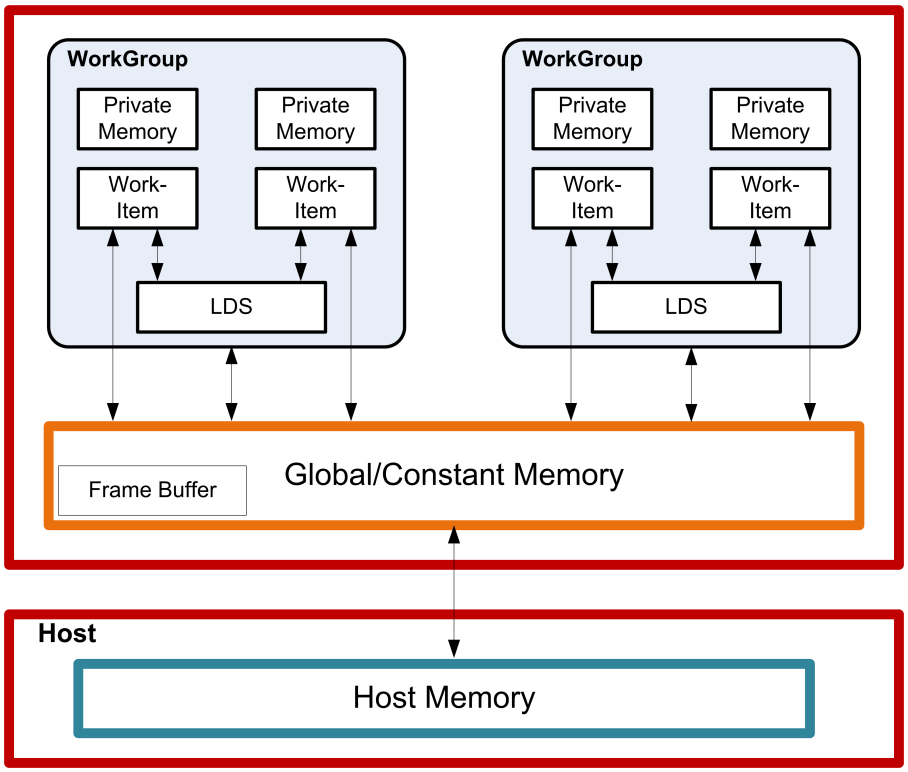


Figure 6. High-Level Memory Configuration

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

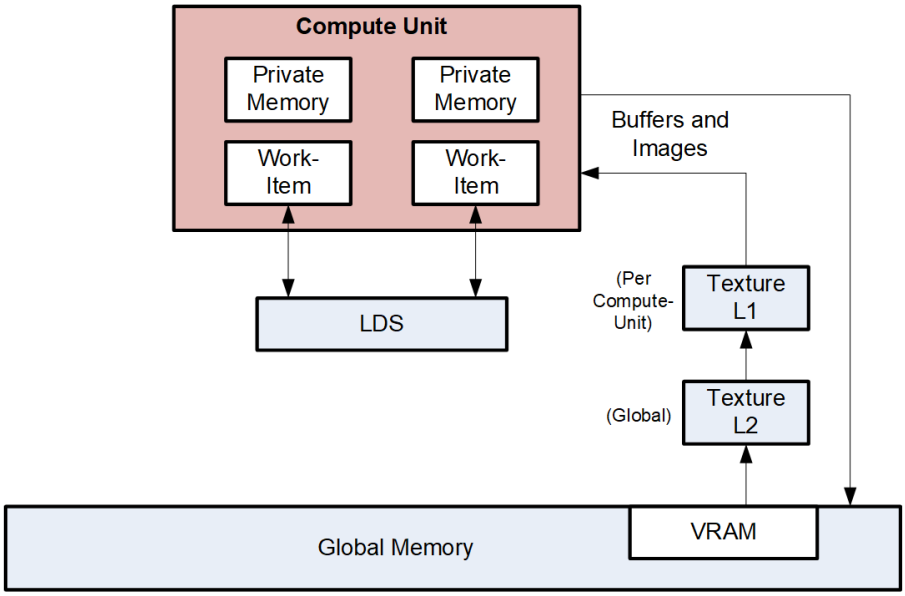
There are 64 kB memory per compute unit, segmented into 32 banks of 512 Dwords. Each bank is a 256x32 two-port RAM (1R/1W per clock cycle). Dwords are placed in the banks serially, but all banks can execute a store or load simultaneously. One work-group can request up to 64 kB memory. Reads across wavefront 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 can decrease the effective bandwidth of the LDS. To help achieve optimal throughput (optimal efficiency), therefore, it is important to avoid bank conflicts. A knowledge of request scheduling and address mapping can be key to help achieving this.

11.2. Dataflow in Memory Hierarchy

The figure below is a conceptual diagram of the dataflow within the memory structure.



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 workitem's registers, then placed into global memory. To make effective use of the LDS, a kernel 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.)

11.3. LDS Access

The LDS is accessed via indexed or atomic instructions.

11.3.1. Data Share Indexed and Atomic Access

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.

The table below lists and briefly describes the LDS instruction fields.

Table 56. LDS Instruction Fields

Field	Size	Description
OP	7	LDS opcode.
GDS	1	0 = LDS, 1 = GWS.
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.
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 57. LDS Indexed Load/Store

Load / Store	Description
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.
DS_PERMUTE_B32	Forward permute. Does not write any LDS memory. LDS[dst] = src0 returnVal = LDS[thread_id] where thread_id is 0..63.
DS_BPERMUTE_B32	Backward permute. Does not actually write any LDS memory. LDS[thread_id] = src0 where thread_id is 0..63, and returnVal = LDS[dst].

Single Address Instructions

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

Double Address Instructions

```
LDS_Addr0 = LDS_BASE + VGPR[ADDR] + InstrOffset0*ADJ +
LDS_Addr1 = LDS_BASE + VGPR[ADDR] + InstrOffset1*ADJ
   Where ADJ = 4 for 8, 16 and 32-bit data types; and ADJ = 8 for 64-bit.
```

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

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.

Denormal behavior for floating point atomics is controlled via the MODE register's FP_DENORM field. The rounding mode is fixed at "round to nearest even".

11.4. GWS Programming Restriction

All GWS instructions must be immediately followed by:

```
    s_waitcnt 0
```

VGPRs used by any GWS instruction must be even.

Chapter 12. Instructions

This chapter lists, and provides descriptions for, all instructions in the CDNA Generation 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)
- BF16 floating-point 16 bit (Bfloat16 format)
- I8 signed 8-bit integer
- I16 signed 16-bit integer
- 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.

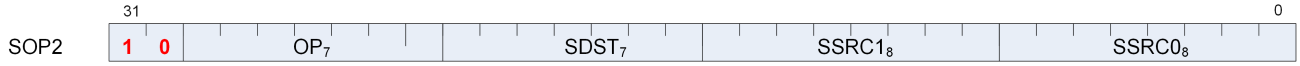
The following abbreviations are used in instruction definitions:

- D = destination
- U = unsigned integer
- S = source
- SCC = scalar condition code
- I = signed integer
- B = bitfield

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

Note: Rounding and Denormal modes apply to all floating-point operations unless otherwise specified in the instruction description.

12.1. SOP2 Instructions



Instructions in this format may use a 32-bit literal constant which occurs immediately after the instruction.

S_ADD_U32 **0**

Add two unsigned inputs, store the result into a scalar register and store the carry-out bit into SCC.

```
tmp = 64'U(S0.u32) + 64'U(S1.u32);
SCC = tmp >= 0x100000000ULL ? 1'1U : 1'0U;
// unsigned overflow or carry-out for S_ADDC_U32.
D0.u32 = tmp.u32
```

S_SUB_U32 1

Subtract the second unsigned input from the first input, store the result into a scalar register and store the carry-out bit into SCC.

```
tmp = S0.u32 - S1.u32;
SCC = S1.u32 > S0.u32 ? 1'1U : 1'0U;
// unsigned overflow or carry-out for S_SUBB_U32.
D0.u32 = tmp.u32
```

S_ADD_I32 2

Add two signed inputs, store the result into a scalar register and store the carry-out bit into SCC.

```
tmp = S0.i32 + S1.i32;
SCC = ((S0.u32[31] == S1.u32[31]) && (S0.u32[31] != tmp.u32[31]));
// signed overflow.
D0.i32 = tmp.i32
```

Notes

This opcode is not suitable for use with S_ADDC_U32 for implementing 64-bit operations.

S_SUB_I32 3

Subtract the second signed input from the first input, store the result into a scalar register and store the carry-out bit into SCC.

```
tmp = S0.i32 - S1.i32;
SCC = ((S0.u32[31] != S1.u32[31]) && (S0.u32[31] != tmp.u32[31]));
// signed overflow.
D0.i32 = tmp.i32
```

Notes

This opcode is not suitable for use with S_SUBB_U32 for implementing 64-bit operations.

S_ADDC_U32 4

Add two unsigned inputs and a carry-in bit, store the result into a scalar register and store the carry-out bit into SCC.

```
tmp = 64'U(S0.u32) + 64'U(S1.u32) + SCC.u64;
SCC = tmp >= 0x100000000ULL ? 1'1U : 1'0U;
// unsigned overflow.
D0.u32 = tmp.u32
```

S_SUBB_U32 5

Subtract the second unsigned input from the first input, subtract the carry-in bit, store the result into a scalar register and store the carry-out bit into SCC.

```
tmp = S0.u32 - S1.u32 - SCC.u32;
SCC = 64'U(S1.u32) + SCC.u64 > 64'U(S0.u32) ? 1'1U : 1'0U;
// unsigned overflow.
D0.u32 = tmp.u32
```

S_MIN_I32 6

Select the minimum of two signed 32-bit integer inputs, store the selected value into a scalar register and set SCC iff the first value is selected.

```
SCC = S0.i32 < S1.i32;
D0.i32 = SCC ? S0.i32 : S1.i32
```

S_MIN_U32 7

Select the minimum of two unsigned 32-bit integer inputs, store the selected value into a scalar register and set SCC iff the first value is selected.

```
SCC = S0.u32 < S1.u32;
D0.u32 = SCC ? S0.u32 : S1.u32
```

S_MAX_I32 8

Select the maximum of two signed 32-bit integer inputs, store the selected value into a scalar register and set SCC iff the first value is selected.

```
SCC = S0.i32 >= S1.i32;
D0.i32 = SCC ? S0.i32 : S1.i32
```

S_MAX_U32 9

Select the maximum of two unsigned 32-bit integer inputs, store the selected value into a scalar register and set SCC iff the first value is selected.

```
SCC = S0.u32 >= S1.u32;
D0.u32 = SCC ? S0.u32 : S1.u32
```

S_CSELECT_B32 10

Select the first input if SCC is true otherwise select the second input, then store the selected input into a scalar register.

```
D0.u32 = SCC ? S0.u32 : S1.u32
```

S_CSELECT_B64 11

Select the first input if SCC is true otherwise select the second input, then store the selected input into a scalar register.

```
D0.u64 = SCC ? S0.u64 : S1.u64
```

S_AND_B32 12

Calculate bitwise AND on two scalar inputs, store the result into a scalar register and set SCC iff the result is nonzero.

```
D0.u32 = (S0.u32 & S1.u32);
SCC = D0.u32 != 0U
```

S_AND_B64 13

Calculate bitwise AND on two scalar inputs, store the result into a scalar register and set SCC iff the result is nonzero.

```
D0.u64 = (S0.u64 & S1.u64);
SCC = D0.u64 != 0ULL
```

S_OR_B32 14

Calculate bitwise OR on two scalar inputs, store the result into a scalar register and set SCC iff the result is nonzero.

```
D0.u32 = (S0.u32 | S1.u32);
SCC = D0.u32 != 0U
```

S_OR_B64 15

Calculate bitwise OR on two scalar inputs, store the result into a scalar register and set SCC iff the result is nonzero.

```
D0.u64 = (S0.u64 | S1.u64);
SCC = D0.u64 != 0ULL
```

S_XOR_B32 16

Calculate bitwise XOR on two scalar inputs, store the result into a scalar register and set SCC iff the result is nonzero.

```
D0.u32 = (S0.u32 ^ S1.u32);
SCC = D0.u32 != 0U
```

S_XOR_B64 17

Calculate bitwise XOR on two scalar inputs, store the result into a scalar register and set SCC iff the result is nonzero.

```
D0.u64 = (S0.u64 ^ S1.u64);
```

```
SCC = D0.u64 != 0ULL
```

S_ANDN2_B32 18

Calculate bitwise AND with the first input and the negation of the second input, store the result into a scalar register and set SCC if the result is nonzero.

```
D0.u32 = (S0.u32 & ~S1.u32);
SCC = D0.u32 != 0U
```

S_ANDN2_B64 19

Calculate bitwise AND with the first input and the negation of the second input, store the result into a scalar register and set SCC if the result is nonzero.

```
D0.u64 = (S0.u64 & ~S1.u64);
SCC = D0.u64 != 0ULL
```

S_ORN2_B32 20

Calculate bitwise OR with the first input and the negation of the second input, store the result into a scalar register and set SCC if the result is nonzero.

```
D0.u32 = (S0.u32 | ~S1.u32);
SCC = D0.u32 != 0U
```

S_ORN2_B64 21

Calculate bitwise OR with the first input and the negation of the second input, store the result into a scalar register and set SCC if the result is nonzero.

```
D0.u64 = (S0.u64 | ~S1.u64);
SCC = D0.u64 != 0ULL
```

S_NAND_B32 22

Calculate bitwise NAND on two scalar inputs, store the result into a scalar register and set SCC if the result is

nonzero.

```
D0.u32 = ~(S0.u32 & S1.u32);
SCC = D0.u32 != 0U
```

S_NAND_B64 23

Calculate bitwise NAND on two scalar inputs, store the result into a scalar register and set SCC if the result is nonzero.

```
D0.u64 = ~(S0.u64 & S1.u64);
SCC = D0.u64 != 0ULL
```

S_NOR_B32 24

Calculate bitwise NOR on two scalar inputs, store the result into a scalar register and set SCC if the result is nonzero.

```
D0.u32 = ~(S0.u32 | S1.u32);
SCC = D0.u32 != 0U
```

S_NOR_B64 25

Calculate bitwise NOR on two scalar inputs, store the result into a scalar register and set SCC if the result is nonzero.

```
D0.u64 = ~(S0.u64 | S1.u64);
SCC = D0.u64 != 0ULL
```

S_XNOR_B32 26

Calculate bitwise XNOR on two scalar inputs, store the result into a scalar register and set SCC if the result is nonzero.

```
D0.u32 = ~(S0.u32 ^ S1.u32);
SCC = D0.u32 != 0U
```

S_XNOR_B64 27

Calculate bitwise XNOR on two scalar inputs, store the result into a scalar register and set SCC if the result is nonzero.

```
D0.u64 = ~(S0.u64 ^ S1.u64);
SCC = D0.u64 != 0ULL
```

S_LSHL_B32 28

Given a shift count in the second scalar input, calculate the logical shift left of the first scalar input, store the result into a scalar register and set SCC iff the result is nonzero.

```
D0.u32 = (S0.u32 << S1[4 : 0].u32);
SCC = D0.u32 != 0U
```

S_LSHL_B64 29

Given a shift count in the second scalar input, calculate the logical shift left of the first scalar input, store the result into a scalar register and set SCC iff the result is nonzero.

```
D0.u64 = (S0.u64 << S1[5 : 0].u32);
SCC = D0.u64 != 0ULL
```

S_LSHR_B32 30

Given a shift count in the second scalar input, calculate the logical shift right of the first scalar input, store the result into a scalar register and set SCC iff the result is nonzero.

```
D0.u32 = (S0.u32 >> S1[4 : 0].u32);
SCC = D0.u32 != 0U
```

S_LSHR_B64 31

Given a shift count in the second scalar input, calculate the logical shift right of the first scalar input, store the result into a scalar register and set SCC iff the result is nonzero.

```
D0.u64 = (S0.u64 >> S1[5 : 0].u32);
```

```
SCC = D0.u64 != 0ULL
```

S_ASHR_I32 32

Given a shift count in the second scalar input, calculate the arithmetic shift right (preserving sign bit) of the first scalar input, store the result into a scalar register and set SCC iff the result is nonzero.

```
D0.i32 = 32'I(signext(S0.i32) >> S1[4 : 0].u32);
SCC = D0.i32 != 0
```

S_ASHR_I64 33

Given a shift count in the second scalar input, calculate the arithmetic shift right (preserving sign bit) of the first scalar input, store the result into a scalar register and set SCC iff the result is nonzero.

```
D0.i64 = (signext(S0.i64) >> S1[5 : 0].u32);
SCC = D0.i64 != 0LL
```

S_BFM_B32 34

Calculate a bitfield mask given a field offset and size and store the result in a scalar register.

```
D0.u32 = (((1U << S0[4 : 0].u32) - 1U) << S1[4 : 0].u32)
```

S_BFM_B64 35

Calculate a bitfield mask given a field offset and size and store the result in a scalar register.

```
D0.u64 = (((1ULL << S0[5 : 0].u32) - 1ULL) << S1[5 : 0].u32)
```

S_MUL_I32 36

Multiply two signed integers and store the result into a scalar register.

```
D0.i32 = S0.i32 * S1.i32
```

S_BFE_U32 37

Extract an unsigned bitfield from the first input using field offset and size encoded in the second input, store the result into a scalar register and set SCC iff the result is nonzero.

```
D0.u32 = ((S0.u32 >> S1[4 : 0].u32) & ((1U << S1[22 : 16].u32) - 1U));
SCC = D0.u32 != 0U
```

S_BFE_I32 38

Extract a signed bitfield from the first input using field offset and size encoded in the second input, store the result into a scalar register and set SCC iff the result is nonzero.

```
tmp.i32 = ((S0.i32 >> S1[4 : 0].u32) & ((1 << S1[22 : 16].u32) - 1));
D0.i32 = signext_from_bit(tmp.i32, S1[22 : 16].u32);
SCC = D0.i32 != 0
```

S_BFE_U64 39

Extract an unsigned bitfield from the first input using field offset and size encoded in the second input, store the result into a scalar register and set SCC iff the result is nonzero.

```
D0.u64 = ((S0.u64 >> S1[5 : 0].u32) & ((1ULL << S1[22 : 16].u32) - 1ULL));
SCC = D0.u64 != 0ULL
```

S_BFE_I64 40

Extract a signed bitfield from the first input using field offset and size encoded in the second input, store the result into a scalar register and set SCC iff the result is nonzero.

```
tmp.i64 = ((S0.i64 >> S1[5 : 0].u32) & ((1LL << S1[22 : 16].u32) - 1LL));
D0.i64 = signext_from_bit(tmp.i64, S1[22 : 16].u32);
SCC = D0.i64 != 0LL
```

S_CBRANCH_G_FORK 41

Conditional branch using branch-stack.

S0 = compare mask (VCC or any SGPR) and S1 = 64-bit byte address of target instruction. See also S_CBRANCH_JOIN.

```
mask_pass = (S0.u64 & EXEC.u64);
mask_fail = (~S0.u64 & EXEC.u64);
if mask_pass == EXEC.u64 then
    PC = 64'I(S1.u64)
elsif mask_fail == EXEC.u64 then
    PC += 4LL
elsif bitCount(mask_fail.b64) < bitCount(mask_pass.b64) then
    EXEC = mask_fail.b64;
    SGPR[WAVE_MODE.CSP.u32 * 4U].b128 = { S1.u64, mask_pass };
    WAVE_MODE.CSP += 3'1U;
    PC += 4LL
else
    EXEC = mask_pass.b64;
    SGPR[WAVE_MODE.CSP.u32 * 4U].b128 = { (PC + 4LL), mask_fail };
    WAVE_MODE.CSP += 3'1U;
    PC = 64'I(S1.u64)
endif
```

S_ABSDIFF_I32 42

Calculate the absolute value of difference between two scalar inputs, store the result into a scalar register and set SCC iff the result is nonzero.

```
D0.i32 = S0.i32 - S1.i32;
if D0.i32 < 0 then
    D0.i32 = -D0.i32
endif;
SCC = D0.i32 != 0
```

Notes

Functional examples:

```
S_ABSDIFF_I32(0x00000002, 0x00000005) => 0x00000003
S_ABSDIFF_I32(0xffffffff, 0x00000000) => 0x00000001
S_ABSDIFF_I32(0x80000000, 0x00000000) => 0x80000000     // Note: result is negative!
S_ABSDIFF_I32(0x80000000, 0x00000001) => 0x7fffffff
S_ABSDIFF_I32(0x80000000, 0xffffffff) => 0x7fffffff
S_ABSDIFF_I32(0x80000000, 0xfffffffe) => 0x7ffffffe
```

S_MUL_HI_U32 44

Multiply two unsigned integers and store the high 32 bits of the result into a scalar register.

```
D0.u32 = 32'U((64'U(S0.u32) * 64'U(S1.u32)) >> 32U)
```

S_MUL_HI_I32 45

Multiply two signed integers and store the high 32 bits of the result into a scalar register.

```
D0.i32 = 32'I((64'I(S0.i32) * 64'I(S1.i32)) >> 32U)
```

S_LSHL1_ADD_U32 46

Calculate the logical shift left of the first input by 1, then add the second input, store the result into a scalar register and set SCC iff the summation results in an unsigned overflow.

```
tmp = (64'U(S0.u32) << 1U) + 64'U(S1.u32);
SCC = tmp >= 0x100000000ULL ? 1'1U : 1'0U;
// unsigned overflow.
D0.u32 = tmp.u32
```

S_LSHL2_ADD_U32 47

Calculate the logical shift left of the first input by 2, then add the second input, store the result into a scalar register and set SCC iff the summation results in an unsigned overflow.

```
tmp = (64'U(S0.u32) << 2U) + 64'U(S1.u32);
SCC = tmp >= 0x100000000ULL ? 1'1U : 1'0U;
// unsigned overflow.
D0.u32 = tmp.u32
```

S_LSHL3_ADD_U32 48

Calculate the logical shift left of the first input by 3, then add the second input, store the result into a scalar register and set SCC iff the summation results in an unsigned overflow.

```
tmp = (64'U(S0.u32) << 3U) + 64'U(S1.u32);
SCC = tmp >= 0x100000000ULL ? 1'1U : 1'0U;
// unsigned overflow.
D0.u32 = tmp.u32
```

S_LSHL4_ADD_U32 49

Calculate the logical shift left of the first input by 4, then add the second input, store the result into a scalar register and set SCC iff the summation results in an unsigned overflow.

```
tmp = (64'U(S0.u32) << 4U) + 64'U(S1.u32);
SCC = tmp >= 0x100000000ULL ? 1'1U : 1'0U;
// unsigned overflow.
D0.u32 = tmp.u32
```

S_PACK_LL_B32_B16 50

Pack two 16-bit scalar values into a scalar register.

```
D0 = { S1[15 : 0].u16, S0[15 : 0].u16 }
```

S_PACK_LH_B32_B16 51

Pack two 16-bit scalar values into a scalar register.

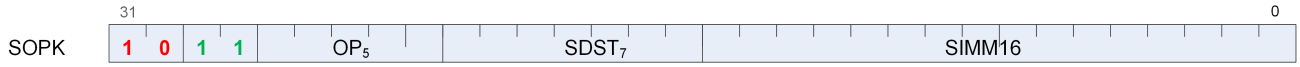
```
D0 = { S1[31 : 16].u16, S0[15 : 0].u16 }
```

S_PACK_HH_B32_B16 52

Pack two 16-bit scalar values into a scalar register.

```
D0 = { S1[31 : 16].u16, S0[31 : 16].u16 }
```

12.2. SOPK Instructions



Instructions in this format may use a 32-bit literal constant which occurs immediately after the instruction.

S_MOVK_I32 0

Sign extend a literal 16-bit constant and store the result into a scalar register.

```
D0.i32 = 32'I(signext(SIMM16.i16))
```

S_CMOVK_I32 1

Move the sign extension of a literal 16-bit constant into a scalar register iff SCC is nonzero.

```
if SCC then
    D0.i32 = 32'I(signext(SIMM16.i16))
endif
```

S_CMPK_EQ_I32 2

Set SCC to 1 iff scalar input is equal to the sign extension of a literal 16-bit constant.

```
SCC = 64'I(S0.i32) == signext(SIMM16.i16)
```

S_CMPK_LG_I32 3

Set SCC to 1 iff scalar input is less than or greater than the sign extension of a literal 16-bit constant.

```
SCC = 64'I(S0.i32) != signext(SIMM16.i16)
```

S_CMPK_GT_I32 4

Set SCC to 1 iff scalar input is greater than the sign extension of a literal 16-bit constant.

```
SCC = 64'I(S0.i32) > signext(SIMM16.i16)
```

S_CMPK_GE_I32 5

Set SCC to 1 iff scalar input is greater than or equal to the sign extension of a literal 16-bit constant.

```
SCC = 64'I(S0.i32) >= signext(SIMM16.i16)
```

S_CMPK_LT_I32 6

Set SCC to 1 iff scalar input is less than the sign extension of a literal 16-bit constant.

```
SCC = 64'I(S0.i32) < signext(SIMM16.i16)
```

S_CMPK_LE_I32 7

Set SCC to 1 iff scalar input is less than or equal to the sign extension of a literal 16-bit constant.

```
SCC = 64'I(S0.i32) <= signext(SIMM16.i16)
```

S_CMPK_EQ_U32 8

Set SCC to 1 iff scalar input is equal to the zero extension of a literal 16-bit constant.

```
SCC = S0.u32 == 32'U(SIMM16.u16)
```

S_CMPK_LG_U32 9

Set SCC to 1 iff scalar input is less than or greater than the zero extension of a literal 16-bit constant.

```
SCC = S0.u32 != 32'U(SIMM16.u16)
```

S_CMPK_GT_U32 10

Set SCC to 1 iff scalar input is greater than the zero extension of a literal 16-bit constant.

```
SCC = S0.u32 > 32'U(SIMM16.u16)
```

S_CMPK_GE_U32 11

Set SCC to 1 iff scalar input is greater than or equal to the zero extension of a literal 16-bit constant.

```
SCC = S0.u32 >= 32'U(SIMM16.u16)
```

S_CMPK_LT_U32 12

Set SCC to 1 iff scalar input is less than the zero extension of a literal 16-bit constant.

```
SCC = S0.u32 < 32'U(SIMM16.u16)
```

S_CMPK_LE_U32 13

Set SCC to 1 iff scalar input is less than or equal to the zero extension of a literal 16-bit constant.

```
SCC = S0.u32 <= 32'U(SIMM16.u16)
```

S_ADDK_I32 14

Add a scalar input and the sign extension of a literal 16-bit constant, store the result into a scalar register and store the carry-out bit into SCC.

```
tmp = D0.i32;
// save value so we can check sign bits for overflow later.
D0.i32 = 32'I(64'I(D0.i32) + signext(SIMM16.i16));
SCC = ((tmp[31] == SIMM16.i16[15]) && (tmp[31] != D0.i32[31]));
// signed overflow.
```

S_MULK_I32 15

Multiply a scalar input with the sign extension of a literal 16-bit constant and store the result into a scalar register.

```
D0.i32 = 32'I(64'I(D0.i32) * signext(SIMM16.i16))
```

S_CBRANCH_I_FORK 16

Conditional branch using branch-stack.

S0 = compare mask (VCC or any SGPR), and SIMM16 = signed DWORD branch offset relative to next instruction. See also S_CBRANCH_JOIN.

```
// Initial setup.
mask_pass = (S0.u64 & EXEC.u64);
mask_fail = (~S0.u64 & EXEC.u64);
target_addr = PC + signext(SIMM16.i32 * 4) + 4LL;
// Decide where to jump to.
if mask_pass == EXEC.u64 then
    PC = target_addr
elsif mask_fail == EXEC.u64 then
    PC += 4LL
elsif bitCount(mask_fail.b64) < bitCount(mask_pass.b64) then
    EXEC = mask_fail.b64;
    SGPR[WAVE_MODE.CSP.u32 * 4U].b128 = { target_addr, mask_pass };
    WAVE_MODE.CSP += 3'1U;
    PC += 4LL
else
    EXEC = mask_pass.b64;
    SGPR[WAVE_MODE.CSP.u32 * 4U].b128 = { (PC + 4LL), mask_fail };
    WAVE_MODE.CSP += 3'1U;
    PC = target_addr
endif
```

S_GETREG_B32 17

Read some or all of a hardware register into the LSBs of destination.

```
hwRegId = SIMM16.u16[5 : 0];
offset = SIMM16.u16[10 : 6];
size = SIMM16.u16[15 : 11].u32 + 1U;
// logical size is in range 1:32
value = HW_REGISTERS[hwRegId];
D0.u32 = 32'U(32'I(value >> offset.u32) & ((1 << size) - 1))
```

S_SETREG_B32 18

Write some or all of the LSBs of source argument into a hardware register.

```
hwRegId = SIMM16.u16[5 : 0];
offset = SIMM16.u16[10 : 6];
size = SIMM16.u16[15 : 11].u32 + 1U;
// logical size is in range 1:32
mask = (1 << size) - 1;
mask = (mask & 32'I(writeableBitMask(hwRegId.u32, WAVE_STATUS.PRIV)));
// Mask of bits we are allowed to modify
value = ((S0.u32 << offset.u32) & mask.u32);
value = (value | 32'U(HW_REGISTERS[hwRegId].i32 & ~mask));
HW_REGISTERS[hwRegId] = value.b32;
// Side-effects may trigger here if certain bits are modified
```

S_SETREG_IMM32_B32 20

Write some or all of the LSBs of a 32-bit literal constant into a hardware register; this instruction requires a 32-bit literal constant.

```
hwRegId = SIMM16.u16[5 : 0];
offset = SIMM16.u16[10 : 6];
size = SIMM16.u16[15 : 11].u32 + 1U;
// logical size is in range 1:32
mask = (1 << size) - 1;
mask = (mask & 32'I(writeableBitMask(hwRegId.u32, WAVE_STATUS.PRIV)));
// Mask of bits we are allowed to modify
value = ((SIMM32.u32 << offset.u32) & mask.u32);
value = (value | 32'U(HW_REGISTERS[hwRegId].i32 & ~mask));
HW_REGISTERS[hwRegId] = value.b32;
// Side-effects may trigger here if certain bits are modified
```

S_CALL_B64 21

Store the address of the next instruction to a scalar register and then jump to a constant offset relative to the current PC.

The literal argument is a signed DWORD offset relative to the PC of the next instruction. The byte address of the instruction immediately *following* this instruction is saved to the destination.

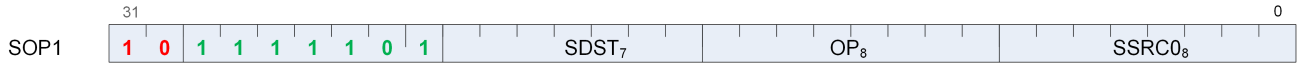
```
D0.i64 = PC + 4LL;
PC = PC + signext(SIMM16.i16 * 16'4) + 4LL
```

Notes

This implements a short subroutine call where the return address (the next instruction after the S_CALL_B64) is saved to D. Long calls should consider S_SWAPPC_B64 instead.

This instruction must be 4 bytes.

12.3. SOP1 Instructions



Instructions in this format may use a 32-bit literal constant which occurs immediately after the instruction.

S_MOV_B32 0

Move scalar input into a scalar register.

```
D0.b32 = S0.b32
```

S_MOV_B64 1

Move scalar input into a scalar register.

```
D0.b64 = S0.b64
```

S_CMOV_B32 2

Move scalar input into a scalar register iff SCC is nonzero.

```
if SCC then
    D0.b32 = S0.b32
endif
```

S_CMOV_B64 3

Move scalar input into a scalar register iff SCC is nonzero.

```
if SCC then
    D0.b64 = S0.b64
endif
```

S_NOT_B32 4

Calculate bitwise negation on a scalar input, store the result into a scalar register and set SCC iff the result is nonzero.

```
D0.u32 = ~S0.u32;
SCC = D0.u32 != 0U
```

S_NOT_B64 5

Calculate bitwise negation on a scalar input, store the result into a scalar register and set SCC iff the result is nonzero.

```
D0.u64 = ~S0.u64;
SCC = D0.u64 != 0ULL
```

S_WQM_B32 6

Given an active pixel mask in a scalar input, calculate whole quad mode mask for that input, store the result into a scalar register and set SCC iff the result is nonzero.

In whole quad mode, if any pixel in a quad is active then all pixels of the quad are marked active.

```
tmp = 0U;
declare i : 6'U;
for i in 6'0U : 6'31U do
    tmp[i] = S0.u32[i & 6'60U +: 6'4U] != 0U
endfor;
D0.u32 = tmp;
SCC = D0.u32 != 0U
```

S_WQM_B64 7

Given an active pixel mask in a scalar input, calculate whole quad mode mask for that input, store the result into a scalar register and set SCC iff the result is nonzero.

In whole quad mode, if any pixel in a quad is active then all pixels of the quad are marked active.

```
tmp = 0ULL;
declare i : 6'U;
for i in 6'0U : 6'63U do
    tmp[i] = S0.u64[i & 6'60U +: 6'4U] != 0ULL
endfor;
D0.u64 = tmp;
SCC = D0.u64 != 0ULL
```

S_BREV_B32 8

Reverse the order of bits in a scalar input and store the result into a scalar register.

```
D0.u32[31 : 0] = S0.u32[0 : 31]
```

S_BREV_B64 9

Reverse the order of bits in a scalar input and store the result into a scalar register.

```
D0.u64[63 : 0] = S0.u64[0 : 63]
```

S_BCNT0_I32_B32 10

Count the number of "0" bits in a scalar input, store the result into a scalar register and set SCC iff the result is nonzero.

```
tmp = 0;
for i in 0 : 31 do
    tmp += S0.u32[i] == 1'0U ? 1 : 0
endfor;
D0.i32 = tmp;
SCC = D0.u32 != 0U
```

Notes

Functional examples:

```
S_BCNT0_I32_B32(0x00000000) => 32
S_BCNT0_I32_B32(0xcccccccc) => 16
S_BCNT0_I32_B32(0xffffffff) => 0
```

S_BCNT0_I32_B64 11

Count the number of "0" bits in a scalar input, store the result into a scalar register and set SCC iff the result is nonzero.

```
tmp = 0;
for i in 0 : 63 do
    tmp += S0.u64[i] == 1'0U ? 1 : 0
endfor;
```

```
D0.i32 = tmp;
SCC = D0.u64 != 0ULL
```

S_BCNT1_I32_B32 12

Count the number of "1" bits in a scalar input, store the result into a scalar register and set SCC iff the result is nonzero.

```
tmp = 0;
for i in 0 : 31 do
    tmp += S0.u32[i] == 1'1U ? 1 : 0
endfor;
D0.i32 = tmp;
SCC = D0.u32 != 0U
```

Notes

Functional examples:

```
S_BCNT1_I32_B32(0x00000000) => 0
S_BCNT1_I32_B32(0xcccccccc) => 16
S_BCNT1_I32_B32(0xffffffff) => 32
```

S_BCNT1_I32_B64 13

Count the number of "1" bits in a scalar input, store the result into a scalar register and set SCC iff the result is nonzero.

```
tmp = 0;
for i in 0 : 63 do
    tmp += S0.u64[i] == 1'1U ? 1 : 0
endfor;
D0.i32 = tmp;
SCC = D0.u64 != 0ULL
```

S_FF0_I32_B32 14

Count the number of trailing "1" bits before the first "0" in a scalar input and store the result into a scalar register. Store -1 if there are no "0" bits in the input.

```
tmp = -1;
// Set if no zeros are found
```

```
for i in 0 : 31 do
    // Search from LSB
    if S0.u32[i] == 1'0U then
        tmp = i;
        break
    endif
endfor;
D0.i32 = tmp
```

Notes

Functional examples:

```
S_FF0_I32_B32(0xaaaaaaaa) => 0
S_FF0_I32_B32(0x55555555) => 1
S_FF0_I32_B32(0x00000000) => 0
S_FF0_I32_B32(0xffffffff) => 0xffffffff
S_FF0_I32_B32(0xfffeffff) => 16
```

S_FF0_I32_B64 15

Count the number of trailing "1" bits before the first "0" in a scalar input and store the result into a scalar register. Store -1 if there are no "0" bits in the input.

```
tmp = -1;
// Set if no zeros are found
for i in 0 : 63 do
    // Search from LSB
    if S0.u64[i] == 1'0U then
        tmp = i;
        break
    endif
endfor;
D0.i32 = tmp
```

S_FF1_I32_B32 16

Count the number of trailing "0" bits before the first "1" in a scalar input and store the result into a scalar register. Store -1 if there are no "1" bits in the input.

```
tmp = -1;
// Set if no ones are found
for i in 0 : 31 do
    // Search from LSB
    if S0.u32[i] == 1'1U then
        tmp = i;
        break
```

```
    endif
endfor;
D0.i32 = tmp
```

Notes

Functional examples:

```
S_FF1_I32_B32(0xaaaaaaaa) => 1
S_FF1_I32_B32(0x55555555) => 0
S_FF1_I32_B32(0x00000000) => 0xffffffff
S_FF1_I32_B32(0xffffffff) => 0
S_FF1_I32_B32(0x00010000) => 16
```

S_FF1_I32_B64 17

Count the number of trailing "0" bits before the first "1" in a scalar input and store the result into a scalar register. Store -1 if there are no "1" bits in the input.

```
tmp = -1;
// Set if no ones are found
for i in 0 : 63 do
    // Search from LSB
    if S0.u64[i] == 1'1U then
        tmp = i;
        break
    endif
endfor;
D0.i32 = tmp
```

S_FLBIT_I32_B32 18

Count the number of leading "0" bits before the first "1" in a scalar input and store the result into a scalar register. Store -1 if there are no "1" bits.

```
tmp = -1;
// Set if no ones are found
for i in 0 : 31 do
    // Search from MSB
    if S0.u32[31 - i] == 1'1U then
        tmp = i;
        break
    endif
endfor;
D0.i32 = tmp
```

Notes

Functional examples:

```
S_FLBIT_I32_B32(0x00000000) => 0xffffffff
S_FLBIT_I32_B32(0x0000cccc) => 16
S_FLBIT_I32_B32(0xffff3333) => 0
S_FLBIT_I32_B32(0x7fffffff) => 1
S_FLBIT_I32_B32(0x80000000) => 0
S_FLBIT_I32_B32(0xffffffff) => 0
```

S_FLBIT_I32_B64 19

Count the number of leading "0" bits before the first "1" in a scalar input and store the result into a scalar register. Store -1 if there are no "1" bits.

```
tmp = -1;
// Set if no ones are found
for i in 0 : 63 do
    // Search from MSB
    if S0.u64[63 - i] == 1'1U then
        tmp = i;
        break
    endif
endfor;
D0.i32 = tmp
```

S_FLBIT_I32 20

Count the number of leading bits that are the same as the sign bit of a scalar input and store the result into a scalar register. Store -1 if all input bits are the same.

```
tmp = -1;
// Set if all bits are the same
for i in 1 : 31 do
    // Search from MSB
    if S0.u32[31 - i] != S0.u32[31] then
        tmp = i;
        break
    endif
endfor;
D0.i32 = tmp
```

Notes

Functional examples:

```
S_FLBIT_I32(0x00000000) => 0xffffffff
S_FLBIT_I32(0x0000cccc) => 16
S_FLBIT_I32(0xffff3333) => 16
S_FLBIT_I32(0x7fffffff) => 1
S_FLBIT_I32(0x80000000) => 1
S_FLBIT_I32(0xffffffff) => 0xffffffff
```

S_FLBIT_I32_I64 21

Count the number of leading bits that are the same as the sign bit of a scalar input and store the result into a scalar register. Store -1 if all input bits are the same.

```
tmp = -1;
// Set if all bits are the same
for i in 1 : 63 do
    // Search from MSB
    if S0.u64[63 - i] != S0.u64[63] then
        tmp = i;
        break
    endif
endfor;
D0.i32 = tmp
```

S_SEXT_I32_I8 22

Sign extend a signed 8 bit scalar input to 32 bits and store the result into a scalar register.

```
D0.i32 = 32'I(signext(S0.i8))
```

S_SEXT_I32_I16 23

Sign extend a signed 16 bit scalar input to 32 bits and store the result into a scalar register.

```
D0.i32 = 32'I(signext(S0.i16))
```

S_BITSET0_B32 24

Given a bit offset in a scalar input, set the indicated bit in the destination scalar register to 0.

```
D0.u32[S0.u32[4 : 0]] = 1'0U
```

S_BITSET0_B64 25

Given a bit offset in a scalar input, set the indicated bit in the destination scalar register to 0.

```
D0.u64[S0.u32[5 : 0]] = 1'0U
```

S_BITSET1_B32 26

Given a bit offset in a scalar input, set the indicated bit in the destination scalar register to 1.

```
D0.u32[S0.u32[4 : 0]] = 1'1U
```

S_BITSET1_B64 27

Given a bit offset in a scalar input, set the indicated bit in the destination scalar register to 1.

```
D0.u64[S0.u32[5 : 0]] = 1'1U
```

S_GETPC_B64 28

Store the address of the next instruction to a scalar register.

The byte address of the instruction immediately *following* this instruction is saved to the destination.

```
D0.i64 = PC + 4LL
```

Notes

This instruction must be 4 bytes.

S_SETPC_B64 29

Jump to an address specified in a scalar register.

The argument is a byte address of the instruction to jump to.

```
PC = S0.i64
```

S_SWAPPC_B64 30

Store the address of the next instruction to a scalar register and then jump to an address specified in the scalar input.

The argument is a byte address of the instruction to jump to. The byte address of the instruction immediately *following* this instruction is saved to the destination.

```
jump_addr = S0.i64;
D0.i64 = PC + 4LL;
PC = jump_addr.i64
```

Notes

This instruction must be 4 bytes.

S_RFE_B64 31

Return from the exception handler. Clear the wave's PRIV bit and then jump to an address specified by the scalar input.

The argument is a byte address of the instruction to jump to; this address is likely derived from the state passed into the trap handler.

This instruction may only be used within a trap handler.

```
WAVE_STATUS.PRIV = 1'0U;
PC = S0.i64
```

S_AND_SAVEEXEC_B64 32

Calculate bitwise AND on the scalar input and the EXEC mask, store the calculated result into the EXEC mask, set SCC iff the calculated result is nonzero and store the *original* value of the EXEC mask into the scalar destination register.

The original EXEC mask is saved to the destination SGPRs before the bitwise operation is performed.

```
saveexec = EXEC.u64;
```

```
EXEC.u64 = (S0.u64 & EXEC.u64);
D0.u64 = saveexec.u64;
SCC = EXEC.u64 != 0ULL
```

S_OR_SAVEEXEC_B64 33

Calculate bitwise OR on the scalar input and the EXEC mask, store the calculated result into the EXEC mask, set SCC iff the calculated result is nonzero and store the *original* value of the EXEC mask into the scalar destination register.

The original EXEC mask is saved to the destination SGPRs before the bitwise operation is performed.

```
saveexec = EXEC.u64;
EXEC.u64 = (S0.u64 | EXEC.u64);
D0.u64 = saveexec.u64;
SCC = EXEC.u64 != 0ULL
```

S_XOR_SAVEEXEC_B64 34

Calculate bitwise XOR on the scalar input and the EXEC mask, store the calculated result into the EXEC mask, set SCC iff the calculated result is nonzero and store the *original* value of the EXEC mask into the scalar destination register.

The original EXEC mask is saved to the destination SGPRs before the bitwise operation is performed.

```
saveexec = EXEC.u64;
EXEC.u64 = (S0.u64 ^ EXEC.u64);
D0.u64 = saveexec.u64;
SCC = EXEC.u64 != 0ULL
```

S_ANDN2_SAVEEXEC_B64 35

Calculate bitwise AND on the scalar input and the negation of the EXEC mask, store the calculated result into the EXEC mask, set SCC iff the calculated result is nonzero and store the *original* value of the EXEC mask into the scalar destination register.

The original EXEC mask is saved to the destination SGPRs before the bitwise operation is performed.

```
saveexec = EXEC.u64;
EXEC.u64 = (S0.u64 & ~EXEC.u64);
D0.u64 = saveexec.u64;
SCC = EXEC.u64 != 0ULL
```

S_ORN2_SAVEEXEC_B64 36

Calculate bitwise OR on the scalar input and the negation of the EXEC mask, store the calculated result into the EXEC mask, set SCC iff the calculated result is nonzero and store the *original* value of the EXEC mask into the scalar destination register.

The original EXEC mask is saved to the destination SGPRs before the bitwise operation is performed.

```
saveexec = EXEC.u64;
EXEC.u64 = (S0.u64 | ~EXEC.u64);
D0.u64 = saveexec.u64;
SCC = EXEC.u64 != 0ULL
```

S_NAND_SAVEEXEC_B64 37

Calculate bitwise NAND on the scalar input and the EXEC mask, store the calculated result into the EXEC mask, set SCC iff the calculated result is nonzero and store the *original* value of the EXEC mask into the scalar destination register.

```
saveexec = EXEC.u64;
EXEC.u64 = ~(S0.u64 & EXEC.u64);
D0.u64 = saveexec.u64;
SCC = EXEC.u64 != 0ULL
```

S_NOR_SAVEEXEC_B64 38

Calculate bitwise NOR on the scalar input and the EXEC mask, store the calculated result into the EXEC mask, set SCC iff the calculated result is nonzero and store the *original* value of the EXEC mask into the scalar destination register.

```
saveexec = EXEC.u64;
EXEC.u64 = ~(S0.u64 | EXEC.u64);
D0.u64 = saveexec.u64;
SCC = EXEC.u64 != 0ULL
```

S_XNOR_SAVEEXEC_B64 39

Calculate bitwise XNOR on the scalar input and the EXEC mask, store the calculated result into the EXEC mask, set SCC iff the calculated result is nonzero and store the *original* value of the EXEC mask into the scalar destination register.

```
saveexec = EXEC.u64;
EXEC.u64 = ~(S0.u64 ^ EXEC.u64);
D0.u64 = saveexec.u64;
SCC = EXEC.u64 != 0ULL
```

S_QUADMASK_B32 40

Reduce a pixel mask from the scalar input into a quad mask, store the result in a scalar register and set SCC iff the result is nonzero.

```
tmp = 0U;
for i in 0 : 7 do
    tmp[i] = S0.u32[i * 4 +: 4] != 0U
endfor;
D0.u32 = tmp;
SCC = D0.u32 != 0U
```

Notes

To perform the inverse operation see S_BITREPLICATE_B64_B32.

S_QUADMASK_B64 41

Reduce a pixel mask from the scalar input into a quad mask, store the result in a scalar register and set SCC iff the result is nonzero.

```
tmp = 0ULL;
for i in 0 : 15 do
    tmp[i] = S0.u64[i * 4 +: 4] != 0ULL
endfor;
D0.u64 = tmp;
SCC = D0.u64 != 0ULL
```

Notes

To perform the inverse operation see S_BITREPLICATE_B64_B32.

S_MOVRELS_B32 42

Move data from a relatively-indexed scalar register into another scalar register.

```
addr = SRC0.u32;
// Raw value from instruction
```

```
addr += M0.u32[31 : 0];
D0.b32 = SGPR[addr].b32
```

Notes

Example: The following instruction sequence performs the move s5 <= s17:

```
    s_mov_b32 m0, 10
    s_movrels_b32 s5, s7
```

S_MOVRELS_B64 43

Move data from a relatively-indexed scalar register into another scalar register.

The index in M0.u and the operand address in SRC0.u must be even for this operation.

```
addr = SRC0.u32;
// Raw value from instruction
addr += M0.u32[31 : 0];
D0.b64 = SGPR[addr].b64
```

S_MOVRELD_B32 44

Move data from a scalar input into a relatively-indexed scalar register.

```
addr = DST.u32;
// Raw value from instruction
addr += M0.u32[31 : 0];
SGPR[addr].b32 = S0.b32
```

Notes

Example: The following instruction sequence performs the move s15 <= s7:

```
    s_mov_b32 m0, 10
    s_movreld_b32 s5, s7
```

S_MOVRELD_B64 45

Move data from a scalar input into a relatively-indexed scalar register.

The index in M0.u and the operand address in DST.u must be even for this operation.

```
addr = DST.u32;
// Raw value from instruction
addr += M0.u32[31 : 0];
SGPR[addr].b64 = S0.b64
```

S_CBRANCH_JOIN 46

Conditional branch join point (end of conditional branch block).

S0 is saved CSP value. See S_CBRANCH_G_FORK and S_CBRANCH_I_FORK for related instructions.

```
saved_csp = S0.u32;
if WAVE_MODE.CSP.u32 == saved_csp then
    PC += 4LL;
    // Second time to JOIN: continue with program.
else
    WAVE_MODE.CSP -= 3'1U;
    // First time to JOIN; jump to other FORK path.
    { PC, EXEC } = SGPR[WAVE_MODE.CSP.u32 * 4U].b128;
    // Read 128 bits from 4 consecutive SGPRs.
endif
```

S_ABS_I32 48

Compute the absolute value of a scalar input, store the result into a scalar register and set SCC iff the result is nonzero.

```
D0.i32 = S0.i32 < 0 ? -S0.i32 : S0.i32;
SCC = D0.i32 != 0
```

Notes

Functional examples:

```
S_ABS_I32(0x00000001) => 0x00000001
S_ABS_I32(0x7fffffff) => 0x7fffffff
S_ABS_I32(0x80000000) => 0x80000000     // Note this is negative!
S_ABS_I32(0x80000001) => 0x7fffffff
S_ABS_I32(0x80000002) => 0x7ffffffe
S_ABS_I32(0xffffffff) => 0x00000001
```

S_SET_GPR_IDX_IDX 50

Set the index used in vector GPR indexing.

S_SET_GPR_IDX_ON, S_SET_GPR_IDX_OFF, S_SET_GPR_IDX_MODE and S_SET_GPR_IDX_IDX are related instructions.

```
M0[7 : 0] = S0.u32[7 : 0].b8
```

S_ANDN1_SAVEEXEC_B64 51

Calculate bitwise AND on the EXEC mask and the negation of the scalar input, store the calculated result into the EXEC mask, set SCC iff the calculated result is nonzero and store the *original* value of the EXEC mask into the scalar destination register.

The original EXEC mask is saved to the destination SGPRs before the bitwise operation is performed.

```
saveexec = EXEC.u64;
EXEC.u64 = (~S0.u64 & EXEC.u64);
D0.u64 = saveexec.u64;
SCC = EXEC.u64 != 0ULL
```

S_ORN1_SAVEEXEC_B64 52

Calculate bitwise OR on the EXEC mask and the negation of the scalar input, store the calculated result into the EXEC mask, set SCC iff the calculated result is nonzero and store the *original* value of the EXEC mask into the scalar destination register.

The original EXEC mask is saved to the destination SGPRs before the bitwise operation is performed.

```
saveexec = EXEC.u64;
EXEC.u64 = (~S0.u64 | EXEC.u64);
D0.u64 = saveexec.u64;
SCC = EXEC.u64 != 0ULL
```

S_ANDN1_WREXEC_B64 53

Calculate bitwise AND on the EXEC mask and the negation of the scalar input, store the calculated result into the EXEC mask and also into the scalar destination register, and set SCC iff the calculated result is nonzero.

Unlike the SAVEEXEC series of opcodes, the value written to destination SGPRs is the result of the bitwise-op result. EXEC and the destination SGPRs have the same value at the end of this instruction. This instruction is intended to help accelerate waterfalling.

```
EXEC.u64 = (~S0.u64 & EXEC.u64);
D0.u64 = EXEC.u64;
SCC = EXEC.u64 != 0ULL
```

S_ANDN2_WREXEC_B64 54

Calculate bitwise AND on the scalar input and the negation of the EXEC mask, store the calculated result into the EXEC mask and also into the scalar destination register, and set SCC iff the calculated result is nonzero.

Unlike the SAVEEXEC series of opcodes, the value written to destination SGPRs is the result of the bitwise-op result. EXEC and the destination SGPRs have the same value at the end of this instruction. This instruction is intended to help accelerate waterfalling.

```
EXEC.u64 = (S0.u64 & ~EXEC.u64);
D0.u64 = EXEC.u64;
SCC = EXEC.u64 != 0ULL
```

Notes

In particular, the following sequence of waterfall code is optimized by using a WREXEC instead of two separate scalar ops:

```
    // V0 holds the index value per lane
    // save exec mask for restore at the end
    s_mov_b64 s2, exec
    // exec mask of remaining (unprocessed) threads
    s_mov_b64 s4, exec
    loop:
    // get the index value for the first active lane
    v_readfirstlane_b32  s0, v0
    // find all other lanes with same index value
    v_cmpx_eq s0, v0
    <OP>        // do the operation using the current EXEC mask. S0 holds the index.
    // mask out thread that was just executed
    // s_andn2_b64  s4, s4, exec
    // s_mov_b64    exec, s4
    s_andn2_wrexec_b64 s4, s4     // replaces above 2 ops
    // repeat until EXEC==0
    s_cbranch_scc1  loop
    s_mov_b64    exec, s2
```

S_BITREPLICATE_B64_B32 55

Substitute each bit of a 32 bit scalar input with two instances of itself and store the result into a 64 bit scalar register.

```
tmp = S0.u32;
for i in 0 : 31 do
    D0.u64[i * 2] = tmp[i];
    D0.u64[i * 2 + 1] = tmp[i]
endfor
```

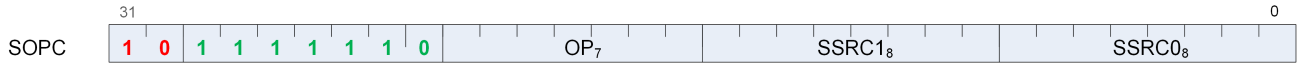
Notes

This opcode can be used to convert a quad mask into a pixel mask; given quad mask in s0, the following sequence produces a pixel mask in s2:

```
    s_bitreplicate_b64 s2, s0
    s_bitreplicate_b64 s2, s2
```

To perform the inverse operation see S_QUADMASK_B64.

12.4. SOPC Instructions



Instructions in this format may use a 32-bit literal constant which occurs immediately after the instruction.

S_CMP_EQ_I32 0

Set SCC to 1 iff the first scalar input is equal to the second scalar input.

```
SCC = S0.i32 == S1.i32
```

Notes

Note that S_CMP_EQ_I32 and S_CMP_EQ_U32 are identical opcodes, but both are provided for symmetry.

S_CMP_LG_I32 1

Set SCC to 1 iff the first scalar input is less than or greater than the second scalar input.

```
SCC = S0.i32 <> S1.i32
```

Notes

Note that S_CMP_LG_I32 and S_CMP_LG_U32 are identical opcodes, but both are provided for symmetry.

S_CMP_GT_I32 2

Set SCC to 1 iff the first scalar input is greater than the second scalar input.

```
SCC = S0.i32 > S1.i32
```

S_CMP_GE_I32 3

Set SCC to 1 iff the first scalar input is greater than or equal to the second scalar input.

```
SCC = S0.i32 >= S1.i32
```

S_CMP_LT_I32 4

Set SCC to 1 iff the first scalar input is less than the second scalar input.

```
SCC = S0.i32 < S1.i32
```

S_CMP_LE_I32 5

Set SCC to 1 iff the first scalar input is less than or equal to the second scalar input.

```
SCC = S0.i32 <= S1.i32
```

S_CMP_EQ_U32 6

Set SCC to 1 iff the first scalar input is equal to the second scalar input.

```
SCC = S0.u32 == S1.u32
```

Notes

Note that S_CMP_EQ_I32 and S_CMP_EQ_U32 are identical opcodes, but both are provided for symmetry.

S_CMP_LG_U32 7

Set SCC to 1 iff the first scalar input is less than or greater than the second scalar input.

```
SCC = S0.u32 <> S1.u32
```

Notes

Note that S_CMP_LG_I32 and S_CMP_LG_U32 are identical opcodes, but both are provided for symmetry.

S_CMP_GT_U32 8

Set SCC to 1 iff the first scalar input is greater than the second scalar input.

```
SCC = S0.u32 > S1.u32
```

S_CMP_GE_U32 9

Set SCC to 1 iff the first scalar input is greater than or equal to the second scalar input.

```
SCC = S0.u32 >= S1.u32
```

S_CMP_LT_U32 10

Set SCC to 1 iff the first scalar input is less than the second scalar input.

```
SCC = S0.u32 < S1.u32
```

S_CMP_LE_U32 11

Set SCC to 1 iff the first scalar input is less than or equal to the second scalar input.

```
SCC = S0.u32 <= S1.u32
```

S_BITCMP0_B32 12

Extract a bit from the first scalar input based on an index in the second scalar input, and set SCC to 1 iff the extracted bit is equal to 0.

```
SCC = S0.u32[S1.u32[4 : 0]] == 1'0U
```

S_BITCMP1_B32 13

Extract a bit from the first scalar input based on an index in the second scalar input, and set SCC to 1 iff the extracted bit is equal to 1.

```
SCC = S0.u32[S1.u32[4 : 0]] == 1'1U
```

S_BITCMP0_B64 14

Extract a bit from the first scalar input based on an index in the second scalar input, and set SCC to 1 iff the extracted bit is equal to 0.

```
SCC = S0.u64[S1.u32[5 : 0]] == 1'0U
```

S_BITCMP1_B64 15

Extract a bit from the first scalar input based on an index in the second scalar input, and set SCC to 1 iff the extracted bit is equal to 1.

```
SCC = S0.u64[S1.u32[5 : 0]] == 1'1U
```

S_SETVSKIP 16

Enables or disables VSKIP mode.

When VSKIP is enabled, no VOP*/M*BUF/MIMG/DS/FLAT instructions are issued. Note that VSKIPped memory instructions do not manipulate the waitcnt counters; as a result, if there are outstanding memory requests the shader may want to issue S_WAITCNT 0 prior to enabling VSKIP, otherwise the shader must be careful not to count VSKIPped instructions in waitcnt calculations.

```
VSKIP = S0.u32[S1.u32[4 : 0]]
```

Notes

Functional examples:

```
    s_setvskip 1, 0     // Enable vskip mode.
    s_setvskip 0, 0     // Disable vskip mode.
```

S_SET_GPR_IDX_ON 17

Enable GPR indexing mode.

Vector operations after this perform relative GPR addressing based on the contents of M0. The index is specified in the SRC0 operand. The raw bits of the SRC1 field are read and used to set the enable bits. S1[0] = VSRC0_REL, S1[1] = VSRC1_REL, S1[2] = VSRC2_REL and S1[3] = VDST_REL.

S_SET_GPR_IDX_ON, S_SET_GPR_IDX_OFF, S_SET_GPR_IDX_MODE and S_SET_GPR_IDX_IDX are related instructions.

```
WAVE_MODE.GPR_IDX_EN = 1'1U;
M0[7 : 0] = S0.u32[7 : 0].b8;
M0[15 : 12] = SRC1.u32[3 : 0].b4;
// this is the direct content of raw S1 field
// Remaining bits of M0 are unmodified.
```

S_CMP_EQ_U64 18

Set SCC to 1 iff the first scalar input is equal to the second scalar input.

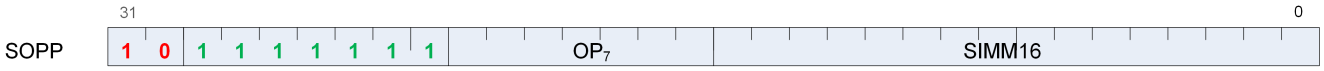
```
SCC = S0.u64 == S1.u64
```

S_CMP_LG_U64 19

Set SCC to 1 iff the first scalar input is less than or greater than the second scalar input.

```
SCC = S0.u64 <> S1.u64
```

12.5. SOPP Instructions



S_NOP 0

Do nothing. Delay issue of next instruction by a small, fixed amount.

Insert 0..15 wait states based on SIMM16[3:0]. 0x0 means the next instruction can issue on the next clock, 0xf means the next instruction can issue 16 clocks later.

```
for i in 0U : SIMM16.u16[3 : 0].u32 do
    nop()
endfor
```

Notes

Examples:

```
    s_nop 0         // Wait 1 cycle.
    s_nop 0xf       // Wait 16 cycles.
```

S_ENDPGM 1

End of program; terminate wavefront.

The hardware implicitly executes S_WAITCNT 0 before executing this instruction. See S_ENDPGM_SAVED for the context-switch version of this instruction and S_ENDPGM_ORDERED_PS_DONE for the POPS critical region version of this instruction.

S_BRANCH 2

Jump to a constant offset relative to the current PC.

The literal argument is a signed DWORD offset relative to the PC of the next instruction.

```
PC = PC + signext(SIMM16.i16 * 16'4) + 4LL;
// short jump.
```

Notes

For a long jump or an indirect jump use S_SETPC_B64.

Examples:

```
    s_branch label    // Set SIMM16 = +4 = 0x0004
    s_nop 0    // 4 bytes
label:
    s_nop 0    // 4 bytes
    s_branch label    // Set SIMM16 = -8 = 0xfff8
```

S_WAKEUP 3

Allow a wave to 'ping' all the other waves in its threadgroup to force them to wake up early from an S_SLEEP instruction.

The ping is ignored if the waves are not sleeping. This allows for efficient polling on a memory location. The waves which are polling can sit in a long S_SLEEP between memory reads, but the wave which writes the value can tell them all to wake up early now that the data is available. This method is also safe from races since any waves that miss the ping resume when they complete their S_SLEEP.

If the wave executing S_WAKEUP is in a threadgroup (in_tg set), then it wakes up all waves associated with the same threadgroup ID. Otherwise, S_WAKEUP is treated as an S_NOP.

S_CBRANCH_SCC0 4

If SCC is 0 then jump to a constant offset relative to the current PC.

The literal argument is a signed DWORD offset relative to the PC of the next instruction.

```
if SCC == 1'0U then
    PC = PC + signext(SIMM16.i16 * 16'4) + 4LL
else
    PC = PC + 4LL
endif
```

S_CBRANCH_SCC1 5

If SCC is 1 then jump to a constant offset relative to the current PC.

The literal argument is a signed DWORD offset relative to the PC of the next instruction.

```
if SCC == 1'1U then
    PC = PC + signext(SIMM16.i16 * 16'4) + 4LL
else
    PC = PC + 4LL
```

```
endif
```

S_CBRANCH_VCCZ 6

If VCCZ is 1 then jump to a constant offset relative to the current PC.

The literal argument is a signed DWORD offset relative to the PC of the next instruction.

```
if VCCZ.u1 == 1'1U then
    PC = PC + signext(SIMM16.i16 * 16'4) + 4LL
else
    PC = PC + 4LL
endif
```

S_CBRANCH_VCCNZ 7

If VCCZ is 0 then jump to a constant offset relative to the current PC.

The literal argument is a signed DWORD offset relative to the PC of the next instruction.

```
if VCCZ.u1 == 1'0U then
    PC = PC + signext(SIMM16.i16 * 16'4) + 4LL
else
    PC = PC + 4LL
endif
```

S_CBRANCH_EXECZ 8

If EXECZ is 1 then jump to a constant offset relative to the current PC.

The literal argument is a signed DWORD offset relative to the PC of the next instruction.

```
if EXECZ.u1 == 1'1U then
    PC = PC + signext(SIMM16.i16 * 16'4) + 4LL
else
    PC = PC + 4LL
endif
```

S_CBRANCH_EXECNZ 9

If EXECZ is 0 then jump to a constant offset relative to the current PC.

The literal argument is a signed DWORD offset relative to the PC of the next instruction.

```
if EXECZ.u1 == 1'0U then
    PC = PC + signext(SIMM16.i16 * 16'4) + 4LL
else
    PC = PC + 4LL
endif
```

S_BARRIER 10

Synchronize waves within a threadgroup.

If not all waves of the threadgroup have been created yet, waits for entire group before proceeding. If some waves in the threadgroup have already terminated, this waits on only the surviving waves. Barriers are legal inside trap handlers.

Barrier instructions do not wait for any counters to go to zero before issuing. If the barrier is being used to protect an outstanding memory operation use the appropriate S_WAITCNT instruction before the barrier.

S_SETKILL 11

Kill this wave if the least significant bit of the immediate constant is 1.

Used primarily for debugging kill wave host command behavior.

S_WAITCNT 12

Wait for the counts of outstanding local data share, vector memory and export instructions to be at or below the specified levels.

```
SIMM16[3:0] = vmcount (vector memory operations) lower bits [3:0],
```

```
SIMM16[6:4] = export/mem-write-data count,
```

```
SIMM16[11:8] = LGKMcnt (scalar-mem/GDS/LDS count),
```

```
SIMM16[15:14] = vmcount (vector memory operations) upper bits [5:4].
```

S_SETHALT 13

Set or clear the HALT status bit.

Set HALT bit to value of SIMM16[0]; 1 = halt, 0 = clear HALT bit. The halt flag is ignored while PRIV == 1 (inside trap handlers) but the shader halts after the handler returns if HALT is still set at that time.

S_SLEEP 14

Cause a wave to sleep for up to ~8000 clocks.

The wave sleeps for (64*(SIMM16[6:0]-1) .. 64*SIMM16[6:0]) clocks. The exact amount of delay is approximate. Compare with S_NOP. When SIMM16[6:0] is zero then no sleep occurs.

Notes

Examples:

```
s_sleep 0       // Wait for 0 clocks.
s_sleep 1       // Wait for 1-64 clocks.
s_sleep 2       // Wait for 65-128 clocks.
```

S_SETPRIO 15

Change wave user priority.

User settable wave priority is set to SIMM16[1:0]. 0 = lowest, 3 = highest. The overall wave priority is {SPIPrio[1:0], UserPrio[1:0], WaveAge[3:0]}.

S_SENDMSG 16

Send a message upstream to graphics control hardware.

SIMM16[9:0] contains the message type.

Notes

S_SENDMSGHALT 17

Send a message to upstream control hardware and then HALT the wavefront; see S_SENDMSG for details.

S_TRAP 18

Enter the trap handler.

This instruction may be generated internally as well in response to a host trap (HT = 1) or an exception. TrapID 0 is reserved for hardware use and should not be used in a shader-generated trap.

```
TrapID = SIMM16.u16[7 : 0];
"Wait for all instructions to complete";
// PC passed into trap handler points to S_TRAP itself,
// *not* to the next instruction.
{ TTMP[1], TTMP[0] } = { 3'0, PCRewind[3 : 0], HT[0], TrapID[7 : 0], PC[47 : 0] };
PC = TBA.i64;
// trap base address
WAVE_STATUS.PRIV = 1'1U
```

S_ICACHE_INV 19

Invalidate entire first level instruction cache.

There must be 16 separate S_NOP instructions or a jump/branch instruction after this instruction to ensure the internal instruction buffers are also invalidated.

S_INCPERFLEVEL 20

Increment performance counter specified in SIMM16[3:0] by 1.

S_DECPERFLEVEL 21

Decrement performance counter specified in SIMM16[3:0] by 1.

S_CBRANCH_CDBGSYS 23

If the system debug flag is set then jump to a constant offset relative to the current PC.

The literal argument is a signed DWORD offset relative to the PC of the next instruction.

```
if WAVE_STATUS.COND_DBG_SYS.u32 != 0U then
    PC = PC + signext(SIMM16.i16 * 16'4) + 4LL
else
    PC = PC + 4LL
endif
```

S_CBRANCH_CDBGUSER 24

If the user debug flag is set then jump to a constant offset relative to the current PC.

The literal argument is a signed DWORD offset relative to the PC of the next instruction.

```
if WAVE_STATUS.COND_DBG_USER.u32 != 0U then
    PC = PC + signext(SIMM16.i16 * 16'4) + 4LL
else
    PC = PC + 4LL
endif
```

S_CBRANCH_CDBGSYS_OR_USER 25

If either the system debug flag or the user debug flag is set then jump to a constant offset relative to the current PC.

The literal argument is a signed DWORD offset relative to the PC of the next instruction.

```
if (WAVE_STATUS.COND_DBG_SYS || WAVE_STATUS.COND_DBG_USER) then
    PC = PC + signext(SIMM16.i16 * 16'4) + 4LL
else
    PC = PC + 4LL
endif
```

S_CBRANCH_CDBGSYS_AND_USER 26

If both the system debug flag and the user debug flag are set then jump to a constant offset relative to the current PC.

The literal argument is a signed DWORD offset relative to the PC of the next instruction.

```
if (WAVE_STATUS.COND_DBG_SYS && WAVE_STATUS.COND_DBG_USER) then
    PC = PC + signext(SIMM16.i16 * 16'4) + 4LL
else
    PC = PC + 4LL
endif
```

S_ENDPGM_SAVED 27

End of program; signal that a wave has been saved by the context-switch trap handler and terminate wavefront.

The hardware implicitly executes S_WAITCNT 0 before executing this instruction.

See S_ENDPGM for additional variants.

S_SET_GPR_IDX_OFF 28

Clear GPR indexing mode.

Vector operations after this do not perform relative GPR addressing regardless of the contents of M0. This instruction does not modify M0.

S_SET_GPR_IDX_ON, S_SET_GPR_IDX_OFF, S_SET_GPR_IDX_MODE and S_SET_GPR_IDX_IDX are related instructions.

```
WAVE_MODE.GPR_IDX_EN = 1'0U
```

S_SET_GPR_IDX_MODE 29

Modify the mode used for vector GPR indexing.

The raw contents of the source field are read and used to set the enable bits. SIMM16[0] = VSRC0_REL, SIMM16[1] = VSRC1_REL, SIMM16[2] = VSRC2_REL and SIMM16[3] = VDST_REL.

S_SET_GPR_IDX_ON, S_SET_GPR_IDX_OFF, S_SET_GPR_IDX_MODE and S_SET_GPR_IDX_IDX are related instructions.

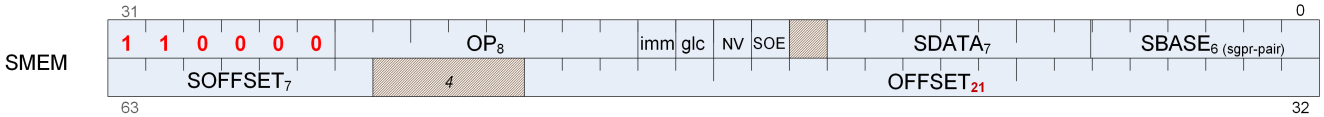
```
M0[15 : 12] = SIMM16.u16[3 : 0].b4
```

12.5.1. Send Message

The S_SENDMSG instruction encodes the message type in M0, and can also send data from the SIMM16 field and in some cases from EXEC.

Message	SIMM16[3:0]	SIMM16[6:4]	Payload
none	0	-	illegal
Interrupt	1	-	M0[23:0] carries data payload
Save wave	4	-	used in context switching
Stall Wave Gen	5	-	stop new wave generation
Halt Waves	6	-	halt all running waves of this vmid
Get Doorbell ID	10	-	Returns doorbell into EXEC, with the doorbell physical address in bits [12:3].

12.6. SMEM Instructions



S_LOAD_DWORD 0

Load 32 bits of data from the scalar memory into a scalar register.

```
SDATA[31 : 0] = MEM[ADDR].b32
```

Notes

If the offset is specified as an SGPR, the SGPR contains an UNSIGNED BYTE offset (the 2 LSBs are ignored).

If the offset is specified as an immediate 21-bit constant, the constant is a SIGNED BYTE offset.

S_LOAD_DWORDX2 1

Load 64 bits of data from the scalar memory into a scalar register.

```
SDATA[31 : 0] = MEM[ADDR].b32;
SDATA[63 : 32] = MEM[ADDR + 4U].b32
```

Notes

If the offset is specified as an SGPR, the SGPR contains an UNSIGNED BYTE offset (the 2 LSBs are ignored).

If the offset is specified as an immediate 21-bit constant, the constant is a SIGNED BYTE offset.

S_LOAD_DWORDX4 2

Load 128 bits of data from the scalar memory into a scalar register.

```
SDATA[31 : 0] = MEM[ADDR].b32;
SDATA[63 : 32] = MEM[ADDR + 4U].b32;
SDATA[95 : 64] = MEM[ADDR + 8U].b32;
SDATA[127 : 96] = MEM[ADDR + 12U].b32
```

Notes

If the offset is specified as an SGPR, the SGPR contains an UNSIGNED BYTE offset (the 2 LSBs are ignored).

If the offset is specified as an immediate 21-bit constant, the constant is a SIGNED BYTE offset.

S_LOAD_DWORDX8 3

Load 256 bits of data from the scalar memory into a scalar register.

```
SDATA[31 : 0] = MEM[ADDR].b32;
SDATA[63 : 32] = MEM[ADDR + 4U].b32;
SDATA[95 : 64] = MEM[ADDR + 8U].b32;
SDATA[127 : 96] = MEM[ADDR + 12U].b32;
SDATA[159 : 128] = MEM[ADDR + 16U].b32;
SDATA[191 : 160] = MEM[ADDR + 20U].b32;
SDATA[223 : 192] = MEM[ADDR + 24U].b32;
SDATA[255 : 224] = MEM[ADDR + 28U].b32
```

Notes

If the offset is specified as an SGPR, the SGPR contains an UNSIGNED BYTE offset (the 2 LSBs are ignored).

If the offset is specified as an immediate 21-bit constant, the constant is a SIGNED BYTE offset.

S_LOAD_DWORDX16 4

Load 512 bits of data from the scalar memory into a scalar register.

```
SDATA[31 : 0] = MEM[ADDR].b32;
SDATA[63 : 32] = MEM[ADDR + 4U].b32;
SDATA[95 : 64] = MEM[ADDR + 8U].b32;
SDATA[127 : 96] = MEM[ADDR + 12U].b32;
SDATA[159 : 128] = MEM[ADDR + 16U].b32;
SDATA[191 : 160] = MEM[ADDR + 20U].b32;
SDATA[223 : 192] = MEM[ADDR + 24U].b32;
SDATA[255 : 224] = MEM[ADDR + 28U].b32;
```

```
SDATA[287 : 256] = MEM[ADDR + 32U].b32;
SDATA[319 : 288] = MEM[ADDR + 36U].b32;
SDATA[351 : 320] = MEM[ADDR + 40U].b32;
SDATA[383 : 352] = MEM[ADDR + 44U].b32;
SDATA[415 : 384] = MEM[ADDR + 48U].b32;
SDATA[447 : 416] = MEM[ADDR + 52U].b32;
SDATA[479 : 448] = MEM[ADDR + 56U].b32;
SDATA[511 : 480] = MEM[ADDR + 60U].b32
```

Notes

If the offset is specified as an SGPR, the SGPR contains an UNSIGNED BYTE offset (the 2 LSBs are ignored).

If the offset is specified as an immediate 21-bit constant, the constant is a SIGNED BYTE offset.

S_SCRATCH_LOAD_DWORD 5

Load 32 bits of data from the scalar scratch aperture into a scalar register.

Notes

If the offset is specified as an SGPR, the SGPR contains an UNSIGNED 64-byte offset, consistent with other scratch operations.

If the offset is specified as an immediate 21-bit constant, the constant is a SIGNED BYTE offset.

S_SCRATCH_LOAD_DWORDX2 6

Load 64 bits of data from the scalar scratch aperture into a scalar register.

Notes

If the offset is specified as an SGPR, the SGPR contains an UNSIGNED 64-byte offset, consistent with other scratch operations.

If the offset is specified as an immediate 21-bit constant, the constant is a SIGNED BYTE offset.

S_SCRATCH_LOAD_DWORDX4 7

Load 128 bits of data from the scalar scratch aperture into a scalar register.

Notes

If the offset is specified as an SGPR, the SGPR contains an UNSIGNED 64-byte offset, consistent with other scratch operations.

If the offset is specified as an immediate 21-bit constant, the constant is a SIGNED BYTE offset.

S_BUFFER_LOAD_DWORD 8

Load 32 bits of data from a scalar buffer surface into a scalar register.

```
SDATA[31 : 0] = MEM[ADDR].b32
```

Notes

If the offset is specified as an SGPR, the SGPR contains an UNSIGNED BYTE offset (the 2 LSBs are ignored).

If the offset is specified as an immediate 21-bit constant, the constant is a SIGNED BYTE offset.

S_BUFFER_LOAD_DWORDX2 9

Load 64 bits of data from a scalar buffer surface into a scalar register.

```
SDATA[31 : 0] = MEM[ADDR].b32;
SDATA[63 : 32] = MEM[ADDR + 4U].b32
```

Notes

If the offset is specified as an SGPR, the SGPR contains an UNSIGNED BYTE offset (the 2 LSBs are ignored).

If the offset is specified as an immediate 21-bit constant, the constant is a SIGNED BYTE offset.

S_BUFFER_LOAD_DWORDX4 10

Load 128 bits of data from a scalar buffer surface into a scalar register.

```
SDATA[31 : 0] = MEM[ADDR].b32;
SDATA[63 : 32] = MEM[ADDR + 4U].b32;
SDATA[95 : 64] = MEM[ADDR + 8U].b32;
SDATA[127 : 96] = MEM[ADDR + 12U].b32
```

Notes

If the offset is specified as an SGPR, the SGPR contains an UNSIGNED BYTE offset (the 2 LSBs are ignored).

If the offset is specified as an immediate 21-bit constant, the constant is a SIGNED BYTE offset.

S_BUFFER_LOAD_DWORDX8 11

Load 256 bits of data from a scalar buffer surface into a scalar register.

```
SDATA[31 : 0] = MEM[ADDR].b32;
SDATA[63 : 32] = MEM[ADDR + 4U].b32;
SDATA[95 : 64] = MEM[ADDR + 8U].b32;
SDATA[127 : 96] = MEM[ADDR + 12U].b32;
SDATA[159 : 128] = MEM[ADDR + 16U].b32;
SDATA[191 : 160] = MEM[ADDR + 20U].b32;
SDATA[223 : 192] = MEM[ADDR + 24U].b32;
SDATA[255 : 224] = MEM[ADDR + 28U].b32
```

Notes

If the offset is specified as an SGPR, the SGPR contains an UNSIGNED BYTE offset (the 2 LSBs are ignored).

If the offset is specified as an immediate 21-bit constant, the constant is a SIGNED BYTE offset.

S_BUFFER_LOAD_DWORDX16 12

Load 512 bits of data from a scalar buffer surface into a scalar register.

```
SDATA[31 : 0] = MEM[ADDR].b32;
SDATA[63 : 32] = MEM[ADDR + 4U].b32;
SDATA[95 : 64] = MEM[ADDR + 8U].b32;
SDATA[127 : 96] = MEM[ADDR + 12U].b32;
SDATA[159 : 128] = MEM[ADDR + 16U].b32;
SDATA[191 : 160] = MEM[ADDR + 20U].b32;
SDATA[223 : 192] = MEM[ADDR + 24U].b32;
SDATA[255 : 224] = MEM[ADDR + 28U].b32;
SDATA[287 : 256] = MEM[ADDR + 32U].b32;
SDATA[319 : 288] = MEM[ADDR + 36U].b32;
SDATA[351 : 320] = MEM[ADDR + 40U].b32;
SDATA[383 : 352] = MEM[ADDR + 44U].b32;
SDATA[415 : 384] = MEM[ADDR + 48U].b32;
SDATA[447 : 416] = MEM[ADDR + 52U].b32;
SDATA[479 : 448] = MEM[ADDR + 56U].b32;
SDATA[511 : 480] = MEM[ADDR + 60U].b32
```

Notes

If the offset is specified as an SGPR, the SGPR contains an UNSIGNED BYTE offset (the 2 LSBs are ignored).

If the offset is specified as an immediate 21-bit constant, the constant is a SIGNED BYTE offset.

S_STORE_DWORD 16

Store 32 bits of data from a scalar register into the scalar memory.

```
MEM[ADDR].b32 = SDATA[31 : 0]
```

Notes

If the offset is specified as an SGPR, the SGPR contains an UNSIGNED BYTE offset (the 2 LSBs are ignored).

If the offset is specified as an immediate 21-bit constant, the constant is a SIGNED BYTE offset.

S_STORE_DWORDX2 17

Store 64 bits of data from a scalar register into the scalar memory.

```
MEM[ADDR].b32 = SDATA[31 : 0];
MEM[ADDR + 4U].b32 = SDATA[63 : 32]
```

Notes

If the offset is specified as an SGPR, the SGPR contains an UNSIGNED BYTE offset (the 2 LSBs are ignored).

If the offset is specified as an immediate 21-bit constant, the constant is a SIGNED BYTE offset.

S_STORE_DWORDX4 18

Store 128 bits of data from a scalar register into the scalar memory.

```
MEM[ADDR].b32 = SDATA[31 : 0];
MEM[ADDR + 4U].b32 = SDATA[63 : 32];
MEM[ADDR + 8U].b32 = SDATA[95 : 64];
MEM[ADDR + 12U].b32 = SDATA[127 : 96]
```

Notes

If the offset is specified as an SGPR, the SGPR contains an UNSIGNED BYTE offset (the 2 LSBs are ignored).

If the offset is specified as an immediate 21-bit constant, the constant is a SIGNED BYTE offset.

S_SCRATCH_STORE_DWORD 21

Store 32 bits of data from a scalar register into the scalar scratch aperture.

```
MEM[ADDR].b32 = SDATA[31 : 0]
```

Notes

If the offset is specified as an SGPR, the SGPR contains an UNSIGNED 64-byte offset, consistent with other scratch operations.

If the offset is specified as an immediate 21-bit constant, the constant is a SIGNED BYTE offset.

S_SCRATCH_STORE_DWORDX2 22

Store 64 bits of data from a scalar register into the scalar scratch aperture.

```
MEM[ADDR].b32 = SDATA[31 : 0];
MEM[ADDR + 4U].b32 = SDATA[63 : 32]
```

Notes

If the offset is specified as an SGPR, the SGPR contains an UNSIGNED 64-byte offset, consistent with other scratch operations.

If the offset is specified as an immediate 21-bit constant, the constant is a SIGNED BYTE offset.

S_SCRATCH_STORE_DWORDX4 23

Store 128 bits of data from a scalar register into the scalar scratch aperture.

```
MEM[ADDR].b32 = SDATA[31 : 0];
MEM[ADDR + 4U].b32 = SDATA[63 : 32];
MEM[ADDR + 8U].b32 = SDATA[95 : 64];
MEM[ADDR + 12U].b32 = SDATA[127 : 96]
```

Notes

If the offset is specified as an SGPR, the SGPR contains an UNSIGNED 64-byte offset, consistent with other scratch operations.

If the offset is specified as an immediate 21-bit constant, the constant is a SIGNED BYTE offset.

S_BUFFER_STORE_DWORD 24

Store 32 bits of data from a scalar register into a scalar buffer surface.

```
MEM[ADDR].b32 = SDATA[31 : 0]
```

Notes

If the offset is specified as an SGPR, the SGPR contains an UNSIGNED BYTE offset (the 2 LSBs are ignored).

If the offset is specified as an immediate 21-bit constant, the constant is a SIGNED BYTE offset.

S_BUFFER_STORE_DWORDX2 25

Store 64 bits of data from a scalar register into a scalar buffer surface.

```
MEM[ADDR].b32 = SDATA[31 : 0];
MEM[ADDR + 4U].b32 = SDATA[63 : 32]
```

Notes

If the offset is specified as an SGPR, the SGPR contains an UNSIGNED BYTE offset (the 2 LSBs are ignored).

If the offset is specified as an immediate 21-bit constant, the constant is a SIGNED BYTE offset.

S_BUFFER_STORE_DWORDX4 26

Store 128 bits of data from a scalar register into a scalar buffer surface.

```
MEM[ADDR].b32 = SDATA[31 : 0];
MEM[ADDR + 4U].b32 = SDATA[63 : 32];
MEM[ADDR + 8U].b32 = SDATA[95 : 64];
MEM[ADDR + 12U].b32 = SDATA[127 : 96]
```

Notes

If the offset is specified as an SGPR, the SGPR contains an UNSIGNED BYTE offset (the 2 LSBs are ignored).

If the offset is specified as an immediate 21-bit constant, the constant is a SIGNED BYTE offset.

S_DCACHE_INV 32

Invalidate the scalar (L0) data cache.

S_DCACHE_WB 33

Write back dirty data in the scalar (L0) data cache.

S_DCACHE_INV_VOL 34

Invalidate the scalar (L0) data cache volatile lines.

S_DCACHE_WB_VOL 35

Write back dirty data in the scalar (L0) data cache volatile lines.

S_MEMTIME 36

Return current 64-bit timestamp.

S_MEMREALTIME 37

Return current 64-bit RTC.

S_DCACHE_DISCARD 40

Discard one dirty scalar (L0) data cache line. A cache line is 64 bytes.

Typically, dirty cachelines (one which have been written by the shader) are written back to memory, but this instruction allows the shader to invalidate and not write back cachelines which it has previously written. This is a performance optimization to be used when the shader knows it no longer needs that data.

Address is calculated the same as S_STORE_DWORD, except the 6 LSBs are ignored to get the 64 byte aligned address. LGKM count is incremented by 1 for this opcode.

S_DCACHE_DISCARD_X2 41

Discard two consecutive dirty scalar (L0) data cache lines. A cache line is 64 bytes.

Typically, dirty cachelines (one which have been written by the shader) are written back to memory, but this instruction allows the shader to invalidate and not write back cachelines which it has previously written. This is a performance optimization to be used when the shader knows it no longer needs that data.

Address is calculated the same as S_STORE_DWORD, except the 6 LSBs are ignored to get the 64 byte aligned address. LGKM count is incremented by 2 for this opcode.

S_BUFFER_ATOMIC_SWAP 64

Swap an unsigned 32-bit integer value in the data register with a location in a scalar buffer surface.

```
tmp = MEM[ADDR].b32;
MEM[ADDR].b32 = DATA.b32;
RETURN_DATA.b32 = tmp
```

S_BUFFER_ATOMIC_CMPSWAP 65

Compare two unsigned 32-bit integer values stored in the data comparison register and a location in a scalar buffer surface. Modify the memory location with a value in the data source register iff the comparison is equal.

```
tmp = MEM[ADDR].u32;
src = DATA[31 : 0].u32;
cmp = DATA[63 : 32].u32;
MEM[ADDR].u32 = tmp == cmp ? src : tmp;
RETURN_DATA.u32 = tmp
```

S_BUFFER_ATOMIC_ADD 66

Add two unsigned 32-bit integer values stored in the data register and a location in a scalar buffer surface.

```
tmp = MEM[ADDR].u32;
MEM[ADDR].u32 += DATA.u32;
RETURN_DATA.u32 = tmp
```

S_BUFFER_ATOMIC_SUB 67

Subtract an unsigned 32-bit integer value stored in the data register from a value stored in a location in a scalar buffer surface.

```
tmp = MEM[ADDR].u32;
MEM[ADDR].u32 -= DATA.u32;
RETURN_DATA.u32 = tmp
```

S_BUFFER_ATOMIC_SMIN 68

Select the minimum of two signed 32-bit integer inputs, given two values stored in the data register and a location in a scalar buffer surface.

```
tmp = MEM[ADDR].i32;
src = DATA.i32;
```

```
MEM[ADDR].i32 = src < tmp ? src : tmp;
RETURN_DATA.i32 = tmp
```

S_BUFFER_ATOMIC_UMIN 69

Select the minimum of two unsigned 32-bit integer inputs, given two values stored in the data register and a location in a scalar buffer surface.

```
tmp = MEM[ADDR].u32;
src = DATA.u32;
MEM[ADDR].u32 = src < tmp ? src : tmp;
RETURN_DATA.u32 = tmp
```

S_BUFFER_ATOMIC_SMAX 70

Select the maximum of two signed 32-bit integer inputs, given two values stored in the data register and a location in a scalar buffer surface.

```
tmp = MEM[ADDR].i32;
src = DATA.i32;
MEM[ADDR].i32 = src >= tmp ? src : tmp;
RETURN_DATA.i32 = tmp
```

S_BUFFER_ATOMIC_UMAX 71

Select the maximum of two unsigned 32-bit integer inputs, given two values stored in the data register and a location in a scalar buffer surface.

```
tmp = MEM[ADDR].u32;
src = DATA.u32;
MEM[ADDR].u32 = src >= tmp ? src : tmp;
RETURN_DATA.u32 = tmp
```

S_BUFFER_ATOMIC_AND 72

Calculate bitwise AND given two unsigned 32-bit integer values stored in the data register and a location in a scalar buffer surface.

```
tmp = MEM[ADDR].b32;
MEM[ADDR].b32 = (tmp & DATA.b32);
```

```
RETURN_DATA.b32 = tmp
```

S_BUFFER_ATOMIC_OR 73

Calculate bitwise OR given two unsigned 32-bit integer values stored in the data register and a location in a scalar buffer surface.

```
tmp = MEM[ADDR].b32;
MEM[ADDR].b32 = (tmp | DATA.b32);
RETURN_DATA.b32 = tmp
```

S_BUFFER_ATOMIC_XOR 74

Calculate bitwise XOR given two unsigned 32-bit integer values stored in the data register and a location in a scalar buffer surface.

```
tmp = MEM[ADDR].b32;
MEM[ADDR].b32 = (tmp ^ DATA.b32);
RETURN_DATA.b32 = tmp
```

S_BUFFER_ATOMIC_INC 75

Increment an unsigned 32-bit integer value from a location in a scalar buffer surface with wraparound to 0 if the value exceeds a value in the data register.

```
tmp = MEM[ADDR].u32;
src = DATA.u32;
MEM[ADDR].u32 = tmp >= src ? 0U : tmp + 1U;
RETURN_DATA.u32 = tmp
```

S_BUFFER_ATOMIC_DEC 76

Decrement an unsigned 32-bit integer value from a location in a scalar buffer surface with wraparound to a value in the data register if the decrement yields a negative value.

```
tmp = MEM[ADDR].u32;
src = DATA.u32;
MEM[ADDR].u32 = ((tmp == 0U) || (tmp > src)) ? src : tmp - 1U;
RETURN_DATA.u32 = tmp
```

S_BUFFER_ATOMIC_SWAP_X2 96

Swap an unsigned 64-bit integer value in the data register with a location in a scalar buffer surface.

```
tmp = MEM[ADDR].b64;
MEM[ADDR].b64 = DATA.b64;
RETURN_DATA.b64 = tmp
```

S_BUFFER_ATOMIC_CMPSWAP_X2 97

Compare two unsigned 64-bit integer values stored in the data comparison register and a location in a scalar buffer surface. Modify the memory location with a value in the data source register iff the comparison is equal.

```
tmp = MEM[ADDR].u64;
src = DATA[63 : 0].u64;
cmp = DATA[127 : 64].u64;
MEM[ADDR].u64 = tmp == cmp ? src : tmp;
RETURN_DATA.u64 = tmp
```

S_BUFFER_ATOMIC_ADD_X2 98

Add two unsigned 64-bit integer values stored in the data register and a location in a scalar buffer surface.

```
tmp = MEM[ADDR].u64;
MEM[ADDR].u64 += DATA.u64;
RETURN_DATA.u64 = tmp
```

S_BUFFER_ATOMIC_SUB_X2 99

Subtract an unsigned 64-bit integer value stored in the data register from a value stored in a location in a scalar buffer surface.

```
tmp = MEM[ADDR].u64;
MEM[ADDR].u64 -= DATA.u64;
RETURN_DATA.u64 = tmp
```

S_BUFFER_ATOMIC_SMIN_X2 100

Select the minimum of two signed 64-bit integer inputs, given two values stored in the data register and a location in a scalar buffer surface.

```
tmp = MEM[ADDR].i64;
src = DATA.i64;
MEM[ADDR].i64 = src < tmp ? src : tmp;
RETURN_DATA.i64 = tmp
```

S_BUFFER_ATOMIC_UMIN_X2 101

Select the minimum of two unsigned 64-bit integer inputs, given two values stored in the data register and a location in a scalar buffer surface.

```
tmp = MEM[ADDR].u64;
src = DATA.u64;
MEM[ADDR].u64 = src < tmp ? src : tmp;
RETURN_DATA.u64 = tmp
```

S_BUFFER_ATOMIC_SMAX_X2 102

Select the maximum of two signed 64-bit integer inputs, given two values stored in the data register and a location in a scalar buffer surface.

```
tmp = MEM[ADDR].i64;
src = DATA.i64;
MEM[ADDR].i64 = src >= tmp ? src : tmp;
RETURN_DATA.i64 = tmp
```

S_BUFFER_ATOMIC_UMAX_X2 103

Select the maximum of two unsigned 64-bit integer inputs, given two values stored in the data register and a location in a scalar buffer surface.

```
tmp = MEM[ADDR].u64;
src = DATA.u64;
MEM[ADDR].u64 = src >= tmp ? src : tmp;
RETURN_DATA.u64 = tmp
```

S_BUFFER_ATOMIC_AND_X2 104

Calculate bitwise AND given two unsigned 64-bit integer values stored in the data register and a location in a scalar buffer surface.

```
tmp = MEM[ADDR].b64;
MEM[ADDR].b64 = (tmp & DATA.b64);
RETURN_DATA.b64 = tmp
```

S_BUFFER_ATOMIC_OR_X2 105

Calculate bitwise OR given two unsigned 64-bit integer values stored in the data register and a location in a scalar buffer surface.

```
tmp = MEM[ADDR].b64;
MEM[ADDR].b64 = (tmp | DATA.b64);
RETURN_DATA.b64 = tmp
```

S_BUFFER_ATOMIC_XOR_X2 106

Calculate bitwise XOR given two unsigned 64-bit integer values stored in the data register and a location in a scalar buffer surface.

```
tmp = MEM[ADDR].b64;
MEM[ADDR].b64 = (tmp ^ DATA.b64);
RETURN_DATA.b64 = tmp
```

S_BUFFER_ATOMIC_INC_X2 107

Increment an unsigned 64-bit integer value from a location in a scalar buffer surface with wraparound to 0 if the value exceeds a value in the data register.

```
tmp = MEM[ADDR].u64;
src = DATA.u64;
MEM[ADDR].u64 = tmp >= src ? 0ULL : tmp + 1ULL;
RETURN_DATA.u64 = tmp
```

S_BUFFER_ATOMIC_DEC_X2 108

Decrement an unsigned 64-bit integer value from a location in a scalar buffer surface with wraparound to a value in the data register if the decrement yields a negative value.

```
tmp = MEM[ADDR].u64;
src = DATA.u64;
MEM[ADDR].u64 = ((tmp == 0ULL) || (tmp > src)) ? src : tmp - 1ULL;
RETURN_DATA.u64 = tmp
```

S_ATOMIC_SWAP 128

Swap an unsigned 32-bit integer value in the data register with a location in the scalar memory.

```
tmp = MEM[ADDR].b32;
MEM[ADDR].b32 = DATA.b32;
RETURN_DATA.b32 = tmp
```

S_ATOMIC_CMPSWAP 129

Compare two unsigned 32-bit integer values stored in the data comparison register and a location in the scalar memory. Modify the memory location with a value in the data source register iff the comparison is equal.

```
tmp = MEM[ADDR].u32;
src = DATA[31 : 0].u32;
cmp = DATA[63 : 32].u32;
MEM[ADDR].u32 = tmp == cmp ? src : tmp;
RETURN_DATA.u32 = tmp
```

S_ATOMIC_ADD 130

Add two unsigned 32-bit integer values stored in the data register and a location in the scalar memory.

```
tmp = MEM[ADDR].u32;
MEM[ADDR].u32 += DATA.u32;
RETURN_DATA.u32 = tmp
```

S_ATOMIC_SUB 131

Subtract an unsigned 32-bit integer value stored in the data register from a value stored in a location in the scalar memory.

```
tmp = MEM[ADDR].u32;
MEM[ADDR].u32 -= DATA.u32;
```

```
RETURN_DATA.u32 = tmp
```

S_ATOMIC_SMIN 132

Select the minimum of two signed 32-bit integer inputs, given two values stored in the data register and a location in the scalar memory.

```
tmp = MEM[ADDR].i32;
src = DATA.i32;
MEM[ADDR].i32 = src < tmp ? src : tmp;
RETURN_DATA.i32 = tmp
```

S_ATOMIC_UMIN 133

Select the minimum of two unsigned 32-bit integer inputs, given two values stored in the data register and a location in the scalar memory.

```
tmp = MEM[ADDR].u32;
src = DATA.u32;
MEM[ADDR].u32 = src < tmp ? src : tmp;
RETURN_DATA.u32 = tmp
```

S_ATOMIC_SMAX 134

Select the maximum of two signed 32-bit integer inputs, given two values stored in the data register and a location in the scalar memory.

```
tmp = MEM[ADDR].i32;
src = DATA.i32;
MEM[ADDR].i32 = src >= tmp ? src : tmp;
RETURN_DATA.i32 = tmp
```

S_ATOMIC_UMAX 135

Select the maximum of two unsigned 32-bit integer inputs, given two values stored in the data register and a location in the scalar memory.

```
tmp = MEM[ADDR].u32;
src = DATA.u32;
MEM[ADDR].u32 = src >= tmp ? src : tmp;
```

```
RETURN_DATA.u32 = tmp
```

S_ATOMIC_AND 136

Calculate bitwise AND given two unsigned 32-bit integer values stored in the data register and a location in the scalar memory.

```
tmp = MEM[ADDR].b32;
MEM[ADDR].b32 = (tmp & DATA.b32);
RETURN_DATA.b32 = tmp
```

S_ATOMIC_OR 137

Calculate bitwise OR given two unsigned 32-bit integer values stored in the data register and a location in the scalar memory.

```
tmp = MEM[ADDR].b32;
MEM[ADDR].b32 = (tmp | DATA.b32);
RETURN_DATA.b32 = tmp
```

S_ATOMIC_XOR 138

Calculate bitwise XOR given two unsigned 32-bit integer values stored in the data register and a location in the scalar memory.

```
tmp = MEM[ADDR].b32;
MEM[ADDR].b32 = (tmp ^ DATA.b32);
RETURN_DATA.b32 = tmp
```

S_ATOMIC_INC 139

Increment an unsigned 32-bit integer value from a location in the scalar memory with wraparound to 0 if the value exceeds a value in the data register.

```
tmp = MEM[ADDR].u32;
src = DATA.u32;
MEM[ADDR].u32 = tmp >= src ? 0U : tmp + 1U;
RETURN_DATA.u32 = tmp
```

S_ATOMIC_DEC 140

Decrement an unsigned 32-bit integer value from a location in the scalar memory with wraparound to a value in the data register if the decrement yields a negative value.

```
tmp = MEM[ADDR].u32;
src = DATA.u32;
MEM[ADDR].u32 = ((tmp == 0U) || (tmp > src)) ? src : tmp - 1U;
RETURN_DATA.u32 = tmp
```

S_ATOMIC_SWAP_X2 160

Swap an unsigned 64-bit integer value in the data register with a location in the scalar memory.

```
tmp = MEM[ADDR].b64;
MEM[ADDR].b64 = DATA.b64;
RETURN_DATA.b64 = tmp
```

S_ATOMIC_CMPSWAP_X2 161

Compare two unsigned 64-bit integer values stored in the data comparison register and a location in the scalar memory. Modify the memory location with a value in the data source register iff the comparison is equal.

```
tmp = MEM[ADDR].u64;
src = DATA[63 : 0].u64;
cmp = DATA[127 : 64].u64;
MEM[ADDR].u64 = tmp == cmp ? src : tmp;
RETURN_DATA.u64 = tmp
```

S_ATOMIC_ADD_X2 162

Add two unsigned 64-bit integer values stored in the data register and a location in the scalar memory.

```
tmp = MEM[ADDR].u64;
MEM[ADDR].u64 += DATA.u64;
RETURN_DATA.u64 = tmp
```

S_ATOMIC_SUB_X2 163

Subtract an unsigned 64-bit integer value stored in the data register from a value stored in a location in the scalar memory.

```
tmp = MEM[ADDR].u64;
MEM[ADDR].u64 -= DATA.u64;
RETURN_DATA.u64 = tmp
```

S_ATOMIC_SMIN_X2 164

Select the minimum of two signed 64-bit integer inputs, given two values stored in the data register and a location in the scalar memory.

```
tmp = MEM[ADDR].i64;
src = DATA.i64;
MEM[ADDR].i64 = src < tmp ? src : tmp;
RETURN_DATA.i64 = tmp
```

S_ATOMIC_UMIN_X2 165

Select the minimum of two unsigned 64-bit integer inputs, given two values stored in the data register and a location in the scalar memory.

```
tmp = MEM[ADDR].u64;
src = DATA.u64;
MEM[ADDR].u64 = src < tmp ? src : tmp;
RETURN_DATA.u64 = tmp
```

S_ATOMIC_SMAX_X2 166

Select the maximum of two signed 64-bit integer inputs, given two values stored in the data register and a location in the scalar memory.

```
tmp = MEM[ADDR].i64;
src = DATA.i64;
MEM[ADDR].i64 = src >= tmp ? src : tmp;
RETURN_DATA.i64 = tmp
```

S_ATOMIC_UMAX_X2 167

Select the maximum of two unsigned 64-bit integer inputs, given two values stored in the data register and a

location in the scalar memory.

```
tmp = MEM[ADDR].u64;
src = DATA.u64;
MEM[ADDR].u64 = src >= tmp ? src : tmp;
RETURN_DATA.u64 = tmp
```

S_ATOMIC_AND_X2 168

Calculate bitwise AND given two unsigned 64-bit integer values stored in the data register and a location in the scalar memory.

```
tmp = MEM[ADDR].b64;
MEM[ADDR].b64 = (tmp & DATA.b64);
RETURN_DATA.b64 = tmp
```

S_ATOMIC_OR_X2 169

Calculate bitwise OR given two unsigned 64-bit integer values stored in the data register and a location in the scalar memory.

```
tmp = MEM[ADDR].b64;
MEM[ADDR].b64 = (tmp | DATA.b64);
RETURN_DATA.b64 = tmp
```

S_ATOMIC_XOR_X2 170

Calculate bitwise XOR given two unsigned 64-bit integer values stored in the data register and a location in the scalar memory.

```
tmp = MEM[ADDR].b64;
MEM[ADDR].b64 = (tmp ^ DATA.b64);
RETURN_DATA.b64 = tmp
```

S_ATOMIC_INC_X2 171

Increment an unsigned 64-bit integer value from a location in the scalar memory with wraparound to 0 if the value exceeds a value in the data register.

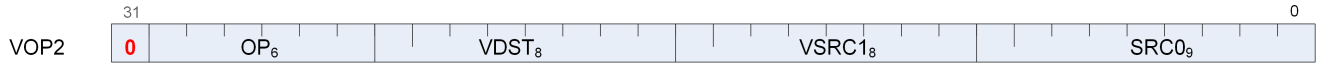
```
tmp = MEM[ADDR].u64;
src = DATA.u64;
MEM[ADDR].u64 = tmp >= src ? 0ULL : tmp + 1ULL;
RETURN_DATA.u64 = tmp
```

S_ATOMIC_DEC_X2 172

Decrement an unsigned 64-bit integer value from a location in the scalar memory with wraparound to a value in the data register if the decrement yields a negative value.

```
tmp = MEM[ADDR].u64;
src = DATA.u64;
MEM[ADDR].u64 = ((tmp == 0ULL) || (tmp > src)) ? src : tmp - 1ULL;
RETURN_DATA.u64 = tmp
```

12.7. VOP2 Instructions



Instructions in this format may use a 32-bit literal constant, DPP or SDWA which occurs immediately after the instruction.

V_CNDMASK_B32 0

Copy data from one of two inputs based on the vector condition code and store the result into a vector register.

```
D0.u32 = VCC.u64[laneId] ? S1.u32 : S0.u32
```

Notes

In VOP3 the VCC source may be a scalar GPR specified in S2.

Floating-point modifiers are valid for this instruction if S0 and S1 are 32-bit floating point values. This instruction is suitable for negating or taking the absolute value of a floating-point value.

V_ADD_F32 1

Add two floating point inputs and store the result into a vector register.

```
D0.f32 = S0.f32 + S1.f32
```

Notes

0.5ULP precision, denormals are supported.

V_SUB_F32 2

Subtract the second floating point input from the first input and store the result into a vector register.

```
D0.f32 = S0.f32 - S1.f32
```

Notes

0.5ULP precision, denormals are supported.

V_SUBREV_F32 3

Subtract the *first* floating point input from the *second* input and store the result into a vector register.

```
D0.f32 = S1.f32 - S0.f32
```

Notes

0.5ULP precision, denormals are supported.

V_FMAC_F64 4

Multiply two floating point inputs and accumulate the result into the destination register using fused multiply add.

```
D0.f64 = fma(S0.f64, S1.f64, D0.f64)
```

V_MUL_F32 5

Multiply two floating point inputs and store the result into a vector register.

```
D0.f32 = S0.f32 * S1.f32
```

Notes

0.5ULP precision, denormals are supported.

V_MUL_I32_I24 6

Multiply two signed 24-bit integer inputs and store the result as a signed 32-bit integer into a vector register.

```
D0.i32 = 32'I(S0.i24) * 32'I(S1.i24)
```

Notes

This opcode is expected to be as efficient as basic single-precision opcodes since it utilizes the single-precision floating point multiplier. See also V_MUL_HI_I32_I24.

V_MUL_HI_I32_I24 7

Multiply two signed 24-bit integer inputs and store the high 32 bits of the result as a signed 32-bit integer into a

vector register.

```
D0.i32 = 32'I((64'I(S0.i24) * 64'I(S1.i24)) >> 32U)
```

Notes

See also V_MUL_I32_I24.

V_MUL_U32_U24 8

Multiply two unsigned 24-bit integer inputs and store the result as an unsigned 32-bit integer into a vector register.

```
D0.u32 = 32'U(S0.u24) * 32'U(S1.u24)
```

Notes

This opcode is expected to be as efficient as basic single-precision opcodes since it utilizes the single-precision floating point multiplier. See also V_MUL_HI_U32_U24.

V_MUL_HI_U32_U24 9

Multiply two unsigned 24-bit integer inputs and store the high 32 bits of the result as an unsigned 32-bit integer into a vector register.

```
D0.u32 = 32'U((64'U(S0.u24) * 64'U(S1.u24)) >> 32U)
```

Notes

See also V_MUL_U32_U24.

V_MIN_F32 10

Select the minimum of two single-precision float inputs and store the result into a vector register.

```
if (WAVE_MODE.IEEE && isSignalNAN(64'F(S0.f32))) then
    D0.f32 = 32'F(cvtToQuietNAN(64'F(S0.f32)))
elsif (WAVE_MODE.IEEE && isSignalNAN(64'F(S1.f32))) then
    D0.f32 = 32'F(cvtToQuietNAN(64'F(S1.f32)))
elsif isNAN(64'F(S0.f32)) then
    D0.f32 = S1.f32
elsif isNAN(64'F(S1.f32)) then
```

```
    D0.f32 = S0.f32
elsif ((64'F(S0.f32) == +0.0) && (64'F(S1.f32) == -0.0)) then
    D0.f32 = S1.f32
elsif ((64'F(S0.f32) == -0.0) && (64'F(S1.f32) == +0.0)) then
    D0.f32 = S0.f32
else
    // Note: there's no IEEE case here like there is for V_MAX_F32.
    D0.f32 = S0.f32 < S1.f32 ? S0.f32 : S1.f32
endif
```

V_MAX_F32 11

Select the maximum of two single-precision float inputs and store the result into a vector register.

```
if (WAVE_MODE.IEEE && isSignalNAN(64'F(S0.f32))) then
    D0.f32 = 32'F(cvtToQuietNAN(64'F(S0.f32)))
elsif (WAVE_MODE.IEEE && isSignalNAN(64'F(S1.f32))) then
    D0.f32 = 32'F(cvtToQuietNAN(64'F(S1.f32)))
elsif isNAN(64'F(S0.f32)) then
    D0.f32 = S1.f32
elsif isNAN(64'F(S1.f32)) then
    D0.f32 = S0.f32
elsif ((64'F(S0.f32) == +0.0) && (64'F(S1.f32) == -0.0)) then
    D0.f32 = S0.f32
elsif ((64'F(S0.f32) == -0.0) && (64'F(S1.f32) == +0.0)) then
    D0.f32 = S1.f32
elsif WAVE_MODE.IEEE then
    D0.f32 = S0.f32 >= S1.f32 ? S0.f32 : S1.f32
else
    D0.f32 = S0.f32 > S1.f32 ? S0.f32 : S1.f32
endif
```

V_MIN_I32 12

Select the minimum of two signed 32-bit integer inputs and store the selected value into a vector register.

```
D0.i32 = S0.i32 < S1.i32 ? S0.i32 : S1.i32
```

V_MAX_I32 13

Select the maximum of two signed 32-bit integer inputs and store the selected value into a vector register.

```
D0.i32 = S0.i32 >= S1.i32 ? S0.i32 : S1.i32
```

V_MIN_U32 14

Select the minimum of two unsigned 32-bit integer inputs and store the selected value into a vector register.

```
D0.u32 = S0.u32 < S1.u32 ? S0.u32 : S1.u32
```

V_MAX_U32 15

Select the maximum of two unsigned 32-bit integer inputs and store the selected value into a vector register.

```
D0.u32 = S0.u32 >= S1.u32 ? S0.u32 : S1.u32
```

V_LSHRREV_B32 16

Given a shift count in the *first* vector input, calculate the logical shift right of the *second* vector input and store the result into a vector register.

```
D0.u32 = (S1.u32 >> S0[4 : 0].u32)
```

V_ASHRREV_I32 17

Given a shift count in the *first* vector input, calculate the arithmetic shift right (preserving sign bit) of the *second* vector input and store the result into a vector register.

```
D0.i32 = (S1.i32 >> S0[4 : 0].u32)
```

V_LSHLREV_B32 18

Given a shift count in the *first* vector input, calculate the logical shift left of the *second* vector input and store the result into a vector register.

```
D0.u32 = (S1.u32 << S0[4 : 0].u32)
```

V_AND_B32 19

Calculate bitwise AND on two vector inputs and store the result into a vector register.

```
D0.u32 = (S0.u32 & S1.u32)
```

Notes

Input and output modifiers not supported.

V_OR_B32 20

Calculate bitwise OR on two vector inputs and store the result into a vector register.

```
D0.u32 = (S0.u32 | S1.u32)
```

Notes

Input and output modifiers not supported.

V_XOR_B32 21

Calculate bitwise XOR on two vector inputs and store the result into a vector register.

```
D0.u32 = (S0.u32 ^ S1.u32)
```

Notes

Input and output modifiers not supported.

V_FMAMK_F32 23

Multiply a single-precision float input with a literal constant and add a second single-precision float input using fused multiply add, and store the result into a vector register.

```
D0.f32 = fma(S0.f32, SIMM32.f32, S1.f32)
```

Notes

This opcode cannot use the VOP3 encoding and cannot use input/output modifiers.

V_FMAAK_F32 24

Multiply two single-precision float inputs and add a literal constant using fused multiply add, and store the result into a vector register.

```
D0.f32 = fma(S0.f32, S1.f32, SIMM32.f32)
```

Notes

This opcode cannot use the VOP3 encoding and cannot use input/output modifiers.

V_ADD_CO_U32 25

Add two unsigned inputs, store the result into a vector register and store the carry-out mask into a scalar register.

```
tmp = 64'U(S0.u32) + 64'U(S1.u32);
VCC.u64[laneId] = tmp >= 0x100000000ULL ? 1'1U : 1'0U;
// VCC is an UNSIGNED overflow/carry-out for V_ADDC_CO_U32.
D0.u32 = tmp.u32
```

Notes

In VOP3 the VCC destination may be an arbitrary SGPR-pair.

Supports saturation (unsigned 32-bit integer domain).

V_SUB_CO_U32 26

Subtract the second unsigned input from the first input, store the result into a vector register and store the carry-out mask into a scalar register.

```
tmp = S0.u32 - S1.u32;
VCC.u64[laneId] = S1.u32 > S0.u32 ? 1'1U : 1'0U;
// VCC is an UNSIGNED overflow/carry-out for V_SUBB_CO_U32.
D0.u32 = tmp.u32
```

Notes

In VOP3 the VCC destination may be an arbitrary SGPR-pair.

Supports saturation (unsigned 32-bit integer domain).

V_SUBREV_CO_U32 27

Subtract the *first* unsigned input from the *second* input, store the result into a vector register and store the carry-out mask into a scalar register.

```
tmp = S1.u32 - S0.u32;
VCC.u64[laneId] = S0.u32 > S1.u32 ? 1'1U : 1'0U;
// VCC is an UNSIGNED overflow/carry-out for V_SUBB_CO_U32.
D0.u32 = tmp.u32
```

Notes

In VOP3 the VCC destination may be an arbitrary SGPR-pair.

Supports saturation (unsigned 32-bit integer domain).

V_ADDC_CO_U32 28

Add two unsigned inputs and a bit from a carry-in mask, store the result into a vector register and store the carry-out mask into a scalar register.

```
tmp = 64'U(S0.u32) + 64'U(S1.u32) + VCC.u64[laneId].u64;
VCC.u64[laneId] = tmp >= 0x100000000ULL ? 1'1U : 1'0U;
// VCC is an UNSIGNED overflow.
D0.u32 = tmp.u32
```

Notes

In VOP3 the VCC destination may be an arbitrary SGPR-pair, and the VCC source comes from the SGPR-pair at S2.u.

Supports saturation (unsigned 32-bit integer domain).

V_SUBB_CO_U32 29

Subtract the second unsigned input from the first input, subtract a bit from the carry-in mask, store the result into a vector register and store the carry-out mask to a scalar register.

```
tmp = S0.u32 - S1.u32 - VCC.u64[laneId].u32;
VCC.u64[laneId] = 64'U(S1.u32) + VCC.u64[laneId].u64 > 64'U(S0.u32) ? 1'1U : 1'0U;
// VCC is an UNSIGNED overflow.
D0.u32 = tmp.u32
```

Notes

In VOP3 the VCC destination may be an arbitrary SGPR-pair, and the VCC source comes from the SGPR-pair at S2.u.

Supports saturation (unsigned 32-bit integer domain).

V_SUBBREV_CO_U32 30

Subtract the *first* unsigned input from the *second* input, subtract a bit from the carry-in mask, store the result into a vector register and store the carry-out mask to a scalar register.

```
tmp = S1.u32 - S0.u32 - VCC.u64[laneId].u32;
VCC.u64[laneId] = 64'U(S1.u32) + VCC.u64[laneId].u64 > 64'U(S0.u32) ? 1'1U : 1'0U;
// VCC is an UNSIGNED overflow.
D0.u32 = tmp.u32
```

Notes

In VOP3 the VCC destination may be an arbitrary SGPR-pair, and the VCC source comes from the SGPR-pair at S2.u.

Supports saturation (unsigned 32-bit integer domain).

V_ADD_F16 31

Add two floating point inputs and store the result into a vector register.

```
D0.f16 = S0.f16 + S1.f16
```

Notes

0.5ULP precision. Supports denormals, round mode, exception flags and saturation.

V_SUB_F16 32

Subtract the second floating point input from the first input and store the result into a vector register.

```
D0.f16 = S0.f16 - S1.f16
```

Notes

0.5ULP precision. Supports denormals, round mode, exception flags and saturation.

V_SUBREV_F16 33

Subtract the *first* floating point input from the *second* input and store the result into a vector register.

```
D0.f16 = S1.f16 - S0.f16
```

Notes

0.5ULP precision. Supports denormals, round mode, exception flags and saturation.

V_MUL_F16 34

Multiply two floating point inputs and store the result into a vector register.

```
D0.f16 = S0.f16 * S1.f16
```

Notes

0.5ULP precision. Supports denormals, round mode, exception flags and saturation.

V_MAC_F16 35

Multiply two floating point inputs and accumulate the result into the destination register. Implements IEEE rules and non-standard rule for OPSEL.

```
tmp = S0.f16 * S1.f16 + D0.f16;
if OPSEL.u4[3] then
    D0 = { tmp.f16, D0[15 : 0] }
else
    D0 = { 16'0, tmp.f16 }
endif
```

Notes

Supports round mode, exception flags, saturation.

V_MADMK_F16 36

Multiply a floating point input with a literal constant and add a second floating point input, and store the result into a vector register. Implements IEEE rules and non-standard rule for OPSEL.

```
tmp = S0.f16 * SIMM32.f16 + S1.f16;
```

```
if OPSEL.u4[3] then
    D0 = { tmp.f16, D0[15 : 0] }
else
    D0 = { 16'0, tmp.f16 }
endif
```

Notes

This opcode cannot use the VOP3 encoding and cannot use input/output modifiers. Supports round mode, exception flags, saturation.

V_MADAK_F16 37

Multiply two floating point inputs and add a literal constant, and store the result into a vector register. Implements IEEE rules and non-standard rule for OPSEL.

```
tmp = S0.f16 * S1.f16 + SIMM32.f16;
// K is a literal constant.
if OPSEL.u4[3] then
    D0 = { tmp.f16, D0[15 : 0] }
else
    D0 = { 16'0, tmp.f16 }
endif
```

Notes

This opcode cannot use the VOP3 encoding and cannot use input/output modifiers. Supports round mode, exception flags, saturation.

V_ADD_U16 38

Add two unsigned 16-bit integer inputs and store the result into a vector register. No carry-in or carry-out support.

```
D0.u16 = S0.u16 + S1.u16
```

Notes

Supports saturation (unsigned 16-bit integer domain).

V_SUB_U16 39

Subtract the second unsigned input from the first input and store the result into a vector register. No carry-in or carry-out support.

```
D0.u16 = S0.u16 - S1.u16
```

Notes

Supports saturation (unsigned 16-bit integer domain).

V_SUBREV_U16 40

Subtract the *first* unsigned input from the *second* input and store the result into a vector register. No carry-in or carry-out support.

```
D0.u16 = S1.u16 - S0.u16
```

Notes

Supports saturation (unsigned 16-bit integer domain).

V_MUL_LO_U16 41

Multiply two unsigned 16-bit integer inputs and store the low bits of the result into a vector register.

```
D0.u16 = S0.u16 * S1.u16
```

Notes

Supports saturation (unsigned 16-bit integer domain).

V_LSHLREV_B16 42

Given a shift count in the *first* vector input, calculate the logical shift left of the *second* vector input and store the result into a vector register.

```
D0.u16 = (S1.u16 << S0[3 : 0].u32)
```

V_LSHRREV_B16 43

Given a shift count in the *first* vector input, calculate the logical shift right of the *second* vector input and store the result into a vector register.

```
D0.u16 = (S1.u16 >> S0[3 : 0].u32)
```

V_ASHRREV_I16 44

Given a shift count in the *first* vector input, calculate the arithmetic shift right (preserving sign bit) of the *second* vector input and store the result into a vector register.

```
D0.i16 = (S1.i16 >> S0[3 : 0].u32)
```

V_MAX_F16 45

Select the maximum of two half-precision float inputs and store the result into a vector register.

```
if (WAVE_MODE.IEEE && isSignalNAN(64'F(S0.f16))) then
    D0.f16 = 16'F(cvtToQuietNAN(64'F(S0.f16)))
elsif (WAVE_MODE.IEEE && isSignalNAN(64'F(S1.f16))) then
    D0.f16 = 16'F(cvtToQuietNAN(64'F(S1.f16)))
elsif isNAN(64'F(S0.f16)) then
    D0.f16 = S1.f16
elsif isNAN(64'F(S1.f16)) then
    D0.f16 = S0.f16
elsif ((64'F(S0.f16) == +0.0) && (64'F(S1.f16) == -0.0)) then
    D0.f16 = S0.f16
elsif ((64'F(S0.f16) == -0.0) && (64'F(S1.f16) == +0.0)) then
    D0.f16 = S1.f16
elsif WAVE_MODE.IEEE then
    D0.f16 = S0.f16 >= S1.f16 ? S0.f16 : S1.f16
else
    D0.f16 = S0.f16 > S1.f16 ? S0.f16 : S1.f16
endif
```

Notes

IEEE compliant. Supports denormals, round mode, exception flags, saturation.

V_MIN_F16 46

Select the minimum of two half-precision float inputs and store the result into a vector register.

```
if (WAVE_MODE.IEEE && isSignalNAN(64'F(S0.f16))) then
    D0.f16 = 16'F(cvtToQuietNAN(64'F(S0.f16)))
elsif (WAVE_MODE.IEEE && isSignalNAN(64'F(S1.f16))) then
    D0.f16 = 16'F(cvtToQuietNAN(64'F(S1.f16)))
elsif isNAN(64'F(S0.f16)) then
```

```
    D0.f16 = S1.f16
elsif isNAN(64'F(S1.f16)) then
    D0.f16 = S0.f16
elsif ((64'F(S0.f16) == +0.0) && (64'F(S1.f16) == -0.0)) then
    D0.f16 = S1.f16
elsif ((64'F(S0.f16) == -0.0) && (64'F(S1.f16) == +0.0)) then
    D0.f16 = S0.f16
else
    // Note: there's no IEEE case here like there is for V_MAX_F16.
    D0.f16 = S0.f16 < S1.f16 ? S0.f16 : S1.f16
endif
```

Notes

IEEE compliant. Supports denormals, round mode, exception flags, saturation.

V_MAX_U16 47

Select the maximum of two unsigned 16-bit integer inputs and store the selected value into a vector register.

```
D0.u16 = S0.u16 >= S1.u16 ? S0.u16 : S1.u16
```

V_MAX_I16 48

Select the maximum of two signed 16-bit integer inputs and store the selected value into a vector register.

```
D0.i16 = S0.i16 >= S1.i16 ? S0.i16 : S1.i16
```

V_MIN_U16 49

Select the minimum of two unsigned 16-bit integer inputs and store the selected value into a vector register.

```
D0.u16 = S0.u16 < S1.u16 ? S0.u16 : S1.u16
```

V_MIN_I16 50

Select the minimum of two signed 16-bit integer inputs and store the selected value into a vector register.

```
D0.i16 = S0.i16 < S1.i16 ? S0.i16 : S1.i16
```

V_LDEXP_F16 51

Multiply the first input, a floating point value, by an integral power of 2 specified in the second input, a signed integer value, and store the floating point result into a vector register.

```
D0.f16 = S0.f16 * 16'F(2.0F ** 32'I(S1.i16))
```

Notes

Compare with the `ldexp()` function in C. Note that the S1 has a format of f16 since floating point literal constants are interpreted as 16 bit value for this opcode.

V_ADD_U32 52

Add two unsigned 32-bit integer inputs and store the result into a vector register. No carry-in or carry-out support.

```
D0.u32 = S0.u32 + S1.u32
```

Notes

Supports saturation (unsigned 32-bit integer domain).

V_SUB_U32 53

Subtract the second unsigned input from the first input and store the result into a vector register. No carry-in or carry-out support.

```
D0.u32 = S0.u32 - S1.u32
```

Notes

Supports saturation (unsigned 32-bit integer domain).

V_SUBREV_U32 54

Subtract the *first* unsigned input from the *second* input and store the result into a vector register. No carry-in or carry-out support.

```
D0.u32 = S1.u32 - S0.u32
```

Notes

Supports saturation (unsigned 32-bit integer domain).

V_DOT2C_F32_F16 55

Compute the dot product of two packed 2-D half-precision float inputs in the single-precision float domain and accumulate with the single-precision float value in the destination register.

```
tmp = D0.f32;
tmp += f16_to_f32(S0[15 : 0].f16) * f16_to_f32(S1[15 : 0].f16);
tmp += f16_to_f32(S0[31 : 16].f16) * f16_to_f32(S1[31 : 16].f16);
D0.f32 = tmp
```

V_DOT2C_I32_I16 56

Compute the dot product of two packed 2-D signed 16-bit integer inputs in the signed 32-bit integer domain and accumulate with the signed 32-bit integer value in the destination register.

```
tmp = D0.i32;
tmp += i16_to_i32(S0[15 : 0].i16) * i16_to_i32(S1[15 : 0].i16);
tmp += i16_to_i32(S0[31 : 16].i16) * i16_to_i32(S1[31 : 16].i16);
D0.i32 = tmp
```

V_DOT4C_I32_I8 57

Compute the dot product of two packed 4-D signed 8-bit integer inputs in the signed 32-bit integer domain and accumulate with the signed 32-bit integer value in the destination register.

```
tmp = D0.i32;
tmp += i8_to_i32(S0[7 : 0].i8) * i8_to_i32(S1[7 : 0].i8);
tmp += i8_to_i32(S0[15 : 8].i8) * i8_to_i32(S1[15 : 8].i8);
tmp += i8_to_i32(S0[23 : 16].i8) * i8_to_i32(S1[23 : 16].i8);
tmp += i8_to_i32(S0[31 : 24].i8) * i8_to_i32(S1[31 : 24].i8);
D0.i32 = tmp
```

V_DOT8C_I32_I4 58

Compute the dot product of two packed 8-D signed 4-bit integer inputs in the signed 32-bit integer domain and accumulate with the signed 32-bit integer value in the destination register.

```
tmp = D0.i32;
tmp += i4_to_i32(S0[3 : 0].i4) * i4_to_i32(S1[3 : 0].i4);
tmp += i4_to_i32(S0[7 : 4].i4) * i4_to_i32(S1[7 : 4].i4);
tmp += i4_to_i32(S0[11 : 8].i4) * i4_to_i32(S1[11 : 8].i4);
tmp += i4_to_i32(S0[15 : 12].i4) * i4_to_i32(S1[15 : 12].i4);
tmp += i4_to_i32(S0[19 : 16].i4) * i4_to_i32(S1[19 : 16].i4);
tmp += i4_to_i32(S0[23 : 20].i4) * i4_to_i32(S1[23 : 20].i4);
tmp += i4_to_i32(S0[27 : 24].i4) * i4_to_i32(S1[27 : 24].i4);
tmp += i4_to_i32(S0[31 : 28].i4) * i4_to_i32(S1[31 : 28].i4);
D0.i32 = tmp
```

V_FMAC_F32 59

Multiply two floating point inputs and accumulate the result into the destination register using fused multiply add.

```
D0.f32 = fma(S0.f32, S1.f32, D0.f32)
```

V_PK_FMAC_F16 60

Multiply two packed half-precision float inputs component-wise and accumulate the result into the destination register using fused multiply add.

```
D0[15 : 0].f16 = fma(S0[15 : 0].f16, S1[15 : 0].f16, D0[15 : 0].f16);
D0[31 : 16].f16 = fma(S0[31 : 16].f16, S1[31 : 16].f16, D0[31 : 16].f16)
```

V_XNOR_B32 61

Calculate bitwise XNOR on two vector inputs and store the result into a vector register.

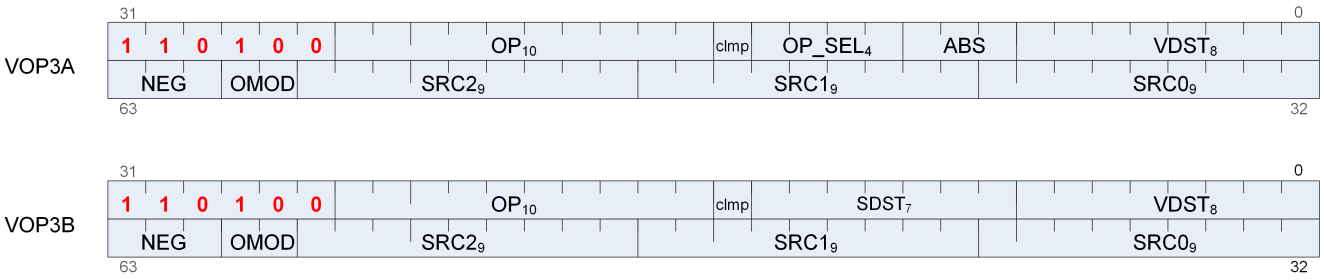
```
D0.u32 = ~(S0.u32 ^ S1.u32)
```

Notes

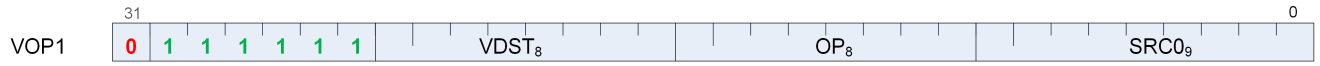
Input and output modifiers not supported.

12.7.1. VOP2 using VOP3 encoding

Instructions in this format may also be encoded as VOP3. This allows access to the extra control bits (e.g. ABS, OMOD) in exchange for not being able to use a literal constant. The VOP3 opcode is: VOP2 opcode + 0x100.



12.8. VOP1 Instructions



Instructions in this format may use a 32-bit literal constant, DPP or SDWA which occurs immediately after the instruction.

V_NOP **0**

Do nothing.

V_MOV_B32 **1**

Move data from a vector input into a vector register.

```
D0.b32 = S0.b32
```

Notes

Floating-point modifiers are valid for this instruction if S0.u is a 32-bit floating point value. This instruction is suitable for negating or taking the absolute value of a floating-point value.

Functional examples:

```
v_mov_b32 v0, v1    // Move into v0 from v1
v_mov_b32 v0, -v1   // Set v0 to the negation of v1
v_mov_b32 v0, abs(v1)   // Set v0 to the absolute value of v1
```

V_READFIRSTLANE_B32 **2**

Read the scalar value in the lowest active lane of the input vector register and store it into a scalar register.

```
declare lane : 32'I;
if EXEC == 0x0LL then
    lane = 0;
    // Force lane 0 if all lanes are disabled
else
    lane = s_ff1_i32_b64(EXEC);
    // Lowest active lane
endif;
D0.b32 = VGPR[lane][SRC0.u32]
```

Notes

Overrides EXEC mask for the VGPR read. Input and output modifiers not supported; this is an untyped operation.

V_CVT_I32_F64 3

Convert from a double-precision float input to a signed 32-bit integer value and store the result into a vector register.

```
D0.i32 = f64_to_i32(S0.f64)
```

Notes

0.5ULP accuracy, out-of-range floating point values (including infinity) saturate. NAN is converted to 0.

Generation of the INEXACT exception is controlled by the CLAMP bit. INEXACT exceptions are enabled for this conversion iff CLAMP == 1.

V_CVT_F64_I32 4

Convert from a signed 32-bit integer input to a double-precision float value and store the result into a vector register.

```
D0.f64 = i32_to_f64(S0.i32)
```

Notes

0ULP accuracy.

V_CVT_F32_I32 5

Convert from a signed 32-bit integer input to a single-precision float value and store the result into a vector register.

```
D0.f32 = i32_to_f32(S0.i32)
```

Notes

0.5ULP accuracy.

V_CVT_F32_U32 6

Convert from an unsigned 32-bit integer input to a single-precision float value and store the result into a vector register.

```
D0.f32 = u32_to_f32(S0.u32)
```

Notes

0.5ULP accuracy.

V_CVT_U32_F32 7

Convert from a single-precision float input to an unsigned 32-bit integer value and store the result into a vector register.

```
D0.u32 = f32_to_u32(S0.f32)
```

Notes

1ULP accuracy, out-of-range floating point values (including infinity) saturate. NAN is converted to 0.

Generation of the INEXACT exception is controlled by the CLAMP bit. INEXACT exceptions are enabled for this conversion iff CLAMP == 1.

V_CVT_I32_F32 8

Convert from a single-precision float input to a signed 32-bit integer value and store the result into a vector register.

```
D0.i32 = f32_to_i32(S0.f32)
```

Notes

1ULP accuracy, out-of-range floating point values (including infinity) saturate. NAN is converted to 0.

Generation of the INEXACT exception is controlled by the CLAMP bit. INEXACT exceptions are enabled for this conversion iff CLAMP == 1.

V_CVT_F16_F32 10

Convert from a single-precision float input to a half-precision float value and store the result into a vector register.

```
D0.f16 = f32_to_f16(S0.f32)
```

Notes

0.5ULP accuracy, supports input modifiers and creates FP16 denormals when appropriate. Flush denorms on output if specified based on DP denorm mode. Output rounding based on DP rounding mode.

V_CVT_F32_F16 11

Convert from a half-precision float input to a single-precision float value and store the result into a vector register.

```
D0.f32 = f16_to_f32(S0.f16)
```

Notes

0ULP accuracy, FP16 denormal inputs are accepted. Flush denorms on input if specified based on DP denorm mode.

V_CVT_RPI_I32_F32 12

Convert from a single-precision float input to a signed 32-bit integer value using round to nearest integer semantics (ignore the default rounding mode) and store the result into a vector register.

```
D0.i32 = f32_to_i32(floor(S0.f32 + 0.5F))
```

Notes

0.5ULP accuracy, denormals are supported.

V_CVT_FLR_I32_F32 13

Convert from a single-precision float input to a signed 32-bit integer value using round-down semantics (ignore the default rounding mode) and store the result into a vector register.

```
D0.i32 = f32_to_i32(floor(S0.f32))
```

Notes

1ULP accuracy, denormals are supported.

V_CVT_OFF_F32_I4 14

Convert from a signed 4-bit integer input to a single-precision float value using an offset table and store the result into a vector register.

Used for interpolation in shader. Lookup table on S0[3:0]:

S0 binary Result
1000 -0.5000f
1001 -0.4375f
1010 -0.3750f
1011 -0.3125f
1100 -0.2500f
1101 -0.1875f
1110 -0.1250f
1111 -0.0625f
0000 +0.0000f
0001 +0.0625f
0010 +0.1250f
0011 +0.1875f
0100 +0.2500f
0101 +0.3125f
0110 +0.3750f
0111 +0.4375f

```
declare CVT_OFF_TABLE : 32'F[16];
D0.f32 = CVT_OFF_TABLE[S0.u32[3 : 0]]
```

V_CVT_F32_F64 15

Convert from a double-precision float input to a single-precision float value and store the result into a vector register.

```
D0.f32 = f64_to_f32(S0.f64)
```

Notes

0.5ULP accuracy, denormals are supported.

V_CVT_F64_F32 16

Convert from a single-precision float input to a double-precision float value and store the result into a vector register.

```
D0.f64 = f32_to_f64(S0.f32)
```

Notes

0ULP accuracy, denormals are supported.

V_CVT_F32_UBYTE0 17

Convert an unsigned byte in byte 0 of the input to a single-precision float value and store the result into a vector register.

```
D0.f32 = u32_to_f32(S0[7 : 0].u32)
```

V_CVT_F32_UBYTE1 18

Convert an unsigned byte in byte 1 of the input to a single-precision float value and store the result into a vector register.

```
D0.f32 = u32_to_f32(S0[15 : 8].u32)
```

V_CVT_F32_UBYTE2 19

Convert an unsigned byte in byte 2 of the input to a single-precision float value and store the result into a vector register.

```
D0.f32 = u32_to_f32(S0[23 : 16].u32)
```

V_CVT_F32_UBYTE3 20

Convert an unsigned byte in byte 3 of the input to a single-precision float value and store the result into a vector register.

```
D0.f32 = u32_to_f32(S0[31 : 24].u32)
```

V_CVT_U32_F64 21

Convert from a double-precision float input to an unsigned 32-bit integer value and store the result into a vector register.

```
D0.u32 = f64_to_u32(S0.f64)
```

Notes

0.5ULP accuracy, out-of-range floating point values (including infinity) saturate. NAN is converted to 0.

Generation of the INEXACT exception is controlled by the CLAMP bit. INEXACT exceptions are enabled for this conversion iff CLAMP == 1.

V_CVT_F64_U32 22

Convert from an unsigned 32-bit integer input to a double-precision float value and store the result into a vector register.

```
D0.f64 = u32_to_f64(S0.u32)
```

Notes

0ULP accuracy.

V_TRUNC_F64 23

Compute the integer part of a double-precision float input using round toward zero semantics and store the result in floating point format into a vector register.

```
D0.f64 = trunc(S0.f64)
```

V_CEIL_F64 24

Round the double-precision float input up to next integer and store the result in floating point format into a vector register.

```
D0.f64 = trunc(S0.f64);
if ((S0.f64 > 0.0) && (S0.f64 != D0.f64)) then
    D0.f64 += 1.0
endif
```

V_RNDNE_F64 25

Round the double-precision float input to the nearest even integer and store the result in floating point format into a vector register.

```
D0.f64 = floor(S0.f64 + 0.5);
if (isEven(floor(S0.f64)) && (fract(S0.f64) == 0.5)) then
    D0.f64 -= 1.0
endif
```

V_FLOOR_F64 26

Round the double-precision float input down to previous integer and store the result in floating point format into a vector register.

```
D0.f64 = trunc(S0.f64);
if ((S0.f64 < 0.0) && (S0.f64 != D0.f64)) then
    D0.f64 += -1.0
endif
```

V_FRACT_F32 27

Compute the fractional portion of a single-precision float input and store the result in floating point format into a vector register.

```
D0.f32 = S0.f32 + -floor(S0.f32)
```

Notes

0.5ULP accuracy, denormals are accepted.

This is intended to comply with the DX specification of fract where the function behaves like an extension of integer modulus; be aware this may differ from how fract() is defined in other domains. For example: fract(-1.2) = 0.8 in DX.

Obey round mode, result clamped to 0x3f7fffff.

V_TRUNC_F32 28

Compute the integer part of a single-precision float input using round toward zero semantics and store the result in floating point format into a vector register.

```
D0.f32 = trunc(S0.f32)
```

V_CEIL_F32 29

Round the single-precision float input up to next integer and store the result in floating point format into a vector register.

```
D0.f32 = trunc(S0.f32);
if ((S0.f32 > 0.0F) && (S0.f32 != D0.f32)) then
    D0.f32 += 1.0F
endif
```

V_RNDNE_F32 30

Round the single-precision float input to the nearest even integer and store the result in floating point format into a vector register.

```
D0.f32 = floor(S0.f32 + 0.5F);
if (isEven(64'F(floor(S0.f32))) && (fract(S0.f32) == 0.5F)) then
    D0.f32 -= 1.0F
endif
```

V_FLOOR_F32 31

Round the single-precision float input down to previous integer and store the result in floating point format into a vector register.

```
D0.f32 = trunc(S0.f32);
if ((S0.f32 < 0.0F) && (S0.f32 != D0.f32)) then
```

```
    D0.f32 += -1.0F
endif
```

V_EXP_F32 32

Calculate 2 raised to the power of the single-precision float input and store the result into a vector register.

```
D0.f32 = pow(2.0F, S0.f32)
```

Notes

1ULP accuracy, denormals are flushed.

Functional examples:

```
V_EXP_F32(0xff800000) => 0x00000000     // exp(-INF) = 0
V_EXP_F32(0x80000000) => 0x3f800000     // exp(-0.0) = 1
V_EXP_F32(0x7f800000) => 0x7f800000     // exp(+INF) = +INF
```

V_LOG_F32 33

Calculate the base 2 logarithm of the single-precision float input and store the result into a vector register.

```
D0.f32 = log2(S0.f32)
```

Notes

1ULP accuracy, denormals are flushed.

Functional examples:

```
V_LOG_F32(0xff800000) => 0xffc00000     // log(-INF) = NAN
V_LOG_F32(0xbf800000) => 0xffc00000     // log(-1.0) = NAN
V_LOG_F32(0x80000000) => 0xff800000     // log(-0.0) = -INF
V_LOG_F32(0x00000000) => 0xff800000     // log(+0.0) = -INF
V_LOG_F32(0x3f800000) => 0x00000000     // log(+1.0) = 0
V_LOG_F32(0x7f800000) => 0x7f800000     // log(+INF) = +INF
```

V_RCP_F32 34

Calculate the reciprocal of the single-precision float input using IEEE rules and store the result into a vector

register.

```
D0.f32 = 1.0F / S0.f32
```

Notes

1ULP accuracy. Accuracy converges to < 0.5ULP when using the Newton-Raphson method and 2 FMA operations. Denormals are flushed.

Functional examples:

```
V_RCP_F32(0xff800000) => 0x80000000     // rcp(-INF) = -0
V_RCP_F32(0xc0000000) => 0xbf000000     // rcp(-2.0) = -0.5
V_RCP_F32(0x80000000) => 0xff800000     // rcp(-0.0) = -INF
V_RCP_F32(0x00000000) => 0x7f800000     // rcp(+0.0) = +INF
V_RCP_F32(0x7f800000) => 0x00000000     // rcp(+INF) = +0
```

V_RCP_IFLAG_F32 35

Calculate the reciprocal of the vector float input in a manner suitable for integer division and store the result into a vector register. This opcode is intended for use as part of an integer division macro.

```
D0.f32 = 1.0F / S0.f32;
// Can only raise integer DIV_BY_ZERO exception
```

Notes

Can raise integer DIV_BY_ZERO exception but cannot raise floating-point exceptions. To be used in an integer reciprocal macro by the compiler with one of the sequences listed below (depending on signed or unsigned operation).

Unsigned usage:
CVT_F32_U32
RCP_IFLAG_F32
MUL_F32 (2**32 - 1)
CVT_U32_F32

Signed usage:
CVT_F32_I32
RCP_IFLAG_F32
MUL_F32 (2**31 - 1)
CVT_I32_F32

V_RSQ_F32 36

Calculate the reciprocal of the square root of the single-precision float input using IEEE rules and store the result into a vector register.

```
D0.f32 = 1.0F / sqrt(S0.f32)
```

Notes

1ULP accuracy, denormals are flushed.

Functional examples:

```
V_RSQ_F32(0xff800000) => 0xffc00000     // rsq(-INF) = NAN
V_RSQ_F32(0x80000000) => 0xff800000     // rsq(-0.0) = -INF
V_RSQ_F32(0x00000000) => 0x7f800000     // rsq(+0.0) = +INF
V_RSQ_F32(0x40800000) => 0x3f000000     // rsq(+4.0) = +0.5
V_RSQ_F32(0x7f800000) => 0x00000000     // rsq(+INF) = +0
```

V_RCP_F64 37

Calculate the reciprocal of the double-precision float input using IEEE rules and store the result into a vector register.

```
D0.f64 = 1.0 / S0.f64
```

Notes

This opcode has (2**29)ULP accuracy and supports denormals.

V_RSQ_F64 38

Calculate the reciprocal of the square root of the double-precision float input using IEEE rules and store the result into a vector register.

```
D0.f64 = 1.0 / sqrt(S0.f64)
```

Notes

This opcode has (2**29)ULP accuracy and supports denormals.

V_SQRT_F32 39

Calculate the square root of the single-precision float input using IEEE rules and store the result into a vector register.

```
D0.f32 = sqrt(S0.f32)
```

Notes

1ULP accuracy, denormals are flushed.

Functional examples:

```
V_SQRT_F32(0xff800000) => 0xffc00000     // sqrt(-INF) = NAN
V_SQRT_F32(0x80000000) => 0x80000000     // sqrt(-0.0) = -0
V_SQRT_F32(0x00000000) => 0x00000000     // sqrt(+0.0) = +0
V_SQRT_F32(0x40800000) => 0x40000000     // sqrt(+4.0) = +2.0
V_SQRT_F32(0x7f800000) => 0x7f800000     // sqrt(+INF) = +INF
```

V_SQRT_F64 40

Calculate the square root of the double-precision float input using IEEE rules and store the result into a vector register.

```
D0.f64 = sqrt(S0.f64)
```

Notes

This opcode has (2**29)ULP accuracy and supports denormals.

V_SIN_F32 41

Calculate the trigonometric sine of a single-precision float value using IEEE rules and store the result into a vector register. The operand is calculated by scaling the vector input by 2 PI.

```
D0.f32 = sin(S0.f32 * 32'F(PI * 2.0))
```

Notes

Denormals are supported. Full range input is supported.

Functional examples:

```
V_SIN_F32(0xff800000) => 0xffc00000     // sin(-INF) = NAN
V_SIN_F32(0xff7fffff) => 0x00000000     // -MaxFloat, finite
```

```
V_SIN_F32(0x80000000) => 0x80000000     // sin(-0.0) = -0
V_SIN_F32(0x3e800000) => 0x3f800000     // sin(0.25) = 1
V_SIN_F32(0x7f800000) => 0xffc00000     // sin(+INF) = NAN
```

V_COS_F32 42

Calculate the trigonometric cosine of a single-precision float value using IEEE rules and store the result into a vector register. The operand is calculated by scaling the vector input by 2 PI.

```
D0.f32 = cos(S0.f32 * 32'F(PI * 2.0))
```

Notes

Denormals are supported. Full range input is supported.

Functional examples:

```
V_COS_F32(0xff800000) => 0xffc00000     // cos(-INF) = NAN
V_COS_F32(0xff7fffff) => 0x3f800000     // -MaxFloat, finite
V_COS_F32(0x80000000) => 0x3f800000     // cos(-0.0) = 1
V_COS_F32(0x3e800000) => 0x00000000     // cos(0.25) = 0
V_COS_F32(0x7f800000) => 0xffc00000     // cos(+INF) = NAN
```

V_NOT_B32 43

Calculate bitwise negation on a vector input and store the result into a vector register.

```
D0.u32 = ~S0.u32
```

Notes

Input and output modifiers not supported.

V_BFREV_B32 44

Reverse the order of bits in a vector input and store the result into a vector register.

```
D0.u32[31 : 0] = S0.u32[0 : 31]
```

Notes

Input and output modifiers not supported.

V_FFBH_U32 45

Count the number of leading "0" bits before the first "1" in a vector input and store the result into a vector register. Store -1 if there are no "1" bits.

```
D0.i32 = -1;
// Set if no ones are found
for i in 0 : 31 do
    // Search from MSB
    if S0.u32[31 - i] == 1'1U then
        D0.i32 = i;
        break
    endif
endfor
```

Notes

Functional examples:

```
V_FFBH_U32(0x00000000) => 0xffffffff
V_FFBH_U32(0x800000ff) => 0
V_FFBH_U32(0x100000ff) => 3
V_FFBH_U32(0x0000ffff) => 16
V_FFBH_U32(0x00000001) => 31
```

V_FFBL_B32 46

Count the number of trailing "0" bits before the first "1" in a vector input and store the result into a vector register. Store -1 if there are no "1" bits in the input.

```
D0.i32 = -1;
// Set if no ones are found
for i in 0 : 31 do
    // Search from LSB
    if S0.u32[i] == 1'1U then
        D0.i32 = i;
        break
    endif
endfor
```

Notes

Functional examples:

```
V_FFBL_B32(0x00000000) => 0xffffffff
V_FFBL_B32(0xff000001) => 0
V_FFBL_B32(0xff000008) => 3
V_FFBL_B32(0xffff0000) => 16
V_FFBL_B32(0x80000000) => 31
```

V_FFBH_I32 47

Count the number of leading bits that are the same as the sign bit of a vector input and store the result into a vector register. Store -1 if all input bits are the same.

```
D0.i32 = -1;
// Set if all bits are the same
for i in 1 : 31 do
    // Search from MSB
    if S0.i32[31 - i] != S0.i32[31] then
        D0.i32 = i;
        break
    endif
endfor
```

Notes

Functional examples:

```
V_FFBH_I32(0x00000000) => 0xffffffff
V_FFBH_I32(0x40000000) => 1
V_FFBH_I32(0x80000000) => 1
V_FFBH_I32(0x0fffffff) => 4
V_FFBH_I32(0xffff0000) => 16
V_FFBH_I32(0xfffffffe) => 31
V_FFBH_I32(0xffffffff) => 0xffffffff
```

V_FREXP_EXP_I32_F64 48

Extract the exponent of a double-precision float input and store the result as a signed 32-bit integer into a vector register.

```
if ((S0.f64 == +INF) || (S0.f64 == -INF) || isNAN(S0.f64)) then
    D0.i32 = 0
else
    D0.i32 = exponent(S0.f64) - 1023 + 1
endif
```

Notes

This operation satisfies the invariant S0.f64 = significand * (2 ** exponent). See also V_FREXP_MANT_F64, which returns the significand. See the C library function `frexp()` for more information.

V_FREXP_MANT_F64 49

Extract the binary significand, or mantissa, of a double-precision float input and store the result as a double-precision float into a vector register.

```
if ((S0.f64 == +INF) || (S0.f64 == -INF) || isNAN(S0.f64)) then
    D0.f64 = S0.f64
else
    D0.f64 = mantissa(S0.f64)
endif
```

Notes

This operation satisfies the invariant S0.f64 = significand * (2 ** exponent). Result range is in (-1.0,-0.5][0.5,1.0) in normal cases. See also V_FREXP_EXP_I_F64, which returns integer exponent. See the C library function `frexp()` for more information.

V_FRACT_F64 50

Compute the fractional portion of a double-precision float input and store the result in floating point format into a vector register.

```
D0.f64 = S0.f64 + -floor(S0.f64)
```

Notes

0.5ULP accuracy, denormals are accepted.

This is intended to comply with the DX specification of fract where the function behaves like an extension of integer modulus; be aware this may differ from how fract() is defined in other domains. For example: fract(-1.2) = 0.8 in DX.

Obey round mode, result clamped to 0x3fefffffffffffff.

V_FREXP_EXP_I32_F32 51

Extract the exponent of a single-precision float input and store the result as a signed 32-bit integer into a vector register.

```
if ((64'F(S0.f32) == +INF) || (64'F(S0.f32) == -INF) || isNAN(64'F(S0.f32))) then
    D0.i32 = 0
```

```
else
    D0.i32 = exponent(S0.f32) - 127 + 1
endif
```

Notes

This operation satisfies the invariant S0.f32 = significand * (2 ** exponent). See also V_FREXP_MANT_F32, which returns the significand. See the C library function `frexp()` for more information.

V_FREXP_MANT_F32 52

Extract the binary significand, or mantissa, of a single-precision float input and store the result as a single-precision float into a vector register.

```
if ((64'F(S0.f32) == +INF) || (64'F(S0.f32) == -INF) || isNAN(64'F(S0.f32))) then
    D0.f32 = S0.f32
else
    D0.f32 = mantissa(S0.f32)
endif
```

Notes

This operation satisfies the invariant S0.f32 = significand * (2 ** exponent). Result range is in (-1.0,-0.5][0.5,1.0) in normal cases. See also V_FREXP_EXP_I_F32, which returns integer exponent. See the C library function `frexp()` for more information.

V_CLREXCP 53

Clear this wave's exception state in the vector ALU.

V_MOV_B64 56

Move data from a 64-bit vector input into a vector register.

```
D0.b64 = S0.b64
```

Notes

Floating-point modifiers are valid for this instruction if S0.u64 is a 64-bit floating point value. This instruction is suitable for negating or taking the absolute value of a floating-point value.

V_CVT_F16_U16 57

Convert from an unsigned 16-bit integer input to a half-precision float value and store the result into a vector register.

```
D0.f16 = u16_to_f16(S0.u16)
```

Notes

0.5ULP accuracy, supports denormals, rounding, exception flags and saturation.

V_CVT_F16_I16 58

Convert from a signed 16-bit integer input to a half-precision float value and store the result into a vector register.

```
D0.f16 = i16_to_f16(S0.i16)
```

Notes

0.5ULP accuracy, supports denormals, rounding, exception flags and saturation.

V_CVT_U16_F16 59

Convert from a half-precision float input to an unsigned 16-bit integer value and store the result into a vector register.

```
D0.u16 = f16_to_u16(S0.f16)
```

Notes

1ULP accuracy, supports rounding, exception flags and saturation. FP16 denormals are accepted. Conversion is done with truncation.

Generation of the INEXACT exception is controlled by the CLAMP bit. INEXACT exceptions are enabled for this conversion iff CLAMP == 1.

V_CVT_I16_F16 60

Convert from a half-precision float input to a signed 16-bit integer value and store the result into a vector register.

```
D0.i16 = f16_to_i16(S0.f16)
```

Notes

1ULP accuracy, supports rounding, exception flags and saturation. FP16 denormals are accepted. Conversion is done with truncation.

Generation of the INEXACT exception is controlled by the CLAMP bit. INEXACT exceptions are enabled for this conversion iff CLAMP == 1.

V_RCP_F16 61

Calculate the reciprocal of the half-precision float input using IEEE rules and store the result into a vector register.

```
D0.f16 = 16'1.0 / S0.f16
```

Notes

0.51ULP accuracy.

Functional examples:

```
V_RCP_F16(0xfc00) => 0x8000     // rcp(-INF) = -0
V_RCP_F16(0xc000) => 0xb800     // rcp(-2.0) = -0.5
V_RCP_F16(0x8000) => 0xfc00     // rcp(-0.0) = -INF
V_RCP_F16(0x0000) => 0x7c00     // rcp(+0.0) = +INF
V_RCP_F16(0x7c00) => 0x0000     // rcp(+INF) = +0
```

V_SQRT_F16 62

Calculate the square root of the half-precision float input using IEEE rules and store the result into a vector register.

```
D0.f16 = sqrt(S0.f16)
```

Notes

0.51ULP accuracy, denormals are supported.

Functional examples:

```
V_SQRT_F16(0xfc00) => 0xfe00     // sqrt(-INF) = NAN
V_SQRT_F16(0x8000) => 0x8000     // sqrt(-0.0) = -0
V_SQRT_F16(0x0000) => 0x0000     // sqrt(+0.0) = +0
V_SQRT_F16(0x4400) => 0x4000     // sqrt(+4.0) = +2.0
```

```
V_SQRT_F16(0x7c00) => 0x7c00     // sqrt(+INF) = +INF
```

V_RSQ_F16 63

Calculate the reciprocal of the square root of the half-precision float input using IEEE rules and store the result into a vector register.

```
D0.f16 = 16'1.0 / sqrt(S0.f16)
```

Notes

0.51ULP accuracy, denormals are supported.

Functional examples:

```
V_RSQ_F16(0xfc00) => 0xfe00     // rsq(-INF) = NAN
V_RSQ_F16(0x8000) => 0xfc00     // rsq(-0.0) = -INF
V_RSQ_F16(0x0000) => 0x7c00     // rsq(+0.0) = +INF
V_RSQ_F16(0x4400) => 0x3800     // rsq(+4.0) = +0.5
V_RSQ_F16(0x7c00) => 0x0000     // rsq(+INF) = +0
```

V_LOG_F16 64

Calculate the base 2 logarithm of the half-precision float input and store the result into a vector register.

```
D0.f16 = log2(S0.f16)
```

Notes

0.51ULP accuracy, denormals are supported.

Functional examples:

```
V_LOG_F16(0xfc00) => 0xfe00     // log(-INF) = NAN
V_LOG_F16(0xbc00) => 0xfe00     // log(-1.0) = NAN
V_LOG_F16(0x8000) => 0xfc00     // log(-0.0) = -INF
V_LOG_F16(0x0000) => 0xfc00     // log(+0.0) = -INF
V_LOG_F16(0x3c00) => 0x0000     // log(+1.0) = 0
V_LOG_F16(0x7c00) => 0x7c00     // log(+INF) = +INF
```

V_EXP_F16 65

Calculate 2 raised to the power of the half-precision float input and store the result into a vector register.

```
D0.f16 = pow(16'2.0, S0.f16)
```

Notes

0.51ULP accuracy, denormals are supported.

Functional examples:

```
V_EXP_F16(0xfc00) => 0x0000     // exp(-INF) = 0
V_EXP_F16(0x8000) => 0x3c00     // exp(-0.0) = 1
V_EXP_F16(0x7c00) => 0x7c00     // exp(+INF) = +INF
```

V_FREXP_MANT_F16 66

Extract the binary significand, or mantissa, of a half-precision float input and store the result as a half-precision float into a vector register.

```
if ((64'F(S0.f16) == +INF) || (64'F(S0.f16) == -INF) || isNAN(64'F(S0.f16))) then
    D0.f16 = S0.f16
else
    D0.f16 = mantissa(S0.f16)
endif
```

Notes

This operation satisfies the invariant S0.f16 = significand * (2 ** exponent). Result range is in (-1.0,-0.5][0.5,1.0) in normal cases. See also V_FREXP_EXP_I_F16, which returns integer exponent. See the C library function `frexp()` for more information.

V_FREXP_EXP_I16_F16 67

Extract the exponent of a half-precision float input and store the result as a signed 16-bit integer into a vector register.

```
if ((64'F(S0.f16) == +INF) || (64'F(S0.f16) == -INF) || isNAN(64'F(S0.f16))) then
    D0.i16 = 16'0
else
    D0.i16 = 16'I(exponent(S0.f16) - 15 + 1)
endif
```

Notes

This operation satisfies the invariant S0.f16 = significand * (2 ** exponent). See also V_FREXP_MANT_F16, which returns the significand. See the C library function `frexp()` for more information.

V_FLOOR_F16 68

Round the half-precision float input down to previous integer and store the result in floating point format into a vector register.

```
D0.f16 = trunc(S0.f16);
if ((S0.f16 < 16'0.0) && (S0.f16 != D0.f16)) then
    D0.f16 += -16'1.0
endif
```

V_CEIL_F16 69

Round the half-precision float input up to next integer and store the result in floating point format into a vector register.

```
D0.f16 = trunc(S0.f16);
if ((S0.f16 > 16'0.0) && (S0.f16 != D0.f16)) then
    D0.f16 += 16'1.0
endif
```

V_TRUNC_F16 70

Compute the integer part of a half-precision float input using round toward zero semantics and store the result in floating point format into a vector register.

```
D0.f16 = trunc(S0.f16)
```

V_RNDNE_F16 71

Round the half-precision float input to the nearest even integer and store the result in floating point format into a vector register.

```
D0.f16 = floor(S0.f16 + 16'0.5);
if (isEven(64'F(floor(S0.f16))) && (fract(S0.f16) == 16'0.5)) then
    D0.f16 -= 16'1.0
endif
```

V_FRACT_F16 72

Compute the fractional portion of a half-precision float input and store the result in floating point format into a vector register.

```
D0.f16 = S0.f16 + -floor(S0.f16)
```

Notes

0.5ULP accuracy, denormals are accepted.

This is intended to comply with the DX specification of fract where the function behaves like an extension of integer modulus; be aware this may differ from how fract() is defined in other domains. For example: fract(-1.2) = 0.8 in DX.

V_SIN_F16 73

Calculate the trigonometric sine of a half-precision float value using IEEE rules and store the result into a vector register. The operand is calculated by scaling the vector input by 2 PI.

```
D0.f16 = sin(S0.f16 * 16'F(PI * 2.0))
```

Notes

Denormals are supported. Full range input is supported.

Functional examples:

```
V_SIN_F16(0xfc00) => 0xfe00     // sin(-INF) = NAN
V_SIN_F16(0xfbff) => 0x0000     // Most negative finite FP16
V_SIN_F16(0x8000) => 0x8000     // sin(-0.0) = -0
V_SIN_F16(0x3400) => 0x3c00     // sin(0.25) = 1
V_SIN_F16(0x7bff) => 0x0000     // Most positive finite FP16
V_SIN_F16(0x7c00) => 0xfe00     // sin(+INF) = NAN
```

V_COS_F16 74

Calculate the trigonometric cosine of a half-precision float value using IEEE rules and store the result into a vector register. The operand is calculated by scaling the vector input by 2 PI.

```
D0.f16 = cos(S0.f16 * 16'F(PI * 2.0))
```

Notes

Denormals are supported. Full range input is supported.

Functional examples:

```
V_COS_F16(0xfc00) => 0xfe00     // cos(-INF) = NAN
V_COS_F16(0xfbff) => 0x3c00     // Most negative finite FP16
V_COS_F16(0x8000) => 0x3c00     // cos(-0.0) = 1
V_COS_F16(0x3400) => 0x0000     // cos(0.25) = 0
V_COS_F16(0x7bff) => 0x3c00     // Most positive finite FP16
V_COS_F16(0x7c00) => 0xfe00     // cos(+INF) = NAN
```

V_CVT_NORM_I16_F16 77

Convert from a half-precision float input to a signed normalized short and store the result into a vector register.

```
D0.i16 = f16_to_snorm(S0.f16)
```

Notes

0.5ULP accuracy, supports rounding, exception flags and saturation, denormals are supported.

V_CVT_NORM_U16_F16 78

Convert from a half-precision float input to an unsigned normalized short and store the result into a vector register.

```
D0.u16 = f16_to_unorm(S0.f16)
```

Notes

0.5ULP accuracy, supports rounding, exception flags and saturation, denormals are supported.

V_SAT_PK_U8_I16 79

Given two 16-bit signed integer inputs, saturate each input over an 8-bit unsigned range, pack the resulting values into a 16-bit word and store the result into a vector register.

```
SAT8 = lambda(n) (
    if n.i32 <= 0 then
```

```
        return 8'0U
    elsif n >= 16'I(0xff) then
        return 8'255U
    else
        return n[7 : 0].u8
    endif);
D0 = { 16'0, SAT8(S0[31 : 16].i16), SAT8(S0[15 : 0].i16) }
```

Notes

Used for 4x16bit data packed as 4x8bit data.

V_SWAP_B32 81

Swap the values in two vector registers.

```
tmp = D0.b32;
D0.b32 = S0.b32;
S0.b32 = tmp
```

Notes

Input and output modifiers not supported; this is an untyped operation.

V_ACCVGPR_MOV_B32 82

Move data from one accumulator register to another accumulator register.

V_CVT_F32_FP8 84

Convert from an FP8 float input to a single-precision float value and store the result into a vector register.

```
if SDWA_SRC0_SEL == BYTE1.b3 then
    D0.f32 = fp8_to_f32(S0[15 : 8].fp8)
elsif SDWA_SRC0_SEL == BYTE2.b3 then
    D0.f32 = fp8_to_f32(S0[23 : 16].fp8)
elsif SDWA_SRC0_SEL == BYTE3.b3 then
    D0.f32 = fp8_to_f32(S0[31 : 24].fp8)
else
    // BYTE0 implied
    D0.f32 = fp8_to_f32(S0[7 : 0].fp8)
endif
```

Notes

SDWA encoding allows SRC0_SEL to control which byte of S0 is converted. Only the BYTE selects of SRC0_SEL are legal. If this instruction is not encoded in SDWA then BYTE0 is implied.

V_CVT_F32_BF8 85

Convert from a BF8 float input to a single-precision float value and store the result into a vector register.

```
if SDWA_SRC0_SEL == BYTE1.b3 then
    D0.f32 = bf8_to_f32(S0[15 : 8].bf8)
elsif SDWA_SRC0_SEL == BYTE2.b3 then
    D0.f32 = bf8_to_f32(S0[23 : 16].bf8)
elsif SDWA_SRC0_SEL == BYTE3.b3 then
    D0.f32 = bf8_to_f32(S0[31 : 24].bf8)
else
    // BYTE0 implied
    D0.f32 = bf8_to_f32(S0[7 : 0].bf8)
endif
```

Notes

SDWA encoding allows SRC0_SEL to control which byte of S0 is converted. Only the BYTE selects of SRC0_SEL are legal. If this instruction is not encoded in SDWA then BYTE0 is implied.

V_CVT_PK_F32_FP8 86

Convert from a packed 2-component FP8 float input to a packed single-precision float value and store the result into a vector register.

```
tmp = SDWA_SRC0_SEL[1 : 0] == WORD1.b2 ? S0[31 : 16] : S0[15 : 0];
D0[31 : 0].f32 = fp8_to_f32(tmp[7 : 0].fp8);
D0[63 : 32].f32 = fp8_to_f32(tmp[15 : 8].fp8)
```

Notes

SDWA encoding allows SRC0_SEL to control which word of S0 is converted. Only the WORD selects of SRC0_SEL are legal. If this instruction is not encoded in SDWA then WORD0 is implied.

V_CVT_PK_F32_BF8 87

Convert from a packed 2-component BF8 float input to a packed single-precision float value and store the result into a vector register.

```
tmp = SDWA_SRC0_SEL[1 : 0] == WORD1.b2 ? S0[31 : 16] : S0[15 : 0];
D0[31 : 0].f32 = bf8_to_f32(tmp[7 : 0].bf8);
```

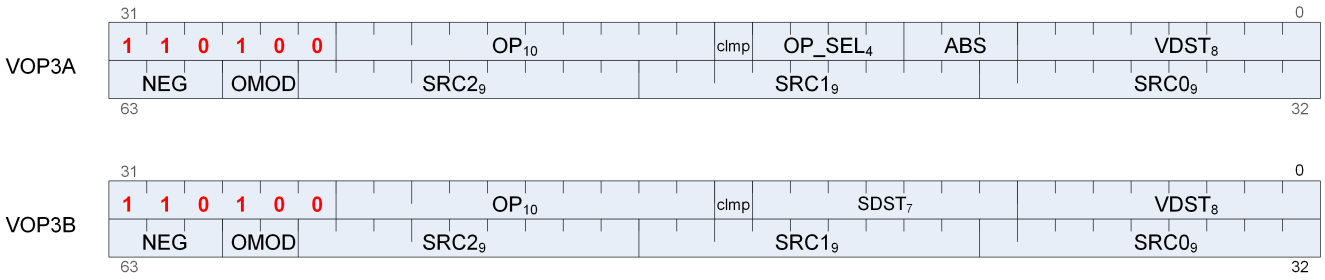
```
D0[63 : 32].f32 = bf8_to_f32(tmp[15 : 8].bf8)
```

Notes

SDWA encoding allows SRC0_SEL to control which word of S0 is converted. Only the WORD selects of SRC0_SEL are legal. If this instruction is not encoded in SDWA then WORD0 is implied.

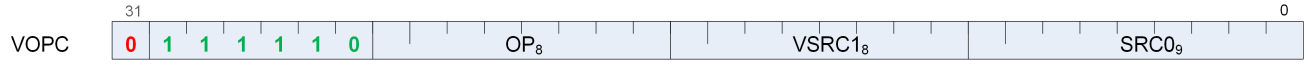
12.8.1. VOP1 using VOP3 encoding

Instructions in this format may also be encoded as VOP3. This allows access to the extra control bits (e.g. ABS, OMOD) in exchange for not being able to use a literal constant. The VOP3 opcode is: VOP2 opcode + 0x140.



12.9. VOPC Instructions

The bitfield map for VOPC is:




```
where:

SRC0  = First operand for instruction.
VSRC1 = Second operand for instruction.
OP    = Instructions.
All VOPC instructions can alternatively be encoded in the VOP3A format.
```

Compare instructions perform the same compare operation on each lane (workItem or thread) using that lane's private data, and producing a 1 bit result per lane into VCC or EXEC.

Instructions in this format may use a 32-bit literal constant which occurs immediately after the instruction.

Most compare instructions fall into one of two categories:

- Those which can use one of 16 compare operations (floating point types). "{COMPF}"
- Those which can use one of 8 compare operations (integer types). "{COMPI}"

The opcode number is such that for these the opcode number can be calculated from a base opcode number for the data type, plus an offset for the specific compare operation.

Table 58. Float Compare Operations

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)

Compare Operation	Opcode Offset	Description
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 59. Instructions with Sixteen Compare Operations

Instruction	Description	Hex Range
V_CMP_{COMPF}_F16	16-bit float compare.	0x20 to 0x2F
V_CMPX_{COMPF}_F16	16-bit float compare. Also writes EXEC.	0x30 to 0x3F
V_CMP_{COMPF}_F32	32-bit float compare.	0x40 to 0x4F
V_CMPX_{COMPF}_F32	32-bit float compare. Also writes EXEC.	0x50 to 0x5F
V_CMPS_{COMPF}_F64	64-bit float compare.	0x60 to 0x6F
V_CMPSX_{COMPF}_F64	64-bit float compare. Also writes EXEC.	0x70 to 0x7F

Table 60. Integer Compare Operations

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 61. Instructions with Eight Compare Operations

Instruction	Description	Hex Range
V_CMP_{COMPI}_I16	16-bit signed integer compare.	0xA0 - 0xA7
V_CMP_{COMPI}_U16	16-bit signed integer compare.	0xA8 - 0xAF
V_CMPX_{COMPI}_I16	16-bit unsigned integer compare. Also writes EXEC.	0xB0 - 0xB7
V_CMPX_{COMPI}_U16	16-bit unsigned integer compare. Also writes EXEC.	0xB8 - 0xBF
V_CMP_{COMPI}_I32	32-bit signed integer compare.	0xC0 - 0xC7
V_CMP_{COMPI}_U32	32-bit signed integer compare.	0xC8 - 0xCF
V_CMPX_{COMPI}_I32	32-bit unsigned integer compare. Also writes EXEC.	0xD0 - 0xD7
V_CMPX_{COMPI}_U32	32-bit unsigned integer compare. Also writes EXEC.	0xD8 - 0xDF
V_CMP_{COMPI}_I64	64-bit signed integer compare.	0xE0 - 0xE7
V_CMP_{COMPI}_U64	64-bit signed integer compare.	0xE8 - 0xEF
V_CMPX_{COMPI}_I64	64-bit unsigned integer compare. Also writes EXEC.	0xF0 - 0xF7
V_CMPX_{COMPI}_U64	64-bit unsigned integer compare. Also writes EXEC.	0xF8 - 0xFF

Table 62. VOPC Compare Opcodes

V_CMP_CLASS_F32 16

Evaluate the IEEE numeric class function specified as a 10 bit mask in the second input on the first input, a single-precision float, and set the vector condition code to the result. Store the result into VCC or a scalar register.

The function reports true if the floating point value is *any* of the numeric types selected in the 10 bit mask according to the following list:

S1.u[0] value is a signaling NAN.
S1.u[1] value is a quiet NAN.
S1.u[2] value is negative infinity.
S1.u[3] value is a negative normal value.
S1.u[4] value is a negative denormal value.
S1.u[5] value is negative zero.
S1.u[6] value is positive zero.
S1.u[7] value is a positive denormal value.
S1.u[8] value is a positive normal value.
S1.u[9] value is positive infinity.

```
declare result : 1'U;
if isSignalNAN(64'F(S0.f32)) then
    result = S1.u32[0]
elsif isQuietNAN(64'F(S0.f32)) then
    result = S1.u32[1]
elsif exponent(S0.f32) == 255 then
    // +-INF
    result = S1.u32[sign(S0.f32) ? 2 : 9]
elsif exponent(S0.f32) > 0 then
    // +-normal value
    result = S1.u32[sign(S0.f32) ? 3 : 8]
elsif 64'F(abs(S0.f32)) > 0.0 then
    // +-denormal value
    result = S1.u32[sign(S0.f32) ? 4 : 7]
else
    // +-0.0
    result = S1.u32[sign(S0.f32) ? 5 : 6]
endif;
D0.u64[laneId] = result;
// D0 = VCC in VOPC encoding.
```

V_CMPX_CLASS_F32 17

Evaluate the IEEE numeric class function specified as a 10 bit mask in the second input on the first input, a single-precision float, and set the vector condition code to the result. Store the result into the EXEC mask and to VCC or a scalar register.

The function reports true if the floating point value is *any* of the numeric types selected in the 10 bit mask according to the following list:

S1.u[0] value is a signaling NAN.
S1.u[1] value is a quiet NAN.
S1.u[2] value is negative infinity.
S1.u[3] value is a negative normal value.
S1.u[4] value is a negative denormal value.
S1.u[5] value is negative zero.
S1.u[6] value is positive zero.
S1.u[7] value is a positive denormal value.
S1.u[8] value is a positive normal value.
S1.u[9] value is positive infinity.

```
declare result : 1'U;
if isSignalNAN(64'F(S0.f32)) then
    result = S1.u32[0]
elsif isQuietNAN(64'F(S0.f32)) then
    result = S1.u32[1]
elsif exponent(S0.f32) == 255 then
    // +-INF
    result = S1.u32[sign(S0.f32) ? 2 : 9]
elsif exponent(S0.f32) > 0 then
    // +-normal value
    result = S1.u32[sign(S0.f32) ? 3 : 8]
elsif 64'F(abs(S0.f32)) > 0.0 then
    // +-denormal value
    result = S1.u32[sign(S0.f32) ? 4 : 7]
else
    // +-0.0
    result = S1.u32[sign(S0.f32) ? 5 : 6]
endif;
EXEC.u64[laneId] = D0.u64[laneId] = result
```

V_CMP_CLASS_F64 18

Evaluate the IEEE numeric class function specified as a 10 bit mask in the second input on the first input, a double-precision float, and set the vector condition code to the result. Store the result into VCC or a scalar register.

The function reports true if the floating point value is *any* of the numeric types selected in the 10 bit mask according to the following list:

S1.u[0] value is a signaling NAN.
S1.u[1] value is a quiet NAN.
S1.u[2] value is negative infinity.
S1.u[3] value is a negative normal value.
S1.u[4] value is a negative denormal value.
S1.u[5] value is negative zero.
S1.u[6] value is positive zero.
S1.u[7] value is a positive denormal value.
S1.u[8] value is a positive normal value.
S1.u[9] value is positive infinity.

```
declare result : 1'U;
if isSignalNAN(S0.f64) then
    result = S1.u32[0]
elsif isQuietNAN(S0.f64) then
    result = S1.u32[1]
elsif exponent(S0.f64) == 2047 then
    // +-INF
    result = S1.u32[sign(S0.f64) ? 2 : 9]
elsif exponent(S0.f64) > 0 then
    // +-normal value
    result = S1.u32[sign(S0.f64) ? 3 : 8]
elsif abs(S0.f64) > 0.0 then
    // +-denormal value
    result = S1.u32[sign(S0.f64) ? 4 : 7]
else
    // +-0.0
    result = S1.u32[sign(S0.f64) ? 5 : 6]
endif;
D0.u64[laneId] = result;
// D0 = VCC in VOPC encoding.
```

V_CMPX_CLASS_F64 19

Evaluate the IEEE numeric class function specified as a 10 bit mask in the second input on the first input, a double-precision float, and set the vector condition code to the result. Store the result into the EXEC mask and to VCC or a scalar register.

The function reports true if the floating point value is *any* of the numeric types selected in the 10 bit mask according to the following list:

S1.u[0] value is a signaling NAN.
S1.u[1] value is a quiet NAN.
S1.u[2] value is negative infinity.
S1.u[3] value is a negative normal value.
S1.u[4] value is a negative denormal value.
S1.u[5] value is negative zero.
S1.u[6] value is positive zero.
S1.u[7] value is a positive denormal value.
S1.u[8] value is a positive normal value.
S1.u[9] value is positive infinity.

```
declare result : 1'U;
if isSignalNAN(S0.f64) then
    result = S1.u32[0]
elsif isQuietNAN(S0.f64) then
    result = S1.u32[1]
elsif exponent(S0.f64) == 2047 then
    // +-INF
    result = S1.u32[sign(S0.f64) ? 2 : 9]
elsif exponent(S0.f64) > 0 then
    // +-normal value
```

```
    result = S1.u32[sign(S0.f64) ? 3 : 8]
elsif abs(S0.f64) > 0.0 then
    // +-denormal value
    result = S1.u32[sign(S0.f64) ? 4 : 7]
else
    // +-0.0
    result = S1.u32[sign(S0.f64) ? 5 : 6]
endif;
EXEC.u64[laneId] = D0.u64[laneId] = result
```

V_CMP_CLASS_F16 20

Evaluate the IEEE numeric class function specified as a 10 bit mask in the second input on the first input, a half-precision float, and set the vector condition code to the result. Store the result into VCC or a scalar register.

The function reports true if the floating point value is *any* of the numeric types selected in the 10 bit mask according to the following list:

S1.u[0] value is a signaling NAN.
S1.u[1] value is a quiet NAN.
S1.u[2] value is negative infinity.
S1.u[3] value is a negative normal value.
S1.u[4] value is a negative denormal value.
S1.u[5] value is negative zero.
S1.u[6] value is positive zero.
S1.u[7] value is a positive denormal value.
S1.u[8] value is a positive normal value.
S1.u[9] value is positive infinity.

```
declare result : 1'U;
if isSignalNAN(64'F(S0.f16)) then
    result = S1.u32[0]
elsif isQuietNAN(64'F(S0.f16)) then
    result = S1.u32[1]
elsif exponent(S0.f16) == 31 then
    // +-INF
    result = S1.u32[sign(S0.f16) ? 2 : 9]
elsif exponent(S0.f16) > 0 then
    // +-normal value
    result = S1.u32[sign(S0.f16) ? 3 : 8]
elsif 64'F(abs(S0.f16)) > 0.0 then
    // +-denormal value
    result = S1.u32[sign(S0.f16) ? 4 : 7]
else
    // +-0.0
    result = S1.u32[sign(S0.f16) ? 5 : 6]
endif;
D0.u64[laneId] = result;
// D0 = VCC in VOPC encoding.
```

Notes

Note that the S1 has a format of f16 since floating point literal constants are interpreted as 16 bit value for this opcode.

V_CMPX_CLASS_F16 21

Evaluate the IEEE numeric class function specified as a 10 bit mask in the second input on the first input, a half-precision float, and set the vector condition code to the result. Store the result into the EXEC mask and to VCC or a scalar register.

The function reports true if the floating point value is *any* of the numeric types selected in the 10 bit mask according to the following list:

S1.u[0] value is a signaling NAN.
S1.u[1] value is a quiet NAN.
S1.u[2] value is negative infinity.
S1.u[3] value is a negative normal value.
S1.u[4] value is a negative denormal value.
S1.u[5] value is negative zero.
S1.u[6] value is positive zero.
S1.u[7] value is a positive denormal value.
S1.u[8] value is a positive normal value.
S1.u[9] value is positive infinity.

```
declare result : 1'U;
if isSignalNAN(64'F(S0.f16)) then
    result = S1.u32[0]
elsif isQuietNAN(64'F(S0.f16)) then
    result = S1.u32[1]
elsif exponent(S0.f16) == 31 then
    // +-INF
    result = S1.u32[sign(S0.f16) ? 2 : 9]
elsif exponent(S0.f16) > 0 then
    // +-normal value
    result = S1.u32[sign(S0.f16) ? 3 : 8]
elsif 64'F(abs(S0.f16)) > 0.0 then
    // +-denormal value
    result = S1.u32[sign(S0.f16) ? 4 : 7]
else
    // +-0.0
    result = S1.u32[sign(S0.f16) ? 5 : 6]
endif;
EXEC.u64[laneId] = D0.u64[laneId] = result
```

Notes

Note that the S1 has a format of f16 since floating point literal constants are interpreted as 16 bit value for this opcode.

V_CMP_F_F16 32

Set the vector condition code to 0. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = 1'0U;
// D0 = VCC in VOPC encoding.
```

V_CMP_LT_F16 33

Set the vector condition code to 1 iff the first input is less than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.f16 < S1.f16;
// D0 = VCC in VOPC encoding.
```

V_CMP_EQ_F16 34

Set the vector condition code to 1 iff the first input is equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.f16 == S1.f16;
// D0 = VCC in VOPC encoding.
```

V_CMP_LE_F16 35

Set the vector condition code to 1 iff the first input is less than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.f16 <= S1.f16;
// D0 = VCC in VOPC encoding.
```

V_CMP_GT_F16 36

Set the vector condition code to 1 iff the first input is greater than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.f16 > S1.f16;
// D0 = VCC in VOPC encoding.
```

V_CMP_LG_F16 37

Set the vector condition code to 1 iff the first input is less than or greater than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.f16 <> S1.f16;
// D0 = VCC in VOPC encoding.
```

V_CMP_GE_F16 38

Set the vector condition code to 1 iff the first input is greater than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.f16 >= S1.f16;
// D0 = VCC in VOPC encoding.
```

V_CMP_O_F16 39

Set the vector condition code to 1 iff the first input is orderable to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = (!isNAN(64'F(S0.f16)) && !isNAN(64'F(S1.f16)));
// D0 = VCC in VOPC encoding.
```

V_CMP_U_F16 40

Set the vector condition code to 1 iff the first input is not orderable to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = (isNAN(64'F(S0.f16)) || isNAN(64'F(S1.f16)));
// D0 = VCC in VOPC encoding.
```

V_CMP_NGE_F16 41

Set the vector condition code to 1 iff the first input is not greater than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = !(S0.f16 >= S1.f16);
// With NAN inputs this is not the same operation as <
// D0 = VCC in VOPC encoding.
```

V_CMP_NLG_F16 42

Set the vector condition code to 1 iff the first input is not less than or greater than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = !(S0.f16 <> S1.f16);
// With NAN inputs this is not the same operation as ==
// D0 = VCC in VOPC encoding.
```

V_CMP_NGT_F16 43

Set the vector condition code to 1 iff the first input is not greater than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = !(S0.f16 > S1.f16);
// With NAN inputs this is not the same operation as <=
// D0 = VCC in VOPC encoding.
```

V_CMP_NLE_F16 44

Set the vector condition code to 1 iff the first input is not less than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = !(S0.f16 <= S1.f16);
// With NAN inputs this is not the same operation as >
// D0 = VCC in VOPC encoding.
```

V_CMP_NEQ_F16 45

Set the vector condition code to 1 iff the first input is not equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = !(S0.f16 == S1.f16);
// With NAN inputs this is not the same operation as !=
```

```
// D0 = VCC in VOPC encoding.
```

V_CMP_NLT_F16 46

Set the vector condition code to 1 iff the first input is not less than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = !(S0.f16 < S1.f16);
// With NAN inputs this is not the same operation as >=
// D0 = VCC in VOPC encoding.
```

V_CMP_TRU_F16 47

Set the vector condition code to 1. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = 1'1U;
// D0 = VCC in VOPC encoding.
```

V_CMPX_F_F16 48

Set the vector condition code to 0. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = 1'0U;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LT_F16 49

Set the vector condition code to 1 iff the first input is less than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.f16 < S1.f16;
// D0 = VCC in VOPC encoding.
```

V_CMPX_EQ_F16 50

Set the vector condition code to 1 iff the first input is equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.f16 == S1.f16;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LE_F16 51

Set the vector condition code to 1 iff the first input is less than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.f16 <= S1.f16;
// D0 = VCC in VOPC encoding.
```

V_CMPX_GT_F16 52

Set the vector condition code to 1 iff the first input is greater than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.f16 > S1.f16;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LG_F16 53

Set the vector condition code to 1 iff the first input is less than or greater than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.f16 <> S1.f16;
// D0 = VCC in VOPC encoding.
```

V_CMPX_GE_F16 54

Set the vector condition code to 1 iff the first input is greater than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.f16 >= S1.f16;
// D0 = VCC in VOPC encoding.
```

V_CMPX_O_F16 55

Set the vector condition code to 1 iff the first input is orderable to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = (!isNAN(64'F(S0.f16)) && !isNAN(64'F(S1.f16)));
// D0 = VCC in VOPC encoding.
```

V_CMPX_U_F16 56

Set the vector condition code to 1 iff the first input is not orderable to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = (isNAN(64'F(S0.f16)) || isNAN(64'F(S1.f16)));
// D0 = VCC in VOPC encoding.
```

V_CMPX_NGE_F16 57

Set the vector condition code to 1 iff the first input is not greater than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = !(S0.f16 >= S1.f16);
// With NAN inputs this is not the same operation as <
// D0 = VCC in VOPC encoding.
```

V_CMPX_NLG_F16 58

Set the vector condition code to 1 iff the first input is not less than or greater than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = !(S0.f16 <> S1.f16);
// With NAN inputs this is not the same operation as ==
// D0 = VCC in VOPC encoding.
```

V_CMPX_NGT_F16 59

Set the vector condition code to 1 iff the first input is not greater than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = !(S0.f16 > S1.f16);
// With NAN inputs this is not the same operation as <=
```

```
// D0 = VCC in VOPC encoding.
```

V_CMPX_NLE_F16 60

Set the vector condition code to 1 iff the first input is not less than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = !(S0.f16 <= S1.f16);
// With NAN inputs this is not the same operation as >
// D0 = VCC in VOPC encoding.
```

V_CMPX_NEQ_F16 61

Set the vector condition code to 1 iff the first input is not equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = !(S0.f16 == S1.f16);
// With NAN inputs this is not the same operation as !=
// D0 = VCC in VOPC encoding.
```

V_CMPX_NLT_F16 62

Set the vector condition code to 1 iff the first input is not less than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = !(S0.f16 < S1.f16);
// With NAN inputs this is not the same operation as >=
// D0 = VCC in VOPC encoding.
```

V_CMPX_TRU_F16 63

Set the vector condition code to 1. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = 1'1U;
// D0 = VCC in VOPC encoding.
```

V_CMP_F_F32 64

Set the vector condition code to 0. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = 1'0U;
// D0 = VCC in VOPC encoding.
```

V_CMP_LT_F32 65

Set the vector condition code to 1 iff the first input is less than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.f32 < S1.f32;
// D0 = VCC in VOPC encoding.
```

V_CMP_EQ_F32 66

Set the vector condition code to 1 iff the first input is equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.f32 == S1.f32;
// D0 = VCC in VOPC encoding.
```

V_CMP_LE_F32 67

Set the vector condition code to 1 iff the first input is less than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.f32 <= S1.f32;
// D0 = VCC in VOPC encoding.
```

V_CMP_GT_F32 68

Set the vector condition code to 1 iff the first input is greater than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.f32 > S1.f32;
// D0 = VCC in VOPC encoding.
```

V_CMP_LG_F32 69

Set the vector condition code to 1 iff the first input is less than or greater than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.f32 <> S1.f32;
// D0 = VCC in VOPC encoding.
```

V_CMP_GE_F32 70

Set the vector condition code to 1 iff the first input is greater than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.f32 >= S1.f32;
// D0 = VCC in VOPC encoding.
```

V_CMP_O_F32 71

Set the vector condition code to 1 iff the first input is orderable to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = (!isNAN(64'F(S0.f32)) && !isNAN(64'F(S1.f32)));
// D0 = VCC in VOPC encoding.
```

V_CMP_U_F32 72

Set the vector condition code to 1 iff the first input is not orderable to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = (isNAN(64'F(S0.f32)) || isNAN(64'F(S1.f32)));
// D0 = VCC in VOPC encoding.
```

V_CMP_NGE_F32 73

Set the vector condition code to 1 iff the first input is not greater than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = !(S0.f32 >= S1.f32);
```

```
// With NAN inputs this is not the same operation as <
// D0 = VCC in VOPC encoding.
```

V_CMP_NLG_F32 74

Set the vector condition code to 1 iff the first input is not less than or greater than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = !(S0.f32 <> S1.f32);
// With NAN inputs this is not the same operation as ==
// D0 = VCC in VOPC encoding.
```

V_CMP_NGT_F32 75

Set the vector condition code to 1 iff the first input is not greater than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = !(S0.f32 > S1.f32);
// With NAN inputs this is not the same operation as <=
// D0 = VCC in VOPC encoding.
```

V_CMP_NLE_F32 76

Set the vector condition code to 1 iff the first input is not less than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = !(S0.f32 <= S1.f32);
// With NAN inputs this is not the same operation as >
// D0 = VCC in VOPC encoding.
```

V_CMP_NEQ_F32 77

Set the vector condition code to 1 iff the first input is not equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = !(S0.f32 == S1.f32);
// With NAN inputs this is not the same operation as !=
// D0 = VCC in VOPC encoding.
```

V_CMP_NLT_F32 78

Set the vector condition code to 1 iff the first input is not less than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = !(S0.f32 < S1.f32);
// With NAN inputs this is not the same operation as >=
// D0 = VCC in VOPC encoding.
```

V_CMP_TRU_F32 79

Set the vector condition code to 1. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = 1'1U;
// D0 = VCC in VOPC encoding.
```

V_CMPX_F_F32 80

Set the vector condition code to 0. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = 1'0U;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LT_F32 81

Set the vector condition code to 1 iff the first input is less than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.f32 < S1.f32;
// D0 = VCC in VOPC encoding.
```

V_CMPX_EQ_F32 82

Set the vector condition code to 1 iff the first input is equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.f32 == S1.f32;
```

```
// D0 = VCC in VOPC encoding.
```

V_CMPX_LE_F32 83

Set the vector condition code to 1 iff the first input is less than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.f32 <= S1.f32;
// D0 = VCC in VOPC encoding.
```

V_CMPX_GT_F32 84

Set the vector condition code to 1 iff the first input is greater than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.f32 > S1.f32;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LG_F32 85

Set the vector condition code to 1 iff the first input is less than or greater than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.f32 <> S1.f32;
// D0 = VCC in VOPC encoding.
```

V_CMPX_GE_F32 86

Set the vector condition code to 1 iff the first input is greater than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.f32 >= S1.f32;
// D0 = VCC in VOPC encoding.
```

V_CMPX_O_F32 87

Set the vector condition code to 1 iff the first input is orderable to the second input. Store the result into the

EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = (!isNAN(64'F(S0.f32)) && !isNAN(64'F(S1.f32)));
// D0 = VCC in VOPC encoding.
```

V_CMPX_U_F32 88

Set the vector condition code to 1 iff the first input is not orderable to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = (isNAN(64'F(S0.f32)) || isNAN(64'F(S1.f32)));
// D0 = VCC in VOPC encoding.
```

V_CMPX_NGE_F32 89

Set the vector condition code to 1 iff the first input is not greater than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = !(S0.f32 >= S1.f32);
// With NAN inputs this is not the same operation as <
// D0 = VCC in VOPC encoding.
```

V_CMPX_NLG_F32 90

Set the vector condition code to 1 iff the first input is not less than or greater than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = !(S0.f32 <> S1.f32);
// With NAN inputs this is not the same operation as ==
// D0 = VCC in VOPC encoding.
```

V_CMPX_NGT_F32 91

Set the vector condition code to 1 iff the first input is not greater than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = !(S0.f32 > S1.f32);
// With NAN inputs this is not the same operation as <=
```

```
// D0 = VCC in VOPC encoding.
```

V_CMPX_NLE_F32 92

Set the vector condition code to 1 iff the first input is not less than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = !(S0.f32 <= S1.f32);
// With NAN inputs this is not the same operation as >
// D0 = VCC in VOPC encoding.
```

V_CMPX_NEQ_F32 93

Set the vector condition code to 1 iff the first input is not equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = !(S0.f32 == S1.f32);
// With NAN inputs this is not the same operation as !=
// D0 = VCC in VOPC encoding.
```

V_CMPX_NLT_F32 94

Set the vector condition code to 1 iff the first input is not less than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = !(S0.f32 < S1.f32);
// With NAN inputs this is not the same operation as >=
// D0 = VCC in VOPC encoding.
```

V_CMPX_TRU_F32 95

Set the vector condition code to 1. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = 1'1U;
// D0 = VCC in VOPC encoding.
```

V_CMP_F_F64 96

Set the vector condition code to 0. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = 1'0U;
// D0 = VCC in VOPC encoding.
```

V_CMP_LT_F64 97

Set the vector condition code to 1 iff the first input is less than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.f64 < S1.f64;
// D0 = VCC in VOPC encoding.
```

V_CMP_EQ_F64 98

Set the vector condition code to 1 iff the first input is equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.f64 == S1.f64;
// D0 = VCC in VOPC encoding.
```

V_CMP_LE_F64 99

Set the vector condition code to 1 iff the first input is less than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.f64 <= S1.f64;
// D0 = VCC in VOPC encoding.
```

V_CMP_GT_F64 100

Set the vector condition code to 1 iff the first input is greater than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.f64 > S1.f64;
// D0 = VCC in VOPC encoding.
```

V_CMP_LG_F64 101

Set the vector condition code to 1 iff the first input is less than or greater than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.f64 <> S1.f64;
// D0 = VCC in VOPC encoding.
```

V_CMP_GE_F64 102

Set the vector condition code to 1 iff the first input is greater than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.f64 >= S1.f64;
// D0 = VCC in VOPC encoding.
```

V_CMP_O_F64 103

Set the vector condition code to 1 iff the first input is orderable to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = (!isNAN(S0.f64) && !isNAN(S1.f64));
// D0 = VCC in VOPC encoding.
```

V_CMP_U_F64 104

Set the vector condition code to 1 iff the first input is not orderable to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = (isNAN(S0.f64) || isNAN(S1.f64));
// D0 = VCC in VOPC encoding.
```

V_CMP_NGE_F64 105

Set the vector condition code to 1 iff the first input is not greater than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = !(S0.f64 >= S1.f64);
```

```
// With NAN inputs this is not the same operation as <
// D0 = VCC in VOPC encoding.
```

V_CMP_NLG_F64 106

Set the vector condition code to 1 iff the first input is not less than or greater than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = !(S0.f64 <> S1.f64);
// With NAN inputs this is not the same operation as ==
// D0 = VCC in VOPC encoding.
```

V_CMP_NGT_F64 107

Set the vector condition code to 1 iff the first input is not greater than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = !(S0.f64 > S1.f64);
// With NAN inputs this is not the same operation as <=
// D0 = VCC in VOPC encoding.
```

V_CMP_NLE_F64 108

Set the vector condition code to 1 iff the first input is not less than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = !(S0.f64 <= S1.f64);
// With NAN inputs this is not the same operation as >
// D0 = VCC in VOPC encoding.
```

V_CMP_NEQ_F64 109

Set the vector condition code to 1 iff the first input is not equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = !(S0.f64 == S1.f64);
// With NAN inputs this is not the same operation as !=
// D0 = VCC in VOPC encoding.
```

V_CMP_NLT_F64 110

Set the vector condition code to 1 iff the first input is not less than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = !(S0.f64 < S1.f64);
// With NAN inputs this is not the same operation as >=
// D0 = VCC in VOPC encoding.
```

V_CMP_TRU_F64 111

Set the vector condition code to 1. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = 1'1U;
// D0 = VCC in VOPC encoding.
```

V_CMPX_F_F64 112

Set the vector condition code to 0. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = 1'0U;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LT_F64 113

Set the vector condition code to 1 iff the first input is less than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.f64 < S1.f64;
// D0 = VCC in VOPC encoding.
```

V_CMPX_EQ_F64 114

Set the vector condition code to 1 iff the first input is equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.f64 == S1.f64;
```

```
// D0 = VCC in VOPC encoding.
```

V_CMPX_LE_F64 115

Set the vector condition code to 1 iff the first input is less than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.f64 <= S1.f64;
// D0 = VCC in VOPC encoding.
```

V_CMPX_GT_F64 116

Set the vector condition code to 1 iff the first input is greater than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.f64 > S1.f64;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LG_F64 117

Set the vector condition code to 1 iff the first input is less than or greater than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.f64 <> S1.f64;
// D0 = VCC in VOPC encoding.
```

V_CMPX_GE_F64 118

Set the vector condition code to 1 iff the first input is greater than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.f64 >= S1.f64;
// D0 = VCC in VOPC encoding.
```

V_CMPX_O_F64 119

Set the vector condition code to 1 iff the first input is orderable to the second input. Store the result into the

EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = (!isNAN(S0.f64) && !isNAN(S1.f64));
// D0 = VCC in VOPC encoding.
```

V_CMPX_U_F64 120

Set the vector condition code to 1 iff the first input is not orderable to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = (isNAN(S0.f64) || isNAN(S1.f64));
// D0 = VCC in VOPC encoding.
```

V_CMPX_NGE_F64 121

Set the vector condition code to 1 iff the first input is not greater than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = !(S0.f64 >= S1.f64);
// With NAN inputs this is not the same operation as <
// D0 = VCC in VOPC encoding.
```

V_CMPX_NLG_F64 122

Set the vector condition code to 1 iff the first input is not less than or greater than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = !(S0.f64 <> S1.f64);
// With NAN inputs this is not the same operation as ==
// D0 = VCC in VOPC encoding.
```

V_CMPX_NGT_F64 123

Set the vector condition code to 1 iff the first input is not greater than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = !(S0.f64 > S1.f64);
// With NAN inputs this is not the same operation as <=
```

```
// D0 = VCC in VOPC encoding.
```

V_CMPX_NLE_F64 124

Set the vector condition code to 1 iff the first input is not less than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = !(S0.f64 <= S1.f64);
// With NAN inputs this is not the same operation as >
// D0 = VCC in VOPC encoding.
```

V_CMPX_NEQ_F64 125

Set the vector condition code to 1 iff the first input is not equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = !(S0.f64 == S1.f64);
// With NAN inputs this is not the same operation as !=
// D0 = VCC in VOPC encoding.
```

V_CMPX_NLT_F64 126

Set the vector condition code to 1 iff the first input is not less than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = !(S0.f64 < S1.f64);
// With NAN inputs this is not the same operation as >=
// D0 = VCC in VOPC encoding.
```

V_CMPX_TRU_F64 127

Set the vector condition code to 1. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = 1'1U;
// D0 = VCC in VOPC encoding.
```

V_CMP_F_I16 160

Set the vector condition code to 0. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = 1'0U;
// D0 = VCC in VOPC encoding.
```

V_CMP_LT_I16 161

Set the vector condition code to 1 iff the first input is less than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.i16 < S1.i16;
// D0 = VCC in VOPC encoding.
```

V_CMP_EQ_I16 162

Set the vector condition code to 1 iff the first input is equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.i16 == S1.i16;
// D0 = VCC in VOPC encoding.
```

V_CMP_LE_I16 163

Set the vector condition code to 1 iff the first input is less than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.i16 <= S1.i16;
// D0 = VCC in VOPC encoding.
```

V_CMP_GT_I16 164

Set the vector condition code to 1 iff the first input is greater than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.i16 > S1.i16;
// D0 = VCC in VOPC encoding.
```

V_CMP_NE_I16 165

Set the vector condition code to 1 iff the first input is not equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.i16 <> S1.i16;
// D0 = VCC in VOPC encoding.
```

V_CMP_GE_I16 166

Set the vector condition code to 1 iff the first input is greater than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.i16 >= S1.i16;
// D0 = VCC in VOPC encoding.
```

V_CMP_T_I16 167

Set the vector condition code to 1. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = 1'1U;
// D0 = VCC in VOPC encoding.
```

V_CMP_F_U16 168

Set the vector condition code to 0. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = 1'0U;
// D0 = VCC in VOPC encoding.
```

V_CMP_LT_U16 169

Set the vector condition code to 1 iff the first input is less than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.u16 < S1.u16;
// D0 = VCC in VOPC encoding.
```

V_CMP_EQ_U16 170

Set the vector condition code to 1 iff the first input is equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.u16 == S1.u16;
// D0 = VCC in VOPC encoding.
```

V_CMP_LE_U16 171

Set the vector condition code to 1 iff the first input is less than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.u16 <= S1.u16;
// D0 = VCC in VOPC encoding.
```

V_CMP_GT_U16 172

Set the vector condition code to 1 iff the first input is greater than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.u16 > S1.u16;
// D0 = VCC in VOPC encoding.
```

V_CMP_NE_U16 173

Set the vector condition code to 1 iff the first input is not equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.u16 <> S1.u16;
// D0 = VCC in VOPC encoding.
```

V_CMP_GE_U16 174

Set the vector condition code to 1 iff the first input is greater than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.u16 >= S1.u16;
// D0 = VCC in VOPC encoding.
```

V_CMP_T_U16 175

Set the vector condition code to 1. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = 1'1U;
// D0 = VCC in VOPC encoding.
```

V_CMPX_F_I16 176

Set the vector condition code to 0. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = 1'0U;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LT_I16 177

Set the vector condition code to 1 iff the first input is less than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.i16 < S1.i16;
// D0 = VCC in VOPC encoding.
```

V_CMPX_EQ_I16 178

Set the vector condition code to 1 iff the first input is equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.i16 == S1.i16;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LE_I16 179

Set the vector condition code to 1 iff the first input is less than or equal to the second input. Store the result into

the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.i16 <= S1.i16;
// D0 = VCC in VOPC encoding.
```

V_CMPX_GT_I16 180

Set the vector condition code to 1 iff the first input is greater than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.i16 > S1.i16;
// D0 = VCC in VOPC encoding.
```

V_CMPX_NE_I16 181

Set the vector condition code to 1 iff the first input is not equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.i16 <> S1.i16;
// D0 = VCC in VOPC encoding.
```

V_CMPX_GE_I16 182

Set the vector condition code to 1 iff the first input is greater than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.i16 >= S1.i16;
// D0 = VCC in VOPC encoding.
```

V_CMPX_T_I16 183

Set the vector condition code to 1. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = 1'1U;
// D0 = VCC in VOPC encoding.
```

V_CMPX_F_U16 184

Set the vector condition code to 0. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = 1'0U;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LT_U16 185

Set the vector condition code to 1 iff the first input is less than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.u16 < S1.u16;
// D0 = VCC in VOPC encoding.
```

V_CMPX_EQ_U16 186

Set the vector condition code to 1 iff the first input is equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.u16 == S1.u16;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LE_U16 187

Set the vector condition code to 1 iff the first input is less than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.u16 <= S1.u16;
// D0 = VCC in VOPC encoding.
```

V_CMPX_GT_U16 188

Set the vector condition code to 1 iff the first input is greater than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.u16 > S1.u16;
// D0 = VCC in VOPC encoding.
```

V_CMPX_NE_U16 189

Set the vector condition code to 1 iff the first input is not equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.u16 <> S1.u16;
// D0 = VCC in VOPC encoding.
```

V_CMPX_GE_U16 190

Set the vector condition code to 1 iff the first input is greater than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.u16 >= S1.u16;
// D0 = VCC in VOPC encoding.
```

V_CMPX_T_U16 191

Set the vector condition code to 1. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = 1'1U;
// D0 = VCC in VOPC encoding.
```

V_CMP_F_I32 192

Set the vector condition code to 0. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = 1'0U;
// D0 = VCC in VOPC encoding.
```

V_CMP_LT_I32 193

Set the vector condition code to 1 iff the first input is less than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.i32 < S1.i32;
```

```
// D0 = VCC in VOPC encoding.
```

V_CMP_EQ_I32 194

Set the vector condition code to 1 iff the first input is equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.i32 == S1.i32;
// D0 = VCC in VOPC encoding.
```

V_CMP_LE_I32 195

Set the vector condition code to 1 iff the first input is less than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.i32 <= S1.i32;
// D0 = VCC in VOPC encoding.
```

V_CMP_GT_I32 196

Set the vector condition code to 1 iff the first input is greater than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.i32 > S1.i32;
// D0 = VCC in VOPC encoding.
```

V_CMP_NE_I32 197

Set the vector condition code to 1 iff the first input is not equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.i32 <> S1.i32;
// D0 = VCC in VOPC encoding.
```

V_CMP_GE_I32 198

Set the vector condition code to 1 iff the first input is greater than or equal to the second input. Store the result

into VCC or a scalar register.

```
D0.u64[laneId] = S0.i32 >= S1.i32;
// D0 = VCC in VOPC encoding.
```

V_CMP_T_I32 199

Set the vector condition code to 1. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = 1'1U;
// D0 = VCC in VOPC encoding.
```

V_CMP_F_U32 200

Set the vector condition code to 0. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = 1'0U;
// D0 = VCC in VOPC encoding.
```

V_CMP_LT_U32 201

Set the vector condition code to 1 iff the first input is less than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.u32 < S1.u32;
// D0 = VCC in VOPC encoding.
```

V_CMP_EQ_U32 202

Set the vector condition code to 1 iff the first input is equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.u32 == S1.u32;
// D0 = VCC in VOPC encoding.
```

V_CMP_LE_U32 203

Set the vector condition code to 1 iff the first input is less than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.u32 <= S1.u32;
// D0 = VCC in VOPC encoding.
```

V_CMP_GT_U32 204

Set the vector condition code to 1 iff the first input is greater than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.u32 > S1.u32;
// D0 = VCC in VOPC encoding.
```

V_CMP_NE_U32 205

Set the vector condition code to 1 iff the first input is not equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.u32 <> S1.u32;
// D0 = VCC in VOPC encoding.
```

V_CMP_GE_U32 206

Set the vector condition code to 1 iff the first input is greater than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.u32 >= S1.u32;
// D0 = VCC in VOPC encoding.
```

V_CMP_T_U32 207

Set the vector condition code to 1. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = 1'1U;
// D0 = VCC in VOPC encoding.
```

V_CMPX_F_I32 208

Set the vector condition code to 0. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = 1'0U;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LT_I32 209

Set the vector condition code to 1 iff the first input is less than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.i32 < S1.i32;
// D0 = VCC in VOPC encoding.
```

V_CMPX_EQ_I32 210

Set the vector condition code to 1 iff the first input is equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.i32 == S1.i32;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LE_I32 211

Set the vector condition code to 1 iff the first input is less than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.i32 <= S1.i32;
// D0 = VCC in VOPC encoding.
```

V_CMPX_GT_I32 212

Set the vector condition code to 1 iff the first input is greater than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.i32 > S1.i32;
// D0 = VCC in VOPC encoding.
```

V_CMPX_NE_I32 213

Set the vector condition code to 1 iff the first input is not equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.i32 <> S1.i32;
// D0 = VCC in VOPC encoding.
```

V_CMPX_GE_I32 214

Set the vector condition code to 1 iff the first input is greater than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.i32 >= S1.i32;
// D0 = VCC in VOPC encoding.
```

V_CMPX_T_I32 215

Set the vector condition code to 1. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = 1'1U;
// D0 = VCC in VOPC encoding.
```

V_CMPX_F_U32 216

Set the vector condition code to 0. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = 1'0U;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LT_U32 217

Set the vector condition code to 1 iff the first input is less than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.u32 < S1.u32;
```

```
// D0 = VCC in VOPC encoding.
```

V_CMPX_EQ_U32 218

Set the vector condition code to 1 iff the first input is equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.u32 == S1.u32;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LE_U32 219

Set the vector condition code to 1 iff the first input is less than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.u32 <= S1.u32;
// D0 = VCC in VOPC encoding.
```

V_CMPX_GT_U32 220

Set the vector condition code to 1 iff the first input is greater than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.u32 > S1.u32;
// D0 = VCC in VOPC encoding.
```

V_CMPX_NE_U32 221

Set the vector condition code to 1 iff the first input is not equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.u32 <> S1.u32;
// D0 = VCC in VOPC encoding.
```

V_CMPX_GE_U32 222

Set the vector condition code to 1 iff the first input is greater than or equal to the second input. Store the result

into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.u32 >= S1.u32;
// D0 = VCC in VOPC encoding.
```

V_CMPX_T_U32 223

Set the vector condition code to 1. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = 1'1U;
// D0 = VCC in VOPC encoding.
```

V_CMP_F_I64 224

Set the vector condition code to 0. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = 1'0U;
// D0 = VCC in VOPC encoding.
```

V_CMP_LT_I64 225

Set the vector condition code to 1 iff the first input is less than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.i64 < S1.i64;
// D0 = VCC in VOPC encoding.
```

V_CMP_EQ_I64 226

Set the vector condition code to 1 iff the first input is equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.i64 == S1.i64;
// D0 = VCC in VOPC encoding.
```

V_CMP_LE_I64 227

Set the vector condition code to 1 iff the first input is less than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.i64 <= S1.i64;
// D0 = VCC in VOPC encoding.
```

V_CMP_GT_I64 228

Set the vector condition code to 1 iff the first input is greater than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.i64 > S1.i64;
// D0 = VCC in VOPC encoding.
```

V_CMP_NE_I64 229

Set the vector condition code to 1 iff the first input is not equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.i64 <> S1.i64;
// D0 = VCC in VOPC encoding.
```

V_CMP_GE_I64 230

Set the vector condition code to 1 iff the first input is greater than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.i64 >= S1.i64;
// D0 = VCC in VOPC encoding.
```

V_CMP_T_I64 231

Set the vector condition code to 1. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = 1'1U;
// D0 = VCC in VOPC encoding.
```

V_CMP_F_U64 232

Set the vector condition code to 0. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = 1'0U;
// D0 = VCC in VOPC encoding.
```

V_CMP_LT_U64 233

Set the vector condition code to 1 iff the first input is less than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.u64 < S1.u64;
// D0 = VCC in VOPC encoding.
```

V_CMP_EQ_U64 234

Set the vector condition code to 1 iff the first input is equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.u64 == S1.u64;
// D0 = VCC in VOPC encoding.
```

V_CMP_LE_U64 235

Set the vector condition code to 1 iff the first input is less than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.u64 <= S1.u64;
// D0 = VCC in VOPC encoding.
```

V_CMP_GT_U64 236

Set the vector condition code to 1 iff the first input is greater than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.u64 > S1.u64;
// D0 = VCC in VOPC encoding.
```

V_CMP_NE_U64 237

Set the vector condition code to 1 iff the first input is not equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.u64 <> S1.u64;
// D0 = VCC in VOPC encoding.
```

V_CMP_GE_U64 238

Set the vector condition code to 1 iff the first input is greater than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.u64 >= S1.u64;
// D0 = VCC in VOPC encoding.
```

V_CMP_T_U64 239

Set the vector condition code to 1. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = 1'1U;
// D0 = VCC in VOPC encoding.
```

V_CMPX_F_I64 240

Set the vector condition code to 0. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = 1'0U;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LT_I64 241

Set the vector condition code to 1 iff the first input is less than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.i64 < S1.i64;
```

```
// D0 = VCC in VOPC encoding.
```

V_CMPX_EQ_I64 242

Set the vector condition code to 1 iff the first input is equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.i64 == S1.i64;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LE_I64 243

Set the vector condition code to 1 iff the first input is less than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.i64 <= S1.i64;
// D0 = VCC in VOPC encoding.
```

V_CMPX_GT_I64 244

Set the vector condition code to 1 iff the first input is greater than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.i64 > S1.i64;
// D0 = VCC in VOPC encoding.
```

V_CMPX_NE_I64 245

Set the vector condition code to 1 iff the first input is not equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.i64 <> S1.i64;
// D0 = VCC in VOPC encoding.
```

V_CMPX_GE_I64 246

Set the vector condition code to 1 iff the first input is greater than or equal to the second input. Store the result

into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.i64 >= S1.i64;
// D0 = VCC in VOPC encoding.
```

V_CMPX_T_I64 247

Set the vector condition code to 1. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = 1'1U;
// D0 = VCC in VOPC encoding.
```

V_CMPX_F_U64 248

Set the vector condition code to 0. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = 1'0U;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LT_U64 249

Set the vector condition code to 1 iff the first input is less than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.u64 < S1.u64;
// D0 = VCC in VOPC encoding.
```

V_CMPX_EQ_U64 250

Set the vector condition code to 1 iff the first input is equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.u64 == S1.u64;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LE_U64 251

Set the vector condition code to 1 iff the first input is less than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.u64 <= S1.u64;
// D0 = VCC in VOPC encoding.
```

V_CMPX_GT_U64 252

Set the vector condition code to 1 iff the first input is greater than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.u64 > S1.u64;
// D0 = VCC in VOPC encoding.
```

V_CMPX_NE_U64 253

Set the vector condition code to 1 iff the first input is not equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.u64 <> S1.u64;
// D0 = VCC in VOPC encoding.
```

V_CMPX_GE_U64 254

Set the vector condition code to 1 iff the first input is greater than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.u64 >= S1.u64;
// D0 = VCC in VOPC encoding.
```

V_CMPX_T_U64 255

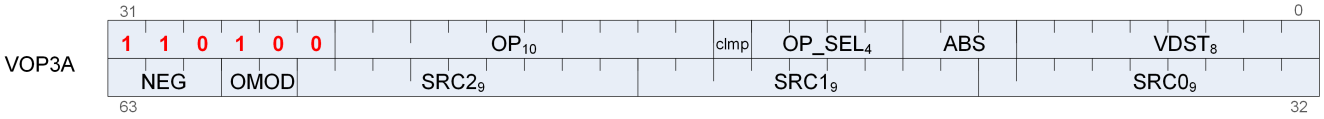
Set the vector condition code to 1. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = 1'1U;
// D0 = VCC in VOPC encoding.
```

12.9.1. VOPC using VOP3A encoding

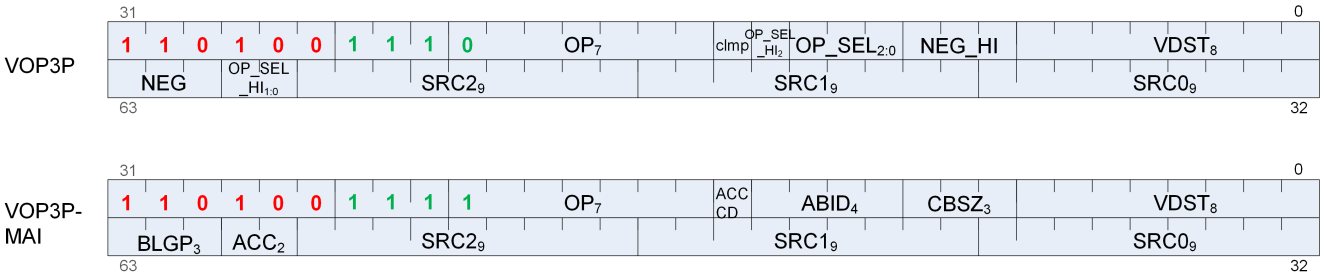
Instructions in this format may also be encoded as VOP3A. This allows access to the extra control bits (e.g. ABS, OMOD) in exchange for not being able to use a literal constant. The VOP3 opcode is: VOP2 opcode + 0x000.

When the CLAMP microcode bit is set to 1, these compare instructions signal an exception when either of the inputs is NaN. When CLAMP is set to zero, NaN does not signal an exception. The second eight VOPC instructions have {OP8} embedded in them. This refers to each of the compare operations listed below.




```
where:

  VDST = Destination for instruction in the VGPR.
  ABS = Floating-point absolute value.
  CLMP = Clamp output.
  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.
```

12.10. VOP3P Instructions



V_PK_MAD_I16 0

Multiply two packed signed 16-bit integer inputs component-wise, add a packed signed 16-bit integer value from a third input component-wise, and store the result into a vector register.

```
tmp[31 : 16].i16 = S0[31 : 16].i16 * S1[31 : 16].i16 + S2[31 : 16].i16;
tmp[15 : 0].i16 = S0[15 : 0].i16 * S1[15 : 0].i16 + S2[15 : 0].i16;
D0.b32 = tmp.b32
```

V_PK_MUL_LO_U16 1

Multiply two packed unsigned 16-bit integer inputs component-wise and store the low bits of each resulting component into a vector register.

```
tmp[31 : 16].u16 = S0[31 : 16].u16 * S1[31 : 16].u16;
tmp[15 : 0].u16 = S0[15 : 0].u16 * S1[15 : 0].u16;
D0.b32 = tmp.b32
```

V_PK_ADD_I16 2

Add two packed signed 16-bit integer inputs component-wise and store the result into a vector register. No carry-in or carry-out support.

```
tmp[31 : 16].i16 = S0[31 : 16].i16 + S1[31 : 16].i16;
tmp[15 : 0].i16 = S0[15 : 0].i16 + S1[15 : 0].i16;
D0.b32 = tmp.b32
```

V_PK_SUB_I16 3

Subtract the second packed signed 16-bit integer input from the first input component-wise and store the result into a vector register. No carry-in or carry-out support.

```
tmp[31 : 16].i16 = S0[31 : 16].i16 - S1[31 : 16].i16;
tmp[15 : 0].i16 = S0[15 : 0].i16 - S1[15 : 0].i16;
D0.b32 = tmp.b32
```

V_PK_LSHLREV_B16 4

Given a packed shift count in the *first* vector input, calculate the component-wise logical shift left of the *second* packed vector input and store the result into a vector register.

```
tmp[31 : 16].u16 = (S1[31 : 16].u16 << S0.u32[19 : 16].u32);
tmp[15 : 0].u16 = (S1[15 : 0].u16 << S0.u32[3 : 0].u32);
D0.b32 = tmp.b32
```

V_PK_LSHRREV_B16 5

Given a packed shift count in the *first* vector input, calculate the component-wise logical shift right of the *second* packed vector input and store the result into a vector register.

```
tmp[31 : 16].u16 = (S1[31 : 16].u16 >> S0.u32[19 : 16].u32);
tmp[15 : 0].u16 = (S1[15 : 0].u16 >> S0.u32[3 : 0].u32);
D0.b32 = tmp.b32
```

V_PK_ASHRREV_I16 6

Given a packed shift count in the *first* vector input, calculate the component-wise arithmetic shift right (preserving sign bit) of the *second* packed vector input and store the result into a vector register.

```
tmp[31 : 16].i16 = (S1[31 : 16].i16 >> S0.u32[19 : 16].u32);
tmp[15 : 0].i16 = (S1[15 : 0].i16 >> S0.u32[3 : 0].u32);
D0.b32 = tmp.b32
```

V_PK_MAX_I16 7

Select the component-wise maximum of two packed signed 16-bit integer inputs and store the selected values into a vector register.

```
tmp[31 : 16].i16 = S0[31 : 16].i16 >= S1[31 : 16].i16 ? S0[31 : 16].i16 : S1[31 : 16].i16;
tmp[15 : 0].i16 = S0[15 : 0].i16 >= S1[15 : 0].i16 ? S0[15 : 0].i16 : S1[15 : 0].i16;
D0.b32 = tmp.b32
```

V_PK_MIN_I16 8

Select the component-wise minimum of two packed signed 16-bit integer inputs and store the selected values into a vector register.

```
tmp[31 : 16].i16 = S0[31 : 16].i16 < S1[31 : 16].i16 ? S0[31 : 16].i16 : S1[31 : 16].i16;
tmp[15 : 0].i16 = S0[15 : 0].i16 < S1[15 : 0].i16 ? S0[15 : 0].i16 : S1[15 : 0].i16;
D0.b32 = tmp.b32
```

V_PK_MAD_U16 9

Multiply two packed unsigned 16-bit integer inputs component-wise, add a packed unsigned 16-bit integer value from a third input component-wise, and store the result into a vector register.

```
tmp[31 : 16].u16 = S0[31 : 16].u16 * S1[31 : 16].u16 + S2[31 : 16].u16;
tmp[15 : 0].u16 = S0[15 : 0].u16 * S1[15 : 0].u16 + S2[15 : 0].u16;
```

```
D0.b32 = tmp.b32
```

V_PK_ADD_U16 10

Add two packed unsigned 16-bit integer inputs component-wise and store the result into a vector register. No carry-in or carry-out support.

```
tmp[31 : 16].u16 = S0[31 : 16].u16 + S1[31 : 16].u16;
tmp[15 : 0].u16 = S0[15 : 0].u16 + S1[15 : 0].u16;
D0.b32 = tmp.b32
```

V_PK_SUB_U16 11

Subtract the second packed unsigned 16-bit integer input from the first input component-wise and store the result into a vector register. No carry-in or carry-out support.

```
tmp[31 : 16].u16 = S0[31 : 16].u16 - S1[31 : 16].u16;
tmp[15 : 0].u16 = S0[15 : 0].u16 - S1[15 : 0].u16;
D0.b32 = tmp.b32
```

V_PK_MAX_U16 12

Select the component-wise maximum of two packed unsigned 16-bit integer inputs and store the selected values into a vector register.

```
tmp[31 : 16].u16 = S0[31 : 16].u16 >= S1[31 : 16].u16 ? S0[31 : 16].u16 : S1[31 : 16].u16;
tmp[15 : 0].u16 = S0[15 : 0].u16 >= S1[15 : 0].u16 ? S0[15 : 0].u16 : S1[15 : 0].u16;
D0.b32 = tmp.b32
```

V_PK_MIN_U16 13

Select the component-wise minimum of two packed unsigned 16-bit integer inputs and store the selected values into a vector register.

```
tmp[31 : 16].u16 = S0[31 : 16].u16 < S1[31 : 16].u16 ? S0[31 : 16].u16 : S1[31 : 16].u16;
tmp[15 : 0].u16 = S0[15 : 0].u16 < S1[15 : 0].u16 ? S0[15 : 0].u16 : S1[15 : 0].u16;
D0.b32 = tmp.b32
```

V_PK_FMA_F16 14

Multiply two packed half-precision float inputs component-wise and add a third input component-wise using fused multiply add, and store the result into a vector register.

```
declare tmp : 32'B;
tmp[31 : 16].f16 = fma(S0[31 : 16].f16, S1[31 : 16].f16, S2[31 : 16].f16);
tmp[15 : 0].f16 = fma(S0[15 : 0].f16, S1[15 : 0].f16, S2[15 : 0].f16);
D0.b32 = tmp
```

V_PK_ADD_F16 15

Add two packed half-precision float inputs component-wise and store the result into a vector register. No carry-in or carry-out support.

```
tmp[31 : 16].f16 = S0[31 : 16].f16 + S1[31 : 16].f16;
tmp[15 : 0].f16 = S0[15 : 0].f16 + S1[15 : 0].f16;
D0.b32 = tmp.b32
```

V_PK_MUL_F16 16

Multiply two packed half-precision float inputs component-wise and store the result into a vector register.

```
tmp[31 : 16].f16 = S0[31 : 16].f16 * S1[31 : 16].f16;
tmp[15 : 0].f16 = S0[15 : 0].f16 * S1[15 : 0].f16;
D0.b32 = tmp.b32
```

V_PK_MIN_F16 17

Select the component-wise minimum of two packed half-precision float inputs and store the result into a vector register.

```
tmp[31 : 16].f16 = v_min_f16(S0[31 : 16].f16, S1[31 : 16].f16);
tmp[15 : 0].f16 = v_min_f16(S0[15 : 0].f16, S1[15 : 0].f16);
D0.b32 = tmp.b32
```

V_PK_MAX_F16 18

Select the component-wise maximum of two packed half-precision float inputs and store the result into a

vector register.

```
tmp[31 : 16].f16 = v_max_f16(S0[31 : 16].f16, S1[31 : 16].f16);
tmp[15 : 0].f16 = v_max_f16(S0[15 : 0].f16, S1[15 : 0].f16);
D0.b32 = tmp.b32
```

V_MAD_MIX_F32 32

Multiply two inputs and add a third input where the inputs are a mix of half-precision float and single-precision float values. Store the result into a vector register.

Size and location of the three inputs are controlled by `{ OPSEL_HI[i], OPSEL[i] }`: `0`=`src[31:0]`, `1`=`src[31:0]`, `2`=`src[15:0]`, `3`=`src[31:16]`. For MIX opcodes the NEG_HI instruction field acts as an absolute-value modifier for the three inputs.

```
declare in : 32'F[3];
declare S : 32'B[3];
for i in 0 : 2 do
    if !OPSEL_HI.u3[i] then
        in[i] = S[i].f32
    elsif OPSEL.u3[i] then
        in[i] = f16_to_f32(S[i][31 : 16].f16)
    else
        in[i] = f16_to_f32(S[i][15 : 0].f16)
    endif
endfor;
D0[31 : 0].f32 = in[0] * in[1] + in[2]
```

Notes

V_MAD_MIXLO_F16 33

Multiply two inputs and add a third input where the inputs are a mix of half-precision float and single-precision float values. Convert the result to a half-precision float. Store the result into the low bits of a vector register.

Size and location of the three inputs are controlled by `{ OPSEL_HI[i], OPSEL[i] }`: `0`=`src[31:0]`, `1`=`src[31:0]`, `2`=`src[15:0]`, `3`=`src[31:16]`. For MIX opcodes the NEG_HI instruction field acts as an absolute-value modifier for the three inputs.

```
declare in : 32'F[3];
declare S : 32'B[3];
for i in 0 : 2 do
    if !OPSEL_HI.u3[i] then
        in[i] = S[i].f32
    elsif OPSEL.u3[i] then
        in[i] = f16_to_f32(S[i][31 : 16].f16)
```

```
    else
        in[i] = f16_to_f32(S[i][15 : 0].f16)
    endif
endfor;
D0[15 : 0].f16 = f32_to_f16(in[0] * in[1] + in[2])
```

Notes

V_MAD_MIXHI_F16 34

Multiply two inputs and add a third input where the inputs are a mix of half-precision float and single-precision float values. Convert the result to a half-precision float. Store the result into the high bits of a vector register.

Size and location of the three inputs are controlled by `{ OPSEL_HI[i], OPSEL[i] }`: `0=src[31:0]`, `1=src[31:0]`, `2=src[15:0]`, `3=src[31:16]`. For MIX opcodes the NEG_HI instruction field acts as an absolute-value modifier for the three inputs.

```
declare in : 32'F[3];
declare S : 32'B[3];
for i in 0 : 2 do
    if !OPSEL_HI.u3[i] then
        in[i] = S[i].f32
    elsif OPSEL.u3[i] then
        in[i] = f16_to_f32(S[i][31 : 16].f16)
    else
        in[i] = f16_to_f32(S[i][15 : 0].f16)
    endif
endfor;
D0[31 : 16].f16 = f32_to_f16(in[0] * in[1] + in[2])
```

Notes

V_DOT2_F32_F16 35

Compute the dot product of two packed 2-D half-precision float inputs in the single-precision float domain, add a single-precision float value from the third input and store the result into a vector register.

```
tmp = S2.f32;
tmp += f16_to_f32(S0[15 : 0].f16) * f16_to_f32(S1[15 : 0].f16);
tmp += f16_to_f32(S0[31 : 16].f16) * f16_to_f32(S1[31 : 16].f16);
D0.f32 = tmp
```

V_DOT2_I32_I16 38

Compute the dot product of two packed 2-D signed 16-bit integer inputs in the signed 32-bit integer domain, add a signed 32-bit integer value from the third input and store the result into a vector register.

```
tmp = S2.i32;
tmp += i16_to_i32(S0[15 : 0].i16) * i16_to_i32(S1[15 : 0].i16);
tmp += i16_to_i32(S0[31 : 16].i16) * i16_to_i32(S1[31 : 16].i16);
D0.i32 = tmp
```

V_DOT2_U32_U16 39

Compute the dot product of two packed 2-D unsigned 16-bit integer inputs in the unsigned 32-bit integer domain, add an unsigned 32-bit integer value from the third input and store the result into a vector register.

```
tmp = S2.u32;
tmp += u16_to_u32(S0[15 : 0].u16) * u16_to_u32(S1[15 : 0].u16);
tmp += u16_to_u32(S0[31 : 16].u16) * u16_to_u32(S1[31 : 16].u16);
D0.u32 = tmp
```

V_DOT4_I32_I8 40

Compute the dot product of two packed 4-D signed 8-bit integer inputs in the signed 32-bit integer domain, add a signed 32-bit integer value from the third input and store the result into a vector register.

```
tmp = S2.i32;
tmp += i8_to_i32(S0[7 : 0].i8) * i8_to_i32(S1[7 : 0].i8);
tmp += i8_to_i32(S0[15 : 8].i8) * i8_to_i32(S1[15 : 8].i8);
tmp += i8_to_i32(S0[23 : 16].i8) * i8_to_i32(S1[23 : 16].i8);
tmp += i8_to_i32(S0[31 : 24].i8) * i8_to_i32(S1[31 : 24].i8);
D0.i32 = tmp
```

V_DOT4_U32_U8 41

Compute the dot product of two packed 4-D unsigned 8-bit integer inputs in the unsigned 32-bit integer domain, add an unsigned 32-bit integer value from the third input and store the result into a vector register.

```
tmp = S2.u32;
tmp += u8_to_u32(S0[7 : 0].u8) * u8_to_u32(S1[7 : 0].u8);
tmp += u8_to_u32(S0[15 : 8].u8) * u8_to_u32(S1[15 : 8].u8);
tmp += u8_to_u32(S0[23 : 16].u8) * u8_to_u32(S1[23 : 16].u8);
tmp += u8_to_u32(S0[31 : 24].u8) * u8_to_u32(S1[31 : 24].u8);
D0.u32 = tmp
```

V_DOT8_I32_I4 42

Compute the dot product of two packed 8-D signed 4-bit integer inputs in the signed 32-bit integer domain, add a signed 32-bit integer value from the third input and store the result into a vector register.

```
tmp = S2.i32;
tmp += i4_to_i32(S0[3 : 0].i4) * i4_to_i32(S1[3 : 0].i4);
tmp += i4_to_i32(S0[7 : 4].i4) * i4_to_i32(S1[7 : 4].i4);
tmp += i4_to_i32(S0[11 : 8].i4) * i4_to_i32(S1[11 : 8].i4);
tmp += i4_to_i32(S0[15 : 12].i4) * i4_to_i32(S1[15 : 12].i4);
tmp += i4_to_i32(S0[19 : 16].i4) * i4_to_i32(S1[19 : 16].i4);
tmp += i4_to_i32(S0[23 : 20].i4) * i4_to_i32(S1[23 : 20].i4);
tmp += i4_to_i32(S0[27 : 24].i4) * i4_to_i32(S1[27 : 24].i4);
tmp += i4_to_i32(S0[31 : 28].i4) * i4_to_i32(S1[31 : 28].i4);
D0.i32 = tmp
```

V_DOT8_U32_U4 43

Compute the dot product of two packed 8-D unsigned 4-bit integer inputs in the unsigned 32-bit integer domain, add an unsigned 32-bit integer value from the third input and store the result into a vector register.

```
tmp = S2.u32;
tmp += u4_to_u32(S0[3 : 0].u4) * u4_to_u32(S1[3 : 0].u4);
tmp += u4_to_u32(S0[7 : 4].u4) * u4_to_u32(S1[7 : 4].u4);
tmp += u4_to_u32(S0[11 : 8].u4) * u4_to_u32(S1[11 : 8].u4);
tmp += u4_to_u32(S0[15 : 12].u4) * u4_to_u32(S1[15 : 12].u4);
tmp += u4_to_u32(S0[19 : 16].u4) * u4_to_u32(S1[19 : 16].u4);
tmp += u4_to_u32(S0[23 : 20].u4) * u4_to_u32(S1[23 : 20].u4);
tmp += u4_to_u32(S0[27 : 24].u4) * u4_to_u32(S1[27 : 24].u4);
tmp += u4_to_u32(S0[31 : 28].u4) * u4_to_u32(S1[31 : 28].u4);
D0.u32 = tmp
```

V_PK_FMA_F32 48

Multiply two packed single-precision float inputs component-wise and add a third input component-wise using fused multiply add, and store the result into a vector register.

```
declare tmp : 32'B;
tmp[63 : 32].f32 = fma(S0[63 : 32].f32, S1[63 : 32].f32, S2[63 : 32].f32);
tmp[31 : 0].f32 = fma(S0[31 : 0].f32, S1[31 : 0].f32, S2[31 : 0].f32);
D0.b32 = tmp
```

V_PK_MUL_F32 49

Multiply two packed single-precision float inputs component-wise and store the result into a vector register.

```
tmp[63 : 32].f32 = S0[63 : 32].f32 * S1[63 : 32].f32;
tmp[31 : 0].f32 = S0[31 : 0].f32 * S1[31 : 0].f32;
D0.b32 = tmp.b32
```

V_PK_ADD_F32 50

Add two packed single-precision float inputs component-wise and store the result into a vector register. No carry-in or carry-out support.

```
tmp[63 : 32].f32 = S0[63 : 32].f32 + S1[63 : 32].f32;
tmp[31 : 0].f32 = S0[31 : 0].f32 + S1[31 : 0].f32;
D0.b32 = tmp.b32
```

V_PK_MOV_B32 51

Move data from two vector inputs into two vector registers.

```
tmp0.u32 = S0.u32[OPSEL[0].i32 * 32 + 31 : OPSEL[0].i32 * 32];
tmp1.u32 = S1.u32[OPSEL[1].i32 * 32 + 31 : OPSEL[1].i32 * 32];
D0.u32[31 : 0] = tmp0.u32;
D0.u32[63 : 32] = tmp1.u32
```

Notes

The source operands are treated as 64 bit and are subject to alignment restrictions for both SGPR and VGPR.

For two VGPR inputs this opcode can be used as an arbitrary gather by using OP_SEL to select either the even VGPR specified or the next odd VGPR.

```
    v_pk_mov_b32 v0, v2, v4 op_sel:[0,1] // evaluates v0 <- v2 and v1 <- v5.
```

Due to scalar broadcast restrictions if two SGPRs are specified as operands, they must be the same SGPR.

```
    v_pk_mov_b32 v0, s6, s6 op_sel:[0,1] // 64-bit move from scalar s[6:7].
```

V_MFMA_F32_16X16X8_XF32 62

Multiply the 16x8 matrix in the first input by the 8x16 matrix in the second input and add the 16x16 matrix in

the third input using fused multiply add. Store the resulting matrix into vector registers.

```
D = A (16x8) * B (8x16) + C (16x16)
```

Each operand contains a single matrix whose elements are distributed across all lanes of the wave. A single matrix multiply is computed and the row-column dot products are distributed across the vector ALU for higher performance.

Matrices A and B are single-precision float format. Matrices C and D are single-precision float format. XF32 is a FP32 operation with FP32 inputs and outputs but implemented at reduced intermediate precision where mantissa is truncated to 10 bits (not including leading 1 for non-zero values) and results are accumulated into FP32 value with 23 bit mantissa.

Notes

This instruction performs 4 passes.

V_MFMA_F32_32X32X4_XF32 63

Multiply the 32x4 matrix in the first input by the 4x32 matrix in the second input and add the 32x32 matrix in the third input using fused multiply add. Store the resulting matrix into vector registers.

```
D = A (32x4) * B (4x32) + C (32x32)
```

Each operand contains a single matrix whose elements are distributed across all lanes of the wave. A single matrix multiply is computed and the row-column dot products are distributed across the vector ALU for higher performance.

Matrices A and B are single-precision float format. Matrices C and D are single-precision float format.

Notes

This instruction performs 8 passes.

V_MFMA_F32_32X32X1_2B_F32 64

Multiply the 32x1 matrix in the first input by the 1x32 matrix in the second input and add the 32x32 matrix in the third input using fused multiply add. Store the resulting matrix into vector registers.

```
D = A (32x1) * B (1x32) + C (32x32)
```

This instruction performs 2 matrix multiplies. Each operand contains 2 matrices back to back, and each matrix has elements distributed across all lanes of the wave. Each matrix multiple is computed and the row-column dot products are distributed across the vector ALU for higher performance. The result matrices are stored

back-to-back in the destination vector registers.

Matrices A and B are single-precision float format. Matrices C and D are single-precision float format.

Notes

This instruction performs 16 passes.

V_MFMA_F32_16X16X1_4B_F32 65

Multiply the 16x1 matrix in the first input by the 1x16 matrix in the second input and add the 16x16 matrix in the third input using fused multiply add. Store the resulting matrix into vector registers.

```
D = A (16x1) * B (1x16) + C (16x16)
```

This instruction performs 4 matrix multiplies. Each operand contains 4 matrices back to back, and each matrix has elements distributed across all lanes of the wave. Each matrix multiple is computed and the row-column dot products are distributed across the vector ALU for higher performance. The result matrices are stored back-to-back in the destination vector registers.

Matrices A and B are single-precision float format. Matrices C and D are single-precision float format.

Notes

This instruction performs 8 passes.

V_MFMA_F32_4X4X1_16B_F32 66

Multiply the 4x1 matrix in the first input by the 1x4 matrix in the second input and add the 4x4 matrix in the third input using fused multiply add. Store the resulting matrix into vector registers.

```
D = A (4x1) * B (1x4) + C (4x4)
```

This instruction performs 16 matrix multiplies. Each operand contains 16 matrices back to back, and each matrix has elements distributed across all lanes of the wave. Each matrix multiple is computed and the row-column dot products are distributed across the vector ALU for higher performance. The result matrices are stored back-to-back in the destination vector registers.

Matrices A and B are single-precision float format. Matrices C and D are single-precision float format.

Notes

This instruction performs 2 passes.

V_MFMA_F32_32X32X2_F32 68

Multiply the 32x2 matrix in the first input by the 2x32 matrix in the second input and add the 32x32 matrix in the third input using fused multiply add. Store the resulting matrix into vector registers.

```
D = A (32x2) * B (2x32) + C (32x32)
```

Each operand contains a single matrix whose elements are distributed across all lanes of the wave. A single matrix multiply is computed and the row-column dot products are distributed across the vector ALU for higher performance.

Matrices A and B are single-precision float format. Matrices C and D are single-precision float format.

Notes

This instruction performs 16 passes.

V_MFMA_F32_16X16X4_F32 69

Multiply the 16x4 matrix in the first input by the 4x16 matrix in the second input and add the 16x16 matrix in the third input using fused multiply add. Store the resulting matrix into vector registers.

```
D = A (16x4) * B (4x16) + C (16x16)
```

Each operand contains a single matrix whose elements are distributed across all lanes of the wave. A single matrix multiply is computed and the row-column dot products are distributed across the vector ALU for higher performance.

Matrices A and B are single-precision float format. Matrices C and D are single-precision float format.

Notes

This instruction performs 8 passes.

V_MFMA_F32_32X32X4_2B_F16 72

Multiply the 32x4 matrix in the first input by the 4x32 matrix in the second input and add the 32x32 matrix in the third input using fused multiply add. Store the resulting matrix into vector registers.

```
D = A (32x4) * B (4x32) + C (32x32)
```

This instruction performs 2 matrix multiplies. Each operand contains 2 matrices back to back, and each matrix has elements distributed across all lanes of the wave. Each matrix multiple is computed and the row-column dot products are distributed across the vector ALU for higher performance. The result matrices are stored

back-to-back in the destination vector registers.

Matrices A and B are half-precision float format. Matrices C and D are single-precision float format.

Notes

This instruction performs 16 passes.

V_MFMA_F32_16X16X4_4B_F16 73

Multiply the 16x4 matrix in the first input by the 4x16 matrix in the second input and add the 16x16 matrix in the third input using fused multiply add. Store the resulting matrix into vector registers.

```
D = A (16x4) * B (4x16) + C (16x16)
```

This instruction performs 4 matrix multiplies. Each operand contains 4 matrices back to back, and each matrix has elements distributed across all lanes of the wave. Each matrix multiple is computed and the row-column dot products are distributed across the vector ALU for higher performance. The result matrices are stored back-to-back in the destination vector registers.

Matrices A and B are half-precision float format. Matrices C and D are single-precision float format.

Notes

This instruction performs 8 passes.

V_MFMA_F32_4X4X4_16B_F16 74

Multiply the 4x4 matrix in the first input by the 4x4 matrix in the second input and add the 4x4 matrix in the third input using fused multiply add. Store the resulting matrix into vector registers.

```
D = A (4x4) * B (4x4) + C (4x4)
```

This instruction performs 16 matrix multiplies. Each operand contains 16 matrices back to back, and each matrix has elements distributed across all lanes of the wave. Each matrix multiple is computed and the row-column dot products are distributed across the vector ALU for higher performance. The result matrices are stored back-to-back in the destination vector registers.

Matrices A and B are half-precision float format. Matrices C and D are single-precision float format.

Notes

This instruction performs 2 passes.

V_MFMA_F32_32X32X8_F16 76

Multiply the 32x8 matrix in the first input by the 8x32 matrix in the second input and add the 32x32 matrix in the third input using fused multiply add. Store the resulting matrix into vector registers.

```
D = A (32x8) * B (8x32) + C (32x32)
```

Each operand contains a single matrix whose elements are distributed across all lanes of the wave. A single matrix multiply is computed and the row-column dot products are distributed across the vector ALU for higher performance.

Matrices A and B are half-precision float format. Matrices C and D are single-precision float format.

Notes

This instruction performs 8 passes.

V_MFMA_F32_16X16X16_F16 77

Multiply the 16x16 matrix in the first input by the 16x16 matrix in the second input and add the 16x16 matrix in the third input using fused multiply add. Store the resulting matrix into vector registers.

```
D = A (16x16) * B (16x16) + C (16x16)
```

Each operand contains a single matrix whose elements are distributed across all lanes of the wave. A single matrix multiply is computed and the row-column dot products are distributed across the vector ALU for higher performance.

Matrices A and B are half-precision float format. Matrices C and D are single-precision float format.

Notes

This instruction performs 4 passes.

V_MFMA_I32_32X32X4_2B_I8 80

Multiply the 32x4 matrix in the first input by the 4x32 matrix in the second input and add the 32x32 matrix in the third input using fused multiply add. Store the resulting matrix into vector registers.

```
D = A (32x4) * B (4x32) + C (32x32)
```

This instruction performs 2 matrix multiplies. Each operand contains 2 matrices back to back, and each matrix has elements distributed across all lanes of the wave. Each matrix multiple is computed and the row-column dot products are distributed across the vector ALU for higher performance. The result matrices are stored

back-to-back in the destination vector registers.

Matrices A and B are signed 8-bit integer format. Matrices C and D are single-precision float format.

Notes

This instruction performs 16 passes.

V_MFMA_I32_16X16X4_4B_I8 81

Multiply the 16x4 matrix in the first input by the 4x16 matrix in the second input and add the 16x16 matrix in the third input using fused multiply add. Store the resulting matrix into vector registers.

```
D = A (16x4) * B (4x16) + C (16x16)
```

This instruction performs 4 matrix multiplies. Each operand contains 4 matrices back to back, and each matrix has elements distributed across all lanes of the wave. Each matrix multiple is computed and the row-column dot products are distributed across the vector ALU for higher performance. The result matrices are stored back-to-back in the destination vector registers.

Matrices A and B are signed 8-bit integer format. Matrices C and D are single-precision float format.

Notes

This instruction performs 8 passes.

V_MFMA_I32_4X4X4_16B_I8 82

Multiply the 4x4 matrix in the first input by the 4x4 matrix in the second input and add the 4x4 matrix in the third input using fused multiply add. Store the resulting matrix into vector registers.

```
D = A (4x4) * B (4x4) + C (4x4)
```

This instruction performs 16 matrix multiplies. Each operand contains 16 matrices back to back, and each matrix has elements distributed across all lanes of the wave. Each matrix multiple is computed and the row-column dot products are distributed across the vector ALU for higher performance. The result matrices are stored back-to-back in the destination vector registers.

Matrices A and B are signed 8-bit integer format. Matrices C and D are single-precision float format.

Notes

This instruction performs 2 passes.

V_MFMA_I32_32X32X16_I8 86

Multiply the 32x16 matrix in the first input by the 16x32 matrix in the second input and add the 32x32 matrix in the third input using fused multiply add. Store the resulting matrix into vector registers.

```
D = A (32x16) * B (16x32) + C (32x32)
```

Each operand contains a single matrix whose elements are distributed across all lanes of the wave. A single matrix multiply is computed and the row-column dot products are distributed across the vector ALU for higher performance.

Matrices A and B are signed 8-bit integer format. Matrices C and D are single-precision float format.

Notes

This instruction performs 8 passes.

V_MFMA_I32_16X16X32_I8 87

Multiply the 16x32 matrix in the first input by the 32x16 matrix in the second input and add the 16x16 matrix in the third input using fused multiply add. Store the resulting matrix into vector registers.

```
D = A (16x32) * B (32x16) + C (16x16)
```

Each operand contains a single matrix whose elements are distributed across all lanes of the wave. A single matrix multiply is computed and the row-column dot products are distributed across the vector ALU for higher performance.

Matrices A and B are signed 8-bit integer format. Matrices C and D are single-precision float format.

Notes

This instruction performs 4 passes.

V_ACCVGPR_READ 88

Move 32 bits of data from an accumulator vector register into an architectural vector register.

V_ACCVGPR_WRITE 89

Move 32 bits of data from an architectural vector register into an accumulator vector register.

V_MFMA_F32_32X32X4_2B_BF16 93

Multiply the 32x4 matrix in the first input by the 4x32 matrix in the second input and add the 32x32 matrix in the third input using fused multiply add. Store the resulting matrix into vector registers.

```
D = A (32x4) * B (4x32) + C (32x32)
```

This instruction performs 2 matrix multiplies. Each operand contains 2 matrices back to back, and each matrix has elements distributed across all lanes of the wave. Each matrix multiple is computed and the row-column dot products are distributed across the vector ALU for higher performance. The result matrices are stored back-to-back in the destination vector registers.

Matrices A and B are BF16 float format. Matrices C and D are single-precision float format.

Notes

This instruction performs 16 passes.

V_MFMA_F32_16X16X4_4B_BF16 94

Multiply the 16x4 matrix in the first input by the 4x16 matrix in the second input and add the 16x16 matrix in the third input using fused multiply add. Store the resulting matrix into vector registers.

```
D = A (16x4) * B (4x16) + C (16x16)
```

This instruction performs 4 matrix multiplies. Each operand contains 4 matrices back to back, and each matrix has elements distributed across all lanes of the wave. Each matrix multiple is computed and the row-column dot products are distributed across the vector ALU for higher performance. The result matrices are stored back-to-back in the destination vector registers.

Matrices A and B are BF16 float format. Matrices C and D are single-precision float format.

Notes

This instruction performs 8 passes.

V_MFMA_F32_4X4X4_16B_BF16 95

Multiply the 4x4 matrix in the first input by the 4x4 matrix in the second input and add the 4x4 matrix in the third input using fused multiply add. Store the resulting matrix into vector registers.

```
D = A (4x4) * B (4x4) + C (4x4)
```

This instruction performs 16 matrix multiplies. Each operand contains 16 matrices back to back, and each

matrix has elements distributed across all lanes of the wave. Each matrix multiple is computed and the row-column dot products are distributed across the vector ALU for higher performance. The result matrices are stored back-to-back in the destination vector registers.

Matrices A and B are BF16 float format. Matrices C and D are single-precision float format.

Notes

This instruction performs 2 passes.

V_MFMA_F32_32X32X8_BF16 96

Multiply the 32x8 matrix in the first input by the 8x32 matrix in the second input and add the 32x32 matrix in the third input using fused multiply add. Store the resulting matrix into vector registers.

```
D = A (32x8) * B (8x32) + C (32x32)
```

Each operand contains a single matrix whose elements are distributed across all lanes of the wave. A single matrix multiply is computed and the row-column dot products are distributed across the vector ALU for higher performance.

Matrices A and B are BF16 float format. Matrices C and D are single-precision float format.

Notes

This instruction performs 8 passes.

V_MFMA_F32_16X16X16_BF16 97

Multiply the 16x16 matrix in the first input by the 16x16 matrix in the second input and add the 16x16 matrix in the third input using fused multiply add. Store the resulting matrix into vector registers.

```
D = A (16x16) * B (16x16) + C (16x16)
```

Each operand contains a single matrix whose elements are distributed across all lanes of the wave. A single matrix multiply is computed and the row-column dot products are distributed across the vector ALU for higher performance.

Matrices A and B are BF16 float format. Matrices C and D are single-precision float format.

Notes

This instruction performs 4 passes.

V_SMFMAC_F32_16X16X32_F16 98

Multiply the 16x32 sparse matrix in the first input by the 32x16 matrix in the second input and accumulate the result into the 16x16 matrix stored in the destination registers using fused multiply add. Sparse indexes for the first matrix are given in the third input.

```
D = A (sparse 16x32) * B (32x16) + D (16x16)
```

Each operand contains a single matrix whose elements are distributed across all lanes of the wave. A single matrix multiply is computed and the row-column dot products are distributed across the vector ALU for higher performance.

Matrix A is a sparse matrix in half-precision float format, consuming half the physical storage of a dense matrix with same dimensions. Matrix B is a dense matrix in half-precision float format. Matrix D is single-precision float format and is both the output and the accumulate input.

2 out of every 4 elements on the K axis of matrix A are zero. The sparse indexes are used to determine which 2 elements are zero.

Notes

This instruction performs 4 passes.

V_SMFMAC_F32_32X32X16_F16 100

Multiply the 32x16 sparse matrix in the first input by the 16x32 matrix in the second input and accumulate the result into the 32x32 matrix stored in the destination registers using fused multiply add. Sparse indexes for the first matrix are given in the third input.

```
D = A (sparse 32x16) * B (16x32) + D (32x32)
```

Each operand contains a single matrix whose elements are distributed across all lanes of the wave. A single matrix multiply is computed and the row-column dot products are distributed across the vector ALU for higher performance.

Matrix A is a sparse matrix in half-precision float format, consuming half the physical storage of a dense matrix with same dimensions. Matrix B is a dense matrix in half-precision float format. Matrix D is single-precision float format and is both the output and the accumulate input.

2 out of every 4 elements on the K axis of matrix A are zero. The sparse indexes are used to determine which 2 elements are zero.

Notes

This instruction performs 8 passes.

V_SMFMAC_F32_16X16X32_BF16 102

Multiply the 16x32 sparse matrix in the first input by the 32x16 matrix in the second input and accumulate the result into the 16x16 matrix stored in the destination registers using fused multiply add. Sparse indexes for the first matrix are given in the third input.

```
D = A (sparse 16x32) * B (32x16) + D (16x16)
```

Each operand contains a single matrix whose elements are distributed across all lanes of the wave. A single matrix multiply is computed and the row-column dot products are distributed across the vector ALU for higher performance.

Matrix A is a sparse matrix in BF16 float format, consuming half the physical storage of a dense matrix with same dimensions. Matrix B is a dense matrix in BF16 float format. Matrix D is single-precision float format and is both the output and the accumulate input.

2 out of every 4 elements on the K axis of matrix A are zero. The sparse indexes are used to determine which 2 elements are zero.

Notes

This instruction performs 4 passes.

V_SMFMAC_F32_32X32X16_BF16 104

Multiply the 32x16 sparse matrix in the first input by the 16x32 matrix in the second input and accumulate the result into the 32x32 matrix stored in the destination registers using fused multiply add. Sparse indexes for the first matrix are given in the third input.

```
D = A (sparse 32x16) * B (16x32) + D (32x32)
```

Each operand contains a single matrix whose elements are distributed across all lanes of the wave. A single matrix multiply is computed and the row-column dot products are distributed across the vector ALU for higher performance.

Matrix A is a sparse matrix in BF16 float format, consuming half the physical storage of a dense matrix with same dimensions. Matrix B is a dense matrix in BF16 float format. Matrix D is single-precision float format and is both the output and the accumulate input.

2 out of every 4 elements on the K axis of matrix A are zero. The sparse indexes are used to determine which 2 elements are zero.

Notes

This instruction performs 8 passes.

V_SMFMAC_I32_16X16X64_I8 106

Multiply the 16x64 sparse matrix in the first input by the 64x16 matrix in the second input and accumulate the result into the 16x16 matrix stored in the destination registers using fused multiply add. Sparse indexes for the first matrix are given in the third input.

```
D = A (sparse 16x64) * B (64x16) + D (16x16)
```

Each operand contains a single matrix whose elements are distributed across all lanes of the wave. A single matrix multiply is computed and the row-column dot products are distributed across the vector ALU for higher performance.

Matrix A is a sparse matrix in signed 8-bit integer format, consuming half the physical storage of a dense matrix with same dimensions. Matrix B is a dense matrix in signed 8-bit integer format. Matrix D is signed 32-bit integer format and is both the output and the accumulate input.

2 out of every 4 elements on the K axis of matrix A are zero. The sparse indexes are used to determine which 2 elements are zero.

Notes

This instruction performs 4 passes.

V_SMFMAC_I32_32X32X32_I8 108

Multiply the 32x32 sparse matrix in the first input by the 32x32 matrix in the second input and accumulate the result into the 32x32 matrix stored in the destination registers using fused multiply add. Sparse indexes for the first matrix are given in the third input.

```
D = A (sparse 32x32) * B (32x32) + D (32x32)
```

Each operand contains a single matrix whose elements are distributed across all lanes of the wave. A single matrix multiply is computed and the row-column dot products are distributed across the vector ALU for higher performance.

Matrix A is a sparse matrix in signed 8-bit integer format, consuming half the physical storage of a dense matrix with same dimensions. Matrix B is a dense matrix in signed 8-bit integer format. Matrix D is signed 32-bit integer format and is both the output and the accumulate input.

2 out of every 4 elements on the K axis of matrix A are zero. The sparse indexes are used to determine which 2 elements are zero.

Notes

This instruction performs 8 passes.

V_MFMA_F64_16X16X4_F64 110

Multiply the 16x4 matrix in the first input by the 4x16 matrix in the second input and add the 16x16 matrix in the third input using fused multiply add. Store the resulting matrix into vector registers.

```
D = A (16x4) * B (4x16) + C (16x16)
```

Each operand contains a single matrix whose elements are distributed across all lanes of the wave. A single matrix multiply is computed and the row-column dot products are distributed across the vector ALU for higher performance.

Matrices A and B are double-precision float format. Matrices C and D are single-precision float format.

Notes

This instruction performs 8 passes.

V_MFMA_F64_4X4X4_4B_F64 111

Multiply the 4x4 matrix in the first input by the 4x4 matrix in the second input and add the 4x4 matrix in the third input using fused multiply add. Store the resulting matrix into vector registers.

```
D = A (4x4) * B (4x4) + C (4x4)
```

This instruction performs 4 matrix multiplies. Each operand contains 4 matrices back to back, and each matrix has elements distributed across all lanes of the wave. Each matrix multiple is computed and the row-column dot products are distributed across the vector ALU for higher performance. The result matrices are stored back-to-back in the destination vector registers.

Matrices A and B are double-precision float format. Matrices C and D are single-precision float format.

Notes

This instruction performs 4 passes.

V_MFMA_F32_16X16X32_BF8_BF8 112

Multiply the 16x32 matrix in the first input by the 32x16 matrix in the second input and add the 16x16 matrix in the third input using fused multiply add. Store the resulting matrix into vector registers.

```
D = A (16x32) * B (32x16) + C (16x16)
```

Each operand contains a single matrix whose elements are distributed across all lanes of the wave. A single matrix multiply is computed and the row-column dot products are distributed across the vector ALU for higher

performance.

Matrix A is BF8 float format. Matrix B is BF8 float format. Matrices C and D are single-precision float format.

Notes

This instruction performs 4 passes.

V_MFMA_F32_16X16X32_BF8_FP8 113

Multiply the 16x32 matrix in the first input by the 32x16 matrix in the second input and add the 16x16 matrix in the third input using fused multiply add. Store the resulting matrix into vector registers.

```
D = A (16x32) * B (32x16) + C (16x16)
```

Each operand contains a single matrix whose elements are distributed across all lanes of the wave. A single matrix multiply is computed and the row-column dot products are distributed across the vector ALU for higher performance.

Matrix A is BF8 float format. Matrix B is FP8 float format. Matrices C and D are single-precision float format.

Notes

This instruction performs 4 passes.

V_MFMA_F32_16X16X32_FP8_BF8 114

Multiply the 16x32 matrix in the first input by the 32x16 matrix in the second input and add the 16x16 matrix in the third input using fused multiply add. Store the resulting matrix into vector registers.

```
D = A (16x32) * B (32x16) + C (16x16)
```

Each operand contains a single matrix whose elements are distributed across all lanes of the wave. A single matrix multiply is computed and the row-column dot products are distributed across the vector ALU for higher performance.

Matrix A is FP8 float format. Matrix B is BF8 float format. Matrices C and D are single-precision float format.

Notes

This instruction performs 4 passes.

V_MFMA_F32_16X16X32_FP8_FP8 115

Multiply the 16x32 matrix in the first input by the 32x16 matrix in the second input and add the 16x16 matrix in the third input using fused multiply add. Store the resulting matrix into vector registers.

```
D = A (16x32) * B (32x16) + C (16x16)
```

Each operand contains a single matrix whose elements are distributed across all lanes of the wave. A single matrix multiply is computed and the row-column dot products are distributed across the vector ALU for higher performance.

Matrix A is FP8 float format. Matrix B is FP8 float format. Matrices C and D are single-precision float format.

Notes

This instruction performs 4 passes.

V_MFMA_F32_32X32X16_BF8_BF8 116

Multiply the 32x16 matrix in the first input by the 16x32 matrix in the second input and add the 32x32 matrix in the third input using fused multiply add. Store the resulting matrix into vector registers.

```
D = A (32x16) * B (16x32) + C (32x32)
```

Each operand contains a single matrix whose elements are distributed across all lanes of the wave. A single matrix multiply is computed and the row-column dot products are distributed across the vector ALU for higher performance.

Matrix A is BF8 float format. Matrix B is BF8 float format. Matrices C and D are single-precision float format.

Notes

This instruction performs 8 passes.

V_MFMA_F32_32X32X16_BF8_FP8 117

Multiply the 32x16 matrix in the first input by the 16x32 matrix in the second input and add the 32x32 matrix in the third input using fused multiply add. Store the resulting matrix into vector registers.

```
D = A (32x16) * B (16x32) + C (32x32)
```

Each operand contains a single matrix whose elements are distributed across all lanes of the wave. A single matrix multiply is computed and the row-column dot products are distributed across the vector ALU for higher performance.

Matrix A is BF8 float format. Matrix B is FP8 float format. Matrices C and D are single-precision float format.

Notes

This instruction performs 8 passes.

V_MFMA_F32_32X32X16_FP8_BF8 118

Multiply the 32x16 matrix in the first input by the 16x32 matrix in the second input and add the 32x32 matrix in the third input using fused multiply add. Store the resulting matrix into vector registers.

```
D = A (32x16) * B (16x32) + C (32x32)
```

Each operand contains a single matrix whose elements are distributed across all lanes of the wave. A single matrix multiply is computed and the row-column dot products are distributed across the vector ALU for higher performance.

Matrix A is FP8 float format. Matrix B is BF8 float format. Matrices C and D are single-precision float format.

Notes

This instruction performs 8 passes.

V_MFMA_F32_32X32X16_FP8_FP8 119

Multiply the 32x16 matrix in the first input by the 16x32 matrix in the second input and add the 32x32 matrix in the third input using fused multiply add. Store the resulting matrix into vector registers.

```
D = A (32x16) * B (16x32) + C (32x32)
```

Each operand contains a single matrix whose elements are distributed across all lanes of the wave. A single matrix multiply is computed and the row-column dot products are distributed across the vector ALU for higher performance.

Matrix A is FP8 float format. Matrix B is FP8 float format. Matrices C and D are single-precision float format.

Notes

This instruction performs 8 passes.

V_SMFMAC_F32_16X16X64_BF8_BF8 120

Multiply the 16x64 sparse matrix in the first input by the 64x16 matrix in the second input and accumulate the result into the 16x16 matrix stored in the destination registers using fused multiply add. Sparse indexes for the first matrix are given in the third input.

```
D = A (sparse 16x64) * B (64x16) + D (16x16)
```

Each operand contains a single matrix whose elements are distributed across all lanes of the wave. A single matrix multiply is computed and the row-column dot products are distributed across the vector ALU for higher performance.

Matrix A is a sparse matrix in BF8 float format, consuming half the physical storage of a dense matrix with same dimensions. Matrix B is a dense matrix in BF8 float format. Matrix D is single-precision float format and is both the output and the accumulate input.

2 out of every 4 elements on the K axis of matrix A are zero. The sparse indexes are used to determine which 2 elements are zero.

Notes

This instruction performs 4 passes.

V_SMFMAC_F32_16X16X64_BF8_FP8 121

Multiply the 16x64 sparse matrix in the first input by the 64x16 matrix in the second input and accumulate the result into the 16x16 matrix stored in the destination registers using fused multiply add. Sparse indexes for the first matrix are given in the third input.

```
D = A (sparse 16x64) * B (64x16) + D (16x16)
```

Each operand contains a single matrix whose elements are distributed across all lanes of the wave. A single matrix multiply is computed and the row-column dot products are distributed across the vector ALU for higher performance.

Matrix A is a sparse matrix in BF8 float format, consuming half the physical storage of a dense matrix with same dimensions. Matrix B is a dense matrix in FP8 float format. Matrix D is single-precision float format and is both the output and the accumulate input.

2 out of every 4 elements on the K axis of matrix A are zero. The sparse indexes are used to determine which 2 elements are zero.

Notes

This instruction performs 4 passes.

V_SMFMAC_F32_16X16X64_FP8_BF8 122

Multiply the 16x64 sparse matrix in the first input by the 64x16 matrix in the second input and accumulate the result into the 16x16 matrix stored in the destination registers using fused multiply add. Sparse indexes for the first matrix are given in the third input.

```
D = A (sparse 16x64) * B (64x16) + D (16x16)
```

Each operand contains a single matrix whose elements are distributed across all lanes of the wave. A single matrix multiply is computed and the row-column dot products are distributed across the vector ALU for higher performance.

Matrix A is a sparse matrix in FP8 float format, consuming half the physical storage of a dense matrix with same dimensions. Matrix B is a dense matrix in BF8 float format. Matrix D is single-precision float format and is both the output and the accumulate input.

2 out of every 4 elements on the K axis of matrix A are zero. The sparse indexes are used to determine which 2 elements are zero.

Notes

This instruction performs 4 passes.

V_SMFMAC_F32_16X16X64_FP8_FP8 123

Multiply the 16x64 sparse matrix in the first input by the 64x16 matrix in the second input and accumulate the result into the 16x16 matrix stored in the destination registers using fused multiply add. Sparse indexes for the first matrix are given in the third input.

```
D = A (sparse 16x64) * B (64x16) + D (16x16)
```

Each operand contains a single matrix whose elements are distributed across all lanes of the wave. A single matrix multiply is computed and the row-column dot products are distributed across the vector ALU for higher performance.

Matrix A is a sparse matrix in FP8 float format, consuming half the physical storage of a dense matrix with same dimensions. Matrix B is a dense matrix in FP8 float format. Matrix D is single-precision float format and is both the output and the accumulate input.

2 out of every 4 elements on the K axis of matrix A are zero. The sparse indexes are used to determine which 2 elements are zero.

Notes

This instruction performs 4 passes.

V_SMFMAC_F32_32X32X32_BF8_BF8 124

Multiply the 32x32 sparse matrix in the first input by the 32x32 matrix in the second input and accumulate the result into the 32x32 matrix stored in the destination registers using fused multiply add. Sparse indexes for the first matrix are given in the third input.

```
D = A (sparse 32x32) * B (32x32) + D (32x32)
```

Each operand contains a single matrix whose elements are distributed across all lanes of the wave. A single matrix multiply is computed and the row-column dot products are distributed across the vector ALU for higher performance.

Matrix A is a sparse matrix in BF8 float format, consuming half the physical storage of a dense matrix with same dimensions. Matrix B is a dense matrix in BF8 float format. Matrix D is single-precision float format and is both the output and the accumulate input.

2 out of every 4 elements on the K axis of matrix A are zero. The sparse indexes are used to determine which 2 elements are zero.

Notes

This instruction performs 8 passes.

V_SMFMAC_F32_32X32X32_BF8_FP8 125

Multiply the 32x32 sparse matrix in the first input by the 32x32 matrix in the second input and accumulate the result into the 32x32 matrix stored in the destination registers using fused multiply add. Sparse indexes for the first matrix are given in the third input.

```
D = A (sparse 32x32) * B (32x32) + D (32x32)
```

Each operand contains a single matrix whose elements are distributed across all lanes of the wave. A single matrix multiply is computed and the row-column dot products are distributed across the vector ALU for higher performance.

Matrix A is a sparse matrix in BF8 float format, consuming half the physical storage of a dense matrix with same dimensions. Matrix B is a dense matrix in FP8 float format. Matrix D is single-precision float format and is both the output and the accumulate input.

2 out of every 4 elements on the K axis of matrix A are zero. The sparse indexes are used to determine which 2 elements are zero.

Notes

This instruction performs 8 passes.

V_SMFMAC_F32_32X32X32_FP8_BF8 126

Multiply the 32x32 sparse matrix in the first input by the 32x32 matrix in the second input and accumulate the result into the 32x32 matrix stored in the destination registers using fused multiply add. Sparse indexes for the first matrix are given in the third input.

```
D = A (sparse 32x32) * B (32x32) + D (32x32)
```

Each operand contains a single matrix whose elements are distributed across all lanes of the wave. A single matrix multiply is computed and the row-column dot products are distributed across the vector ALU for higher performance.

Matrix A is a sparse matrix in FP8 float format, consuming half the physical storage of a dense matrix with same dimensions. Matrix B is a dense matrix in BF8 float format. Matrix D is single-precision float format and is both the output and the accumulate input.

2 out of every 4 elements on the K axis of matrix A are zero. The sparse indexes are used to determine which 2 elements are zero.

Notes

This instruction performs 8 passes.

V_SMFMAC_F32_32X32X32_FP8_FP8 127

Multiply the 32x32 sparse matrix in the first input by the 32x32 matrix in the second input and accumulate the result into the 32x32 matrix stored in the destination registers using fused multiply add. Sparse indexes for the first matrix are given in the third input.

```
D = A (sparse 32x32) * B (32x32) + D (32x32)
```

Each operand contains a single matrix whose elements are distributed across all lanes of the wave. A single matrix multiply is computed and the row-column dot products are distributed across the vector ALU for higher performance.

Matrix A is a sparse matrix in FP8 float format, consuming half the physical storage of a dense matrix with same dimensions. Matrix B is a dense matrix in FP8 float format. Matrix D is single-precision float format and is both the output and the accumulate input.

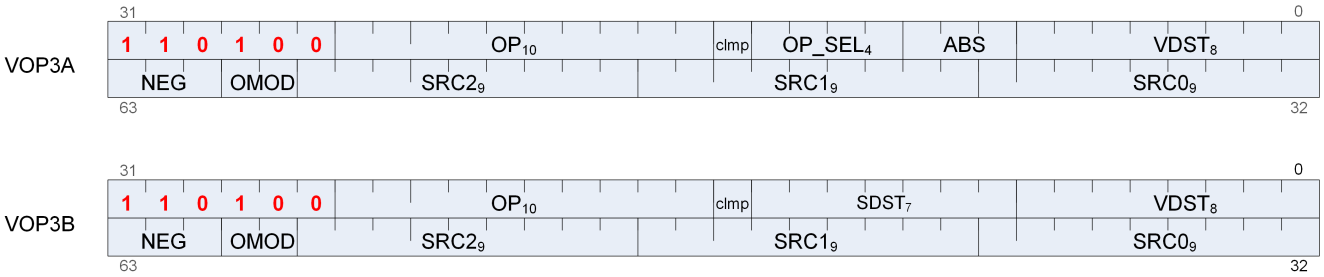
2 out of every 4 elements on the K axis of matrix A are zero. The sparse indexes are used to determine which 2 elements are zero.

Notes

This instruction performs 8 passes.

12.11. VOP3A & VOP3B Instructions

VOP3 instructions use one of two encodings:



VOP3B this encoding allows specifying a unique scalar destination, and is used only for:
V_ADD_CO_U32
V_SUB_CO_U32
V_SUBREV_CO_U32
V_ADDC_CO_U32
V_SUBB_CO_U32
V_SUBBREV_CO_U32
V_DIV_SCALE_F32
V_DIV_SCALE_F64
V_MAD_U64_U32
V_MAD_I64_I32

VOP3A all other VALU instructions use this encoding

V_NOP 384

Do nothing.

V_MOV_B32 385

Move data from a vector input into a vector register.

```
D0.b32 = S0.b32
```

Notes

Floating-point modifiers are valid for this instruction if S0.u is a 32-bit floating point value. This instruction is suitable for negating or taking the absolute value of a floating-point value.

Functional examples:

```
v_mov_b32 v0, v1    // Move into v0 from v1
v_mov_b32 v0, -v1   // Set v0 to the negation of v1
v_mov_b32 v0, abs(v1)   // Set v0 to the absolute value of v1
```

V_READFIRSTLANE_B32 386

Read the scalar value in the lowest active lane of the input vector register and store it into a scalar register.

```
declare lane : 32'I;
if EXEC == 0x0LL then
    lane = 0;
    // Force lane 0 if all lanes are disabled
else
    lane = s_ff1_i32_b64(EXEC);
    // Lowest active lane
endif;
D0.b32 = VGPR[lane][SRC0.u32]
```

Notes

Overrides EXEC mask for the VGPR read. Input and output modifiers not supported; this is an untyped operation.

V_CVT_I32_F64 387

Convert from a double-precision float input to a signed 32-bit integer value and store the result into a vector register.

```
D0.i32 = f64_to_i32(S0.f64)
```

Notes

0.5ULP accuracy, out-of-range floating point values (including infinity) saturate. NAN is converted to 0.

Generation of the INEXACT exception is controlled by the CLAMP bit. INEXACT exceptions are enabled for this conversion iff CLAMP == 1.

V_CVT_F64_I32 388

Convert from a signed 32-bit integer input to a double-precision float value and store the result into a vector register.

```
D0.f64 = i32_to_f64(S0.i32)
```

Notes

0ULP accuracy.

V_CVT_F32_I32 389

Convert from a signed 32-bit integer input to a single-precision float value and store the result into a vector register.

```
D0.f32 = i32_to_f32(S0.i32)
```

Notes

0.5ULP accuracy.

V_CVT_F32_U32 390

Convert from an unsigned 32-bit integer input to a single-precision float value and store the result into a vector register.

```
D0.f32 = u32_to_f32(S0.u32)
```

Notes

0.5ULP accuracy.

V_CVT_U32_F32 391

Convert from a single-precision float input to an unsigned 32-bit integer value and store the result into a vector register.

```
D0.u32 = f32_to_u32(S0.f32)
```

Notes

1ULP accuracy, out-of-range floating point values (including infinity) saturate. NAN is converted to 0.

Generation of the INEXACT exception is controlled by the CLAMP bit. INEXACT exceptions are enabled for this conversion iff CLAMP == 1.

V_CVT_I32_F32 392

Convert from a single-precision float input to a signed 32-bit integer value and store the result into a vector register.

```
D0.i32 = f32_to_i32(S0.f32)
```

Notes

1ULP accuracy, out-of-range floating point values (including infinity) saturate. NAN is converted to 0.

Generation of the INEXACT exception is controlled by the CLAMP bit. INEXACT exceptions are enabled for this conversion iff CLAMP == 1.

V_CVT_F16_F32 394

Convert from a single-precision float input to a half-precision float value and store the result into a vector register.

```
D0.f16 = f32_to_f16(S0.f32)
```

Notes

0.5ULP accuracy, supports input modifiers and creates FP16 denormals when appropriate. Flush denorms on output if specified based on DP denorm mode. Output rounding based on DP rounding mode.

V_CVT_F32_F16 395

Convert from a half-precision float input to a single-precision float value and store the result into a vector register.

```
D0.f32 = f16_to_f32(S0.f16)
```

Notes

0ULP accuracy, FP16 denormal inputs are accepted. Flush denorms on input if specified based on DP denorm mode.

V_CVT_RPI_I32_F32 396

Convert from a single-precision float input to a signed 32-bit integer value using round to nearest integer semantics (ignore the default rounding mode) and store the result into a vector register.

```
D0.i32 = f32_to_i32(floor(S0.f32 + 0.5F))
```

Notes

0.5ULP accuracy, denormals are supported.

V_CVT_FLR_I32_F32 397

Convert from a single-precision float input to a signed 32-bit integer value using round-down semantics (ignore the default rounding mode) and store the result into a vector register.

```
D0.i32 = f32_to_i32(floor(S0.f32))
```

Notes

1ULP accuracy, denormals are supported.

V_CVT_OFF_F32_I4 398

Convert from a signed 4-bit integer input to a single-precision float value using an offset table and store the result into a vector register.

Used for interpolation in shader. Lookup table on S0[3:0]:

S0 binary Result
1000 -0.5000f
1001 -0.4375f
1010 -0.3750f
1011 -0.3125f
1100 -0.2500f
1101 -0.1875f
1110 -0.1250f
1111 -0.0625f
0000 +0.0000f
0001 +0.0625f
0010 +0.1250f
0011 +0.1875f
0100 +0.2500f
0101 +0.3125f
0110 +0.3750f
0111 +0.4375f

```
declare CVT_OFF_TABLE : 32'F[16];
D0.f32 = CVT_OFF_TABLE[S0.u32[3 : 0]]
```

V_CVT_F32_F64 399

Convert from a double-precision float input to a single-precision float value and store the result into a vector register.

```
D0.f32 = f64_to_f32(S0.f64)
```

Notes

0.5ULP accuracy, denormals are supported.

V_CVT_F64_F32 400

Convert from a single-precision float input to a double-precision float value and store the result into a vector register.

```
D0.f64 = f32_to_f64(S0.f32)
```

Notes

0ULP accuracy, denormals are supported.

V_CVT_F32_UBYTE0 401

Convert an unsigned byte in byte 0 of the input to a single-precision float value and store the result into a vector register.

```
D0.f32 = u32_to_f32(S0[7 : 0].u32)
```

V_CVT_F32_UBYTE1 402

Convert an unsigned byte in byte 1 of the input to a single-precision float value and store the result into a vector register.

```
D0.f32 = u32_to_f32(S0[15 : 8].u32)
```

V_CVT_F32_UBYTE2 403

Convert an unsigned byte in byte 2 of the input to a single-precision float value and store the result into a vector register.

```
D0.f32 = u32_to_f32(S0[23 : 16].u32)
```

V_CVT_F32_UBYTE3 404

Convert an unsigned byte in byte 3 of the input to a single-precision float value and store the result into a vector register.

```
D0.f32 = u32_to_f32(S0[31 : 24].u32)
```

V_CVT_U32_F64 405

Convert from a double-precision float input to an unsigned 32-bit integer value and store the result into a vector register.

```
D0.u32 = f64_to_u32(S0.f64)
```

Notes

0.5ULP accuracy, out-of-range floating point values (including infinity) saturate. NAN is converted to 0.

Generation of the INEXACT exception is controlled by the CLAMP bit. INEXACT exceptions are enabled for this conversion iff CLAMP == 1.

V_CVT_F64_U32 406

Convert from an unsigned 32-bit integer input to a double-precision float value and store the result into a vector register.

```
D0.f64 = u32_to_f64(S0.u32)
```

Notes

0ULP accuracy.

V_TRUNC_F64 407

Compute the integer part of a double-precision float input using round toward zero semantics and store the result in floating point format into a vector register.

```
D0.f64 = trunc(S0.f64)
```

V_CEIL_F64 408

Round the double-precision float input up to next integer and store the result in floating point format into a vector register.

```
D0.f64 = trunc(S0.f64);
if ((S0.f64 > 0.0) && (S0.f64 != D0.f64)) then
    D0.f64 += 1.0
endif
```

V_RNDNE_F64 409

Round the double-precision float input to the nearest even integer and store the result in floating point format into a vector register.

```
D0.f64 = floor(S0.f64 + 0.5);
if (isEven(floor(S0.f64)) && (fract(S0.f64) == 0.5)) then
    D0.f64 -= 1.0
endif
```

V_FLOOR_F64 410

Round the double-precision float input down to previous integer and store the result in floating point format into a vector register.

```
D0.f64 = trunc(S0.f64);
if ((S0.f64 < 0.0) && (S0.f64 != D0.f64)) then
    D0.f64 += -1.0
endif
```

V_FRACT_F32 411

Compute the fractional portion of a single-precision float input and store the result in floating point format into a vector register.

```
D0.f32 = S0.f32 + -floor(S0.f32)
```

Notes

0.5ULP accuracy, denormals are accepted.

This is intended to comply with the DX specification of fract where the function behaves like an extension of integer modulus; be aware this may differ from how fract() is defined in other domains. For example: fract(-1.2) = 0.8 in DX.

Obey round mode, result clamped to 0x3f7fffff.

V_TRUNC_F32 412

Compute the integer part of a single-precision float input using round toward zero semantics and store the result in floating point format into a vector register.

```
D0.f32 = trunc(S0.f32)
```

V_CEIL_F32 413

Round the single-precision float input up to next integer and store the result in floating point format into a vector register.

```
D0.f32 = trunc(S0.f32);
if ((S0.f32 > 0.0F) && (S0.f32 != D0.f32)) then
    D0.f32 += 1.0F
endif
```

V_RNDNE_F32 414

Round the single-precision float input to the nearest even integer and store the result in floating point format into a vector register.

```
D0.f32 = floor(S0.f32 + 0.5F);
if (isEven(64'F(floor(S0.f32))) && (fract(S0.f32) == 0.5F)) then
    D0.f32 -= 1.0F
endif
```

V_FLOOR_F32 415

Round the single-precision float input down to previous integer and store the result in floating point format into a vector register.

```
D0.f32 = trunc(S0.f32);
if ((S0.f32 < 0.0F) && (S0.f32 != D0.f32)) then
    D0.f32 += -1.0F
endif
```

V_EXP_F32 416

Calculate 2 raised to the power of the single-precision float input and store the result into a vector register.

```
D0.f32 = pow(2.0F, S0.f32)
```

Notes

1ULP accuracy, denormals are flushed.

Functional examples:

```
V_EXP_F32(0xff800000) => 0x00000000     // exp(-INF) = 0
V_EXP_F32(0x80000000) => 0x3f800000     // exp(-0.0) = 1
V_EXP_F32(0x7f800000) => 0x7f800000     // exp(+INF) = +INF
```

V_LOG_F32 417

Calculate the base 2 logarithm of the single-precision float input and store the result into a vector register.

```
D0.f32 = log2(S0.f32)
```

Notes

1ULP accuracy, denormals are flushed.

Functional examples:

```
V_LOG_F32(0xff800000) => 0xffc00000     // log(-INF) = NAN
V_LOG_F32(0xbf800000) => 0xffc00000     // log(-1.0) = NAN
V_LOG_F32(0x80000000) => 0xff800000     // log(-0.0) = -INF
V_LOG_F32(0x00000000) => 0xff800000     // log(+0.0) = -INF
V_LOG_F32(0x3f800000) => 0x00000000     // log(+1.0) = 0
V_LOG_F32(0x7f800000) => 0x7f800000     // log(+INF) = +INF
```

V_RCP_F32 418

Calculate the reciprocal of the single-precision float input using IEEE rules and store the result into a vector register.

```
D0.f32 = 1.0F / S0.f32
```

Notes

1ULP accuracy. Accuracy converges to < 0.5ULP when using the Newton-Raphson method and 2 FMA operations. Denormals are flushed.

Functional examples:

```
V_RCP_F32(0xff800000) => 0x80000000     // rcp(-INF) = -0
V_RCP_F32(0xc0000000) => 0xbf000000     // rcp(-2.0) = -0.5
V_RCP_F32(0x80000000) => 0xff800000     // rcp(-0.0) = -INF
V_RCP_F32(0x00000000) => 0x7f800000     // rcp(+0.0) = +INF
V_RCP_F32(0x7f800000) => 0x00000000     // rcp(+INF) = +0
```

V_RCP_IFLAG_F32 419

Calculate the reciprocal of the vector float input in a manner suitable for integer division and store the result into a vector register. This opcode is intended for use as part of an integer division macro.

```
D0.f32 = 1.0F / S0.f32;
// Can only raise integer DIV_BY_ZERO exception
```

Notes

Can raise integer DIV_BY_ZERO exception but cannot raise floating-point exceptions. To be used in an integer reciprocal macro by the compiler with one of the sequences listed below (depending on signed or unsigned operation).

Unsigned usage:
CVT_F32_U32
RCP_IFLAG_F32
MUL_F32 (2**32 - 1)
CVT_U32_F32

Signed usage:
CVT_F32_I32
RCP_IFLAG_F32
MUL_F32 (2**31 - 1)
CVT_I32_F32

V_RSQ_F32 420

Calculate the reciprocal of the square root of the single-precision float input using IEEE rules and store the result into a vector register.

```
D0.f32 = 1.0F / sqrt(S0.f32)
```

Notes

1ULP accuracy, denormals are flushed.

Functional examples:

```
V_RSQ_F32(0xff800000) => 0xffc00000     // rsq(-INF) = NAN
V_RSQ_F32(0x80000000) => 0xff800000     // rsq(-0.0) = -INF
V_RSQ_F32(0x00000000) => 0x7f800000     // rsq(+0.0) = +INF
V_RSQ_F32(0x40800000) => 0x3f000000     // rsq(+4.0) = +0.5
V_RSQ_F32(0x7f800000) => 0x00000000     // rsq(+INF) = +0
```

V_RCP_F64 421

Calculate the reciprocal of the double-precision float input using IEEE rules and store the result into a vector register.

```
D0.f64 = 1.0 / S0.f64
```

Notes

This opcode has (2**29)ULP accuracy and supports denormals.

V_RSQ_F64 422

Calculate the reciprocal of the square root of the double-precision float input using IEEE rules and store the result into a vector register.

```
D0.f64 = 1.0 / sqrt(S0.f64)
```

Notes

This opcode has (2**29)ULP accuracy and supports denormals.

V_SQRT_F32 423

Calculate the square root of the single-precision float input using IEEE rules and store the result into a vector register.

```
D0.f32 = sqrt(S0.f32)
```

Notes

1ULP accuracy, denormals are flushed.

Functional examples:

```
V_SQRT_F32(0xff800000) => 0xffc00000     // sqrt(-INF) = NAN
V_SQRT_F32(0x80000000) => 0x80000000     // sqrt(-0.0) = -0
V_SQRT_F32(0x00000000) => 0x00000000     // sqrt(+0.0) = +0
V_SQRT_F32(0x40800000) => 0x40000000     // sqrt(+4.0) = +2.0
V_SQRT_F32(0x7f800000) => 0x7f800000     // sqrt(+INF) = +INF
```

V_SQRT_F64 424

Calculate the square root of the double-precision float input using IEEE rules and store the result into a vector register.

```
D0.f64 = sqrt(S0.f64)
```

Notes

This opcode has (2**29)ULP accuracy and supports denormals.

V_SIN_F32 425

Calculate the trigonometric sine of a single-precision float value using IEEE rules and store the result into a vector register. The operand is calculated by scaling the vector input by 2 PI.

```
D0.f32 = sin(S0.f32 * 32'F(PI * 2.0))
```

Notes

Denormals are supported. Full range input is supported.

Functional examples:

```
V_SIN_F32(0xff800000) => 0xffc00000     // sin(-INF) = NAN
V_SIN_F32(0xff7fffff) => 0x00000000     // -MaxFloat, finite
V_SIN_F32(0x80000000) => 0x80000000     // sin(-0.0) = -0
V_SIN_F32(0x3e800000) => 0x3f800000     // sin(0.25) = 1
V_SIN_F32(0x7f800000) => 0xffc00000     // sin(+INF) = NAN
```

V_COS_F32 426

Calculate the trigonometric cosine of a single-precision float value using IEEE rules and store the result into a vector register. The operand is calculated by scaling the vector input by 2 PI.

```
D0.f32 = cos(S0.f32 * 32'F(PI * 2.0))
```

Notes

Denormals are supported. Full range input is supported.

Functional examples:

```
V_COS_F32(0xff800000) => 0xffc00000     // cos(-INF) = NAN
V_COS_F32(0xff7fffff) => 0x3f800000     // -MaxFloat, finite
V_COS_F32(0x80000000) => 0x3f800000     // cos(-0.0) = 1
V_COS_F32(0x3e800000) => 0x00000000     // cos(0.25) = 0
V_COS_F32(0x7f800000) => 0xffc00000     // cos(+INF) = NAN
```

V_NOT_B32 427

Calculate bitwise negation on a vector input and store the result into a vector register.

```
D0.u32 = ~S0.u32
```

Notes

Input and output modifiers not supported.

V_BFREV_B32 428

Reverse the order of bits in a vector input and store the result into a vector register.

```
D0.u32[31 : 0] = S0.u32[0 : 31]
```

Notes

Input and output modifiers not supported.

V_FFBH_U32 429

Count the number of leading "0" bits before the first "1" in a vector input and store the result into a vector register. Store -1 if there are no "1" bits.

```
D0.i32 = -1;
// Set if no ones are found
for i in 0 : 31 do
    // Search from MSB
    if S0.u32[31 - i] == 1'1U then
        D0.i32 = i;
        break
    endif
endfor
```

Notes

Functional examples:

```
V_FFBH_U32(0x00000000) => 0xffffffff
V_FFBH_U32(0x800000ff) => 0
V_FFBH_U32(0x100000ff) => 3
V_FFBH_U32(0x0000ffff) => 16
V_FFBH_U32(0x00000001) => 31
```

V_FFBL_B32 430

Count the number of trailing "0" bits before the first "1" in a vector input and store the result into a vector register. Store -1 if there are no "1" bits in the input.

```
D0.i32 = -1;
// Set if no ones are found
for i in 0 : 31 do
    // Search from LSB
    if S0.u32[i] == 1'1U then
        D0.i32 = i;
        break
    endif
endfor
```

Notes

Functional examples:

```
V_FFBL_B32(0x00000000) => 0xffffffff
V_FFBL_B32(0xff000001) => 0
V_FFBL_B32(0xff000008) => 3
V_FFBL_B32(0xffff0000) => 16
V_FFBL_B32(0x80000000) => 31
```

V_FFBH_I32 431

Count the number of leading bits that are the same as the sign bit of a vector input and store the result into a vector register. Store -1 if all input bits are the same.

```
D0.i32 = -1;
// Set if all bits are the same
for i in 1 : 31 do
    // Search from MSB
    if S0.i32[31 - i] != S0.i32[31] then
        D0.i32 = i;
        break
    endif
endfor
```

Notes

Functional examples:

```
V_FFBH_I32(0x00000000) => 0xffffffff
V_FFBH_I32(0x40000000) => 1
V_FFBH_I32(0x80000000) => 1
V_FFBH_I32(0x0fffffff) => 4
V_FFBH_I32(0xffff0000) => 16
V_FFBH_I32(0xfffffffe) => 31
V_FFBH_I32(0xffffffff) => 0xffffffff
```

V_FREXP_EXP_I32_F64 432

Extract the exponent of a double-precision float input and store the result as a signed 32-bit integer into a vector register.

```
if ((S0.f64 == +INF) || (S0.f64 == -INF) || isNAN(S0.f64)) then
    D0.i32 = 0
else
    D0.i32 = exponent(S0.f64) - 1023 + 1
endif
```

Notes

This operation satisfies the invariant S0.f64 = significand * (2 ** exponent). See also V_FREXP_MANT_F64, which returns the significand. See the C library function `frexp()` for more information.

V_FREXP_MANT_F64 433

Extract the binary significand, or mantissa, of a double-precision float input and store the result as a double-precision float into a vector register.

```
if ((S0.f64 == +INF) || (S0.f64 == -INF) || isNAN(S0.f64)) then
    D0.f64 = S0.f64
else
    D0.f64 = mantissa(S0.f64)
endif
```

Notes

This operation satisfies the invariant S0.f64 = significand * (2 ** exponent). Result range is in (-1.0,-0.5][0.5,1.0) in normal cases. See also V_FREXP_EXP_I_F64, which returns integer exponent. See the C library function `frexp()` for more information.

V_FRACT_F64 434

Compute the fractional portion of a double-precision float input and store the result in floating point format into a vector register.

```
D0.f64 = S0.f64 + -floor(S0.f64)
```

Notes

0.5ULP accuracy, denormals are accepted.

This is intended to comply with the DX specification of fract where the function behaves like an extension of integer modulus; be aware this may differ from how fract() is defined in other domains. For example: fract(-1.2) = 0.8 in DX.

Obey round mode, result clamped to 0x3fefffffffffffff.

V_FREXP_EXP_I32_F32 435

Extract the exponent of a single-precision float input and store the result as a signed 32-bit integer into a vector register.

```
if ((64'F(S0.f32) == +INF) || (64'F(S0.f32) == -INF) || isNAN(64'F(S0.f32))) then
    D0.i32 = 0
else
    D0.i32 = exponent(S0.f32) - 127 + 1
endif
```

Notes

This operation satisfies the invariant S0.f32 = significand * (2 ** exponent). See also V_FREXP_MANT_F32, which returns the significand. See the C library function `frexp()` for more information.

V_FREXP_MANT_F32 436

Extract the binary significand, or mantissa, of a single-precision float input and store the result as a single-precision float into a vector register.

```
if ((64'F(S0.f32) == +INF) || (64'F(S0.f32) == -INF) || isNAN(64'F(S0.f32))) then
    D0.f32 = S0.f32
else
    D0.f32 = mantissa(S0.f32)
endif
```

Notes

This operation satisfies the invariant S0.f32 = significand * (2 ** exponent). Result range is in (-1.0,-0.5][0.5,1.0) in normal cases. See also V_FREXP_EXP_I_F32, which returns integer exponent. See the C library function `frexp()` for more information.

V_CLREXCP 437

Clear this wave's exception state in the vector ALU.

V_MOV_B64 440

Move data from a 64-bit vector input into a vector register.

```
D0.b64 = S0.b64
```

Notes

Floating-point modifiers are valid for this instruction if S0.u64 is a 64-bit floating point value. This instruction is suitable for negating or taking the absolute value of a floating-point value.

V_CVT_F16_U16 441

Convert from an unsigned 16-bit integer input to a half-precision float value and store the result into a vector register.

```
D0.f16 = u16_to_f16(S0.u16)
```

Notes

0.5ULP accuracy, supports denormals, rounding, exception flags and saturation.

V_CVT_F16_I16 442

Convert from a signed 16-bit integer input to a half-precision float value and store the result into a vector register.

```
D0.f16 = i16_to_f16(S0.i16)
```

Notes

0.5ULP accuracy, supports denormals, rounding, exception flags and saturation.

V_CVT_U16_F16 443

Convert from a half-precision float input to an unsigned 16-bit integer value and store the result into a vector register.

```
D0.u16 = f16_to_u16(S0.f16)
```

Notes

1ULP accuracy, supports rounding, exception flags and saturation. FP16 denormals are accepted. Conversion is done with truncation.

Generation of the INEXACT exception is controlled by the CLAMP bit. INEXACT exceptions are enabled for this conversion iff CLAMP == 1.

V_CVT_I16_F16 444

Convert from a half-precision float input to a signed 16-bit integer value and store the result into a vector register.

```
D0.i16 = f16_to_i16(S0.f16)
```

Notes

1ULP accuracy, supports rounding, exception flags and saturation. FP16 denormals are accepted. Conversion is done with truncation.

Generation of the INEXACT exception is controlled by the CLAMP bit. INEXACT exceptions are enabled for this conversion iff CLAMP == 1.

V_RCP_F16 445

Calculate the reciprocal of the half-precision float input using IEEE rules and store the result into a vector register.

```
D0.f16 = 16'1.0 / S0.f16
```

Notes

0.51ULP accuracy.

Functional examples:

```
V_RCP_F16(0xfc00) => 0x8000     // rcp(-INF) = -0
V_RCP_F16(0xc000) => 0xb800     // rcp(-2.0) = -0.5
V_RCP_F16(0x8000) => 0xfc00     // rcp(-0.0) = -INF
V_RCP_F16(0x0000) => 0x7c00     // rcp(+0.0) = +INF
V_RCP_F16(0x7c00) => 0x0000     // rcp(+INF) = +0
```

V_SQRT_F16 446

Calculate the square root of the half-precision float input using IEEE rules and store the result into a vector register.

```
D0.f16 = sqrt(S0.f16)
```

Notes

0.51ULP accuracy, denormals are supported.

Functional examples:

```
V_SQRT_F16(0xfc00) => 0xfe00     // sqrt(-INF) = NAN
```

```
V_SQRT_F16(0x8000) => 0x8000     // sqrt(-0.0) = -0
V_SQRT_F16(0x0000) => 0x0000     // sqrt(+0.0) = +0
V_SQRT_F16(0x4400) => 0x4000     // sqrt(+4.0) = +2.0
V_SQRT_F16(0x7c00) => 0x7c00     // sqrt(+INF) = +INF
```

V_RSQ_F16 447

Calculate the reciprocal of the square root of the half-precision float input using IEEE rules and store the result into a vector register.

```
D0.f16 = 16'1.0 / sqrt(S0.f16)
```

Notes

0.51ULP accuracy, denormals are supported.

Functional examples:

```
V_RSQ_F16(0xfc00) => 0xfe00     // rsq(-INF) = NAN
V_RSQ_F16(0x8000) => 0xfc00     // rsq(-0.0) = -INF
V_RSQ_F16(0x0000) => 0x7c00     // rsq(+0.0) = +INF
V_RSQ_F16(0x4400) => 0x3800     // rsq(+4.0) = +0.5
V_RSQ_F16(0x7c00) => 0x0000     // rsq(+INF) = +0
```

V_LOG_F16 448

Calculate the base 2 logarithm of the half-precision float input and store the result into a vector register.

```
D0.f16 = log2(S0.f16)
```

Notes

0.51ULP accuracy, denormals are supported.

Functional examples:

```
V_LOG_F16(0xfc00) => 0xfe00     // log(-INF) = NAN
V_LOG_F16(0xbc00) => 0xfe00     // log(-1.0) = NAN
V_LOG_F16(0x8000) => 0xfc00     // log(-0.0) = -INF
V_LOG_F16(0x0000) => 0xfc00     // log(+0.0) = -INF
V_LOG_F16(0x3c00) => 0x0000     // log(+1.0) = 0
V_LOG_F16(0x7c00) => 0x7c00     // log(+INF) = +INF
```

V_EXP_F16 449

Calculate 2 raised to the power of the half-precision float input and store the result into a vector register.

```
D0.f16 = pow(16'2.0, S0.f16)
```

Notes

0.51ULP accuracy, denormals are supported.

Functional examples:

```
V_EXP_F16(0xfc00) => 0x0000     // exp(-INF) = 0
V_EXP_F16(0x8000) => 0x3c00     // exp(-0.0) = 1
V_EXP_F16(0x7c00) => 0x7c00     // exp(+INF) = +INF
```

V_FREXP_MANT_F16 450

Extract the binary significand, or mantissa, of a half-precision float input and store the result as a half-precision float into a vector register.

```
if ((64'F(S0.f16) == +INF) || (64'F(S0.f16) == -INF) || isNAN(64'F(S0.f16))) then
    D0.f16 = S0.f16
else
    D0.f16 = mantissa(S0.f16)
endif
```

Notes

This operation satisfies the invariant S0.f16 = significand * (2 ** exponent). Result range is in (-1.0,-0.5][0.5,1.0) in normal cases. See also V_FREXP_EXP_I_F16, which returns integer exponent. See the C library function `frexp()` for more information.

V_FREXP_EXP_I16_F16 451

Extract the exponent of a half-precision float input and store the result as a signed 16-bit integer into a vector register.

```
if ((64'F(S0.f16) == +INF) || (64'F(S0.f16) == -INF) || isNAN(64'F(S0.f16))) then
    D0.i16 = 16'0
else
    D0.i16 = 16'I(exponent(S0.f16) - 15 + 1)
endif
```

Notes

This operation satisfies the invariant S0.f16 = significand * (2 ** exponent). See also V_FREXP_MANT_F16, which returns the significand. See the C library function `frexp()` for more information.

V_FLOOR_F16 452

Round the half-precision float input down to previous integer and store the result in floating point format into a vector register.

```
D0.f16 = trunc(S0.f16);
if ((S0.f16 < 16'0.0) && (S0.f16 != D0.f16)) then
    D0.f16 += -16'1.0
endif
```

V_CEIL_F16 453

Round the half-precision float input up to next integer and store the result in floating point format into a vector register.

```
D0.f16 = trunc(S0.f16);
if ((S0.f16 > 16'0.0) && (S0.f16 != D0.f16)) then
    D0.f16 += 16'1.0
endif
```

V_TRUNC_F16 454

Compute the integer part of a half-precision float input using round toward zero semantics and store the result in floating point format into a vector register.

```
D0.f16 = trunc(S0.f16)
```

V_RNDNE_F16 455

Round the half-precision float input to the nearest even integer and store the result in floating point format into a vector register.

```
D0.f16 = floor(S0.f16 + 16'0.5);
if (isEven(64'F(floor(S0.f16))) && (fract(S0.f16) == 16'0.5)) then
    D0.f16 -= 16'1.0
```

```
endif
```

V_FRACT_F16 456

Compute the fractional portion of a half-precision float input and store the result in floating point format into a vector register.

```
D0.f16 = S0.f16 + -floor(S0.f16)
```

Notes

0.5ULP accuracy, denormals are accepted.

This is intended to comply with the DX specification of fract where the function behaves like an extension of integer modulus; be aware this may differ from how fract() is defined in other domains. For example: fract(-1.2) = 0.8 in DX.

V_SIN_F16 457

Calculate the trigonometric sine of a half-precision float value using IEEE rules and store the result into a vector register. The operand is calculated by scaling the vector input by 2 PI.

```
D0.f16 = sin(S0.f16 * 16'F(PI * 2.0))
```

Notes

Denormals are supported. Full range input is supported.

Functional examples:

```
V_SIN_F16(0xfc00) => 0xfe00     // sin(-INF) = NAN
V_SIN_F16(0xfbff) => 0x0000     // Most negative finite FP16
V_SIN_F16(0x8000) => 0x8000     // sin(-0.0) = -0
V_SIN_F16(0x3400) => 0x3c00     // sin(0.25) = 1
V_SIN_F16(0x7bff) => 0x0000     // Most positive finite FP16
V_SIN_F16(0x7c00) => 0xfe00     // sin(+INF) = NAN
```

V_COS_F16 458

Calculate the trigonometric cosine of a half-precision float value using IEEE rules and store the result into a vector register. The operand is calculated by scaling the vector input by 2 PI.

```
D0.f16 = cos(S0.f16 * 16'F(PI * 2.0))
```

Notes

Denormals are supported. Full range input is supported.

Functional examples:

```
V_COS_F16(0xfc00) => 0xfe00     // cos(-INF) = NAN
V_COS_F16(0xfbff) => 0x3c00     // Most negative finite FP16
V_COS_F16(0x8000) => 0x3c00     // cos(-0.0) = 1
V_COS_F16(0x3400) => 0x0000     // cos(0.25) = 0
V_COS_F16(0x7bff) => 0x3c00     // Most positive finite FP16
V_COS_F16(0x7c00) => 0xfe00     // cos(+INF) = NAN
```

V_CVT_NORM_I16_F16 461

Convert from a half-precision float input to a signed normalized short and store the result into a vector register.

```
D0.i16 = f16_to_snorm(S0.f16)
```

Notes

0.5ULP accuracy, supports rounding, exception flags and saturation, denormals are supported.

V_CVT_NORM_U16_F16 462

Convert from a half-precision float input to an unsigned normalized short and store the result into a vector register.

```
D0.u16 = f16_to_unorm(S0.f16)
```

Notes

0.5ULP accuracy, supports rounding, exception flags and saturation, denormals are supported.

V_SAT_PK_U8_I16 463

Given two 16-bit signed integer inputs, saturate each input over an 8-bit unsigned range, pack the resulting values into a 16-bit word and store the result into a vector register.

```
SAT8 = lambda(n) (
    if n.i32 <= 0 then
        return 8'0U
    elsif n >= 16'I(0xff) then
        return 8'255U
    else
        return n[7 : 0].u8
    endif);
D0 = { 16'0, SAT8(S0[31 : 16].i16), SAT8(S0[15 : 0].i16) }
```

Notes

Used for 4x16bit data packed as 4x8bit data.

V_SWAP_B32 **465**

Swap the values in two vector registers.

```
tmp = D0.b32;
D0.b32 = S0.b32;
S0.b32 = tmp
```

Notes

Input and output modifiers not supported; this is an untyped operation.

V_ACCVGPR_MOV_B32 **466**

Move data from one accumulator register to another accumulator register.

V_CVT_F32_FP8 **468**

Convert from an FP8 float input to a single-precision float value and store the result into a vector register.

```
if SDWA_SRC0_SEL == BYTE1.b3 then
    D0.f32 = fp8_to_f32(S0[15 : 8].fp8)
elsif SDWA_SRC0_SEL == BYTE2.b3 then
    D0.f32 = fp8_to_f32(S0[23 : 16].fp8)
elsif SDWA_SRC0_SEL == BYTE3.b3 then
    D0.f32 = fp8_to_f32(S0[31 : 24].fp8)
else
    // BYTE0 implied
    D0.f32 = fp8_to_f32(S0[7 : 0].fp8)
endif
```

Notes

SDWA encoding allows SRC0_SEL to control which byte of S0 is converted. Only the BYTE selects of SRC0_SEL are legal. If this instruction is not encoded in SDWA then BYTE0 is implied.

V_CVT_F32_BF8 469

Convert from a BF8 float input to a single-precision float value and store the result into a vector register.

```
if SDWA_SRC0_SEL == BYTE1.b3 then
    D0.f32 = bf8_to_f32(S0[15 : 8].bf8)
elsif SDWA_SRC0_SEL == BYTE2.b3 then
    D0.f32 = bf8_to_f32(S0[23 : 16].bf8)
elsif SDWA_SRC0_SEL == BYTE3.b3 then
    D0.f32 = bf8_to_f32(S0[31 : 24].bf8)
else
    // BYTE0 implied
    D0.f32 = bf8_to_f32(S0[7 : 0].bf8)
endif
```

Notes

SDWA encoding allows SRC0_SEL to control which byte of S0 is converted. Only the BYTE selects of SRC0_SEL are legal. If this instruction is not encoded in SDWA then BYTE0 is implied.

V_CVT_PK_F32_FP8 470

Convert from a packed 2-component FP8 float input to a packed single-precision float value and store the result into a vector register.

```
tmp = SDWA_SRC0_SEL[1 : 0] == WORD1.b2 ? S0[31 : 16] : S0[15 : 0];
D0[31 : 0].f32 = fp8_to_f32(tmp[7 : 0].fp8);
D0[63 : 32].f32 = fp8_to_f32(tmp[15 : 8].fp8)
```

Notes

SDWA encoding allows SRC0_SEL to control which word of S0 is converted. Only the WORD selects of SRC0_SEL are legal. If this instruction is not encoded in SDWA then WORD0 is implied.

V_CVT_PK_F32_BF8 471

Convert from a packed 2-component BF8 float input to a packed single-precision float value and store the result into a vector register.

```
tmp = SDWA_SRC0_SEL[1 : 0] == WORD1.b2 ? S0[31 : 16] : S0[15 : 0];
D0[31 : 0].f32 = bf8_to_f32(tmp[7 : 0].bf8);
D0[63 : 32].f32 = bf8_to_f32(tmp[15 : 8].bf8)
```

Notes

SDWA encoding allows SRC0_SEL to control which word of S0 is converted. Only the WORD selects of SRC0_SEL are legal. If this instruction is not encoded in SDWA then WORD0 is implied.

V_CNDMASK_B32 256

Copy data from one of two inputs based on the vector condition code and store the result into a vector register.

```
D0.u32 = VCC.u64[laneId] ? S1.u32 : S0.u32
```

Notes

In VOP3 the VCC source may be a scalar GPR specified in S2.

Floating-point modifiers are valid for this instruction if S0 and S1 are 32-bit floating point values. This instruction is suitable for negating or taking the absolute value of a floating-point value.

V_ADD_F32 257

Add two floating point inputs and store the result into a vector register.

```
D0.f32 = S0.f32 + S1.f32
```

Notes

0.5ULP precision, denormals are supported.

V_SUB_F32 258

Subtract the second floating point input from the first input and store the result into a vector register.

```
D0.f32 = S0.f32 - S1.f32
```

Notes

0.5ULP precision, denormals are supported.

V_SUBREV_F32 259

Subtract the *first* floating point input from the *second* input and store the result into a vector register.

```
D0.f32 = S1.f32 - S0.f32
```

Notes

0.5ULP precision, denormals are supported.

V_FMAC_F64 260

Multiply two floating point inputs and accumulate the result into the destination register using fused multiply add.

```
D0.f64 = fma(S0.f64, S1.f64, D0.f64)
```

V_MUL_F32 261

Multiply two floating point inputs and store the result into a vector register.

```
D0.f32 = S0.f32 * S1.f32
```

Notes

0.5ULP precision, denormals are supported.

V_MUL_I32_I24 262

Multiply two signed 24-bit integer inputs and store the result as a signed 32-bit integer into a vector register.

```
D0.i32 = 32'I(S0.i24) * 32'I(S1.i24)
```

Notes

This opcode is expected to be as efficient as basic single-precision opcodes since it utilizes the single-precision floating point multiplier. See also V_MUL_HI_I32_I24.

V_MUL_HI_I32_I24 263

Multiply two signed 24-bit integer inputs and store the high 32 bits of the result as a signed 32-bit integer into a vector register.

```
D0.i32 = 32'I((64'I(S0.i24) * 64'I(S1.i24)) >> 32U)
```

Notes

See also V_MUL_I32_I24.

V_MUL_U32_U24 264

Multiply two unsigned 24-bit integer inputs and store the result as an unsigned 32-bit integer into a vector register.

```
D0.u32 = 32'U(S0.u24) * 32'U(S1.u24)
```

Notes

This opcode is expected to be as efficient as basic single-precision opcodes since it utilizes the single-precision floating point multiplier. See also V_MUL_HI_U32_U24.

V_MUL_HI_U32_U24 265

Multiply two unsigned 24-bit integer inputs and store the high 32 bits of the result as an unsigned 32-bit integer into a vector register.

```
D0.u32 = 32'U((64'U(S0.u24) * 64'U(S1.u24)) >> 32U)
```

Notes

See also V_MUL_U32_U24.

V_MIN_F32 266

Select the minimum of two single-precision float inputs and store the result into a vector register.

```
if (WAVE_MODE.IEEE && isSignalNAN(64'F(S0.f32))) then
    D0.f32 = 32'F(cvtToQuietNAN(64'F(S0.f32)))
elsif (WAVE_MODE.IEEE && isSignalNAN(64'F(S1.f32))) then
    D0.f32 = 32'F(cvtToQuietNAN(64'F(S1.f32)))
```

```
elsif isNAN(64'F(S0.f32)) then
    D0.f32 = S1.f32
elsif isNAN(64'F(S1.f32)) then
    D0.f32 = S0.f32
elsif ((64'F(S0.f32) == +0.0) && (64'F(S1.f32) == -0.0)) then
    D0.f32 = S1.f32
elsif ((64'F(S0.f32) == -0.0) && (64'F(S1.f32) == +0.0)) then
    D0.f32 = S0.f32
else
    // Note: there's no IEEE case here like there is for V_MAX_F32.
    D0.f32 = S0.f32 < S1.f32 ? S0.f32 : S1.f32
endif
```

V_MAX_F32 267

Select the maximum of two single-precision float inputs and store the result into a vector register.

```
if (WAVE_MODE.IEEE && isSignalNAN(64'F(S0.f32))) then
    D0.f32 = 32'F(cvtToQuietNAN(64'F(S0.f32)))
elsif (WAVE_MODE.IEEE && isSignalNAN(64'F(S1.f32))) then
    D0.f32 = 32'F(cvtToQuietNAN(64'F(S1.f32)))
elsif isNAN(64'F(S0.f32)) then
    D0.f32 = S1.f32
elsif isNAN(64'F(S1.f32)) then
    D0.f32 = S0.f32
elsif ((64'F(S0.f32) == +0.0) && (64'F(S1.f32) == -0.0)) then
    D0.f32 = S0.f32
elsif ((64'F(S0.f32) == -0.0) && (64'F(S1.f32) == +0.0)) then
    D0.f32 = S1.f32
elsif WAVE_MODE.IEEE then
    D0.f32 = S0.f32 >= S1.f32 ? S0.f32 : S1.f32
else
    D0.f32 = S0.f32 > S1.f32 ? S0.f32 : S1.f32
endif
```

V_MIN_I32 268

Select the minimum of two signed 32-bit integer inputs and store the selected value into a vector register.

```
D0.i32 = S0.i32 < S1.i32 ? S0.i32 : S1.i32
```

V_MAX_I32 269

Select the maximum of two signed 32-bit integer inputs and store the selected value into a vector register.

```
D0.i32 = S0.i32 >= S1.i32 ? S0.i32 : S1.i32
```

V_MIN_U32 270

Select the minimum of two unsigned 32-bit integer inputs and store the selected value into a vector register.

```
D0.u32 = S0.u32 < S1.u32 ? S0.u32 : S1.u32
```

V_MAX_U32 271

Select the maximum of two unsigned 32-bit integer inputs and store the selected value into a vector register.

```
D0.u32 = S0.u32 >= S1.u32 ? S0.u32 : S1.u32
```

V_LSHRREV_B32 272

Given a shift count in the *first* vector input, calculate the logical shift right of the *second* vector input and store the result into a vector register.

```
D0.u32 = (S1.u32 >> S0[4 : 0].u32)
```

V_ASHRREV_I32 273

Given a shift count in the *first* vector input, calculate the arithmetic shift right (preserving sign bit) of the *second* vector input and store the result into a vector register.

```
D0.i32 = (S1.i32 >> S0[4 : 0].u32)
```

V_LSHLREV_B32 274

Given a shift count in the *first* vector input, calculate the logical shift left of the *second* vector input and store the result into a vector register.

```
D0.u32 = (S1.u32 << S0[4 : 0].u32)
```

V_AND_B32 275

Calculate bitwise AND on two vector inputs and store the result into a vector register.

```
D0.u32 = (S0.u32 & S1.u32)
```

Notes

Input and output modifiers not supported.

V_OR_B32 276

Calculate bitwise OR on two vector inputs and store the result into a vector register.

```
D0.u32 = (S0.u32 | S1.u32)
```

Notes

Input and output modifiers not supported.

V_XOR_B32 277

Calculate bitwise XOR on two vector inputs and store the result into a vector register.

```
D0.u32 = (S0.u32 ^ S1.u32)
```

Notes

Input and output modifiers not supported.

V_ADD_CO_U32 281

Add two unsigned inputs, store the result into a vector register and store the carry-out mask into a scalar register.

```
tmp = 64'U(S0.u32) + 64'U(S1.u32);
VCC.u64[laneId] = tmp >= 0x100000000ULL ? 1'1U : 1'0U;
// VCC is an UNSIGNED overflow/carry-out for V_ADDC_CO_U32.
D0.u32 = tmp.u32
```

Notes

In VOP3 the VCC destination may be an arbitrary SGPR-pair.

Supports saturation (unsigned 32-bit integer domain).

V_SUB_CO_U32 282

Subtract the second unsigned input from the first input, store the result into a vector register and store the carry-out mask into a scalar register.

```
tmp = S0.u32 - S1.u32;
VCC.u64[laneId] = S1.u32 > S0.u32 ? 1'1U : 1'0U;
// VCC is an UNSIGNED overflow/carry-out for V_SUBB_CO_U32.
D0.u32 = tmp.u32
```

Notes

In VOP3 the VCC destination may be an arbitrary SGPR-pair.

Supports saturation (unsigned 32-bit integer domain).

V_SUBREV_CO_U32 283

Subtract the *first* unsigned input from the *second* input, store the result into a vector register and store the carry-out mask into a scalar register.

```
tmp = S1.u32 - S0.u32;
VCC.u64[laneId] = S0.u32 > S1.u32 ? 1'1U : 1'0U;
// VCC is an UNSIGNED overflow/carry-out for V_SUBB_CO_U32.
D0.u32 = tmp.u32
```

Notes

In VOP3 the VCC destination may be an arbitrary SGPR-pair.

Supports saturation (unsigned 32-bit integer domain).

V_ADDC_CO_U32 284

Add two unsigned inputs and a bit from a carry-in mask, store the result into a vector register and store the carry-out mask into a scalar register.

```
tmp = 64'U(S0.u32) + 64'U(S1.u32) + VCC.u64[laneId].u64;
```

```
VCC.u64[laneId] = tmp >= 0x100000000ULL ? 1'1U : 1'0U;
// VCC is an UNSIGNED overflow.
D0.u32 = tmp.u32
```

Notes

In VOP3 the VCC destination may be an arbitrary SGPR-pair, and the VCC source comes from the SGPR-pair at S2.u.

Supports saturation (unsigned 32-bit integer domain).

V_SUBB_CO_U32 285

Subtract the second unsigned input from the first input, subtract a bit from the carry-in mask, store the result into a vector register and store the carry-out mask to a scalar register.

```
tmp = S0.u32 - S1.u32 - VCC.u64[laneId].u32;
VCC.u64[laneId] = 64'U(S1.u32) + VCC.u64[laneId].u64 > 64'U(S0.u32) ? 1'1U : 1'0U;
// VCC is an UNSIGNED overflow.
D0.u32 = tmp.u32
```

Notes

In VOP3 the VCC destination may be an arbitrary SGPR-pair, and the VCC source comes from the SGPR-pair at S2.u.

Supports saturation (unsigned 32-bit integer domain).

V_SUBBREV_CO_U32 286

Subtract the *first* unsigned input from the *second* input, subtract a bit from the carry-in mask, store the result into a vector register and store the carry-out mask to a scalar register.

```
tmp = S1.u32 - S0.u32 - VCC.u64[laneId].u32;
VCC.u64[laneId] = 64'U(S1.u32) + VCC.u64[laneId].u64 > 64'U(S0.u32) ? 1'1U : 1'0U;
// VCC is an UNSIGNED overflow.
D0.u32 = tmp.u32
```

Notes

In VOP3 the VCC destination may be an arbitrary SGPR-pair, and the VCC source comes from the SGPR-pair at S2.u.

Supports saturation (unsigned 32-bit integer domain).

V_ADD_F16 287

Add two floating point inputs and store the result into a vector register.

```
D0.f16 = S0.f16 + S1.f16
```

Notes

0.5ULP precision. Supports denormals, round mode, exception flags and saturation.

V_SUB_F16 288

Subtract the second floating point input from the first input and store the result into a vector register.

```
D0.f16 = S0.f16 - S1.f16
```

Notes

0.5ULP precision. Supports denormals, round mode, exception flags and saturation.

V_SUBREV_F16 289

Subtract the *first* floating point input from the *second* input and store the result into a vector register.

```
D0.f16 = S1.f16 - S0.f16
```

Notes

0.5ULP precision. Supports denormals, round mode, exception flags and saturation.

V_MUL_F16 290

Multiply two floating point inputs and store the result into a vector register.

```
D0.f16 = S0.f16 * S1.f16
```

Notes

0.5ULP precision. Supports denormals, round mode, exception flags and saturation.

V_MAC_F16 291

Multiply two floating point inputs and accumulate the result into the destination register. Implements IEEE rules and non-standard rule for OPSEL.

```
tmp = S0.f16 * S1.f16 + D0.f16;
if OPSEL.u4[3] then
    D0 = { tmp.f16, D0[15 : 0] }
else
    D0 = { 16'0, tmp.f16 }
endif
```

Notes

Supports round mode, exception flags, saturation.

V_ADD_U16 294

Add two unsigned 16-bit integer inputs and store the result into a vector register. No carry-in or carry-out support.

```
D0.u16 = S0.u16 + S1.u16
```

Notes

Supports saturation (unsigned 16-bit integer domain).

V_SUB_U16 295

Subtract the second unsigned input from the first input and store the result into a vector register. No carry-in or carry-out support.

```
D0.u16 = S0.u16 - S1.u16
```

Notes

Supports saturation (unsigned 16-bit integer domain).

V_SUBREV_U16 296

Subtract the *first* unsigned input from the *second* input and store the result into a vector register. No carry-in or carry-out support.

```
D0.u16 = S1.u16 - S0.u16
```

Notes

Supports saturation (unsigned 16-bit integer domain).

V_MUL_LO_U16 297

Multiply two unsigned 16-bit integer inputs and store the low bits of the result into a vector register.

```
D0.u16 = S0.u16 * S1.u16
```

Notes

Supports saturation (unsigned 16-bit integer domain).

V_LSHLREV_B16 298

Given a shift count in the *first* vector input, calculate the logical shift left of the *second* vector input and store the result into a vector register.

```
D0.u16 = (S1.u16 << S0[3 : 0].u32)
```

V_LSHRREV_B16 299

Given a shift count in the *first* vector input, calculate the logical shift right of the *second* vector input and store the result into a vector register.

```
D0.u16 = (S1.u16 >> S0[3 : 0].u32)
```

V_ASHRREV_I16 300

Given a shift count in the *first* vector input, calculate the arithmetic shift right (preserving sign bit) of the *second* vector input and store the result into a vector register.

```
D0.i16 = (S1.i16 >> S0[3 : 0].u32)
```

V_MAX_F16 301

Select the maximum of two half-precision float inputs and store the result into a vector register.

```
if (WAVE_MODE.IEEE && isSignalNAN(64'F(S0.f16))) then
    D0.f16 = 16'F(cvtToQuietNAN(64'F(S0.f16)))
elsif (WAVE_MODE.IEEE && isSignalNAN(64'F(S1.f16))) then
    D0.f16 = 16'F(cvtToQuietNAN(64'F(S1.f16)))
elsif isNAN(64'F(S0.f16)) then
    D0.f16 = S1.f16
elsif isNAN(64'F(S1.f16)) then
    D0.f16 = S0.f16
elsif ((64'F(S0.f16) == +0.0) && (64'F(S1.f16) == -0.0)) then
    D0.f16 = S0.f16
elsif ((64'F(S0.f16) == -0.0) && (64'F(S1.f16) == +0.0)) then
    D0.f16 = S1.f16
elsif WAVE_MODE.IEEE then
    D0.f16 = S0.f16 >= S1.f16 ? S0.f16 : S1.f16
else
    D0.f16 = S0.f16 > S1.f16 ? S0.f16 : S1.f16
endif
```

Notes

IEEE compliant. Supports denormals, round mode, exception flags, saturation.

V_MIN_F16 302

Select the minimum of two half-precision float inputs and store the result into a vector register.

```
if (WAVE_MODE.IEEE && isSignalNAN(64'F(S0.f16))) then
    D0.f16 = 16'F(cvtToQuietNAN(64'F(S0.f16)))
elsif (WAVE_MODE.IEEE && isSignalNAN(64'F(S1.f16))) then
    D0.f16 = 16'F(cvtToQuietNAN(64'F(S1.f16)))
elsif isNAN(64'F(S0.f16)) then
    D0.f16 = S1.f16
elsif isNAN(64'F(S1.f16)) then
    D0.f16 = S0.f16
elsif ((64'F(S0.f16) == +0.0) && (64'F(S1.f16) == -0.0)) then
    D0.f16 = S1.f16
elsif ((64'F(S0.f16) == -0.0) && (64'F(S1.f16) == +0.0)) then
    D0.f16 = S0.f16
else
    // Note: there's no IEEE case here like there is for V_MAX_F16.
    D0.f16 = S0.f16 < S1.f16 ? S0.f16 : S1.f16
endif
```

Notes

IEEE compliant. Supports denormals, round mode, exception flags, saturation.

V_MAX_U16 303

Select the maximum of two unsigned 16-bit integer inputs and store the selected value into a vector register.

```
D0.u16 = S0.u16 >= S1.u16 ? S0.u16 : S1.u16
```

V_MAX_I16 304

Select the maximum of two signed 16-bit integer inputs and store the selected value into a vector register.

```
D0.i16 = S0.i16 >= S1.i16 ? S0.i16 : S1.i16
```

V_MIN_U16 305

Select the minimum of two unsigned 16-bit integer inputs and store the selected value into a vector register.

```
D0.u16 = S0.u16 < S1.u16 ? S0.u16 : S1.u16
```

V_MIN_I16 306

Select the minimum of two signed 16-bit integer inputs and store the selected value into a vector register.

```
D0.i16 = S0.i16 < S1.i16 ? S0.i16 : S1.i16
```

V_LDEXP_F16 307

Multiply the first input, a floating point value, by an integral power of 2 specified in the second input, a signed integer value, and store the floating point result into a vector register.

```
D0.f16 = S0.f16 * 16'F(2.0F ** 32'I(S1.i16))
```

Notes

Compare with the `ldexp()` function in C. Note that the S1 has a format of f16 since floating point literal constants are interpreted as 16 bit value for this opcode.

V_ADD_U32 308

Add two unsigned 32-bit integer inputs and store the result into a vector register. No carry-in or carry-out support.

```
D0.u32 = S0.u32 + S1.u32
```

Notes

Supports saturation (unsigned 32-bit integer domain).

V_SUB_U32 309

Subtract the second unsigned input from the first input and store the result into a vector register. No carry-in or carry-out support.

```
D0.u32 = S0.u32 - S1.u32
```

Notes

Supports saturation (unsigned 32-bit integer domain).

V_SUBREV_U32 310

Subtract the *first* unsigned input from the *second* input and store the result into a vector register. No carry-in or carry-out support.

```
D0.u32 = S1.u32 - S0.u32
```

Notes

Supports saturation (unsigned 32-bit integer domain).

V_DOT2C_F32_F16 311

Compute the dot product of two packed 2-D half-precision float inputs in the single-precision float domain and accumulate with the single-precision float value in the destination register.

```
tmp = D0.f32;
tmp += f16_to_f32(S0[15 : 0].f16) * f16_to_f32(S1[15 : 0].f16);
```

```
tmp += f16_to_f32(S0[31 : 16].f16) * f16_to_f32(S1[31 : 16].f16);
D0.f32 = tmp
```

V_DOT2C_I32_I16 312

Compute the dot product of two packed 2-D signed 16-bit integer inputs in the signed 32-bit integer domain and accumulate with the signed 32-bit integer value in the destination register.

```
tmp = D0.i32;
tmp += i16_to_i32(S0[15 : 0].i16) * i16_to_i32(S1[15 : 0].i16);
tmp += i16_to_i32(S0[31 : 16].i16) * i16_to_i32(S1[31 : 16].i16);
D0.i32 = tmp
```

V_DOT4C_I32_I8 313

Compute the dot product of two packed 4-D signed 8-bit integer inputs in the signed 32-bit integer domain and accumulate with the signed 32-bit integer value in the destination register.

```
tmp = D0.i32;
tmp += i8_to_i32(S0[7 : 0].i8) * i8_to_i32(S1[7 : 0].i8);
tmp += i8_to_i32(S0[15 : 8].i8) * i8_to_i32(S1[15 : 8].i8);
tmp += i8_to_i32(S0[23 : 16].i8) * i8_to_i32(S1[23 : 16].i8);
tmp += i8_to_i32(S0[31 : 24].i8) * i8_to_i32(S1[31 : 24].i8);
D0.i32 = tmp
```

V_DOT8C_I32_I4 314

Compute the dot product of two packed 8-D signed 4-bit integer inputs in the signed 32-bit integer domain and accumulate with the signed 32-bit integer value in the destination register.

```
tmp = D0.i32;
tmp += i4_to_i32(S0[3 : 0].i4) * i4_to_i32(S1[3 : 0].i4);
tmp += i4_to_i32(S0[7 : 4].i4) * i4_to_i32(S1[7 : 4].i4);
tmp += i4_to_i32(S0[11 : 8].i4) * i4_to_i32(S1[11 : 8].i4);
tmp += i4_to_i32(S0[15 : 12].i4) * i4_to_i32(S1[15 : 12].i4);
tmp += i4_to_i32(S0[19 : 16].i4) * i4_to_i32(S1[19 : 16].i4);
tmp += i4_to_i32(S0[23 : 20].i4) * i4_to_i32(S1[23 : 20].i4);
tmp += i4_to_i32(S0[27 : 24].i4) * i4_to_i32(S1[27 : 24].i4);
tmp += i4_to_i32(S0[31 : 28].i4) * i4_to_i32(S1[31 : 28].i4);
D0.i32 = tmp
```

V_FMAC_F32 315

Multiply two floating point inputs and accumulate the result into the destination register using fused multiply add.

```
D0.f32 = fma(S0.f32, S1.f32, D0.f32)
```

V_PK_FMAC_F16 316

Multiply two packed half-precision float inputs component-wise and accumulate the result into the destination register using fused multiply add.

```
D0[15 : 0].f16 = fma(S0[15 : 0].f16, S1[15 : 0].f16, D0[15 : 0].f16);
D0[31 : 16].f16 = fma(S0[31 : 16].f16, S1[31 : 16].f16, D0[31 : 16].f16)
```

V_XNOR_B32 317

Calculate bitwise XNOR on two vector inputs and store the result into a vector register.

```
D0.u32 = ~(S0.u32 ^ S1.u32)
```

Notes

Input and output modifiers not supported.

V_MAD_I32_I24 450

Multiply two signed 24-bit integer inputs in the signed 32-bit integer domain, add a signed 32-bit integer value from a third input, and store the result as a signed 32-bit integer into a vector register.

```
D0.i32 = 32'I(S0.i24) * 32'I(S1.i24) + S2.i32
```

V_MAD_U32_U24 451

Multiply two unsigned 24-bit integer inputs in the unsigned 32-bit integer domain, add a unsigned 32-bit integer value from a third input, and store the result as an unsigned 32-bit integer into a vector register.

```
D0.u32 = 32'U(S0.u24) * 32'U(S1.u24) + S2.u32
```

V_CUBEID_F32 452

Compute the cubemap face ID of a 3D coordinate specified as three single-precision float inputs. Store the result in *single-precision float* format into a vector register.

```
// Set D0.f = cubemap face ID ({0.0, 1.0, ..., 5.0}).
// XYZ coordinate is given in (S0.f, S1.f, S2.f).
// S0.f = x
// S1.f = y
// S2.f = z
if ((abs(S2.f32) >= abs(S0.f32)) && (abs(S2.f32) >= abs(S1.f32))) then
    if S2.f32 < 0.0F then
        D0.f32 = 5.0F
    else
        D0.f32 = 4.0F
    endif
elsif abs(S1.f32) >= abs(S0.f32) then
    if S1.f32 < 0.0F then
        D0.f32 = 3.0F
    else
        D0.f32 = 2.0F
    endif
else
    if S0.f32 < 0.0F then
        D0.f32 = 1.0F
    else
        D0.f32 = 0.0F
    endif
endif
```

V_CUBESC_F32 453

Compute the cubemap S coordinate of a 3D coordinate specified as three single-precision float inputs. Store the result in single-precision float format into a vector register.

```
// D0.f = cubemap S coordinate.
// XYZ coordinate is given in (S0.f, S1.f, S2.f).
// S0.f = x
// S1.f = y
// S2.f = z
if ((abs(S2.f32) >= abs(S0.f32)) && (abs(S2.f32) >= abs(S1.f32))) then
    if S2.f32 < 0.0F then
        D0.f32 = -S0.f32
    else
        D0.f32 = S0.f32
    endif
```

```
elsif abs(S1.f32) >= abs(S0.f32) then
    D0.f32 = S0.f32
else
    if S0.f32 < 0.0F then
        D0.f32 = S2.f32
    else
        D0.f32 = -S2.f32
    endif
endif
```

V_CUBETC_F32 454

Compute the cubemap T coordinate of a 3D coordinate specified as three single-precision float inputs. Store the result in single-precision float format into a vector register.

```
// D0.f = cubemap T coordinate.
// XYZ coordinate is given in (S0.f, S1.f, S2.f).
// S0.f = x
// S1.f = y
// S2.f = z
if ((abs(S2.f32) >= abs(S0.f32)) && (abs(S2.f32) >= abs(S1.f32))) then
    D0.f32 = -S1.f32
elsif abs(S1.f32) >= abs(S0.f32) then
    if S1.f32 < 0.0F then
        D0.f32 = -S2.f32
    else
        D0.f32 = S2.f32
    endif
else
    D0.f32 = -S1.f32
endif
```

V_CUBEMA_F32 455

Compute the cubemap major axis coordinate of a 3D coordinate specified as three single-precision float inputs. Store the result in single-precision float format into a vector register.

```
// D0.f = 2.0 * cubemap major axis.
// XYZ coordinate is given in (S0.f, S1.f, S2.f).
// S0.f = x
// S1.f = y
// S2.f = z
if ((abs(S2.f32) >= abs(S0.f32)) && (abs(S2.f32) >= abs(S1.f32))) then
    D0.f32 = S2.f32 * 2.0F
elsif abs(S1.f32) >= abs(S0.f32) then
    D0.f32 = S1.f32 * 2.0F
else
    D0.f32 = S0.f32 * 2.0F
```

```
endif
```

V_BFE_U32 456

Extract an unsigned bitfield from the first input using field offset from the second input and size from the third input, then store the result into a vector register.

```
D0.u32 = ((S0.u32 >> S1[4 : 0].u32) & ((1U << S2[4 : 0].u32) - 1U))
```

V_BFE_I32 457

Extract a signed bitfield from the first input using field offset from the second input and size from the third input, then store the result into a vector register.

```
tmp.i32 = ((S0.i32 >> S1[4 : 0].u32) & ((1 << S2[4 : 0].u32) - 1));
D0.i32 = signext_from_bit(tmp.i32, S2[4 : 0].u32)
```

V_BFI_B32 458

Overwrite a bitfield in the third input with a bitfield from the second input using a mask from the first input, then store the result into a vector register.

```
D0.u32 = ((S0.u32 & S1.u32) | (~S0.u32 & S2.u32))
```

V_FMA_F32 459

Multiply two single-precision float inputs and add a third input using fused multiply add, and store the result into a vector register.

```
D0.f32 = fma(S0.f32, S1.f32, S2.f32)
```

Notes

0.5ULP accuracy, denormals are supported.

V_FMA_F64 460

Multiply two double-precision float inputs and add a third input using fused multiply add, and store the result into a vector register.

```
D0.f64 = fma(S0.f64, S1.f64, S2.f64)
```

Notes

0.5ULP accuracy, denormals are supported.

V_LERP_U8 461

Average two 4-D vectors stored as packed bytes in the first two inputs with rounding control provided by the third input, then store the result into a vector register. Each byte in the third input acts as a rounding mode for the corresponding element; if the LSB is set then 0.5 rounds up, otherwise 0.5 truncates.

```
tmp = ((S0.u32[31 : 24] + S1.u32[31 : 24] + S2.u32[24].u8) >> 1U << 24U);
tmp += ((S0.u32[23 : 16] + S1.u32[23 : 16] + S2.u32[16].u8) >> 1U << 16U);
tmp += ((S0.u32[15 : 8] + S1.u32[15 : 8] + S2.u32[8].u8) >> 1U << 8U);
tmp += ((S0.u32[7 : 0] + S1.u32[7 : 0] + S2.u32[0].u8) >> 1U);
D0.u32 = tmp.u32
```

V_ALIGNBIT_B32 462

Align a 64-bit value encoded in the first two inputs to a bit position specified in the third input, then store the result into a 32-bit vector register.

```
D0.u32 = 32'U(({ S0.u32, S1.u32 } >> S2.u32[4 : 0].u32) & 0xffffffffLL)
```

Notes

S0 carries the MSBs and S1 carries the LSBs of the value being aligned.

V_ALIGNBYTE_B32 463

Align a 64-bit value encoded in the first two inputs to a byte position specified in the third input, then store the result into a 32-bit vector register.

```
D0.u32 = 32'U(({ S0.u32, S1.u32 } >> (S2.u32[1 : 0].u32 * 8U)) & 0xffffffffLL)
```

Notes

S0 carries the MSBs and S1 carries the LSBs of the value being aligned.

V_MIN3_F32 464

Select the minimum of three single-precision float inputs and store the selected value into a vector register.

```
D0.f32 = v_min_f32(v_min_f32(S0.f32, S1.f32), S2.f32)
```

V_MIN3_I32 465

Select the minimum of three signed 32-bit integer inputs and store the selected value into a vector register.

```
D0.i32 = v_min_i32(v_min_i32(S0.i32, S1.i32), S2.i32)
```

V_MIN3_U32 466

Select the minimum of three unsigned 32-bit integer inputs and store the selected value into a vector register.

```
D0.u32 = v_min_u32(v_min_u32(S0.u32, S1.u32), S2.u32)
```

V_MAX3_F32 467

Select the maximum of three single-precision float inputs and store the selected value into a vector register.

```
D0.f32 = v_max_f32(v_max_f32(S0.f32, S1.f32), S2.f32)
```

V_MAX3_I32 468

Select the maximum of three signed 32-bit integer inputs and store the selected value into a vector register.

```
D0.i32 = v_max_i32(v_max_i32(S0.i32, S1.i32), S2.i32)
```

V_MAX3_U32 469

Select the maximum of three unsigned 32-bit integer inputs and store the selected value into a vector register.

```
D0.u32 = v_max_u32(v_max_u32(S0.u32, S1.u32), S2.u32)
```

V_MED3_F32 470

Select the median of three single-precision float values and store the selected value into a vector register.

```
if (isNAN(64'F(S0.f32)) || isNAN(64'F(S1.f32)) || isNAN(64'F(S2.f32))) then
    D0.f32 = v_min3_f32(S0.f32, S1.f32, S2.f32)
elsif v_max3_f32(S0.f32, S1.f32, S2.f32) == S0.f32 then
    D0.f32 = v_max_f32(S1.f32, S2.f32)
elsif v_max3_f32(S0.f32, S1.f32, S2.f32) == S1.f32 then
    D0.f32 = v_max_f32(S0.f32, S2.f32)
else
    D0.f32 = v_max_f32(S0.f32, S1.f32)
endif
```

V_MED3_I32 471

Select the median of three signed 32-bit integer values and store the selected value into a vector register.

```
if v_max3_i32(S0.i32, S1.i32, S2.i32) == S0.i32 then
    D0.i32 = v_max_i32(S1.i32, S2.i32)
elsif v_max3_i32(S0.i32, S1.i32, S2.i32) == S1.i32 then
    D0.i32 = v_max_i32(S0.i32, S2.i32)
else
    D0.i32 = v_max_i32(S0.i32, S1.i32)
endif
```

V_MED3_U32 472

Select the median of three unsigned 32-bit integer values and store the selected value into a vector register.

```
if v_max3_u32(S0.u32, S1.u32, S2.u32) == S0.u32 then
    D0.u32 = v_max_u32(S1.u32, S2.u32)
elsif v_max3_u32(S0.u32, S1.u32, S2.u32) == S1.u32 then
    D0.u32 = v_max_u32(S0.u32, S2.u32)
else
    D0.u32 = v_max_u32(S0.u32, S1.u32)
endif
```

V_SAD_U8 473

Calculate the sum of absolute differences of elements in two packed 4-component unsigned 8-bit integer inputs, add an unsigned 32-bit integer value from the third input and store the result into a vector register.

```
ABSDIFF = lambda(x, y) (
    x > y ? x - y : y - x);
// UNSIGNED comparison
tmp = S2.u32;
tmp += 32'U(ABSDIFF(S0.u32[7 : 0], S1.u32[7 : 0]));
tmp += 32'U(ABSDIFF(S0.u32[15 : 8], S1.u32[15 : 8]));
tmp += 32'U(ABSDIFF(S0.u32[23 : 16], S1.u32[23 : 16]));
tmp += 32'U(ABSDIFF(S0.u32[31 : 24], S1.u32[31 : 24]));
D0.u32 = tmp
```

Notes

Overflow into the upper bits is allowed.

V_SAD_HI_U8 474

Calculate the sum of absolute differences of elements in two packed 4-component unsigned 8-bit integer inputs, shift the sum left by 16 bits, add an unsigned 32-bit integer value from the third input and store the result into a vector register.

```
D0.u32 = (32'U(v_sad_u8(S0, S1, 0U)) << 16U) + S2.u32
```

Notes

Overflow into the upper bits is allowed.

V_SAD_U16 475

Calculate the sum of absolute differences of elements in two packed 2-component unsigned 16-bit integer inputs, add an unsigned 32-bit integer value from the third input and store the result into a vector register.

```
ABSDIFF = lambda(x, y) (
    x > y ? x - y : y - x);
// UNSIGNED comparison
tmp = S2.u32;
tmp += ABSDIFF(S0[15 : 0].u16, S1[15 : 0].u16);
tmp += ABSDIFF(S0[31 : 16].u16, S1[31 : 16].u16);
D0.u32 = tmp
```

V_SAD_U32 476

Calculate the absolute difference of two unsigned 32-bit integer inputs, add an unsigned 32-bit integer value from the third input and store the result into a vector register.

```
ABSDIFF = lambda(x, y) (
    x > y ? x - y : y - x);
// UNSIGNED comparison
D0.u32 = ABSDIFF(S0.u32, S1.u32) + S2.u32
```

V_CVT_PK_U8_F32 477

Convert a single-precision float value from the first input to an unsigned 8-bit integer value and pack the result into one byte of the third input using the second input as a byte select. Store the result into a vector register.

```
tmp = (S2.u32 & 32'U(~(0xff << (S1.u32[1 : 0].u32 * 8U))));
tmp = (tmp | ((32'U(f32_to_u8(S0.f32)) & 255U) << (S1.u32[1 : 0].u32 * 8U)));
D0.u32 = tmp
```

V_DIV_FIXUP_F32 478

Given a single-precision float quotient in the first input, a denominator in the second input and a numerator in the third input, detect and apply corner cases related to division, including divide by zero, NaN inputs and overflow, and modify the quotient accordingly. Generate any invalid, denormal and divide-by-zero exceptions that are a result of the division. Store the modified quotient into a vector register.

This operation handles corner cases in a division macro such as divide by zero and NaN inputs. This operation is well defined when the quotient is approximately equal to the numerator divided by the denominator. Other inputs produce a predictable result but may not be mathematically useful.

```
sign_out = (sign(S1.f32) ^ sign(S2.f32));
if isNAN(64'F(S2.f32)) then
    D0.f32 = 32'F(cvtToQuietNAN(64'F(S2.f32)))
elsif isNAN(64'F(S1.f32)) then
    D0.f32 = 32'F(cvtToQuietNAN(64'F(S1.f32)))
elsif ((64'F(S1.f32) == 0.0) && (64'F(S2.f32) == 0.0)) then
    // 0/0
    D0.f32 = 32'F(0xffc00000)
elsif ((64'F(abs(S1.f32)) == +INF) && (64'F(abs(S2.f32)) == +INF)) then
    // inf/inf
    D0.f32 = 32'F(0xffc00000)
elsif ((64'F(S1.f32) == 0.0) || (64'F(abs(S2.f32)) == +INF)) then
    // x/0, or inf/y
    D0.f32 = sign_out ? -INF.f32 : +INF.f32
elsif ((64'F(abs(S1.f32)) == +INF) || (64'F(S2.f32) == 0.0)) then
    // x/inf, 0/y
    D0.f32 = sign_out ? -0.0F : 0.0F
```

```
elsif exponent(S2.f32) - exponent(S1.f32) < -150 then
    D0.f32 = sign_out ? -UNDERFLOW_F32 : UNDERFLOW_F32
elsif exponent(S1.f32) == 255 then
    D0.f32 = sign_out ? -OVERFLOW_F32 : OVERFLOW_F32
else
    D0.f32 = sign_out ? -abs(S0.f32) : abs(S0.f32)
endif
```

Notes

This operation is the final step of a high precision division macro and handles all exceptional cases of division.

V_DIV_FIXUP_F64 479

Given a double-precision float quotient in the first input, a denominator in the second input and a numerator in the third input, detect and apply corner cases related to division, including divide by zero, NaN inputs and overflow, and modify the quotient accordingly. Generate any invalid, denormal and divide-by-zero exceptions that are a result of the division. Store the modified quotient into a vector register.

This operation handles corner cases in a division macro such as divide by zero and NaN inputs. This operation is well defined when the quotient is approximately equal to the numerator divided by the denominator. Other inputs produce a predictable result but may not be mathematically useful.

```
sign_out = (sign(S1.f64) ^ sign(S2.f64));
if isNAN(S2.f64) then
    D0.f64 = cvtToQuietNAN(S2.f64)
elsif isNAN(S1.f64) then
    D0.f64 = cvtToQuietNAN(S1.f64)
elsif ((S1.f64 == 0.0) && (S2.f64 == 0.0)) then
    // 0/0
    D0.f64 = 64'F(0xfff8000000000000LL)
elsif ((abs(S1.f64) == +INF) && (abs(S2.f64) == +INF)) then
    // inf/inf
    D0.f64 = 64'F(0xfff8000000000000LL)
elsif ((S1.f64 == 0.0) || (abs(S2.f64) == +INF)) then
    // x/0, or inf/y
    D0.f64 = sign_out ? -INF : +INF
elsif ((abs(S1.f64) == +INF) || (S2.f64 == 0.0)) then
    // x/inf, 0/y
    D0.f64 = sign_out ? -0.0 : 0.0
elsif exponent(S2.f64) - exponent(S1.f64) < -1075 then
    D0.f64 = sign_out ? -UNDERFLOW_F64 : UNDERFLOW_F64
elsif exponent(S1.f64) == 2047 then
    D0.f64 = sign_out ? -OVERFLOW_F64 : OVERFLOW_F64
else
    D0.f64 = sign_out ? -abs(S0.f64) : abs(S0.f64)
endif
```

Notes

This operation is the final step of a high precision division macro and handles all exceptional cases of division.

V_DIV_SCALE_F32 480

Given a single-precision float value to scale in the first input, a denominator in the second input and a numerator in the third input, scale the first input for division if required to avoid subnormal terms appearing during application of the Newton-Raphson correction method. Store the scaled result into a vector register and set the vector condition code iff post-scaling is required.

This operation is designed for use in a high precision division macro. The first input should be the same value as either the second or third input; other scale values produce predictable results but may not be mathematically useful. The vector condition code is used by V_DIV_FMAS_F32 to determine if the quotient requires post-scaling.

```
VCC = 0x0LL;
if ((64'F(S2.f32) == 0.0) || (64'F(S1.f32) == 0.0)) then
    D0.f32 = NAN.f32
elsif exponent(S2.f32) - exponent(S1.f32) >= 96 then
    // N/D near MAX_FLOAT_F32
    VCC = 0x1LL;
    if S0.f32 == S1.f32 then
        // Only scale the denominator
        D0.f32 = ldexp(S0.f32, 64)
    endif
elsif S1.f32 == DENORM.f32 then
    D0.f32 = ldexp(S0.f32, 64)
elsif ((1.0 / 64'F(S1.f32) == DENORM.f64) && (S2.f32 / S1.f32 == DENORM.f32)) then
    VCC = 0x1LL;
    if S0.f32 == S1.f32 then
        // Only scale the denominator
        D0.f32 = ldexp(S0.f32, 64)
    endif
elsif 1.0 / 64'F(S1.f32) == DENORM.f64 then
    D0.f32 = ldexp(S0.f32, -64)
elsif S2.f32 / S1.f32 == DENORM.f32 then
    VCC = 0x1LL;
    if S0.f32 == S2.f32 then
        // Only scale the numerator
        D0.f32 = ldexp(S0.f32, 64)
    endif
elsif exponent(S2.f32) <= 23 then
    // Numerator is tiny
    D0.f32 = ldexp(S0.f32, 64)
endif
```

Notes

V_DIV_SCALE_F32, V_DIV_FMAS_F32 and V_DIV_FIXUP_F32 are all designed for use in a high precision division macro that utilizes V_RCP_F32 and V_MUL_F32 to compute the approximate result and then applies two steps of the Newton-Raphson method to converge to the quotient. If subnormal terms appear during this calculation then a loss of precision occurs. This loss of precision can be avoided by scaling the inputs and then post-scaling the quotient after Newton-Raphson is applied.

V_DIV_SCALE_F64 481

Given a double-precision float value to scale in the first input, a denominator in the second input and a numerator in the third input, scale the first input for division if required to avoid subnormal terms appearing during application of the Newton-Raphson correction method. Store the scaled result into a vector register and set the vector condition code iff post-scaling is required.

This operation is designed for use in a high precision division macro. The first input should be the same value as either the second or third input; other scale values produce predictable results but may not be mathematically useful. The vector condition code is used by V_DIV_FMAS_F64 to determine if the quotient requires post-scaling.

```
VCC = 0x0LL;
if ((S2.f64 == 0.0) || (S1.f64 == 0.0)) then
    D0.f64 = NAN.f64
elsif exponent(S2.f64) - exponent(S1.f64) >= 768 then
    // N/D near MAX_FLOAT_F64
    VCC = 0x1LL;
    if S0.f64 == S1.f64 then
        // Only scale the denominator
        D0.f64 = ldexp(S0.f64, 128)
    endif
elsif S1.f64 == DENORM.f64 then
    D0.f64 = ldexp(S0.f64, 128)
elsif ((1.0 / S1.f64 == DENORM.f64) && (S2.f64 / S1.f64 == DENORM.f64)) then
    VCC = 0x1LL;
    if S0.f64 == S1.f64 then
        // Only scale the denominator
        D0.f64 = ldexp(S0.f64, 128)
    endif
elsif 1.0 / S1.f64 == DENORM.f64 then
    D0.f64 = ldexp(S0.f64, -128)
elsif S2.f64 / S1.f64 == DENORM.f64 then
    VCC = 0x1LL;
    if S0.f64 == S2.f64 then
        // Only scale the numerator
        D0.f64 = ldexp(S0.f64, 128)
    endif
elsif exponent(S2.f64) <= 53 then
    // Numerator is tiny
    D0.f64 = ldexp(S0.f64, 128)
endif
```

Notes

V_DIV_SCALE_F64, V_DIV_FMAS_F64 and V_DIV_FIXUP_F64 are all designed for use in a high precision division macro that utilizes V_RCP_F64 and V_MUL_F64 to compute the approximate result and then applies two steps of the Newton-Raphson method to converge to the quotient. If subnormal terms appear during this calculation then a loss of precision occurs. This loss of precision can be avoided by scaling the inputs and then post-scaling the quotient after Newton-Raphson is applied.

V_DIV_FMAS_F32 482

Multiply two single-precision float inputs and add a third input using fused multiply add, then scale the exponent of the result by a fixed factor if the vector condition code is set. Store the result into a vector register.

This operation is designed for use in floating point division macros and relies on V_DIV_SCALE_F32 to set the vector condition code iff the quotient requires post-scaling.

```
if VCC.u64[laneId] then
    D0.f32 = 2.0F ** 32 * fma(S0.f32, S1.f32, S2.f32)
else
    D0.f32 = fma(S0.f32, S1.f32, S2.f32)
endif
```

Notes

Input denormals are not flushed but output flushing is allowed.

V_DIV_SCALE_F32, V_DIV_FMAS_F32 and V_DIV_FIXUP_F32 are all designed for use in a high precision division macro that utilizes V_RCP_F32 and V_MUL_F32 to compute the approximate result and then applies two steps of the Newton-Raphson method to converge to the quotient. If subnormal terms appear during this calculation then a loss of precision occurs. This loss of precision can be avoided by scaling the inputs and then post-scaling the quotient after Newton-Raphson is applied.

V_DIV_FMAS_F64 483

Multiply two double-precision float inputs and add a third input using fused multiply add, then scale the exponent of the result by a fixed factor if the vector condition code is set. Store the result into a vector register.

This operation is designed for use in floating point division macros and relies on V_DIV_SCALE_F64 to set the vector condition code iff the quotient requires post-scaling.

```
if VCC.u64[laneId] then
    D0.f64 = 2.0 ** 64 * fma(S0.f64, S1.f64, S2.f64)
else
    D0.f64 = fma(S0.f64, S1.f64, S2.f64)
endif
```

Notes

Input denormals are not flushed but output flushing is allowed.

V_DIV_SCALE_F64, V_DIV_FMAS_F64 and V_DIV_FIXUP_F64 are all designed for use in a high precision division macro that utilizes V_RCP_F64 and V_MUL_F64 to compute the approximate result and then applies two steps of the Newton-Raphson method to converge to the quotient. If subnormal terms appear during this calculation then a loss of precision occurs. This loss of precision can be avoided by scaling the inputs and then post-scaling the quotient after Newton-Raphson is applied.

V_MSAD_U8 484

Calculate the sum of absolute differences of elements in two packed 4-component unsigned 8-bit integer inputs, except that elements where the *second* input (known as the reference input) is zero are not included in the sum. Add an unsigned 32-bit integer value from the third input and store the result into a vector register.

```
ABSDIFF = lambda(x, y) (
    x > y ? x - y : y - x);
// UNSIGNED comparison
tmp = S2.u32;
tmp += S1.u32[7 : 0] == 8'0U ? 0U : 32'U(ABSDIFF(S0.u32[7 : 0], S1.u32[7 : 0]));
tmp += S1.u32[15 : 8] == 8'0U ? 0U : 32'U(ABSDIFF(S0.u32[15 : 8], S1.u32[15 : 8]));
tmp += S1.u32[23 : 16] == 8'0U ? 0U : 32'U(ABSDIFF(S0.u32[23 : 16], S1.u32[23 : 16]));
tmp += S1.u32[31 : 24] == 8'0U ? 0U : 32'U(ABSDIFF(S0.u32[31 : 24], S1.u32[31 : 24]));
D0.u32 = tmp
```

Notes

Overflow into the upper bits is allowed.

V_QSAD_PK_U16_U8 485

Perform the V_SAD_U8 operation four times using different slices of the first array, all entries of the second array and each entry of the third array. Truncate each result to 16 bits, pack the values into a 4-entry array and store the array into a vector register. The first input is an 8-entry array of unsigned 8-bit integers, the second input is a 4-entry array of unsigned 8-bit integers and the third input is a 4-entry array of unsigned 16-bit integers.

```
tmp[63 : 48] = 16'B(v_sad_u8(S0[55 : 24], S1[31 : 0], S2[63 : 48].u32));
tmp[47 : 32] = 16'B(v_sad_u8(S0[47 : 16], S1[31 : 0], S2[47 : 32].u32));
tmp[31 : 16] = 16'B(v_sad_u8(S0[39 : 8], S1[31 : 0], S2[31 : 16].u32));
tmp[15 : 0] = 16'B(v_sad_u8(S0[31 : 0], S1[31 : 0], S2[15 : 0].u32));
D0.b64 = tmp.b64
```

V_MQSAD_PK_U16_U8 486

Perform the V_MSAD_U8 operation four times using different slices of the first array, all entries of the second array and each entry of the third array. Truncate each result to 16 bits, pack the values into a 4-entry array and store the array into a vector register. The first input is an 8-entry array of unsigned 8-bit integers, the second input is a 4-entry array of unsigned 8-bit integers and the third input is a 4-entry array of unsigned 16-bit integers.

```
tmp[63 : 48] = 16'B(v_msad_u8(S0[55 : 24], S1[31 : 0], S2[63 : 48].u32));
tmp[47 : 32] = 16'B(v_msad_u8(S0[47 : 16], S1[31 : 0], S2[47 : 32].u32));
tmp[31 : 16] = 16'B(v_msad_u8(S0[39 : 8], S1[31 : 0], S2[31 : 16].u32));
tmp[15 : 0] = 16'B(v_msad_u8(S0[31 : 0], S1[31 : 0], S2[15 : 0].u32));
```

```
D0.b64 = tmp.b64
```

V_MQSAD_U32_U8 487

Perform the V_MSAD_U8 operation four times using different slices of the first array, all entries of the second array and each entry of the third array. Pack each 32-bit value into a 4-entry array and store the array into a vector register. The first input is an 8-entry array of unsigned 8-bit integers, the second input is a 4-entry array of unsigned 8-bit integers and the third input is a 4-entry array of unsigned 32-bit integers.

```
tmp[127 : 96] = 32'B(v_msad_u8(S0[55 : 24], S1[31 : 0], S2[127 : 96].u32));
tmp[95 : 64] = 32'B(v_msad_u8(S0[47 : 16], S1[31 : 0], S2[95 : 64].u32));
tmp[63 : 32] = 32'B(v_msad_u8(S0[39 : 8], S1[31 : 0], S2[63 : 32].u32));
tmp[31 : 0] = 32'B(v_msad_u8(S0[31 : 0], S1[31 : 0], S2[31 : 0].u32));
D0.b128 = tmp.b128
```

V_MAD_U64_U32 488

Multiply two unsigned integer inputs, add a third unsigned integer input, store the result into a 64-bit vector register and store the overflow/carryout into a scalar mask register.

```
{ D1.u1, D0.u64 } = 65'B(65'U(S0.u32) * 65'U(S1.u32) + 65'U(S2.u64))
```

Notes

In VOP3 the VCC destination may be an arbitrary SGPR-pair.

V_MAD_I64_I32 489

Multiply two signed integer inputs, add a third signed integer input, store the result into a 64-bit vector register and store the overflow/carryout into a scalar mask register.

```
{ D1.i1, D0.i64 } = 65'B(65'I(S0.i32) * 65'I(S1.i32) + 65'I(S2.i64))
```

Notes

In VOP3 the VCC destination may be an arbitrary SGPR-pair.

V_MAD_LEGACY_F16 490

Multiply add of FP16 values. Implements IEEE rules and non-standard rule for OPSEL.

```
tmp = S0.f16 * S1.f16 + S2.f16;
if OPSEL.u4[3] then
    D0 = { tmp.f16, D0[15 : 0] }
else
    D0 = { 16'0, tmp.f16 }
endif
```

Notes

Supports round mode, exception flags, saturation.

If OPSEL[3] is 0 Result is written to 16 LSBs of destination VGPR and hi 16 bits are written as 0 (this is different from V_MAD_F16).

If OPSEL[3] is 1 Result is written to 16 MSBs of destination VGPR and lo 16 bits are preserved.

V_MAD_LEGACY_U16 491

Multiply add of unsigned short values. Has non-standard rule for OPSEL.

```
tmp = S0.u16 * S1.u16 + S2.u16;
if OPSEL.u4[3] then
    D0 = { tmp.u16, D0[15 : 0] }
else
    D0 = { 16'0, tmp.u16 }
endif
```

Notes

Supports saturation (unsigned 16-bit integer domain).

If OPSEL[3] is 0 Result is written to 16 LSBs of destination VGPR and hi 16 bits are written as 0 (this is different from V_MAD_U16).

If OPSEL[3] is 1 Result is written to 16 MSBs of destination VGPR and lo 16 bits are preserved.

V_MAD_LEGACY_I16 492

Multiply add of signed short values. Has non-standard rule for OPSEL.

```
tmp = S0.i16 * S1.i16 + S2.i16;
if OPSEL.u4[3] then
    D0 = { tmp.i16, D0[15 : 0] }
else
    D0 = { 16'0, tmp.i16 }
endif
```

Notes

Supports saturation (signed 16-bit integer domain).

If OPSEL[3] is 0 Result is written to 16 LSBs of destination VGPR and hi 16 bits are written as 0 (this is different from V_MAD_I16).

If OPSEL[3] is 1 Result is written to 16 MSBs of destination VGPR and lo 16 bits are preserved.

V_PERM_B32 493

Permute a 64-bit value constructed from two vector inputs (most significant bits come from the first input) using a per-lane selector from the third input. The lane selector allows each byte of the result to choose from any of the 8 input bytes, perform sign extension or pad with 0/1 bits. Store the result into a vector register.

```
BYTE_PERMUTE = lambda(data, sel) (
    declare in : 8'B[8];
    for i in 0 : 7 do
        in[i] = data[i * 8 + 7 : i * 8].b8
    endfor;
    if sel.u32 >= 13U then
        return 8'0xff
    elsif sel.u32 == 12U then
        return 8'0x0
    elsif sel.u32 == 11U then
        return in[7][7].b8 * 8'0xff
    elsif sel.u32 == 10U then
        return in[5][7].b8 * 8'0xff
    elsif sel.u32 == 9U then
        return in[3][7].b8 * 8'0xff
    elsif sel.u32 == 8U then
        return in[1][7].b8 * 8'0xff
    else
        return in[sel]
    endif);
D0[31 : 24] = BYTE_PERMUTE({ S0.u32, S1.u32 }, S2.u32[31 : 24]);
D0[23 : 16] = BYTE_PERMUTE({ S0.u32, S1.u32 }, S2.u32[23 : 16]);
D0[15 : 8] = BYTE_PERMUTE({ S0.u32, S1.u32 }, S2.u32[15 : 8]);
D0[7 : 0] = BYTE_PERMUTE({ S0.u32, S1.u32 }, S2.u32[7 : 0])
```

Notes

Selects 0 through 7 select the corresponding byte of the 64-bit input value.

Selects 8 through 11 are useful in modeling sign extension of a smaller-precision signed integer to a larger-precision result by replicating the leading bit of a selected byte.

Selects 12 and 13 return padding values of 0 and 1 bits respectively.

Note the MSBs of the 64-bit value being selected are stored in S0. This is counterintuitive for a little-endian architecture.

V_FMA_LEGACY_F16 494

Fused half precision multiply add. Implements IEEE rules and non-standard rule for OPSEL.

```
tmp = fma(S0.f16, S1.f16, S2.f16);
if OPSEL.u4[3] then
    D0 = { tmp.f16, D0[15 : 0] }
else
    D0 = { 16'0, tmp.f16 }
endif
```

V_DIV_FIXUP_LEGACY_F16 495

Half precision division fixup. Has non-standard rule for OPSEL.

S0 = Quotient, S1 = Denominator, S2 = Numerator.

Given a numerator, denominator, and quotient from a divide, this opcode detects and applies specific case numerics, touching up the quotient if necessary. This opcode also generates invalid, denorm and divide by zero exceptions caused by the division.

```
sign_out = (sign(S1.f16) ^ sign(S2.f16));
if isNAN(64'F(S2.f16)) then
    tmp = cvtToQuietNAN(64'F(S2.f16))
elsif isNAN(64'F(S1.f16)) then
    tmp = cvtToQuietNAN(64'F(S1.f16))
elsif ((64'F(S1.f16) == 0.0) && (64'F(S2.f16) == 0.0)) then
    // 0/0
    tmp = 16'F(0xfe00)
elsif ((64'F(abs(S1.f16)) == +INF) && (64'F(abs(S2.f16)) == +INF)) then
    // inf/inf
    tmp = 16'F(0xfe00)
elsif ((64'F(S1.f16) == 0.0) || (64'F(abs(S2.f16)) == +INF)) then
    // x/0, or inf/y
    tmp = sign_out ? -INF : +INF
elsif ((64'F(abs(S1.f16)) == +INF) || (64'F(S2.f16) == 0.0)) then
    // x/inf, 0/y
    tmp = sign_out ? -0.0 : 0.0
else
    tmp = sign_out ? -abs(S0.f16) : abs(S0.f16)
endif;
if OPSEL.u4[3] then
    D0 = { tmp.f16, D0[15 : 0] }
else
    D0 = { 16'0, tmp.f16 }
endif
```

V_CVT_PKACCUM_U8_F32 496

Convert a single-precision float value in the first input to an unsigned 8-bit integer value and store the result into one byte of the destination register using the second input as a byte select.

```
byte = S1.u32[1 : 0];
bit = byte.u32 * 8U;
D0.u32[bit + 7U : bit] = 32'U(f32_to_u8(S0.f32))
```

Notes

This opcode uses src_c to pass destination in as a source.

V_MAD_U32_U16 497

Multiply two unsigned 16-bit integer inputs in the unsigned 32-bit integer domain, add an unsigned 32-bit integer value from a third input, and store the result as an unsigned 32-bit integer into a vector register.

```
D0.u32 = 32'U(S0.u16) * 32'U(S1.u16) + S2.u32
```

V_MAD_I32_I16 498

Multiply two signed 16-bit integer inputs in the signed 32-bit integer domain, add a signed 32-bit integer value from a third input, and store the result as a signed 32-bit integer into a vector register.

```
D0.i32 = 32'I(S0.i16) * 32'I(S1.i16) + S2.i32
```

V_XAD_U32 499

Calculate bitwise XOR of the first two vector inputs, then add the third vector input to the intermediate result, then store the final result into a vector register.

```
D0.u32 = (S0.u32 ^ S1.u32) + S2.u32
```

Notes

No carryin/carryout and no saturation. This opcode is designed to help accelerate the SHA256 hash algorithm.

V_MIN3_F16 500

Select the minimum of three half-precision float inputs and store the selected value into a vector register.

```
D0.f16 = v_min_f16(v_min_f16(S0.f16, S1.f16), S2.f16)
```

V_MIN3_I16 501

Select the minimum of three signed 16-bit integer inputs and store the selected value into a vector register.

```
D0.i16 = v_min_i16(v_min_i16(S0.i16, S1.i16), S2.i16)
```

V_MIN3_U16 502

Select the minimum of three unsigned 16-bit integer inputs and store the selected value into a vector register.

```
D0.u16 = v_min_u16(v_min_u16(S0.u16, S1.u16), S2.u16)
```

V_MAX3_F16 503

Select the maximum of three half-precision float inputs and store the selected value into a vector register.

```
D0.f16 = v_max_f16(v_max_f16(S0.f16, S1.f16), S2.f16)
```

V_MAX3_I16 504

Select the maximum of three signed 16-bit integer inputs and store the selected value into a vector register.

```
D0.i16 = v_max_i16(v_max_i16(S0.i16, S1.i16), S2.i16)
```

V_MAX3_U16 505

Select the maximum of three unsigned 16-bit integer inputs and store the selected value into a vector register.

```
D0.u16 = v_max_u16(v_max_u16(S0.u16, S1.u16), S2.u16)
```

V_MED3_F16 506

Select the median of three half-precision float values and store the selected value into a vector register.

```
if (isNAN(64'F(S0.f16)) || isNAN(64'F(S1.f16)) || isNAN(64'F(S2.f16))) then
    D0.f16 = v_min3_f16(S0.f16, S1.f16, S2.f16)
elsif v_max3_f16(S0.f16, S1.f16, S2.f16) == S0.f16 then
    D0.f16 = v_max_f16(S1.f16, S2.f16)
elsif v_max3_f16(S0.f16, S1.f16, S2.f16) == S1.f16 then
    D0.f16 = v_max_f16(S0.f16, S2.f16)
else
    D0.f16 = v_max_f16(S0.f16, S1.f16)
endif
```

V_MED3_I16 507

Select the median of three signed 16-bit integer values and store the selected value into a vector register.

```
if v_max3_i16(S0.i16, S1.i16, S2.i16) == S0.i16 then
    D0.i16 = v_max_i16(S1.i16, S2.i16)
elsif v_max3_i16(S0.i16, S1.i16, S2.i16) == S1.i16 then
    D0.i16 = v_max_i16(S0.i16, S2.i16)
else
    D0.i16 = v_max_i16(S0.i16, S1.i16)
endif
```

V_MED3_U16 508

Select the median of three unsigned 16-bit integer values and store the selected value into a vector register.

```
if v_max3_u16(S0.u16, S1.u16, S2.u16) == S0.u16 then
    D0.u16 = v_max_u16(S1.u16, S2.u16)
elsif v_max3_u16(S0.u16, S1.u16, S2.u16) == S1.u16 then
    D0.u16 = v_max_u16(S0.u16, S2.u16)
else
    D0.u16 = v_max_u16(S0.u16, S1.u16)
endif
```

V_LSHL_ADD_U32 509

Given a shift count in the second input, calculate the logical shift left of the first input, then add the third input to the intermediate result, then store the final result into a vector register.

```
D0.u32 = (S0.u32 << S1.u32[4 : 0].u32) + S2.u32
```

V_ADD_LSHL_U32 510

Add the first two integer inputs, then given a shift count in the third input, calculate the logical shift left of the intermediate result, then store the final result into a vector register.

```
D0.u32 = ((S0.u32 + S1.u32) << S2.u32[4 : 0].u32)
```

V_ADD3_U32 511

Add three unsigned inputs and store the result into a vector register. No carry-in or carry-out support.

```
D0.u32 = S0.u32 + S1.u32 + S2.u32
```

V_LSHL_OR_B32 512

Given a shift count in the second input, calculate the logical shift left of the first input, then calculate the bitwise OR of the intermediate result and the third input, then store the final result into a vector register.

```
D0.u32 = ((S0.u32 << S1.u32[4 : 0].u32) | S2.u32)
```

V_AND_OR_B32 513

Calculate bitwise AND on the first two vector inputs, then compute the bitwise OR of the intermediate result and the third vector input, then store the final result into a vector register.

```
D0.u32 = ((S0.u32 & S1.u32) | S2.u32)
```

Notes

Input and output modifiers not supported.

V_OR3_B32 514

Calculate the bitwise OR of three vector inputs and store the result into a vector register.

```
D0.u32 = (S0.u32 | S1.u32 | S2.u32)
```

Notes

Input and output modifiers not supported.

V_MAD_F16 515

Multiply two half-precision float inputs and add a third input, and store the result into a vector register.

```
D0.f16 = S0.f16 * S1.f16 + S2.f16
```

Notes

Supports round mode, exception flags, saturation. 1ULP accuracy, denormals are flushed.

If OPSEL[3] is 0 Result is written to 16 LSBs of destination VGPR and hi 16 bits are preserved.

If OPSEL[3] is 1 Result is written to 16 MSBs of destination VGPR and lo 16 bits are preserved.

V_MAD_U16 516

Multiply two unsigned 16-bit integer inputs, add an unsigned 16-bit integer value from a third input, and store the result into a vector register.

```
D0.u16 = S0.u16 * S1.u16 + S2.u16
```

Notes

Supports saturation (unsigned 16-bit integer domain).

If OPSEL[3] is 0 the result is written to 16 LSBs of destination VGPR and the high 16 bits are preserved.

If OPSEL[3] is 1 the result is written to 16 MSBs of destination VGPR and the low 16 bits are preserved.

V_MAD_I16 517

Multiply two signed 16-bit integer inputs, add a signed 16-bit integer value from a third input, and store the result into a vector register.

```
D0.i16 = S0.i16 * S1.i16 + S2.i16
```

Notes

Supports saturation (signed 16-bit integer domain).

If OPSEL[3] is 0 the result is written to 16 LSBs of destination VGPR and the high 16 bits are preserved.

If OPSEL[3] is 1 the result is written to 16 MSBs of destination VGPR and the low 16 bits are preserved.

V_FMA_F16 518

Multiply two half-precision float inputs and add a third input using fused multiply add, and store the result into a vector register.

```
D0.f16 = fma(S0.f16, S1.f16, S2.f16)
```

Notes

0.5ULP accuracy, denormals are supported.

If OPSEL[3] is 0 Result is written to 16 LSBs of destination VGPR and hi 16 bits are preserved.

If OPSEL[3] is 1 Result is written to 16 MSBs of destination VGPR and lo 16 bits are preserved.

V_DIV_FIXUP_F16 519

Given a half-precision float quotient in the first input, a denominator in the second input and a numerator in the third input, detect and apply corner cases related to division, including divide by zero, NaN inputs and overflow, and modify the quotient accordingly. Generate any invalid, denormal and divide-by-zero exceptions that are a result of the division. Store the modified quotient into a vector register.

This operation handles corner cases in a division macro such as divide by zero and NaN inputs. This operation is well defined when the quotient is approximately equal to the numerator divided by the denominator. Other inputs produce a predictable result but may not be mathematically useful.

```
sign_out = (sign(S1.f16) ^ sign(S2.f16));
if isNAN(64'F(S2.f16)) then
    D0.f16 = 16'F(cvtToQuietNAN(64'F(S2.f16)))
elsif isNAN(64'F(S1.f16)) then
    D0.f16 = 16'F(cvtToQuietNAN(64'F(S1.f16)))
elsif ((64'F(S1.f16) == 0.0) && (64'F(S2.f16) == 0.0)) then
    // 0/0
    D0.f16 = 16'F(0xfe00)
elsif ((64'F(abs(S1.f16)) == +INF) && (64'F(abs(S2.f16)) == +INF)) then
    // inf/inf
```

```
    D0.f16 = 16'F(0xfe00)
elsif ((64'F(S1.f16) == 0.0) || (64'F(abs(S2.f16)) == +INF)) then
    // x/0, or inf/y
    D0.f16 = sign_out ? -INF.f16 : +INF.f16
elsif ((64'F(abs(S1.f16)) == +INF) || (64'F(S2.f16) == 0.0)) then
    // x/inf, 0/y
    D0.f16 = sign_out ? -16'0.0 : 16'0.0
else
    D0.f16 = sign_out ? -abs(S0.f16) : abs(S0.f16)
endif
```

Notes

This operation is the final step of a high precision division macro and handles all exceptional cases of division.

If OPSEL[3] is 0 Result is written to 16 LSBs of destination VGPR and hi 16 bits are preserved.

If OPSEL[3] is 1 Result is written to 16 MSBs of destination VGPR and lo 16 bits are preserved.

V_LSHL_ADD_U64 520

Given a shift count in the second input, calculate the logical shift left of the first input, then add the third input to the intermediate result, then store the final result into a vector register.

For this opcode the shift count must be between 0 and 4, higher shift counts are unsupported.

```
D0.u64 = (S0.u64 << S1.u32[2 : 0].u32) + S2.u64
```

Notes

The design treats unsupported shift counts as a shift of zero.

V_ADD_F64 640

Add two floating point inputs and store the result into a vector register.

```
D0.f64 = S0.f64 + S1.f64
```

Notes

0.5ULP precision, denormals are supported.

V_MUL_F64 641

Multiply two floating point inputs and store the result into a vector register.

```
D0.f64 = S0.f64 * S1.f64
```

Notes

0.5ULP precision, denormals are supported.

V_MIN_F64 642

Select the minimum of two double-precision float inputs and store the result into a vector register.

```
if (WAVE_MODE.IEEE && isSignalNAN(S0.f64)) then
    D0.f64 = cvtToQuietNAN(S0.f64)
elsif (WAVE_MODE.IEEE && isSignalNAN(S1.f64)) then
    D0.f64 = cvtToQuietNAN(S1.f64)
elsif isNAN(S0.f64) then
    D0.f64 = S1.f64
elsif isNAN(S1.f64) then
    D0.f64 = S0.f64
elsif ((S0.f64 == +0.0) && (S1.f64 == -0.0)) then
    D0.f64 = S1.f64
elsif ((S0.f64 == -0.0) && (S1.f64 == +0.0)) then
    D0.f64 = S0.f64
else
    // Note: there's no IEEE case here like there is for V_MAX_F64.
    D0.f64 = S0.f64 < S1.f64 ? S0.f64 : S1.f64
endif
```

V_MAX_F64 643

Select the maximum of two double-precision float inputs and store the result into a vector register.

```
if (WAVE_MODE.IEEE && isSignalNAN(S0.f64)) then
    D0.f64 = cvtToQuietNAN(S0.f64)
elsif (WAVE_MODE.IEEE && isSignalNAN(S1.f64)) then
    D0.f64 = cvtToQuietNAN(S1.f64)
elsif isNAN(S0.f64) then
    D0.f64 = S1.f64
elsif isNAN(S1.f64) then
    D0.f64 = S0.f64
elsif ((S0.f64 == +0.0) && (S1.f64 == -0.0)) then
    D0.f64 = S0.f64
elsif ((S0.f64 == -0.0) && (S1.f64 == +0.0)) then
    D0.f64 = S1.f64
elsif WAVE_MODE.IEEE then
    D0.f64 = S0.f64 >= S1.f64 ? S0.f64 : S1.f64
else
```

```
    D0.f64 = S0.f64 > S1.f64 ? S0.f64 : S1.f64
endif
```

V_LDEXP_F64 644

Multiply the first input, a floating point value, by an integral power of 2 specified in the second input, a signed integer value, and store the floating point result into a vector register.

```
D0.f64 = S0.f64 * 2.0 ** S1.i32
```

Notes

Compare with the `ldexp()` function in C.

V_MUL_LO_U32 645

Multiply two unsigned 32-bit integer inputs and store the result into a vector register.

```
D0.u32 = S0.u32 * S1.u32
```

Notes

To multiply integers with small magnitudes consider V_MUL_U32_U24, which is intended to be a more efficient implementation.

V_MUL_HI_U32 646

Multiply two unsigned 32-bit integer inputs and store the high 32 bits of the result into a vector register.

```
D0.u32 = 32'U((64'U(S0.u32) * 64'U(S1.u32)) >> 32U)
```

Notes

To multiply integers with small magnitudes consider V_MUL_HI_U32_U24, which is intended to be a more efficient implementation.

V_MUL_HI_I32 647

Multiply two signed 32-bit integer inputs and store the high 32 bits of the result into a vector register.

```
D0.i32 = 32'I((64'I(S0.i32) * 64'I(S1.i32)) >> 32U)
```

Notes

To multiply integers with small magnitudes consider V_MUL_HI_I32_I24, which is intended to be a more efficient implementation.

V_LDEXP_F32 648

Multiply the first input, a floating point value, by an integral power of 2 specified in the second input, a signed integer value, and store the floating point result into a vector register.

```
D0.f32 = S0.f32 * 2.0F ** S1.i32
```

Notes

Compare with the `ldexp()` function in C.

V_READLANE_B32 649

Read the scalar value in the specified lane of the first input where the lane select is in the second input. Store the result into a scalar register.

```
lane = S1.u32[5 : 0];
// Lane select
D0.b32 = VGPR[lane][SRC0.u32]
```

Notes

Overrides EXEC mask for the VGPR read. Input and output modifiers not supported; this is an untyped operation.

V_WRITELANE_B32 650

Write the scalar value in the first input into the specified lane of a vector register where the lane select is in the second input.

```
lane = S1.u32[5 : 0];
// Lane select
VGPR[lane][VDST.u32] = S0.b32
```

Notes

Overrides EXEC mask for the VGPR write. Input and output modifiers not supported; this is an untyped operation.

V_BCNT_U32_B32 651

Count the number of "1" bits in the vector input and store the result into a vector register.

```
tmp = S1.u32;
for i in 0 : 31 do
    tmp += S0[i].u32;
    // count i'th bit
endfor;
D0.u32 = tmp
```

V_MBCNT_LO_U32_B32 652

For each lane 0 <= *N* < 32, examine the *N* least significant bits of the first input and count how many of those bits are "1". For each lane 32 <= *N* < 64, all "1" bits in the first input are counted. Add this count to the value in the second input and store the result into a vector register.

In conjunction with V_MBCNT_HI_U32_B32 and with a vector condition code as input, this counts the number of lanes at or below the current lane number that have set their vector condition code bit.

```
ThreadMask = (1LL << laneId.u32) - 1LL;
MaskedValue = (S0.u32 & ThreadMask[31 : 0].u32);
tmp = S1.u32;
for i in 0 : 31 do
    tmp += MaskedValue[i] == 1'1U ? 1U : 0U
endfor;
D0.u32 = tmp
```

Notes

See also V_MBCNT_HI_U32_B32.

V_MBCNT_HI_U32_B32 653

For each lane 32 <= *N* < 64, examine the *N* least significant bits of the first input and count how many of those bits are "1". For lane positions 0 <= *N* < 32 *no* bits are examined and the count is zero. Add this count to the value in the second input and store the result into a vector register.

In conjunction with V_MBCNT_LO_U32_B32 and with a vector condition code as input, this counts the number of lanes at or below the current lane number that have set their vector condition code bit.

```
ThreadMask = (1LL << laneId.u32) - 1LL;
MaskedValue = (S0.u32 & ThreadMask[63 : 32].u32);
tmp = S1.u32;
for i in 0 : 31 do
    tmp += MaskedValue[i] == 1'1U ? 1U : 0U
endfor;
D0.u32 = tmp
```

Notes

Example to compute each lane's position in 0..63:

```
    v_mbcnt_lo_u32_b32 v0, -1, 0
    v_mbcnt_hi_u32_b32 v0, -1, v0
    // v0 now contains laneId
```

Example to compute each lane's position in a list of all lanes whose VCC bits are set, where the first lane with VCC set is assigned position 1, the second lane with VCC set is assigned position 2, etc.:

```
    v_mbcnt_lo_u32_b32 v0, vcc_lo, 0
    v_mbcnt_hi_u32_b32 v0, vcc_hi, v0 // Note vcc_hi is passed in for second instruction
    // v0 now contains position among lanes with VCC=1
```

See also V_MBCNT_LO_U32_B32.

V_LSHLREV_B64 655

Given a shift count in the *first* vector input, calculate the logical shift left of the *second* vector input and store the result into a vector register.

```
D0.u64 = (S1.u64 << S0[5 : 0].u32)
```

V_LSHRREV_B64 656

Given a shift count in the *first* vector input, calculate the logical shift right of the *second* vector input and store the result into a vector register.

```
D0.u64 = (S1.u64 >> S0[5 : 0].u32)
```

V_ASHRREV_I64 657

Given a shift count in the *first* vector input, calculate the arithmetic shift right (preserving sign bit) of the *second* vector input and store the result into a vector register.

```
D0.i64 = (S1.i64 >> S0[5 : 0].u32)
```

V_TRIG_PREOP_F64 658

Look up a 53-bit segment of 2/PI using an integer segment select in the second input. Scale the intermediate result by the exponent from the first double-precision float input and store the double-precision float result into a vector register.

This operation returns an aligned, double precision segment of 2/PI needed to do trigonometric argument reduction on the floating point input. Multiple segments can be accessed using the first input. Rounding is toward zero. Large floating point inputs (with an exponent > 1968) are scaled to avoid loss of precision through denormalization.

```
shift = 32'I(S1[4 : 0].u32) * 53;
if exponent(S0.f64) > 1077 then
    shift += exponent(S0.f64) - 1077
endif;
// (2.0/PI) == 0.{b_1200, b_1199, b_1198, ..., b_1, b_0}
// b_1200 is the MSB of the fractional part of 2.0/PI
// Left shift operation indicates which bits are brought
// into the whole part of the number.
// Only whole part of result is kept.
result = 64'F((1201'B(2.0 / PI)[1200 : 0] << shift.u32) & 1201'0x1fffffffffffff);
scale = -53 - shift;
if exponent(S0.f64) >= 1968 then
    scale += 128
endif;
D0.f64 = ldexp(result, scale)
```

Notes

For a more complete treatment of trigonometric argument reduction refer to *Argument Reduction for Huge Arguments: Good to the Last Bit,* K. C. Ng et.al., March 1992, available online.

V_BFM_B32 659

Calculate a bitfield mask given a field offset and size and store the result into a vector register.

```
D0.u32 = (((1U << S0[4 : 0].u32) - 1U) << S1[4 : 0].u32)
```

V_CVT_PKNORM_I16_F32 660

Convert from two single-precision float inputs to a packed signed normalized short and store the result into a vector register.

```
declare tmp : 32'B;
tmp[15 : 0].i16 = f32_to_snorm(S0.f32);
tmp[31 : 16].i16 = f32_to_snorm(S1.f32);
D0 = tmp.b32
```

V_CVT_PKNORM_U16_F32 661

Convert from two single-precision float inputs to a packed unsigned normalized short and store the result into a vector register.

```
declare tmp : 32'B;
tmp[15 : 0].u16 = f32_to_unorm(S0.f32);
tmp[31 : 16].u16 = f32_to_unorm(S1.f32);
D0 = tmp.b32
```

V_CVT_PKRTZ_F16_F32 662

Convert two single-precision float inputs to a packed half-precision float value using round toward zero semantics (ignore the current rounding mode), and store the result into a vector register.

```
prev_mode = ROUND_MODE;
ROUND_MODE = ROUND_TOWARD_ZERO;
tmp[15 : 0].f16 = f32_to_f16(S0.f32);
tmp[31 : 16].f16 = f32_to_f16(S1.f32);
D0 = tmp.b32;
ROUND_MODE = prev_mode;
// Round-toward-zero regardless of current round mode setting in hardware.
```

Notes

This opcode is intended for use with 16-bit compressed exports. See V_CVT_F16_F32 for a version that respects the current rounding mode.

V_CVT_PK_U16_U32 663

Convert from two unsigned 32-bit integer inputs to a packed unsigned 16-bit integer value and store the result into a vector register.

```
declare tmp : 32'B;
tmp[15 : 0].u16 = u32_to_u16(S0.u32);
tmp[31 : 16].u16 = u32_to_u16(S1.u32);
D0 = tmp.b32
```

V_CVT_PK_I16_I32 664

Convert from two signed 32-bit integer inputs to a packed signed 16-bit integer value and store the result into a vector register.

```
declare tmp : 32'B;
tmp[15 : 0].i16 = i32_to_i16(S0.i32);
tmp[31 : 16].i16 = i32_to_i16(S1.i32);
D0 = tmp.b32
```

V_CVT_PKNORM_I16_F16 665

Convert from two half-precision float inputs to a packed signed normalized short and store the result into a vector register.

```
declare tmp : 32'B;
tmp[15 : 0].i16 = f16_to_snorm(S0.f16);
tmp[31 : 16].i16 = f16_to_snorm(S1.f16);
D0 = tmp.b32
```

V_CVT_PKNORM_U16_F16 666

Convert from two half-precision float inputs to a packed unsigned normalized short and store the result into a vector register.

```
declare tmp : 32'B;
tmp[15 : 0].u16 = f16_to_unorm(S0.f16);
tmp[31 : 16].u16 = f16_to_unorm(S1.f16);
D0 = tmp.b32
```

V_ADD_I32 668

Add two signed 32-bit integer inputs and store the result into a vector register. No carry-in or carry-out support.

```
D0.i32 = S0.i32 + S1.i32
```

Notes

Supports saturation (signed 32-bit integer domain).

V_SUB_I32 669

Subtract the second signed input from the first input and store the result into a vector register. No carry-in or carry-out support.

```
D0.i32 = S0.i32 - S1.i32
```

Notes

Supports saturation (signed 32-bit integer domain).

V_ADD_I16 670

Add two signed 16-bit integer inputs and store the result into a vector register. No carry-in or carry-out support.

```
D0.i16 = S0.i16 + S1.i16
```

Notes

Supports saturation (signed 16-bit integer domain).

V_SUB_I16 671

Subtract the second signed input from the first input and store the result into a vector register. No carry-in or carry-out support.

```
D0.i16 = S0.i16 - S1.i16
```

Notes

Supports saturation (signed 16-bit integer domain).

V_PACK_B32_F16 672

Pack two half-precision float values into a single 32-bit value and store the result into a vector register.

```
D0[31 : 16].f16 = S1.f16;
D0[15 : 0].f16 = S0.f16
```

V_MUL_LEGACY_F32 673

Multiply two floating point inputs and store the result in a vector register. Follows DX9 rules where 0.0 times anything produces 0.0 (this differs from other APIs when the other input is infinity or NaN).

```
if ((64'F(S0.f32) == 0.0) || (64'F(S1.f32) == 0.0)) then
    // DX9 rules, 0.0 * x = 0.0
    D0.f32 = 0.0F
else
    D0.f32 = S0.f32 * S1.f32
endif
```

V_CVT_PK_FP8_F32 674

Convert from two single-precision float inputs to a packed FP8 float value with round to nearest even semantics and store the result into 16 bits of a vector register using OPSEL.

```
prev_mode = ROUND_MODE;
ROUND_MODE = ROUND_NEAREST_EVEN;
if OPSEL[3].u32 == 0U then
    VGPR[laneId][VDST.u32][15 : 0].b16 = { f32_to_fp8(S1.f32), f32_to_fp8(S0.f32) };
    // D0[31:16] are preserved
else
    VGPR[laneId][VDST.u32][31 : 16].b16 = { f32_to_fp8(S1.f32), f32_to_fp8(S0.f32) };
    // D0[15:0] are preserved
endif;
ROUND_MODE = prev_mode
```

Notes

Round to nearest even. Ignores OMOD and clamp.

V_CVT_PK_BF8_F32 675

Convert from two single-precision float inputs to a packed BF8 float value with round to nearest even semantics and store the result into 16 bits of a vector register using OPSEL.

```
prev_mode = ROUND_MODE;
```

```
ROUND_MODE = ROUND_NEAREST_EVEN;
if OPSEL[3].u32 == 0U then
    VGPR[laneId][VDST.u32][15 : 0].b16 = { f32_to_bf8(S1.f32), f32_to_bf8(S0.f32) };
    // D0[31:16] are preserved
else
    VGPR[laneId][VDST.u32][31 : 16].b16 = { f32_to_bf8(S1.f32), f32_to_bf8(S0.f32) };
    // D0[15:0] are preserved
endif;
ROUND_MODE = prev_mode
```

Notes

Round to nearest even. Ignores OMOD and clamp.

V_CVT_SR_FP8_F32 676

Convert from a single-precision float input to an FP8 value with stochastic rounding using seed data from the second input. Store the result into 8 bits of a vector register using OPSEL to determine which byte of the destination to overwrite.

```
prev_mode = ROUND_MODE;
ROUND_MODE = ROUND_NEAREST_EVEN;
s = sign(S0.f32);
e = exponent(S0.f32);
m = 23'U(32'U(23'B(mantissa(S0.f32))) + S1[31 : 12].u32);
tmp = float32(s, e, m);
// Add stochastic value to mantissa, wrap around on overflow
if OPSEL[3 : 2].u2 == 2'0U then
    VGPR[laneId][VDST.u32][7 : 0].fp8 = f32_to_fp8(tmp.f32)
elsif OPSEL[3 : 2].u2 == 2'1U then
    VGPR[laneId][VDST.u32][15 : 8].fp8 = f32_to_fp8(tmp.f32)
elsif OPSEL[3 : 2].u2 == 2'2U then
    VGPR[laneId][VDST.u32][23 : 16].fp8 = f32_to_fp8(tmp.f32)
else
    VGPR[laneId][VDST.u32][31 : 24].fp8 = f32_to_fp8(tmp.f32)
endif;
// Unwritten bytes of D are preserved.
ROUND_MODE = prev_mode
```

Notes

Stochastic rounding. Ignores OMOD and clamp.

V_CVT_SR_BF8_F32 677

Convert from a single-precision float input to a BF8 value with stochastic rounding using seed data from the second input. Store the result into 8 bits of a vector register using OPSEL to determine which byte of the destination to overwrite.

```
prev_mode = ROUND_MODE;
ROUND_MODE = ROUND_NEAREST_EVEN;
s = sign(S0.f32);
e = exponent(S0.f32);
m = 23'U(32'U(23'B(mantissa(S0.f32))) + S1[31 : 11].u32);
tmp = float32(s, e, m);
// Add stochastic value to mantissa, wrap around on overflow
if OPSEL[3 : 2].u2 == 2'0U then
    VGPR[laneId][VDST.u32][7 : 0].bf8 = f32_to_bf8(tmp.f32)
elsif OPSEL[3 : 2].u2 == 2'1U then
    VGPR[laneId][VDST.u32][15 : 8].bf8 = f32_to_bf8(tmp.f32)
elsif OPSEL[3 : 2].u2 == 2'2U then
    VGPR[laneId][VDST.u32][23 : 16].bf8 = f32_to_bf8(tmp.f32)
else
    VGPR[laneId][VDST.u32][31 : 24].bf8 = f32_to_bf8(tmp.f32)
endif;
// Unwritten bytes of D are preserved.
ROUND_MODE = prev_mode
```

Notes

Stochastic rounding. Ignores OMOD and clamp.

V_CMP_CLASS_F32 16

Evaluate the IEEE numeric class function specified as a 10 bit mask in the second input on the first input, a single-precision float, and set the vector condition code to the result. Store the result into VCC or a scalar register.

The function reports true if the floating point value is *any* of the numeric types selected in the 10 bit mask according to the following list:

S1.u[0] value is a signaling NAN.
S1.u[1] value is a quiet NAN.
S1.u[2] value is negative infinity.
S1.u[3] value is a negative normal value.
S1.u[4] value is a negative denormal value.
S1.u[5] value is negative zero.
S1.u[6] value is positive zero.
S1.u[7] value is a positive denormal value.
S1.u[8] value is a positive normal value.
S1.u[9] value is positive infinity.

```
declare result : 1'U;
if isSignalNAN(64'F(S0.f32)) then
    result = S1.u32[0]
elsif isQuietNAN(64'F(S0.f32)) then
    result = S1.u32[1]
elsif exponent(S0.f32) == 255 then
    // +-INF
    result = S1.u32[sign(S0.f32) ? 2 : 9]
```

```
elsif exponent(S0.f32) > 0 then
    // +-normal value
    result = S1.u32[sign(S0.f32) ? 3 : 8]
elsif 64'F(abs(S0.f32)) > 0.0 then
    // +-denormal value
    result = S1.u32[sign(S0.f32) ? 4 : 7]
else
    // +-0.0
    result = S1.u32[sign(S0.f32) ? 5 : 6]
endif;
D0.u64[laneId] = result;
// D0 = VCC in VOPC encoding.
```

V_CMPX_CLASS_F32 17

Evaluate the IEEE numeric class function specified as a 10 bit mask in the second input on the first input, a single-precision float, and set the vector condition code to the result. Store the result into the EXEC mask and to VCC or a scalar register.

The function reports true if the floating point value is *any* of the numeric types selected in the 10 bit mask according to the following list:

S1.u[0] value is a signaling NAN.
S1.u[1] value is a quiet NAN.
S1.u[2] value is negative infinity.
S1.u[3] value is a negative normal value.
S1.u[4] value is a negative denormal value.
S1.u[5] value is negative zero.
S1.u[6] value is positive zero.
S1.u[7] value is a positive denormal value.
S1.u[8] value is a positive normal value.
S1.u[9] value is positive infinity.

```
declare result : 1'U;
if isSignalNAN(64'F(S0.f32)) then
    result = S1.u32[0]
elsif isQuietNAN(64'F(S0.f32)) then
    result = S1.u32[1]
elsif exponent(S0.f32) == 255 then
    // +-INF
    result = S1.u32[sign(S0.f32) ? 2 : 9]
elsif exponent(S0.f32) > 0 then
    // +-normal value
    result = S1.u32[sign(S0.f32) ? 3 : 8]
elsif 64'F(abs(S0.f32)) > 0.0 then
    // +-denormal value
    result = S1.u32[sign(S0.f32) ? 4 : 7]
else
    // +-0.0
    result = S1.u32[sign(S0.f32) ? 5 : 6]
endif;
```

```
EXEC.u64[laneId] = D0.u64[laneId] = result
```

V_CMP_CLASS_F64 18

Evaluate the IEEE numeric class function specified as a 10 bit mask in the second input on the first input, a double-precision float, and set the vector condition code to the result. Store the result into VCC or a scalar register.

The function reports true if the floating point value is *any* of the numeric types selected in the 10 bit mask according to the following list:

S1.u[0] value is a signaling NAN.
S1.u[1] value is a quiet NAN.
S1.u[2] value is negative infinity.
S1.u[3] value is a negative normal value.
S1.u[4] value is a negative denormal value.
S1.u[5] value is negative zero.
S1.u[6] value is positive zero.
S1.u[7] value is a positive denormal value.
S1.u[8] value is a positive normal value.
S1.u[9] value is positive infinity.

```
declare result : 1'U;
if isSignalNAN(S0.f64) then
    result = S1.u32[0]
elsif isQuietNAN(S0.f64) then
    result = S1.u32[1]
elsif exponent(S0.f64) == 2047 then
    // +-INF
    result = S1.u32[sign(S0.f64) ? 2 : 9]
elsif exponent(S0.f64) > 0 then
    // +-normal value
    result = S1.u32[sign(S0.f64) ? 3 : 8]
elsif abs(S0.f64) > 0.0 then
    // +-denormal value
    result = S1.u32[sign(S0.f64) ? 4 : 7]
else
    // +-0.0
    result = S1.u32[sign(S0.f64) ? 5 : 6]
endif;
D0.u64[laneId] = result;
// D0 = VCC in VOPC encoding.
```

V_CMPX_CLASS_F64 19

Evaluate the IEEE numeric class function specified as a 10 bit mask in the second input on the first input, a double-precision float, and set the vector condition code to the result. Store the result into the EXEC mask and to VCC or a scalar register.

The function reports true if the floating point value is *any* of the numeric types selected in the 10 bit mask according to the following list:

S1.u[0] value is a signaling NAN.
S1.u[1] value is a quiet NAN.
S1.u[2] value is negative infinity.
S1.u[3] value is a negative normal value.
S1.u[4] value is a negative denormal value.
S1.u[5] value is negative zero.
S1.u[6] value is positive zero.
S1.u[7] value is a positive denormal value.
S1.u[8] value is a positive normal value.
S1.u[9] value is positive infinity.

```
declare result : 1'U;
if isSignalNAN(S0.f64) then
    result = S1.u32[0]
elsif isQuietNAN(S0.f64) then
    result = S1.u32[1]
elsif exponent(S0.f64) == 2047 then
    // +-INF
    result = S1.u32[sign(S0.f64) ? 2 : 9]
elsif exponent(S0.f64) > 0 then
    // +-normal value
    result = S1.u32[sign(S0.f64) ? 3 : 8]
elsif abs(S0.f64) > 0.0 then
    // +-denormal value
    result = S1.u32[sign(S0.f64) ? 4 : 7]
else
    // +-0.0
    result = S1.u32[sign(S0.f64) ? 5 : 6]
endif;
EXEC.u64[laneId] = D0.u64[laneId] = result
```

V_CMP_CLASS_F16 20

Evaluate the IEEE numeric class function specified as a 10 bit mask in the second input on the first input, a half-precision float, and set the vector condition code to the result. Store the result into VCC or a scalar register.

The function reports true if the floating point value is *any* of the numeric types selected in the 10 bit mask according to the following list:

S1.u[0] value is a signaling NAN.
S1.u[1] value is a quiet NAN.
S1.u[2] value is negative infinity.
S1.u[3] value is a negative normal value.
S1.u[4] value is a negative denormal value.
S1.u[5] value is negative zero.
S1.u[6] value is positive zero.

S1.u[7] value is a positive denormal value.
S1.u[8] value is a positive normal value.
S1.u[9] value is positive infinity.

```
declare result : 1'U;
if isSignalNAN(64'F(S0.f16)) then
    result = S1.u32[0]
elsif isQuietNAN(64'F(S0.f16)) then
    result = S1.u32[1]
elsif exponent(S0.f16) == 31 then
    // +-INF
    result = S1.u32[sign(S0.f16) ? 2 : 9]
elsif exponent(S0.f16) > 0 then
    // +-normal value
    result = S1.u32[sign(S0.f16) ? 3 : 8]
elsif 64'F(abs(S0.f16)) > 0.0 then
    // +-denormal value
    result = S1.u32[sign(S0.f16) ? 4 : 7]
else
    // +-0.0
    result = S1.u32[sign(S0.f16) ? 5 : 6]
endif;
D0.u64[laneId] = result;
// D0 = VCC in VOPC encoding.
```

Notes

Note that the S1 has a format of f16 since floating point literal constants are interpreted as 16 bit value for this opcode.

V_CMPX_CLASS_F16 21

Evaluate the IEEE numeric class function specified as a 10 bit mask in the second input on the first input, a half-precision float, and set the vector condition code to the result. Store the result into the EXEC mask and to VCC or a scalar register.

The function reports true if the floating point value is *any* of the numeric types selected in the 10 bit mask according to the following list:

S1.u[0] value is a signaling NAN.
S1.u[1] value is a quiet NAN.
S1.u[2] value is negative infinity.
S1.u[3] value is a negative normal value.
S1.u[4] value is a negative denormal value.
S1.u[5] value is negative zero.
S1.u[6] value is positive zero.
S1.u[7] value is a positive denormal value.
S1.u[8] value is a positive normal value.
S1.u[9] value is positive infinity.

```
declare result : 1'U;
if isSignalNAN(64'F(S0.f16)) then
    result = S1.u32[0]
elsif isQuietNAN(64'F(S0.f16)) then
    result = S1.u32[1]
elsif exponent(S0.f16) == 31 then
    // +-INF
    result = S1.u32[sign(S0.f16) ? 2 : 9]
elsif exponent(S0.f16) > 0 then
    // +-normal value
    result = S1.u32[sign(S0.f16) ? 3 : 8]
elsif 64'F(abs(S0.f16)) > 0.0 then
    // +-denormal value
    result = S1.u32[sign(S0.f16) ? 4 : 7]
else
    // +-0.0
    result = S1.u32[sign(S0.f16) ? 5 : 6]
endif;
EXEC.u64[laneId] = D0.u64[laneId] = result
```

Notes

Note that the S1 has a format of f16 since floating point literal constants are interpreted as 16 bit value for this opcode.

V_CMP_F_F16 32

Set the vector condition code to 0. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = 1'0U;
// D0 = VCC in VOPC encoding.
```

V_CMP_LT_F16 33

Set the vector condition code to 1 iff the first input is less than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.f16 < S1.f16;
// D0 = VCC in VOPC encoding.
```

V_CMP_EQ_F16 34

Set the vector condition code to 1 iff the first input is equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.f16 == S1.f16;
// D0 = VCC in VOPC encoding.
```

V_CMP_LE_F16 35

Set the vector condition code to 1 iff the first input is less than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.f16 <= S1.f16;
// D0 = VCC in VOPC encoding.
```

V_CMP_GT_F16 36

Set the vector condition code to 1 iff the first input is greater than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.f16 > S1.f16;
// D0 = VCC in VOPC encoding.
```

V_CMP_LG_F16 37

Set the vector condition code to 1 iff the first input is less than or greater than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.f16 <> S1.f16;
// D0 = VCC in VOPC encoding.
```

V_CMP_GE_F16 38

Set the vector condition code to 1 iff the first input is greater than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.f16 >= S1.f16;
// D0 = VCC in VOPC encoding.
```

V_CMP_O_F16 39

Set the vector condition code to 1 iff the first input is orderable to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = (!isNAN(64'F(S0.f16)) && !isNAN(64'F(S1.f16)));
// D0 = VCC in VOPC encoding.
```

V_CMP_U_F16 40

Set the vector condition code to 1 iff the first input is not orderable to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = (isNAN(64'F(S0.f16)) || isNAN(64'F(S1.f16)));
// D0 = VCC in VOPC encoding.
```

V_CMP_NGE_F16 41

Set the vector condition code to 1 iff the first input is not greater than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = !(S0.f16 >= S1.f16);
// With NAN inputs this is not the same operation as <
// D0 = VCC in VOPC encoding.
```

V_CMP_NLG_F16 42

Set the vector condition code to 1 iff the first input is not less than or greater than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = !(S0.f16 <> S1.f16);
// With NAN inputs this is not the same operation as ==
// D0 = VCC in VOPC encoding.
```

V_CMP_NGT_F16 43

Set the vector condition code to 1 iff the first input is not greater than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = !(S0.f16 > S1.f16);
// With NAN inputs this is not the same operation as <=
```

```
// D0 = VCC in VOPC encoding.
```

V_CMP_NLE_F16 44

Set the vector condition code to 1 iff the first input is not less than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = !(S0.f16 <= S1.f16);
// With NAN inputs this is not the same operation as >
// D0 = VCC in VOPC encoding.
```

V_CMP_NEQ_F16 45

Set the vector condition code to 1 iff the first input is not equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = !(S0.f16 == S1.f16);
// With NAN inputs this is not the same operation as !=
// D0 = VCC in VOPC encoding.
```

V_CMP_NLT_F16 46

Set the vector condition code to 1 iff the first input is not less than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = !(S0.f16 < S1.f16);
// With NAN inputs this is not the same operation as >=
// D0 = VCC in VOPC encoding.
```

V_CMP_TRU_F16 47

Set the vector condition code to 1. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = 1'1U;
// D0 = VCC in VOPC encoding.
```

V_CMPX_F_F16 48

Set the vector condition code to 0. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = 1'0U;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LT_F16 49

Set the vector condition code to 1 iff the first input is less than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.f16 < S1.f16;
// D0 = VCC in VOPC encoding.
```

V_CMPX_EQ_F16 50

Set the vector condition code to 1 iff the first input is equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.f16 == S1.f16;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LE_F16 51

Set the vector condition code to 1 iff the first input is less than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.f16 <= S1.f16;
// D0 = VCC in VOPC encoding.
```

V_CMPX_GT_F16 52

Set the vector condition code to 1 iff the first input is greater than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.f16 > S1.f16;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LG_F16 53

Set the vector condition code to 1 iff the first input is less than or greater than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.f16 <> S1.f16;
// D0 = VCC in VOPC encoding.
```

V_CMPX_GE_F16 54

Set the vector condition code to 1 iff the first input is greater than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.f16 >= S1.f16;
// D0 = VCC in VOPC encoding.
```

V_CMPX_O_F16 55

Set the vector condition code to 1 iff the first input is orderable to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = (!isNAN(64'F(S0.f16)) && !isNAN(64'F(S1.f16)));
// D0 = VCC in VOPC encoding.
```

V_CMPX_U_F16 56

Set the vector condition code to 1 iff the first input is not orderable to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = (isNAN(64'F(S0.f16)) || isNAN(64'F(S1.f16)));
// D0 = VCC in VOPC encoding.
```

V_CMPX_NGE_F16 57

Set the vector condition code to 1 iff the first input is not greater than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = !(S0.f16 >= S1.f16);
```

```
// With NAN inputs this is not the same operation as <
// D0 = VCC in VOPC encoding.
```

V_CMPX_NLG_F16 58

Set the vector condition code to 1 iff the first input is not less than or greater than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = !(S0.f16 <> S1.f16);
// With NAN inputs this is not the same operation as ==
// D0 = VCC in VOPC encoding.
```

V_CMPX_NGT_F16 59

Set the vector condition code to 1 iff the first input is not greater than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = !(S0.f16 > S1.f16);
// With NAN inputs this is not the same operation as <=
// D0 = VCC in VOPC encoding.
```

V_CMPX_NLE_F16 60

Set the vector condition code to 1 iff the first input is not less than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = !(S0.f16 <= S1.f16);
// With NAN inputs this is not the same operation as >
// D0 = VCC in VOPC encoding.
```

V_CMPX_NEQ_F16 61

Set the vector condition code to 1 iff the first input is not equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = !(S0.f16 == S1.f16);
// With NAN inputs this is not the same operation as !=
// D0 = VCC in VOPC encoding.
```

V_CMPX_NLT_F16 62

Set the vector condition code to 1 iff the first input is not less than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = !(S0.f16 < S1.f16);
// With NAN inputs this is not the same operation as >=
// D0 = VCC in VOPC encoding.
```

V_CMPX_TRU_F16 63

Set the vector condition code to 1. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = 1'1U;
// D0 = VCC in VOPC encoding.
```

V_CMP_F_F32 64

Set the vector condition code to 0. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = 1'0U;
// D0 = VCC in VOPC encoding.
```

V_CMP_LT_F32 65

Set the vector condition code to 1 iff the first input is less than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.f32 < S1.f32;
// D0 = VCC in VOPC encoding.
```

V_CMP_EQ_F32 66

Set the vector condition code to 1 iff the first input is equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.f32 == S1.f32;
```

```
// D0 = VCC in VOPC encoding.
```

V_CMP_LE_F32 67

Set the vector condition code to 1 iff the first input is less than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.f32 <= S1.f32;
// D0 = VCC in VOPC encoding.
```

V_CMP_GT_F32 68

Set the vector condition code to 1 iff the first input is greater than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.f32 > S1.f32;
// D0 = VCC in VOPC encoding.
```

V_CMP_LG_F32 69

Set the vector condition code to 1 iff the first input is less than or greater than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.f32 <> S1.f32;
// D0 = VCC in VOPC encoding.
```

V_CMP_GE_F32 70

Set the vector condition code to 1 iff the first input is greater than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.f32 >= S1.f32;
// D0 = VCC in VOPC encoding.
```

V_CMP_O_F32 71

Set the vector condition code to 1 iff the first input is orderable to the second input. Store the result into VCC or

a scalar register.

```
D0.u64[laneId] = (!isNAN(64'F(S0.f32)) && !isNAN(64'F(S1.f32)));
// D0 = VCC in VOPC encoding.
```

V_CMP_U_F32 72

Set the vector condition code to 1 iff the first input is not orderable to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = (isNAN(64'F(S0.f32)) || isNAN(64'F(S1.f32)));
// D0 = VCC in VOPC encoding.
```

V_CMP_NGE_F32 73

Set the vector condition code to 1 iff the first input is not greater than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = !(S0.f32 >= S1.f32);
// With NAN inputs this is not the same operation as <
// D0 = VCC in VOPC encoding.
```

V_CMP_NLG_F32 74

Set the vector condition code to 1 iff the first input is not less than or greater than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = !(S0.f32 <> S1.f32);
// With NAN inputs this is not the same operation as ==
// D0 = VCC in VOPC encoding.
```

V_CMP_NGT_F32 75

Set the vector condition code to 1 iff the first input is not greater than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = !(S0.f32 > S1.f32);
// With NAN inputs this is not the same operation as <=
```

```
// D0 = VCC in VOPC encoding.
```

V_CMP_NLE_F32 76

Set the vector condition code to 1 iff the first input is not less than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = !(S0.f32 <= S1.f32);
// With NAN inputs this is not the same operation as >
// D0 = VCC in VOPC encoding.
```

V_CMP_NEQ_F32 77

Set the vector condition code to 1 iff the first input is not equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = !(S0.f32 == S1.f32);
// With NAN inputs this is not the same operation as !=
// D0 = VCC in VOPC encoding.
```

V_CMP_NLT_F32 78

Set the vector condition code to 1 iff the first input is not less than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = !(S0.f32 < S1.f32);
// With NAN inputs this is not the same operation as >=
// D0 = VCC in VOPC encoding.
```

V_CMP_TRU_F32 79

Set the vector condition code to 1. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = 1'1U;
// D0 = VCC in VOPC encoding.
```

V_CMPX_F_F32 80

Set the vector condition code to 0. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = 1'0U;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LT_F32 81

Set the vector condition code to 1 iff the first input is less than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.f32 < S1.f32;
// D0 = VCC in VOPC encoding.
```

V_CMPX_EQ_F32 82

Set the vector condition code to 1 iff the first input is equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.f32 == S1.f32;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LE_F32 83

Set the vector condition code to 1 iff the first input is less than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.f32 <= S1.f32;
// D0 = VCC in VOPC encoding.
```

V_CMPX_GT_F32 84

Set the vector condition code to 1 iff the first input is greater than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.f32 > S1.f32;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LG_F32 85

Set the vector condition code to 1 iff the first input is less than or greater than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.f32 <> S1.f32;
// D0 = VCC in VOPC encoding.
```

V_CMPX_GE_F32 86

Set the vector condition code to 1 iff the first input is greater than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.f32 >= S1.f32;
// D0 = VCC in VOPC encoding.
```

V_CMPX_O_F32 87

Set the vector condition code to 1 iff the first input is orderable to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = (!isNAN(64'F(S0.f32)) && !isNAN(64'F(S1.f32)));
// D0 = VCC in VOPC encoding.
```

V_CMPX_U_F32 88

Set the vector condition code to 1 iff the first input is not orderable to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = (isNAN(64'F(S0.f32)) || isNAN(64'F(S1.f32)));
// D0 = VCC in VOPC encoding.
```

V_CMPX_NGE_F32 89

Set the vector condition code to 1 iff the first input is not greater than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = !(S0.f32 >= S1.f32);
```

```
// With NAN inputs this is not the same operation as <
// D0 = VCC in VOPC encoding.
```

V_CMPX_NLG_F32 90

Set the vector condition code to 1 iff the first input is not less than or greater than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = !(S0.f32 <> S1.f32);
// With NAN inputs this is not the same operation as ==
// D0 = VCC in VOPC encoding.
```

V_CMPX_NGT_F32 91

Set the vector condition code to 1 iff the first input is not greater than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = !(S0.f32 > S1.f32);
// With NAN inputs this is not the same operation as <=
// D0 = VCC in VOPC encoding.
```

V_CMPX_NLE_F32 92

Set the vector condition code to 1 iff the first input is not less than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = !(S0.f32 <= S1.f32);
// With NAN inputs this is not the same operation as >
// D0 = VCC in VOPC encoding.
```

V_CMPX_NEQ_F32 93

Set the vector condition code to 1 iff the first input is not equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = !(S0.f32 == S1.f32);
// With NAN inputs this is not the same operation as !=
// D0 = VCC in VOPC encoding.
```

V_CMPX_NLT_F32 94

Set the vector condition code to 1 iff the first input is not less than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = !(S0.f32 < S1.f32);
// With NAN inputs this is not the same operation as >=
// D0 = VCC in VOPC encoding.
```

V_CMPX_TRU_F32 95

Set the vector condition code to 1. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = 1'1U;
// D0 = VCC in VOPC encoding.
```

V_CMP_F_F64 96

Set the vector condition code to 0. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = 1'0U;
// D0 = VCC in VOPC encoding.
```

V_CMP_LT_F64 97

Set the vector condition code to 1 iff the first input is less than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.f64 < S1.f64;
// D0 = VCC in VOPC encoding.
```

V_CMP_EQ_F64 98

Set the vector condition code to 1 iff the first input is equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.f64 == S1.f64;
```

```
// D0 = VCC in VOPC encoding.
```

V_CMP_LE_F64 99

Set the vector condition code to 1 iff the first input is less than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.f64 <= S1.f64;
// D0 = VCC in VOPC encoding.
```

V_CMP_GT_F64 100

Set the vector condition code to 1 iff the first input is greater than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.f64 > S1.f64;
// D0 = VCC in VOPC encoding.
```

V_CMP_LG_F64 101

Set the vector condition code to 1 iff the first input is less than or greater than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.f64 <> S1.f64;
// D0 = VCC in VOPC encoding.
```

V_CMP_GE_F64 102

Set the vector condition code to 1 iff the first input is greater than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.f64 >= S1.f64;
// D0 = VCC in VOPC encoding.
```

V_CMP_O_F64 103

Set the vector condition code to 1 iff the first input is orderable to the second input. Store the result into VCC or

a scalar register.

```
D0.u64[laneId] = (!isNAN(S0.f64) && !isNAN(S1.f64));
// D0 = VCC in VOPC encoding.
```

V_CMP_U_F64 104

Set the vector condition code to 1 iff the first input is not orderable to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = (isNAN(S0.f64) || isNAN(S1.f64));
// D0 = VCC in VOPC encoding.
```

V_CMP_NGE_F64 105

Set the vector condition code to 1 iff the first input is not greater than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = !(S0.f64 >= S1.f64);
// With NAN inputs this is not the same operation as <
// D0 = VCC in VOPC encoding.
```

V_CMP_NLG_F64 106

Set the vector condition code to 1 iff the first input is not less than or greater than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = !(S0.f64 <> S1.f64);
// With NAN inputs this is not the same operation as ==
// D0 = VCC in VOPC encoding.
```

V_CMP_NGT_F64 107

Set the vector condition code to 1 iff the first input is not greater than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = !(S0.f64 > S1.f64);
// With NAN inputs this is not the same operation as <=
```

```
// D0 = VCC in VOPC encoding.
```

V_CMP_NLE_F64 108

Set the vector condition code to 1 iff the first input is not less than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = !(S0.f64 <= S1.f64);
// With NAN inputs this is not the same operation as >
// D0 = VCC in VOPC encoding.
```

V_CMP_NEQ_F64 109

Set the vector condition code to 1 iff the first input is not equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = !(S0.f64 == S1.f64);
// With NAN inputs this is not the same operation as !=
// D0 = VCC in VOPC encoding.
```

V_CMP_NLT_F64 110

Set the vector condition code to 1 iff the first input is not less than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = !(S0.f64 < S1.f64);
// With NAN inputs this is not the same operation as >=
// D0 = VCC in VOPC encoding.
```

V_CMP_TRU_F64 111

Set the vector condition code to 1. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = 1'1U;
// D0 = VCC in VOPC encoding.
```

V_CMPX_F_F64 112

Set the vector condition code to 0. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = 1'0U;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LT_F64 113

Set the vector condition code to 1 iff the first input is less than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.f64 < S1.f64;
// D0 = VCC in VOPC encoding.
```

V_CMPX_EQ_F64 114

Set the vector condition code to 1 iff the first input is equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.f64 == S1.f64;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LE_F64 115

Set the vector condition code to 1 iff the first input is less than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.f64 <= S1.f64;
// D0 = VCC in VOPC encoding.
```

V_CMPX_GT_F64 116

Set the vector condition code to 1 iff the first input is greater than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.f64 > S1.f64;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LG_F64 117

Set the vector condition code to 1 iff the first input is less than or greater than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.f64 <> S1.f64;
// D0 = VCC in VOPC encoding.
```

V_CMPX_GE_F64 118

Set the vector condition code to 1 iff the first input is greater than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.f64 >= S1.f64;
// D0 = VCC in VOPC encoding.
```

V_CMPX_O_F64 119

Set the vector condition code to 1 iff the first input is orderable to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = (!isNAN(S0.f64) && !isNAN(S1.f64));
// D0 = VCC in VOPC encoding.
```

V_CMPX_U_F64 120

Set the vector condition code to 1 iff the first input is not orderable to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = (isNAN(S0.f64) || isNAN(S1.f64));
// D0 = VCC in VOPC encoding.
```

V_CMPX_NGE_F64 121

Set the vector condition code to 1 iff the first input is not greater than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = !(S0.f64 >= S1.f64);
```

```
// With NAN inputs this is not the same operation as <
// D0 = VCC in VOPC encoding.
```

V_CMPX_NLG_F64 122

Set the vector condition code to 1 iff the first input is not less than or greater than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = !(S0.f64 <> S1.f64);
// With NAN inputs this is not the same operation as ==
// D0 = VCC in VOPC encoding.
```

V_CMPX_NGT_F64 123

Set the vector condition code to 1 iff the first input is not greater than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = !(S0.f64 > S1.f64);
// With NAN inputs this is not the same operation as <=
// D0 = VCC in VOPC encoding.
```

V_CMPX_NLE_F64 124

Set the vector condition code to 1 iff the first input is not less than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = !(S0.f64 <= S1.f64);
// With NAN inputs this is not the same operation as >
// D0 = VCC in VOPC encoding.
```

V_CMPX_NEQ_F64 125

Set the vector condition code to 1 iff the first input is not equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = !(S0.f64 == S1.f64);
// With NAN inputs this is not the same operation as !=
// D0 = VCC in VOPC encoding.
```

V_CMPX_NLT_F64 126

Set the vector condition code to 1 iff the first input is not less than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = !(S0.f64 < S1.f64);
// With NAN inputs this is not the same operation as >=
// D0 = VCC in VOPC encoding.
```

V_CMPX_TRU_F64 127

Set the vector condition code to 1. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = 1'1U;
// D0 = VCC in VOPC encoding.
```

V_CMP_F_I16 160

Set the vector condition code to 0. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = 1'0U;
// D0 = VCC in VOPC encoding.
```

V_CMP_LT_I16 161

Set the vector condition code to 1 iff the first input is less than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.i16 < S1.i16;
// D0 = VCC in VOPC encoding.
```

V_CMP_EQ_I16 162

Set the vector condition code to 1 iff the first input is equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.i16 == S1.i16;
```

```
// D0 = VCC in VOPC encoding.
```

V_CMP_LE_I16 163

Set the vector condition code to 1 iff the first input is less than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.i16 <= S1.i16;
// D0 = VCC in VOPC encoding.
```

V_CMP_GT_I16 164

Set the vector condition code to 1 iff the first input is greater than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.i16 > S1.i16;
// D0 = VCC in VOPC encoding.
```

V_CMP_NE_I16 165

Set the vector condition code to 1 iff the first input is not equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.i16 <> S1.i16;
// D0 = VCC in VOPC encoding.
```

V_CMP_GE_I16 166

Set the vector condition code to 1 iff the first input is greater than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.i16 >= S1.i16;
// D0 = VCC in VOPC encoding.
```

V_CMP_T_I16 167

Set the vector condition code to 1. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = 1'1U;
// D0 = VCC in VOPC encoding.
```

V_CMP_F_U16 168

Set the vector condition code to 0. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = 1'0U;
// D0 = VCC in VOPC encoding.
```

V_CMP_LT_U16 169

Set the vector condition code to 1 iff the first input is less than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.u16 < S1.u16;
// D0 = VCC in VOPC encoding.
```

V_CMP_EQ_U16 170

Set the vector condition code to 1 iff the first input is equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.u16 == S1.u16;
// D0 = VCC in VOPC encoding.
```

V_CMP_LE_U16 171

Set the vector condition code to 1 iff the first input is less than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.u16 <= S1.u16;
// D0 = VCC in VOPC encoding.
```

V_CMP_GT_U16 172

Set the vector condition code to 1 iff the first input is greater than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.u16 > S1.u16;
// D0 = VCC in VOPC encoding.
```

V_CMP_NE_U16 173

Set the vector condition code to 1 iff the first input is not equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.u16 <> S1.u16;
// D0 = VCC in VOPC encoding.
```

V_CMP_GE_U16 174

Set the vector condition code to 1 iff the first input is greater than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.u16 >= S1.u16;
// D0 = VCC in VOPC encoding.
```

V_CMP_T_U16 175

Set the vector condition code to 1. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = 1'1U;
// D0 = VCC in VOPC encoding.
```

V_CMPX_F_I16 176

Set the vector condition code to 0. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = 1'0U;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LT_I16 177

Set the vector condition code to 1 iff the first input is less than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.i16 < S1.i16;
// D0 = VCC in VOPC encoding.
```

V_CMPX_EQ_I16 178

Set the vector condition code to 1 iff the first input is equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.i16 == S1.i16;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LE_I16 179

Set the vector condition code to 1 iff the first input is less than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.i16 <= S1.i16;
// D0 = VCC in VOPC encoding.
```

V_CMPX_GT_I16 180

Set the vector condition code to 1 iff the first input is greater than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.i16 > S1.i16;
// D0 = VCC in VOPC encoding.
```

V_CMPX_NE_I16 181

Set the vector condition code to 1 iff the first input is not equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.i16 <> S1.i16;
```

```
// D0 = VCC in VOPC encoding.
```

V_CMPX_GE_I16 182

Set the vector condition code to 1 iff the first input is greater than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.i16 >= S1.i16;
// D0 = VCC in VOPC encoding.
```

V_CMPX_T_I16 183

Set the vector condition code to 1. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = 1'1U;
// D0 = VCC in VOPC encoding.
```

V_CMPX_F_U16 184

Set the vector condition code to 0. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = 1'0U;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LT_U16 185

Set the vector condition code to 1 iff the first input is less than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.u16 < S1.u16;
// D0 = VCC in VOPC encoding.
```

V_CMPX_EQ_U16 186

Set the vector condition code to 1 iff the first input is equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.u16 == S1.u16;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LE_U16 187

Set the vector condition code to 1 iff the first input is less than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.u16 <= S1.u16;
// D0 = VCC in VOPC encoding.
```

V_CMPX_GT_U16 188

Set the vector condition code to 1 iff the first input is greater than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.u16 > S1.u16;
// D0 = VCC in VOPC encoding.
```

V_CMPX_NE_U16 189

Set the vector condition code to 1 iff the first input is not equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.u16 <> S1.u16;
// D0 = VCC in VOPC encoding.
```

V_CMPX_GE_U16 190

Set the vector condition code to 1 iff the first input is greater than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.u16 >= S1.u16;
// D0 = VCC in VOPC encoding.
```

V_CMPX_T_U16 191

Set the vector condition code to 1. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = 1'1U;
// D0 = VCC in VOPC encoding.
```

V_CMP_F_I32 192

Set the vector condition code to 0. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = 1'0U;
// D0 = VCC in VOPC encoding.
```

V_CMP_LT_I32 193

Set the vector condition code to 1 iff the first input is less than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.i32 < S1.i32;
// D0 = VCC in VOPC encoding.
```

V_CMP_EQ_I32 194

Set the vector condition code to 1 iff the first input is equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.i32 == S1.i32;
// D0 = VCC in VOPC encoding.
```

V_CMP_LE_I32 195

Set the vector condition code to 1 iff the first input is less than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.i32 <= S1.i32;
// D0 = VCC in VOPC encoding.
```

V_CMP_GT_I32 196

Set the vector condition code to 1 iff the first input is greater than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.i32 > S1.i32;
// D0 = VCC in VOPC encoding.
```

V_CMP_NE_I32 197

Set the vector condition code to 1 iff the first input is not equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.i32 <> S1.i32;
// D0 = VCC in VOPC encoding.
```

V_CMP_GE_I32 198

Set the vector condition code to 1 iff the first input is greater than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.i32 >= S1.i32;
// D0 = VCC in VOPC encoding.
```

V_CMP_T_I32 199

Set the vector condition code to 1. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = 1'1U;
// D0 = VCC in VOPC encoding.
```

V_CMP_F_U32 200

Set the vector condition code to 0. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = 1'0U;
// D0 = VCC in VOPC encoding.
```

V_CMP_LT_U32 201

Set the vector condition code to 1 iff the first input is less than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.u32 < S1.u32;
// D0 = VCC in VOPC encoding.
```

V_CMP_EQ_U32 202

Set the vector condition code to 1 iff the first input is equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.u32 == S1.u32;
// D0 = VCC in VOPC encoding.
```

V_CMP_LE_U32 203

Set the vector condition code to 1 iff the first input is less than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.u32 <= S1.u32;
// D0 = VCC in VOPC encoding.
```

V_CMP_GT_U32 204

Set the vector condition code to 1 iff the first input is greater than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.u32 > S1.u32;
// D0 = VCC in VOPC encoding.
```

V_CMP_NE_U32 205

Set the vector condition code to 1 iff the first input is not equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.u32 <> S1.u32;
// D0 = VCC in VOPC encoding.
```

V_CMP_GE_U32 206

Set the vector condition code to 1 iff the first input is greater than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.u32 >= S1.u32;
// D0 = VCC in VOPC encoding.
```

V_CMP_T_U32 207

Set the vector condition code to 1. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = 1'1U;
// D0 = VCC in VOPC encoding.
```

V_CMPX_F_I32 208

Set the vector condition code to 0. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = 1'0U;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LT_I32 209

Set the vector condition code to 1 iff the first input is less than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.i32 < S1.i32;
// D0 = VCC in VOPC encoding.
```

V_CMPX_EQ_I32 210

Set the vector condition code to 1 iff the first input is equal to the second input. Store the result into the EXEC

mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.i32 == S1.i32;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LE_I32 211

Set the vector condition code to 1 iff the first input is less than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.i32 <= S1.i32;
// D0 = VCC in VOPC encoding.
```

V_CMPX_GT_I32 212

Set the vector condition code to 1 iff the first input is greater than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.i32 > S1.i32;
// D0 = VCC in VOPC encoding.
```

V_CMPX_NE_I32 213

Set the vector condition code to 1 iff the first input is not equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.i32 <> S1.i32;
// D0 = VCC in VOPC encoding.
```

V_CMPX_GE_I32 214

Set the vector condition code to 1 iff the first input is greater than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.i32 >= S1.i32;
// D0 = VCC in VOPC encoding.
```

V_CMPX_T_I32 215

Set the vector condition code to 1. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = 1'1U;
// D0 = VCC in VOPC encoding.
```

V_CMPX_F_U32 216

Set the vector condition code to 0. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = 1'0U;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LT_U32 217

Set the vector condition code to 1 iff the first input is less than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.u32 < S1.u32;
// D0 = VCC in VOPC encoding.
```

V_CMPX_EQ_U32 218

Set the vector condition code to 1 iff the first input is equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.u32 == S1.u32;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LE_U32 219

Set the vector condition code to 1 iff the first input is less than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.u32 <= S1.u32;
// D0 = VCC in VOPC encoding.
```

V_CMPX_GT_U32 220

Set the vector condition code to 1 iff the first input is greater than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.u32 > S1.u32;
// D0 = VCC in VOPC encoding.
```

V_CMPX_NE_U32 221

Set the vector condition code to 1 iff the first input is not equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.u32 <> S1.u32;
// D0 = VCC in VOPC encoding.
```

V_CMPX_GE_U32 222

Set the vector condition code to 1 iff the first input is greater than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.u32 >= S1.u32;
// D0 = VCC in VOPC encoding.
```

V_CMPX_T_U32 223

Set the vector condition code to 1. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = 1'1U;
// D0 = VCC in VOPC encoding.
```

V_CMP_F_I64 224

Set the vector condition code to 0. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = 1'0U;
// D0 = VCC in VOPC encoding.
```

V_CMP_LT_I64 225

Set the vector condition code to 1 iff the first input is less than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.i64 < S1.i64;
// D0 = VCC in VOPC encoding.
```

V_CMP_EQ_I64 226

Set the vector condition code to 1 iff the first input is equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.i64 == S1.i64;
// D0 = VCC in VOPC encoding.
```

V_CMP_LE_I64 227

Set the vector condition code to 1 iff the first input is less than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.i64 <= S1.i64;
// D0 = VCC in VOPC encoding.
```

V_CMP_GT_I64 228

Set the vector condition code to 1 iff the first input is greater than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.i64 > S1.i64;
// D0 = VCC in VOPC encoding.
```

V_CMP_NE_I64 229

Set the vector condition code to 1 iff the first input is not equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.i64 <> S1.i64;
// D0 = VCC in VOPC encoding.
```

V_CMP_GE_I64 230

Set the vector condition code to 1 iff the first input is greater than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.i64 >= S1.i64;
// D0 = VCC in VOPC encoding.
```

V_CMP_T_I64 231

Set the vector condition code to 1. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = 1'1U;
// D0 = VCC in VOPC encoding.
```

V_CMP_F_U64 232

Set the vector condition code to 0. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = 1'0U;
// D0 = VCC in VOPC encoding.
```

V_CMP_LT_U64 233

Set the vector condition code to 1 iff the first input is less than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.u64 < S1.u64;
// D0 = VCC in VOPC encoding.
```

V_CMP_EQ_U64 234

Set the vector condition code to 1 iff the first input is equal to the second input. Store the result into VCC or a

scalar register.

```
D0.u64[laneId] = S0.u64 == S1.u64;
// D0 = VCC in VOPC encoding.
```

V_CMP_LE_U64 235

Set the vector condition code to 1 iff the first input is less than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.u64 <= S1.u64;
// D0 = VCC in VOPC encoding.
```

V_CMP_GT_U64 236

Set the vector condition code to 1 iff the first input is greater than the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.u64 > S1.u64;
// D0 = VCC in VOPC encoding.
```

V_CMP_NE_U64 237

Set the vector condition code to 1 iff the first input is not equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.u64 <> S1.u64;
// D0 = VCC in VOPC encoding.
```

V_CMP_GE_U64 238

Set the vector condition code to 1 iff the first input is greater than or equal to the second input. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = S0.u64 >= S1.u64;
// D0 = VCC in VOPC encoding.
```

V_CMP_T_U64 239

Set the vector condition code to 1. Store the result into VCC or a scalar register.

```
D0.u64[laneId] = 1'1U;
// D0 = VCC in VOPC encoding.
```

V_CMPX_F_I64 240

Set the vector condition code to 0. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = 1'0U;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LT_I64 241

Set the vector condition code to 1 iff the first input is less than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.i64 < S1.i64;
// D0 = VCC in VOPC encoding.
```

V_CMPX_EQ_I64 242

Set the vector condition code to 1 iff the first input is equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.i64 == S1.i64;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LE_I64 243

Set the vector condition code to 1 iff the first input is less than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.i64 <= S1.i64;
// D0 = VCC in VOPC encoding.
```

V_CMPX_GT_I64 244

Set the vector condition code to 1 iff the first input is greater than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.i64 > S1.i64;
// D0 = VCC in VOPC encoding.
```

V_CMPX_NE_I64 245

Set the vector condition code to 1 iff the first input is not equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.i64 <> S1.i64;
// D0 = VCC in VOPC encoding.
```

V_CMPX_GE_I64 246

Set the vector condition code to 1 iff the first input is greater than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.i64 >= S1.i64;
// D0 = VCC in VOPC encoding.
```

V_CMPX_T_I64 247

Set the vector condition code to 1. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = 1'1U;
// D0 = VCC in VOPC encoding.
```

V_CMPX_F_U64 248

Set the vector condition code to 0. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = 1'0U;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LT_U64 249

Set the vector condition code to 1 iff the first input is less than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.u64 < S1.u64;
// D0 = VCC in VOPC encoding.
```

V_CMPX_EQ_U64 250

Set the vector condition code to 1 iff the first input is equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.u64 == S1.u64;
// D0 = VCC in VOPC encoding.
```

V_CMPX_LE_U64 251

Set the vector condition code to 1 iff the first input is less than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.u64 <= S1.u64;
// D0 = VCC in VOPC encoding.
```

V_CMPX_GT_U64 252

Set the vector condition code to 1 iff the first input is greater than the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.u64 > S1.u64;
// D0 = VCC in VOPC encoding.
```

V_CMPX_NE_U64 253

Set the vector condition code to 1 iff the first input is not equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.u64 <> S1.u64;
// D0 = VCC in VOPC encoding.
```

V_CMPX_GE_U64 254

Set the vector condition code to 1 iff the first input is greater than or equal to the second input. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = S0.u64 >= S1.u64;
// D0 = VCC in VOPC encoding.
```

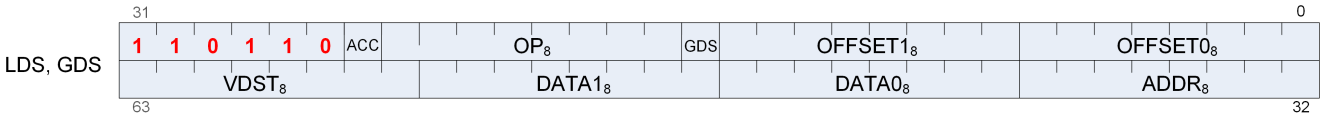
V_CMPX_T_U64 255

Set the vector condition code to 1. Store the result into the EXEC mask and to VCC or a scalar register.

```
EXEC.u64[laneId] = D0.u64[laneId] = 1'1U;
// D0 = VCC in VOPC encoding.
```

12.12. LDS & GWS 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 LDS/GWS is:



GWS instructions operate only on a single lane (the first lane active in the EXEC mask).

```
where:
OFFSET0 = Unsigned byte offset added to the address from the ADDR VGPR.
OFFSET1 = Unsigned byte offset added to the address from the ADDR VGPR.
GDS = Set if GWS, 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.
```

All instructions with RTN in the name return the value that was in memory before the operation was performed.

DS_ADD_U32 0

Add two unsigned 32-bit integer values stored in the data register and a location in a data share.

```
tmp = MEM[ADDR].u32;
MEM[ADDR].u32 += DATA.u32;
RETURN_DATA.u32 = tmp
```

DS_SUB_U32 1

Subtract an unsigned 32-bit integer value stored in the data register from a value stored in a location in a data share.

```
tmp = MEM[ADDR].u32;
MEM[ADDR].u32 -= DATA.u32;
RETURN_DATA.u32 = tmp
```

DS_RSUB_U32 2

Subtract an unsigned 32-bit integer value stored in a location in a data share from a value stored in the data register.

```
tmp = MEM[ADDR].u32;
MEM[ADDR].u32 = DATA.u32 - MEM[ADDR].u32;
RETURN_DATA.u32 = tmp
```

DS_INC_U32 3

Increment an unsigned 32-bit integer value from a location in a data share with wraparound to 0 if the value exceeds a value in the data register.

```
tmp = MEM[ADDR].u32;
src = DATA.u32;
MEM[ADDR].u32 = tmp >= src ? 0U : tmp + 1U;
RETURN_DATA.u32 = tmp
```

DS_DEC_U32 4

Decrement an unsigned 32-bit integer value from a location in a data share with wraparound to a value in the data register if the decrement yields a negative value.

```
tmp = MEM[ADDR].u32;
src = DATA.u32;
MEM[ADDR].u32 = ((tmp == 0U) || (tmp > src)) ? src : tmp - 1U;
RETURN_DATA.u32 = tmp
```

DS_MIN_I32 5

Select the minimum of two signed 32-bit integer inputs, given two values stored in the data register and a location in a data share.

```
tmp = MEM[ADDR].i32;
src = DATA.i32;
MEM[ADDR].i32 = src < tmp ? src : tmp;
RETURN_DATA.i32 = tmp
```

DS_MAX_I32 6

Select the maximum of two signed 32-bit integer inputs, given two values stored in the data register and a

location in a data share.

```
tmp = MEM[ADDR].i32;
src = DATA.i32;
MEM[ADDR].i32 = src >= tmp ? src : tmp;
RETURN_DATA.i32 = tmp
```

DS_MIN_U32 7

Select the minimum of two unsigned 32-bit integer inputs, given two values stored in the data register and a location in a data share.

```
tmp = MEM[ADDR].u32;
src = DATA.u32;
MEM[ADDR].u32 = src < tmp ? src : tmp;
RETURN_DATA.u32 = tmp
```

DS_MAX_U32 8

Select the maximum of two unsigned 32-bit integer inputs, given two values stored in the data register and a location in a data share.

```
tmp = MEM[ADDR].u32;
src = DATA.u32;
MEM[ADDR].u32 = src >= tmp ? src : tmp;
RETURN_DATA.u32 = tmp
```

DS_AND_B32 9

Calculate bitwise AND given two unsigned 32-bit integer values stored in the data register and a location in a data share.

```
tmp = MEM[ADDR].b32;
MEM[ADDR].b32 = (tmp & DATA.b32);
RETURN_DATA.b32 = tmp
```

DS_OR_B32 10

Calculate bitwise OR given two unsigned 32-bit integer values stored in the data register and a location in a data share.

```
tmp = MEM[ADDR].b32;
MEM[ADDR].b32 = (tmp | DATA.b32);
RETURN_DATA.b32 = tmp
```

DS_XOR_B32 11

Calculate bitwise XOR given two unsigned 32-bit integer values stored in the data register and a location in a data share.

```
tmp = MEM[ADDR].b32;
MEM[ADDR].b32 = (tmp ^ DATA.b32);
RETURN_DATA.b32 = tmp
```

DS_MSKOR_B32 12

Calculate masked bitwise OR on an unsigned 32-bit integer location in a data share, given mask value and bits to OR in the data registers.

```
tmp = MEM[ADDR].b32;
MEM[ADDR].b32 = ((tmp & ~DATA.b32) | DATA2.b32);
RETURN_DATA.b32 = tmp
```

DS_WRITE_B32 13

Store 32 bits of data from a vector input register into a data share.

```
MEM[ADDR + OFFSET.u32 * 4U].b32 = DATA[31 : 0]
```

DS_WRITE2_B32 14

Store 32 bits of data from one vector input register and then 32 bits of data from a second vector input register into a data share.

```
MEM[ADDR + OFFSET0.u32 * 4U].b32 = DATA[31 : 0];
MEM[ADDR + OFFSET1.u32 * 4U].b32 = DATA2[31 : 0]
```

DS_WRITE2ST64_B32 15

Store 32 bits of data from one vector input register and then 32 bits of data from a second vector input register into a data share. Treat each offset as an index and multiply by a stride of 64 elements (256 bytes) to generate an offset for each DS address.

```
MEM[ADDR + OFFSET0.u32 * 256U].b32 = DATA[31 : 0];
MEM[ADDR + OFFSET1.u32 * 256U].b32 = DATA2[31 : 0]
```

DS_CMPST_B32 16

Compare an unsigned 32-bit integer value in the data comparison register with a location in a data share, and modify the memory location with a value in the data source register if the comparison is equal.

```
tmp = MEM[ADDR].b32;
src = DATA2.b32;
cmp = DATA.b32;
MEM[ADDR].b32 = tmp == cmp ? src : tmp;
RETURN_DATA.b32 = tmp
```

Notes

Caution, the order of `src` and `cmp` are the opposite of the BUFFER_ATOMIC_CMPSWAP opcode.

DS_CMPST_F32 17

Compare a single-precision float value in the data comparison register with a location in a data share, and modify the memory location with a value in the data source register if the comparison is equal.

```
tmp = MEM[ADDR].f32;
src = DATA2.f32;
cmp = DATA.f32;
MEM[ADDR].f32 = tmp == cmp ? src : tmp;
RETURN_DATA.f32 = tmp
```

Notes

Caution, the order of `src` and `cmp` are the opposite of the BUFFER_ATOMIC_CMPSWAP opcode.

DS_MIN_F32 18

Select the minimum of two single-precision float inputs, given two values stored in the data register and a location in a data share.

```
tmp = MEM[ADDR].f32;
src = DATA.f32;
MEM[ADDR].f32 = src < tmp ? src : tmp;
RETURN_DATA.f32 = tmp
```

Notes

Floating-point compare handles NAN/INF/denorm.

DS_MAX_F32 19

Select the maximum of two single-precision float inputs, given two values stored in the data register and a location in a data share.

```
tmp = MEM[ADDR].f32;
src = DATA.f32;
MEM[ADDR].f32 = src > tmp ? src : tmp;
RETURN_DATA.f32 = tmp
```

Notes

Floating-point compare handles NAN/INF/denorm.

DS_NOP 20

Do nothing.

DS_ADD_F32 21

Add two single-precision float values stored in the data register and a location in a data share.

```
tmp = MEM[ADDR].f32;
MEM[ADDR].f32 += DATA.f32;
RETURN_DATA.f32 = tmp
```

Notes

Floating-point addition handles NAN/INF/denorm.

DS_PK_ADD_F16 23

Add a packed 2-component half-precision float value in the data register to a location in a data share.

```
tmp = MEM[ADDR];
src = DATA;
dst[31 : 16].f16 = tmp[31 : 16].f16 + src[31 : 16].f16;
dst[15 : 0].f16 = tmp[15 : 0].f16 + src[15 : 0].f16;
MEM[ADDR] = dst.b32;
RETURN_DATA = tmp
```

Notes

Floating-point addition handles NAN/INF/denorm.

DS_PK_ADD_BF16 24

Add a packed 2-component BF16 float value in the data register to a location in a data share.

```
tmp = MEM[ADDR];
src = DATA;
dst[31 : 16].bf16 = tmp[31 : 16].bf16 + src[31 : 16].bf16;
dst[15 : 0].bf16 = tmp[15 : 0].bf16 + src[15 : 0].bf16;
MEM[ADDR] = dst.b32;
RETURN_DATA = tmp
```

Notes

Floating-point addition handles NAN/INF/denorm.

DS_WRITE_ADDTID_B32 29

Store 32 bits of data from a vector input register into a data share. The memory base address is provided as an immediate value and the lane ID is used as an offset.

```
declare OFFSET0 : 8'U;
declare OFFSET1 : 8'U;
MEM[32'I({ OFFSET1, OFFSET0 } + M0[15 : 0]) + laneID.i32 * 4].u32 = DATA0.u32
```

DS_WRITE_B8 30

Store 8 bits of data from a vector register into a data share.

```
MEM[ADDR].b8 = DATA[7 : 0]
```

DS_WRITE_B16 31

Store 16 bits of data from a vector register into a data share.

```
MEM[ADDR].b16 = DATA[15 : 0]
```

DS_ADD_RTN_U32 32

Add two unsigned 32-bit integer values stored in the data register and a location in a data share. Store the original value from data share into a vector register.

```
tmp = MEM[ADDR].u32;
MEM[ADDR].u32 += DATA.u32;
RETURN_DATA.u32 = tmp
```

DS_SUB_RTN_U32 33

Subtract an unsigned 32-bit integer value stored in the data register from a value stored in a location in a data share. Store the original value from data share into a vector register.

```
tmp = MEM[ADDR].u32;
MEM[ADDR].u32 -= DATA.u32;
RETURN_DATA.u32 = tmp
```

DS_RSUB_RTN_U32 34

Subtract an unsigned 32-bit integer value stored in a location in a data share from a value stored in the data register. Store the original value from data share into a vector register.

```
tmp = MEM[ADDR].u32;
MEM[ADDR].u32 = DATA.u32 - MEM[ADDR].u32;
RETURN_DATA.u32 = tmp
```

DS_INC_RTN_U32 35

Increment an unsigned 32-bit integer value from a location in a data share with wraparound to 0 if the value exceeds a value in the data register. Store the original value from data share into a vector register.

```
tmp = MEM[ADDR].u32;
src = DATA.u32;
MEM[ADDR].u32 = tmp >= src ? 0U : tmp + 1U;
RETURN_DATA.u32 = tmp
```

DS_DEC_RTN_U32 36

Decrement an unsigned 32-bit integer value from a location in a data share with wraparound to a value in the data register if the decrement yields a negative value. Store the original value from data share into a vector register.

```
tmp = MEM[ADDR].u32;
src = DATA.u32;
MEM[ADDR].u32 = ((tmp == 0U) || (tmp > src)) ? src : tmp - 1U;
RETURN_DATA.u32 = tmp
```

DS_MIN_RTN_I32 37

Select the minimum of two signed 32-bit integer inputs, given two values stored in the data register and a location in a data share. Store the original value from data share into a vector register.

```
tmp = MEM[ADDR].i32;
src = DATA.i32;
MEM[ADDR].i32 = src < tmp ? src : tmp;
RETURN_DATA.i32 = tmp
```

DS_MAX_RTN_I32 38

Select the maximum of two signed 32-bit integer inputs, given two values stored in the data register and a location in a data share. Store the original value from data share into a vector register.

```
tmp = MEM[ADDR].i32;
src = DATA.i32;
MEM[ADDR].i32 = src >= tmp ? src : tmp;
RETURN_DATA.i32 = tmp
```

DS_MIN_RTN_U32 39

Select the minimum of two unsigned 32-bit integer inputs, given two values stored in the data register and a location in a data share. Store the original value from data share into a vector register.

```
tmp = MEM[ADDR].u32;
src = DATA.u32;
MEM[ADDR].u32 = src < tmp ? src : tmp;
RETURN_DATA.u32 = tmp
```

DS_MAX_RTN_U32 40

Select the maximum of two unsigned 32-bit integer inputs, given two values stored in the data register and a location in a data share. Store the original value from data share into a vector register.

```
tmp = MEM[ADDR].u32;
src = DATA.u32;
MEM[ADDR].u32 = src >= tmp ? src : tmp;
RETURN_DATA.u32 = tmp
```

DS_AND_RTN_B32 41

Calculate bitwise AND given two unsigned 32-bit integer values stored in the data register and a location in a data share. Store the original value from data share into a vector register.

```
tmp = MEM[ADDR].b32;
MEM[ADDR].b32 = (tmp & DATA.b32);
RETURN_DATA.b32 = tmp
```

DS_OR_RTN_B32 42

Calculate bitwise OR given two unsigned 32-bit integer values stored in the data register and a location in a data share. Store the original value from data share into a vector register.

```
tmp = MEM[ADDR].b32;
MEM[ADDR].b32 = (tmp | DATA.b32);
RETURN_DATA.b32 = tmp
```

DS_XOR_RTN_B32 43

Calculate bitwise XOR given two unsigned 32-bit integer values stored in the data register and a location in a data share. Store the original value from data share into a vector register.

```
tmp = MEM[ADDR].b32;
```

```
MEM[ADDR].b32 = (tmp ^ DATA.b32);
RETURN_DATA.b32 = tmp
```

DS_MSKOR_RTN_B32 44

Calculate masked bitwise OR on an unsigned 32-bit integer location in a data share, given mask value and bits to OR in the data registers.

```
tmp = MEM[ADDR].b32;
MEM[ADDR].b32 = ((tmp & ~DATA.b32) | DATA2.b32);
RETURN_DATA.b32 = tmp
```

DS_WRXCHG_RTN_B32 45

Swap an unsigned 32-bit integer value in the data register with a location in a data share.

```
tmp = MEM[ADDR].b32;
MEM[ADDR].b32 = DATA.b32;
RETURN_DATA.b32 = tmp
```

DS_WRXCHG2_RTN_B32 46

Swap two unsigned 32-bit integer values in the data registers with two locations in a data share.

```
addr1 = ADDR_BASE.u32 + OFFSET0.u32 * 4U;
addr2 = ADDR_BASE.u32 + OFFSET1.u32 * 4U;
tmp1 = MEM[addr1].b32;
tmp2 = MEM[addr2].b32;
MEM[addr1].b32 = DATA.b32;
MEM[addr2].b32 = DATA2.b32;
// Note DATA2 can be any other register
RETURN_DATA[31 : 0] = tmp1;
RETURN_DATA[63 : 32] = tmp2
```

DS_WRXCHG2ST64_RTN_B32 47

Swap two unsigned 32-bit integer values in the data registers with two locations in a data share. Treat each offset as an index and multiply by a stride of 64 elements (256 bytes) to generate an offset for each DS address.

```
addr1 = ADDR_BASE.u32 + OFFSET0.u32 * 256U;
```

```
addr2 = ADDR_BASE.u32 + OFFSET1.u32 * 256U;
tmp1 = MEM[addr1].b32;
tmp2 = MEM[addr2].b32;
MEM[addr1].b32 = DATA.b32;
MEM[addr2].b32 = DATA2.b32;
// Note DATA2 can be any other register
RETURN_DATA[31 : 0] = tmp1;
RETURN_DATA[63 : 32] = tmp2
```

DS_CMPST_RTN_B32 48

Compare an unsigned 32-bit integer value in the data comparison register with a location in a data share, and modify the memory location with a value in the data source register if the comparison is equal.

```
tmp = MEM[ADDR].b32;
src = DATA2.b32;
cmp = DATA.b32;
MEM[ADDR].b32 = tmp == cmp ? src : tmp;
RETURN_DATA.b32 = tmp
```

Notes

Caution, the order of `src` and `cmp` are the opposite of the BUFFER_ATOMIC_CMPSWAP opcode.

DS_CMPST_RTN_F32 49

Compare a single-precision float value in the data comparison register with a location in a data share, and modify the memory location with a value in the data source register if the comparison is equal.

```
tmp = MEM[ADDR].f32;
src = DATA2.f32;
cmp = DATA.f32;
MEM[ADDR].f32 = tmp == cmp ? src : tmp;
RETURN_DATA.f32 = tmp
```

Notes

Caution, the order of `src` and `cmp` are the opposite of the BUFFER_ATOMIC_CMPSWAP opcode.

DS_MIN_RTN_F32 50

Select the minimum of two single-precision float inputs, given two values stored in the data register and a location in a data share. Store the original value from data share into a vector register.

```
tmp = MEM[ADDR].f32;
src = DATA.f32;
MEM[ADDR].f32 = src < tmp ? src : tmp;
RETURN_DATA.f32 = tmp
```

Notes

Floating-point compare handles NAN/INF/denorm.

DS_MAX_RTN_F32 51

Select the maximum of two single-precision float inputs, given two values stored in the data register and a location in a data share. Store the original value from data share into a vector register.

```
tmp = MEM[ADDR].f32;
src = DATA.f32;
MEM[ADDR].f32 = src > tmp ? src : tmp;
RETURN_DATA.f32 = tmp
```

Notes

Floating-point compare handles NAN/INF/denorm.

DS_WRAP_RTN_B32 52

Given a minuend from a location in data share and a subtrahend from a vector register, subtract the two values *iff* the result is nonnegative; otherwise add a value from a second vector register to the memory location.

This calculation provides flexible wraparound semantics for subtraction.

```
tmp = MEM[ADDR].u32;
MEM[ADDR].u32 = tmp >= DATA.u32 ? tmp - DATA.u32 : tmp + DATA2.u32;
RETURN_DATA = tmp
```

Notes

This instruction is designed to for use in ring buffer management.

DS_ADD_RTN_F32 53

Add two single-precision float values stored in the data register and a location in a data share. Store the original value from data share into a vector register.

```
tmp = MEM[ADDR].f32;
MEM[ADDR].f32 += DATA.f32;
RETURN_DATA.f32 = tmp
```

Notes

Floating-point addition handles NAN/INF/denorm.

DS_READ_B32 54

Load 32 bits of data from a data share into a vector register.

```
RETURN_DATA[31 : 0] = MEM[ADDR + OFFSET.u32 * 4U].b32
```

DS_READ2_B32 55

Load 32 bits of data from one location in a data share and then 32 bits of data from a second location in a data share and store the results into a 64-bit vector register.

```
RETURN_DATA[31 : 0] = MEM[ADDR + OFFSET0.u32 * 4U].b32;
RETURN_DATA[63 : 32] = MEM[ADDR + OFFSET1.u32 * 4U].b32
```

DS_READ2ST64_B32 56

Load 32 bits of data from one location in a data share and then 32 bits of data from a second location in a data share and store the results into a 64-bit vector register. Treat each offset as an index and multiply by a stride of 64 elements (256 bytes) to generate an offset for each DS address.

```
RETURN_DATA[31 : 0] = MEM[ADDR + OFFSET0.u32 * 256U].b32;
RETURN_DATA[63 : 32] = MEM[ADDR + OFFSET1.u32 * 256U].b32
```

DS_READ_I8 57

Load 8 bits of signed data from a data share, sign extend to 32 bits and store the result into a vector register.

```
RETURN_DATA.i32 = 32'I(signext(MEM[ADDR].i8))
```

DS_READ_U8 58

Load 8 bits of unsigned data from a data share, zero extend to 32 bits and store the result into a vector register.

```
RETURN_DATA.u32 = 32'U({ 24'0U, MEM[ADDR].u8 })
```

DS_READ_I16 59

Load 16 bits of signed data from a data share, sign extend to 32 bits and store the result into a vector register.

```
RETURN_DATA.i32 = 32'I(signext(MEM[ADDR].i16))
```

DS_READ_U16 60

Load 16 bits of unsigned data from a data share, zero extend to 32 bits and store the result into a vector register.

```
RETURN_DATA.u32 = 32'U({ 16'0U, MEM[ADDR].u16 })
```

DS_SWIZZLE_B32 61

Dword swizzle, no data is written to LDS memory.

Swizzles input thread data based on offset mask and returns; note does not read or write the DS memory banks.

Note that reading from an invalid thread results in 0x0.

This opcode supports two specific modes, FFT and rotate, plus two basic modes which swizzle in groups of 4 or 32 consecutive threads.

The FFT mode (offset >= 0xe000) swizzles the input based on offset[4:0] to support FFT calculation. Example swizzles using input {1, 2, ... 20} are:

```
Offset[4:0]: Swizzle
0x00: {1,11,9,19,5,15,d,1d,3,13,b,1b,7,17,f,1f,2,12,a,1a,6,16,e,1e,4,14,c,1c,8,18,10,20}
0x10: {1,9,5,d,3,b,7,f,2,a,6,e,4,c,8,10,11,19,15,1d,13,1b,17,1f,12,1a,16,1e,14,1c,18,20}
0x1f: No swizzle
```

The rotate mode (offset >= 0xc000 and offset < 0xe000) rotates the input either left (offset[10] == 0) or right (offset[10] == 1) a number of threads equal to offset[9:5]. The rotate mode also uses a mask value which can alter the rotate result. For example, mask == 1 swaps the odd threads across every other even thread (rotate left), or even threads across every other odd thread (rotate right).

```
Offset[9:5]: Swizzle
0x01, mask=0, rotate left:
{2,3,4,5,6,7,8,9,a,b,c,d,e,f,10,11,12,13,14,15,16,17,18,19,1a,1b,1c,1d,1e,1f,20,1}
0x01, mask=0, rotate right:
{20,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f,10,11,12,13,14,15,16,17,18,19,1a,1b,1c,1d,1e,1f}
0x01, mask=1, rotate left:
{1,4,3,6,5,8,7,a,9,c,b,e,d,10,f,12,11,14,13,16,15,18,17,1a,19,1c,1b,1e,1d,20,1f,2}
0x01, mask=1, rotate right:
{1f,2,1,4,3,6,5,8,7,a,9,c,b,e,d,10,f,12,11,14,13,16,15,18,17,1a,19,1c,1b,1e,1d,20}
```

If offset < 0xc000, one of the basic swizzle modes is used based on offset[15]. If offset[15] == 1, groups of 4 consecutive threads are swizzled together. If offset[15] == 0, all 32 threads are swizzled together.

The first basic swizzle mode (when offset[15] == 1) allows full data sharing between a group of 4 consecutive threads. Any thread within the group of 4 can get data from any other thread within the group of 4, specified by the corresponding offset bits --- [1:0] for the first thread, [3:2] for the second thread, [5:4] for the third thread, [7:6] for the fourth thread. Note that the offset bits apply to all groups of 4 within a wavefront; thus if offset[1:0] == 1, then thread0 grabs thread1, thread4 grabs thread5, etc.

The second basic swizzle mode (when offset[15] == 0) allows limited data sharing between 32 consecutive threads. In this case, the offset is used to specify a 5-bit xor-mask, 5-bit or-mask, and 5-bit and-mask used to generate a thread mapping. Note that the offset bits apply to each group of 32 within a wavefront. The details of the thread mapping are listed below. Some example usages:

SWAPX16 : xor_mask = 0x10, or_mask = 0x00, and_mask = 0x1f

SWAPX8 : xor_mask = 0x08, or_mask = 0x00, and_mask = 0x1f

SWAPX4 : xor_mask = 0x04, or_mask = 0x00, and_mask = 0x1f

SWAPX2 : xor_mask = 0x02, or_mask = 0x00, and_mask = 0x1f

SWAPX1 : xor_mask = 0x01, or_mask = 0x00, and_mask = 0x1f

REVERSEX32 : xor_mask = 0x1f, or_mask = 0x00, and_mask = 0x1f

REVERSEX16 : xor_mask = 0x0f, or_mask = 0x00, and_mask = 0x1f

REVERSEX8 : xor_mask = 0x07, or_mask = 0x00, and_mask = 0x1f

REVERSEX4 : xor_mask = 0x03, or_mask = 0x00, and_mask = 0x1f

REVERSEX2 : xor_mask = 0x01 or_mask = 0x00, and_mask = 0x1f

BCASTX32: xor_mask = 0x00, or_mask = thread, and_mask = 0x00

BCASTX16: xor_mask = 0x00, or_mask = thread, and_mask = 0x10

BCASTX8: xor_mask = 0x00, or_mask = thread, and_mask = 0x18

BCASTX4: xor_mask = 0x00, or_mask = thread, and_mask = 0x1c

BCASTX2: xor_mask = 0x00, or_mask = thread, and_mask = 0x1e

Pseudocode follows:

```
offset = offset1:offset0;

if (offset >= 0xe000) {
    // FFT decomposition
    mask = offset[4:0];
    for (i = 0; i < 64; i++) {
        j = reverse_bits(i & 0x1f);
        j = (j >> count_ones(mask));
        j |= (i & mask);
        j |= i & 0x20;
        thread_out[i] = thread_valid[j] ? thread_in[j] : 0;
    }
```

```
} elsif (offset >= 0xc000) {
    // rotate
    rotate = offset[9:5];
    mask = offset[4:0];
    if (offset[10]) {
        rotate = -rotate;
    }
    for (i = 0; i < 64; i++) {
        j = (i & mask) | ((i + rotate) & ~mask);
        j |= i & 0x20;
        thread_out[i] = thread_valid[j] ? thread_in[j] : 0;
    }
```

```
} elsif (offset[15]) {
    // full data sharing within 4 consecutive threads
    for (i = 0; i < 64; 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;
    }
```

```
} else { // offset[15] == 0
    // limited data sharing within 32 consecutive threads
    xor_mask = offset[14:10];
    or_mask = offset[9:5];
    and_mask = offset[4:0];
    for (i = 0; i < 64; i++) {
        j = (((i & 0x1f) & and_mask) | or_mask) ^ xor_mask;
        j |= (i & 0x20); // which group of 32
        thread_out[i] = thread_valid[j] ? thread_in[j] : 0;
    }
}
```

DS_PERMUTE_B32 62

Forward permute. This does not access LDS memory and may be called even if no LDS memory is allocated to the wave. It uses LDS to implement an arbitrary swizzle across threads in a wavefront.

Note the address passed in is the thread ID multiplied by 4.

If multiple sources map to the same destination lane, standard LDS arbitration rules determine which write wins.

See also DS_BPERMUTE_B32.

```
// VGPR[laneId][index] is the VGPR RAM
// VDST, ADDR and DATA0 are from the microcode DS encoding
declare tmp : 32'B[64];
declare OFFSET : 16'U;
declare DATA0 : 32'U;
declare VDST : 32'U;
for i in 0 : 63 do
    tmp[i] = 0x0
endfor;
for i in 0 : 63 do
    // If a source thread is disabled, it does not propagate data.
    if EXEC[i].u1 then
        // ADDR needs to be divided by 4.
        // High-order bits are ignored.
        dst_lane = (VGPR[i][ADDR].u32 + OFFSET.u32) / 4U % 64U;
        tmp[dst_lane] = VGPR[i][DATA0]
    endif
endfor;
// Copy data into destination VGPRs. If multiple sources
// select the same destination thread, the highest-numbered
// source thread wins.
for i in 0 : 63 do
    if EXEC[i].u1 then
        VGPR[i][VDST] = tmp[i]
    endif
endfor
```

Notes

Examples (simplified 4-thread wavefronts):

VGPR[SRC0] = { A, B, C, D }
VGPR[ADDR] = { 0, 0, 12, 4 }
EXEC = 0xF, OFFSET = 0
VGPR[VDST] = { B, D, 0, C }

VGPR[SRC0] = { A, B, C, D }
VGPR[ADDR] = { 0, 0, 12, 4 }
EXEC = 0xA, OFFSET = 0
VGPR[VDST] = { -, D, -, 0 }

DS_BPERMUTE_B32 63

Backward permute. This does not access LDS memory and may be called even if no LDS memory is allocated to the wave. It uses LDS hardware to implement an arbitrary swizzle across threads in a wavefront.

Note the address passed in is the thread ID multiplied by 4.

Note that EXEC mask is applied to both VGPR read and write. If src_lane selects a disabled thread then zero is returned.

See also DS_PERMUTE_B32.

```
// VGPR[laneId][index] is the VGPR RAM
// VDST, ADDR and DATA0 are from the microcode DS encoding
declare tmp : 32'B[64];
declare OFFSET : 16'U;
declare DATA0 : 32'U;
declare VDST : 32'U;
for i in 0 : 63 do
    tmp[i] = 0x0
endfor;
for i in 0 : 63 do
    // ADDR needs to be divided by 4.
    // High-order bits are ignored.
    src_lane = (VGPR[i][ADDR].u32 + OFFSET.u32) / 4U % 64U;
    // EXEC is applied to the source VGPR reads.
    if EXEC[src_lane].u1 then
        tmp[i] = VGPR[src_lane][DATA0]
    endif
endfor;
// Copy data into destination VGPRs. Some source
// data may be broadcast to multiple lanes.
for i in 0 : 63 do
    if EXEC[i].u1 then
        VGPR[i][VDST] = tmp[i]
    endif
endfor
```

Notes

Examples (simplified 4-thread wavefronts):

VGPR[SRC0] = { A, B, C, D }
VGPR[ADDR] = { 0, 0, 12, 4 }
EXEC = 0xF, OFFSET = 0
VGPR[VDST] = { A, A, D, B }

VGPR[SRC0] = { A, B, C, D }
VGPR[ADDR] = { 0, 0, 12, 4 }
EXEC = 0xA, OFFSET = 0
VGPR[VDST] = { -, 0, -, B }

DS_ADD_U64 64

Add two unsigned 64-bit integer values stored in the data register and a location in a data share.

```
tmp = MEM[ADDR].u64;
MEM[ADDR].u64 += DATA.u64;
RETURN_DATA.u64 = tmp
```

DS_SUB_U64 65

Subtract an unsigned 64-bit integer value stored in the data register from a value stored in a location in a data share.

```
tmp = MEM[ADDR].u64;
MEM[ADDR].u64 -= DATA.u64;
RETURN_DATA.u64 = tmp
```

DS_RSUB_U64 66

Subtract an unsigned 64-bit integer value stored in a location in a data share from a value stored in the data register.

```
tmp = MEM[ADDR].u64;
MEM[ADDR].u64 = DATA.u64 - MEM[ADDR].u64;
RETURN_DATA.u64 = tmp
```

DS_INC_U64 67

Increment an unsigned 64-bit integer value from a location in a data share with wraparound to 0 if the value exceeds a value in the data register.

```
tmp = MEM[ADDR].u64;
src = DATA.u64;
MEM[ADDR].u64 = tmp >= src ? 0ULL : tmp + 1ULL;
RETURN_DATA.u64 = tmp
```

DS_DEC_U64 68

Decrement an unsigned 64-bit integer value from a location in a data share with wraparound to a value in the data register if the decrement yields a negative value.

```
tmp = MEM[ADDR].u64;
src = DATA.u64;
MEM[ADDR].u64 = ((tmp == 0ULL) || (tmp > src)) ? src : tmp - 1ULL;
RETURN_DATA.u64 = tmp
```

DS_MIN_I64 69

Select the minimum of two signed 64-bit integer inputs, given two values stored in the data register and a location in a data share.

```
tmp = MEM[ADDR].i64;
src = DATA.i64;
MEM[ADDR].i64 = src < tmp ? src : tmp;
RETURN_DATA.i64 = tmp
```

DS_MAX_I64 70

Select the maximum of two signed 64-bit integer inputs, given two values stored in the data register and a location in a data share.

```
tmp = MEM[ADDR].i64;
src = DATA.i64;
MEM[ADDR].i64 = src >= tmp ? src : tmp;
RETURN_DATA.i64 = tmp
```

DS_MIN_U64 71

Select the minimum of two unsigned 64-bit integer inputs, given two values stored in the data register and a location in a data share.

```
tmp = MEM[ADDR].u64;
src = DATA.u64;
MEM[ADDR].u64 = src < tmp ? src : tmp;
RETURN_DATA.u64 = tmp
```

DS_MAX_U64 72

Select the maximum of two unsigned 64-bit integer inputs, given two values stored in the data register and a location in a data share.

```
tmp = MEM[ADDR].u64;
src = DATA.u64;
MEM[ADDR].u64 = src >= tmp ? src : tmp;
RETURN_DATA.u64 = tmp
```

DS_AND_B64 73

Calculate bitwise AND given two unsigned 64-bit integer values stored in the data register and a location in a data share.

```
tmp = MEM[ADDR].b64;
MEM[ADDR].b64 = (tmp & DATA.b64);
RETURN_DATA.b64 = tmp
```

DS_OR_B64 74

Calculate bitwise OR given two unsigned 64-bit integer values stored in the data register and a location in a data share.

```
tmp = MEM[ADDR].b64;
MEM[ADDR].b64 = (tmp | DATA.b64);
RETURN_DATA.b64 = tmp
```

DS_XOR_B64 75

Calculate bitwise XOR given two unsigned 64-bit integer values stored in the data register and a location in a data share.

```
tmp = MEM[ADDR].b64;
MEM[ADDR].b64 = (tmp ^ DATA.b64);
RETURN_DATA.b64 = tmp
```

DS_MSKOR_B64 76

Calculate masked bitwise OR on an unsigned 64-bit integer location in a data share, given mask value and bits to OR in the data registers.

```
tmp = MEM[ADDR].b64;
MEM[ADDR].b64 = ((tmp & ~DATA.b64) | DATA2.b64);
RETURN_DATA.b64 = tmp
```

DS_WRITE_B64 77

Store 64 bits of data from a vector input register into a data share.

```
MEM[ADDR + OFFSET.u32 * 8U].b32 = DATA[31 : 0];
MEM[ADDR + OFFSET.u32 * 8U + 4U].b32 = DATA[63 : 32]
```

DS_WRITE2_B64 78

Store 64 bits of data from one vector input register and then 64 bits of data from a second vector input register into a data share.

```
MEM[ADDR + OFFSET0.u32 * 8U].b32 = DATA[31 : 0];
MEM[ADDR + OFFSET0.u32 * 8U + 4U].b32 = DATA[63 : 32];
MEM[ADDR + OFFSET1.u32 * 8U].b32 = DATA2[31 : 0];
MEM[ADDR + OFFSET1.u32 * 8U + 4U].b32 = DATA2[63 : 32]
```

DS_WRITE2ST64_B64 79

Store 64 bits of data from one vector input register and then 64 bits of data from a second vector input register into a data share. Treat each offset as an index and multiply by a stride of 64 elements (256 bytes) to generate an offset for each DS address.

```
MEM[ADDR + OFFSET0.u32 * 512U].b32 = DATA[31 : 0];
MEM[ADDR + OFFSET0.u32 * 512U + 4U].b32 = DATA[63 : 32];
MEM[ADDR + OFFSET1.u32 * 512U].b32 = DATA2[31 : 0];
MEM[ADDR + OFFSET1.u32 * 512U + 4U].b32 = DATA2[63 : 32]
```

DS_CMPST_B64 80

Compare an unsigned 64-bit integer value in the data comparison register with a location in a data share, and modify the memory location with a value in the data source register if the comparison is equal.

```
tmp = MEM[ADDR].b64;
src = DATA2.b64;
```

```
cmp = DATA.b64;
MEM[ADDR].b64 = tmp == cmp ? src : tmp;
RETURN_DATA.b64 = tmp
```

Notes

Caution, the order of `src` and `cmp` are the opposite of the BUFFER_ATOMIC_CMPSWAP opcode.

DS_CMPST_F64 81

Compare a double-precision float value in the data comparison register with a location in a data share, and modify the memory location with a value in the data source register if the comparison is equal.

```
tmp = MEM[ADDR].f64;
src = DATA2.f64;
cmp = DATA.f64;
MEM[ADDR].f64 = tmp == cmp ? src : tmp;
RETURN_DATA.f64 = tmp
```

Notes

Caution, the order of `src` and `cmp` are the opposite of the BUFFER_ATOMIC_CMPSWAP opcode.

DS_MIN_F64 82

Select the minimum of two double-precision float inputs, given two values stored in the data register and a location in a data share.

```
tmp = MEM[ADDR].f64;
src = DATA.f64;
MEM[ADDR].f64 = src < tmp ? src : tmp;
RETURN_DATA.f64 = tmp
```

Notes

Floating-point compare handles NAN/INF/denorm.

DS_MAX_F64 83

Select the maximum of two double-precision float inputs, given two values stored in the data register and a location in a data share.

```
tmp = MEM[ADDR].f64;
src = DATA.f64;
```

```
MEM[ADDR].f64 = src > tmp ? src : tmp;
RETURN_DATA.f64 = tmp
```

Notes

Floating-point compare handles NAN/INF/denorm.

DS_WRITE_B8_D16_HI 84

Store 8 bits of data from the high bits of a vector register into a data share.

```
MEM[ADDR].b8 = DATA[23 : 16]
```

DS_WRITE_B16_D16_HI 85

Store 16 bits of data from the high bits of a vector register into a data share.

```
MEM[ADDR].b16 = DATA[31 : 16]
```

DS_READ_U8_D16 86

Load 8 bits of unsigned data from a data share, zero extend to 16 bits and store the result into the low 16 bits of a vector register.

```
RETURN_DATA[15 : 0].u16 = 16'U({ 8'0U, MEM[ADDR].u8 });
// RETURN_DATA[31:16] is preserved.
```

DS_READ_U8_D16_HI 87

Load 8 bits of unsigned data from a data share, zero extend to 16 bits and store the result into the high 16 bits of a vector register.

```
RETURN_DATA[31 : 16].u16 = 16'U({ 8'0U, MEM[ADDR].u8 });
// RETURN_DATA[15:0] is preserved.
```

DS_READ_I8_D16 88

Load 8 bits of signed data from a data share, sign extend to 16 bits and store the result into the low 16 bits of a vector register.

```
RETURN_DATA[15 : 0].i16 = 16'I(signext(MEM[ADDR].i8));
// RETURN_DATA[31:16] is preserved.
```

DS_READ_I8_D16_HI 89

Load 8 bits of signed data from a data share, sign extend to 16 bits and store the result into the high 16 bits of a vector register.

```
RETURN_DATA[31 : 16].i16 = 16'I(signext(MEM[ADDR].i8));
// RETURN_DATA[15:0] is preserved.
```

DS_READ_U16_D16 90

Load 16 bits of unsigned data from a data share and store the result into the low 16 bits of a vector register.

```
RETURN_DATA[15 : 0].u16 = MEM[ADDR].u16;
// RETURN_DATA[31:16] is preserved.
```

DS_READ_U16_D16_HI 91

Load 16 bits of unsigned data from a data share and store the result into the high 16 bits of a vector register.

```
RETURN_DATA[31 : 16].u16 = MEM[ADDR].u16;
// RETURN_DATA[15:0] is preserved.
```

DS_ADD_F64 92

Add a double-precision float value in the data register to a location in a data share.

```
tmp = MEM[ADDR].f64;
MEM[ADDR].f64 += DATA.f64;
RETURN_DATA = tmp
```

Notes

Floating-point addition handles NAN/INF/denorm.

DS_ADD_RTN_U64 96

Add two unsigned 64-bit integer values stored in the data register and a location in a data share. Store the original value from data share into a vector register.

```
tmp = MEM[ADDR].u64;
MEM[ADDR].u64 += DATA.u64;
RETURN_DATA.u64 = tmp
```

DS_SUB_RTN_U64 97

Subtract an unsigned 64-bit integer value stored in the data register from a value stored in a location in a data share. Store the original value from data share into a vector register.

```
tmp = MEM[ADDR].u64;
MEM[ADDR].u64 -= DATA.u64;
RETURN_DATA.u64 = tmp
```

DS_RSUB_RTN_U64 98

Subtract an unsigned 64-bit integer value stored in a location in a data share from a value stored in the data register. Store the original value from data share into a vector register.

```
tmp = MEM[ADDR].u64;
MEM[ADDR].u64 = DATA.u64 - MEM[ADDR].u64;
RETURN_DATA.u64 = tmp
```

DS_INC_RTN_U64 99

Increment an unsigned 64-bit integer value from a location in a data share with wraparound to 0 if the value exceeds a value in the data register. Store the original value from data share into a vector register.

```
tmp = MEM[ADDR].u64;
src = DATA.u64;
MEM[ADDR].u64 = tmp >= src ? 0ULL : tmp + 1ULL;
RETURN_DATA.u64 = tmp
```

DS_DEC_RTN_U64 100

Decrement an unsigned 64-bit integer value from a location in a data share with wraparound to a value in the data register if the decrement yields a negative value. Store the original value from data share into a vector register.

```
tmp = MEM[ADDR].u64;
src = DATA.u64;
MEM[ADDR].u64 = ((tmp == 0ULL) || (tmp > src)) ? src : tmp - 1ULL;
RETURN_DATA.u64 = tmp
```

DS_MIN_RTN_I64 101

Select the minimum of two signed 64-bit integer inputs, given two values stored in the data register and a location in a data share. Store the original value from data share into a vector register.

```
tmp = MEM[ADDR].i64;
src = DATA.i64;
MEM[ADDR].i64 = src < tmp ? src : tmp;
RETURN_DATA.i64 = tmp
```

DS_MAX_RTN_I64 102

Select the maximum of two signed 64-bit integer inputs, given two values stored in the data register and a location in a data share. Store the original value from data share into a vector register.

```
tmp = MEM[ADDR].i64;
src = DATA.i64;
MEM[ADDR].i64 = src >= tmp ? src : tmp;
RETURN_DATA.i64 = tmp
```

DS_MIN_RTN_U64 103

Select the minimum of two unsigned 64-bit integer inputs, given two values stored in the data register and a location in a data share. Store the original value from data share into a vector register.

```
tmp = MEM[ADDR].u64;
src = DATA.u64;
MEM[ADDR].u64 = src < tmp ? src : tmp;
RETURN_DATA.u64 = tmp
```

DS_MAX_RTN_U64 104

Select the maximum of two unsigned 64-bit integer inputs, given two values stored in the data register and a location in a data share. Store the original value from data share into a vector register.

```
tmp = MEM[ADDR].u64;
src = DATA.u64;
MEM[ADDR].u64 = src >= tmp ? src : tmp;
RETURN_DATA.u64 = tmp
```

DS_AND_RTN_B64 105

Calculate bitwise AND given two unsigned 64-bit integer values stored in the data register and a location in a data share. Store the original value from data share into a vector register.

```
tmp = MEM[ADDR].b64;
MEM[ADDR].b64 = (tmp & DATA.b64);
RETURN_DATA.b64 = tmp
```

DS_OR_RTN_B64 106

Calculate bitwise OR given two unsigned 64-bit integer values stored in the data register and a location in a data share. Store the original value from data share into a vector register.

```
tmp = MEM[ADDR].b64;
MEM[ADDR].b64 = (tmp | DATA.b64);
RETURN_DATA.b64 = tmp
```

DS_XOR_RTN_B64 107

Calculate bitwise XOR given two unsigned 64-bit integer values stored in the data register and a location in a data share. Store the original value from data share into a vector register.

```
tmp = MEM[ADDR].b64;
MEM[ADDR].b64 = (tmp ^ DATA.b64);
RETURN_DATA.b64 = tmp
```

DS_MSKOR_RTN_B64 108

Calculate masked bitwise OR on an unsigned 64-bit integer location in a data share, given mask value and bits to OR in the data registers.

```
tmp = MEM[ADDR].b64;
MEM[ADDR].b64 = ((tmp & ~DATA.b64) | DATA2.b64);
RETURN_DATA.b64 = tmp
```

DS_WRXCHG_RTN_B64 109

Swap an unsigned 64-bit integer value in the data register with a location in a data share.

```
tmp = MEM[ADDR].b64;
MEM[ADDR].b64 = DATA.b64;
RETURN_DATA.b64 = tmp
```

DS_WRXCHG2_RTN_B64 110

Swap two unsigned 64-bit integer values in the data registers with two locations in a data share.

```
addr1 = ADDR_BASE.u32 + OFFSET0.u32 * 8U;
addr2 = ADDR_BASE.u32 + OFFSET1.u32 * 8U;
tmp1 = MEM[addr1].b64;
tmp2 = MEM[addr2].b64;
MEM[addr1].b64 = DATA.b64;
MEM[addr2].b64 = DATA2.b64;
// Note DATA2 can be any other register
RETURN_DATA[63 : 0] = tmp1;
RETURN_DATA[127 : 64] = tmp2
```

DS_WRXCHG2ST64_RTN_B64 111

Swap two unsigned 64-bit integer values in the data registers with two locations in a data share. Treat each offset as an index and multiply by a stride of 64 elements (256 bytes) to generate an offset for each DS address.

```
addr1 = ADDR_BASE.u32 + OFFSET0.u32 * 512U;
addr2 = ADDR_BASE.u32 + OFFSET1.u32 * 512U;
tmp1 = MEM[addr1].b64;
tmp2 = MEM[addr2].b64;
MEM[addr1].b64 = DATA.b64;
MEM[addr2].b64 = DATA2.b64;
// Note DATA2 can be any other register
RETURN_DATA[63 : 0] = tmp1;
RETURN_DATA[127 : 64] = tmp2
```

DS_CMPST_RTN_B64 112

Compare an unsigned 64-bit integer value in the data comparison register with a location in a data share, and modify the memory location with a value in the data source register if the comparison is equal.

```
tmp = MEM[ADDR].b64;
src = DATA2.b64;
cmp = DATA.b64;
MEM[ADDR].b64 = tmp == cmp ? src : tmp;
RETURN_DATA.b64 = tmp
```

Notes

Caution, the order of `src` and `cmp` are the opposite of the BUFFER_ATOMIC_CMPSWAP opcode.

DS_CMPST_RTN_F64 113

Compare a double-precision float value in the data comparison register with a location in a data share, and modify the memory location with a value in the data source register if the comparison is equal.

```
tmp = MEM[ADDR].f64;
src = DATA2.f64;
cmp = DATA.f64;
MEM[ADDR].f64 = tmp == cmp ? src : tmp;
RETURN_DATA.f64 = tmp
```

Notes

Caution, the order of `src` and `cmp` are the opposite of the BUFFER_ATOMIC_CMPSWAP opcode.

DS_MIN_RTN_F64 114

Select the minimum of two double-precision float inputs, given two values stored in the data register and a location in a data share. Store the original value from data share into a vector register.

```
tmp = MEM[ADDR].f64;
src = DATA.f64;
MEM[ADDR].f64 = src < tmp ? src : tmp;
RETURN_DATA.f64 = tmp
```

Notes

Floating-point compare handles NAN/INF/denorm.

DS_MAX_RTN_F64 115

Select the maximum of two double-precision float inputs, given two values stored in the data register and a location in a data share. Store the original value from data share into a vector register.

```
tmp = MEM[ADDR].f64;
src = DATA.f64;
MEM[ADDR].f64 = src > tmp ? src : tmp;
RETURN_DATA.f64 = tmp
```

Notes

Floating-point compare handles NAN/INF/denorm.

DS_READ_B64 118

Load 64 bits of data from a data share into a vector register.

```
RETURN_DATA[31 : 0] = MEM[ADDR + OFFSET.u32 * 8U].b32;
RETURN_DATA[63 : 32] = MEM[ADDR + OFFSET.u32 * 8U + 4U].b32
```

DS_READ2_B64 119

Load 64 bits of data from one location in a data share and then 64 bits of data from a second location in a data share and store the results into a 128-bit vector register.

```
RETURN_DATA[31 : 0] = MEM[ADDR + OFFSET0.u32 * 8U].b32;
RETURN_DATA[63 : 32] = MEM[ADDR + OFFSET0.u32 * 8U + 4U].b32;
RETURN_DATA[95 : 64] = MEM[ADDR + OFFSET1.u32 * 8U].b32;
RETURN_DATA[127 : 96] = MEM[ADDR + OFFSET1.u32 * 8U + 4U].b32
```

DS_READ2ST64_B64 120

Load 64 bits of data from one location in a data share and then 64 bits of data from a second location in a data share and store the results into a 128-bit vector register. Treat each offset as an index and multiply by a stride of 64 elements (256 bytes) to generate an offset for each DS address.

```
RETURN_DATA[31 : 0] = MEM[ADDR + OFFSET0.u32 * 512U].b32;
RETURN_DATA[63 : 32] = MEM[ADDR + OFFSET0.u32 * 512U + 4U].b32;
RETURN_DATA[95 : 64] = MEM[ADDR + OFFSET1.u32 * 512U].b32;
RETURN_DATA[127 : 96] = MEM[ADDR + OFFSET1.u32 * 512U + 4U].b32
```

DS_ADD_RTN_F64 124

Add a double-precision float value in the data register to a location in a data share. Store the original value from data share into a vector register.

```
tmp = MEM[ADDR].f64;
MEM[ADDR].f64 += DATA.f64;
RETURN_DATA = tmp
```

Notes

Floating-point addition handles NAN/INF/denorm.

DS_CONDXCHG32_RTN_B64 126

Perform 2 conditional write exchanges, where each conditional write exchange writes a 32 bit value from a data register to a location in data share iff the most significant bit of the data value is set.

```
declare OFFSET0 : 8'U;
declare OFFSET1 : 8'U;
declare RETURN_DATA : 32'U[2];
ADDR = S0.u32;
DATA = S1.u64;
offset = { OFFSET1, OFFSET0 };
ADDR0 = ((ADDR + offset.u32) & 0xfff8U);
ADDR1 = ADDR0 + 4U;
RETURN_DATA[0] = LDS[ADDR0].u32;
if DATA[31] then
    LDS[ADDR0] = { 1'0, DATA[30 : 0] }
endif;
RETURN_DATA[1] = LDS[ADDR1].u32;
if DATA[63] then
    LDS[ADDR1] = { 1'0, DATA[62 : 32] }
endif
```

DS_GWS_SEMA_RELEASE_ALL 152

GDS Only: The GWS resource (rid) indicated processes this opcode by updating the counter and labeling the specified resource as a semaphore.

```
// Determine the GWS resource to work on
rid[5:0] = gds_base[5:0] + offset0[5:0];

// Incr the state counter of the resource
state.counter[rid] = state.wave_in_queue;
```

```
state.type = SEMAPHORE;
return rd_done; //release calling wave
```

This action releases ALL queued waves; it has no effect if no waves are present.

DS_GWS_INIT 153

GDS Only: Initialize a barrier or semaphore resource.

```
// Determine the GWS resource to work on
rid[5:0] = gds_base[5:0] + offset0[5:0];

// Get the value to use in init
index = find_first_valid(vector mask)
value = DATA[thread: index]

// Set the state of the resource
state.counter[rid] = lsb(value); //limit #waves
state.flag[rid] = 0;
return rd_done; //release calling wave
```

CAUTION: The VGPR operand MUST be even-aligned for this instruction. Only 32 bits are used but hardware treats this instruction as a 64 bit read.

DS_GWS_SEMA_V 154

GDS Only: The GWS resource indicated processes this opcode by updating the counter and labeling the resource as a semaphore.

```
//Determine the GWS resource to work on
rid[5:0] = gds_base[5:0] + offset0[5:0];

//Incr the state counter of the resource
state.counter[rid] += 1;
state.type = SEMAPHORE;
return rd_done; //release calling wave
```

This action releases one wave if any are queued in this resource.

DS_GWS_SEMA_BR 155

GDS Only: The GWS resource indicated processes this opcode by updating the counter by the bulk release delivered count and labeling the resource as a semaphore.

```
//Determine the GWS resource to work on
```

```
rid[5:0] = gds_base[5:0] + offset0[5:0];
index =  find first valid (vector mask)
count = DATA[thread: index];

//Add count to the resource state counter
state.counter[rid] += count;
state.type = SEMAPHORE;
return rd_done; //release calling wave
```

This action releases count number of waves, promptly if queued, or as they arrive from the noted resource.

CAUTION: The VGPR operand MUST be even-aligned for this instruction. Only 32 bits are used but hardware treats this instruction as a 64 bit read.

DS_GWS_SEMA_P 156

GDS Only: The GWS resource indicated processes this opcode by queueing it until counter enables a release and then decrementing the counter of the resource as a semaphore.

```
//Determine the GWS resource to work on
rid[5:0] = gds_base[5:0] + offset0[5:0];
state.type = SEMAPHORE;
ENQUEUE until(state[rid].counter > 0)
state[rid].counter -= 1;
return rd_done;
```

DS_GWS_BARRIER 157

GDS Only: The GWS resource indicated processes this opcode by queueing it until barrier is satisfied. The number of waves needed is passed in as DATA of first valid thread.

```
//Determine the GWS resource to work on
rid[5:0] = gds_base[5:0] + OFFSET0[5:0];
index =  find first valid (vector mask);
value = DATA[thread: index];

// Input Decision Machine
state.type[rid] = BARRIER;
if(state[rid].counter <= 0) then
    thread[rid].flag = state[rid].flag;
    ENQUEUE;
    state[rid].flag = !state.flag;
    state[rid].counter = value;
    return rd_done;
else
    state[rid].counter -= 1;
    thread.flag = state[rid].flag;
    ENQUEUE;
```

```
endif.
```

Since the waves deliver the count for the next barrier, this function can have a different size barrier for each occurrence.

```
// Release Machine
if(state.type == BARRIER) then
    if(state.flag != thread.flag) then
        return rd_done;
    endif;
endif.
```

CAUTION: The VGPR operand MUST be even-aligned for this instruction. Only 32 bits are used but hardware treats this instruction as a 64 bit read.

DS_READ_ADDTID_B32 182

Load 32 bits of data from a data share into a vector register. The memory base address is provided as an immediate value and the lane ID is used as an offset.

```
declare OFFSET0 : 8'U;
declare OFFSET1 : 8'U;
RETURN_DATA.u32 = MEM[32'I({ OFFSET1, OFFSET0 } + M0[15 : 0]) + laneID.i32 * 4].u32
```

DS_PK_ADD_RTN_F16 183

Add a packed 2-component half-precision float value in the data register to a location in a data share. Store the original value from data share into a vector register.

```
tmp = MEM[ADDR];
src = DATA;
dst[31 : 16].f16 = tmp[31 : 16].f16 + src[31 : 16].f16;
dst[15 : 0].f16 = tmp[15 : 0].f16 + src[15 : 0].f16;
MEM[ADDR] = dst.b32;
RETURN_DATA = tmp
```

Notes

Floating-point addition handles NAN/INF/denorm.

DS_PK_ADD_RTN_BF16 184

Add a packed 2-component BF16 float value in the data register to a location in a data share. Store the original

value from data share into a vector register.

```
tmp = MEM[ADDR];
src = DATA;
dst[31 : 16].bf16 = tmp[31 : 16].bf16 + src[31 : 16].bf16;
dst[15 : 0].bf16 = tmp[15 : 0].bf16 + src[15 : 0].bf16;
MEM[ADDR] = dst.b32;
RETURN_DATA = tmp
```

Notes

Floating-point addition handles NAN/INF/denorm.

DS_CONSUME **189**

LDS & GDS. Subtract (count_bits(exec_mask)) from the value stored in DS memory at (M0.base + instr_offset). Return the pre-operation value to VGPRs.

DS_APPEND **190**

LDS & GDS. Add (count_bits(exec_mask)) to the value stored in DS memory at (M0.base + instr_offset). Return the pre-operation value to VGPRs.

DS_WRITE_B96 **222**

Store 96 bits of data from a vector input register into a data share.

```
MEM[ADDR + OFFSET.u32 * 12U].b32 = DATA[31 : 0];
MEM[ADDR + OFFSET.u32 * 12U + 4U].b32 = DATA[63 : 32];
MEM[ADDR + OFFSET.u32 * 12U + 8U].b32 = DATA[95 : 64]
```

DS_WRITE_B128 **223**

Store 128 bits of data from a vector input register into a data share.

```
MEM[ADDR + OFFSET.u32 * 16U].b32 = DATA[31 : 0];
MEM[ADDR + OFFSET.u32 * 16U + 4U].b32 = DATA[63 : 32];
MEM[ADDR + OFFSET.u32 * 16U + 8U].b32 = DATA[95 : 64];
MEM[ADDR + OFFSET.u32 * 16U + 12U].b32 = DATA[127 : 96]
```

DS_READ_B96 254

Load 96 bits of data from a data share into a vector register.

```
RETURN_DATA[31 : 0] = MEM[ADDR + OFFSET.u32 * 12U].b32;
RETURN_DATA[63 : 32] = MEM[ADDR + OFFSET.u32 * 12U + 4U].b32;
RETURN_DATA[95 : 64] = MEM[ADDR + OFFSET.u32 * 12U + 8U].b32
```

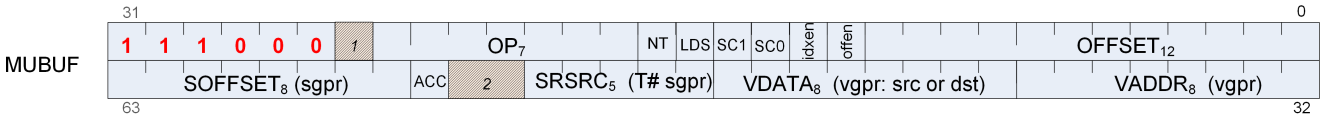
DS_READ_B128 255

Load 128 bits of data from a data share into a vector register.

```
RETURN_DATA[31 : 0] = MEM[ADDR + OFFSET.u32 * 16U].b32;
RETURN_DATA[63 : 32] = MEM[ADDR + OFFSET.u32 * 16U + 4U].b32;
RETURN_DATA[95 : 64] = MEM[ADDR + OFFSET.u32 * 16U + 8U].b32;
RETURN_DATA[127 : 96] = MEM[ADDR + OFFSET.u32 * 16U + 12U].b32
```

12.13. MUBUF Instructions

The bitfield map of the MUBUF format is:




```
where:

OFFSET  = Unsigned immediate byte offset.
OFFEN   = Send offset either as VADDR or as zero..
IDXEN   = Send index either as VADDR or as zero.
LDS     = Data read from/written to LDS or VGPR.
OP      = Instruction Opcode.
VADDR   = VGPR address source.
VDATA   = Destination vector GPR.
SRSRC   = Scalar GPR that specifies resource constant.
ACC     = Return to ACC VGPRs
SC      = Scope
NT      = Non-Temporal
SOFFSET = Byte offset added to the memory address of an SGPR.
```

BUFFER_LOAD_FORMAT_X 0

Load 1-component formatted data from a buffer surface, convert the data to 32 bit integral or floating point format, then store the result into a vector register. The resource descriptor specifies the data format of the surface.

```
VDATA[31 : 0].b32 = ConvertFromFormat(MEM[TADDR.X]);
// Mem access size depends on format
```

BUFFER_LOAD_FORMAT_XY 1

Load 2-component formatted data from a buffer surface, convert the data to 32 bit integral or floating point format, then store the result into a vector register. The resource descriptor specifies the data format of the surface.

```
VDATA[31 : 0].b32 = ConvertFromFormat(MEM[TADDR.X]);
// Mem access size depends on format
VDATA[63 : 32].b32 = ConvertFromFormat(MEM[TADDR.Y])
```

BUFFER_LOAD_FORMAT_XYZ 2

Load 3-component formatted data from a buffer surface, convert the data to 32 bit integral or floating point

format, then store the result into a vector register. The resource descriptor specifies the data format of the surface.

```
VDATA[31 : 0].b32 = ConvertFromFormat(MEM[TADDR.X]);
// Mem access size depends on format
VDATA[63 : 32].b32 = ConvertFromFormat(MEM[TADDR.Y]);
VDATA[95 : 64].b32 = ConvertFromFormat(MEM[TADDR.Z])
```

BUFFER_LOAD_FORMAT_XYZW 3

Load 4-component formatted data from a buffer surface, convert the data to 32 bit integral or floating point format, then store the result into a vector register. The resource descriptor specifies the data format of the surface.

```
VDATA[31 : 0].b32 = ConvertFromFormat(MEM[TADDR.X]);
// Mem access size depends on format
VDATA[63 : 32].b32 = ConvertFromFormat(MEM[TADDR.Y]);
VDATA[95 : 64].b32 = ConvertFromFormat(MEM[TADDR.Z]);
VDATA[127 : 96].b32 = ConvertFromFormat(MEM[TADDR.W])
```

BUFFER_STORE_FORMAT_X 4

Convert 32 bits of data from vector input registers into 1-component formatted data and store the data into a buffer surface. The instruction specifies the data format of the surface, overriding the resource descriptor.

```
MEM[TADDR.X] = ConvertToFormat(VDATA[31 : 0].b32);
// Mem access size depends on format
```

BUFFER_STORE_FORMAT_XY 5

Convert 64 bits of data from vector input registers into 2-component formatted data and store the data into a buffer surface. The instruction specifies the data format of the surface, overriding the resource descriptor.

```
MEM[TADDR.X] = ConvertToFormat(VDATA[31 : 0].b32);
// Mem access size depends on format
MEM[TADDR.Y] = ConvertToFormat(VDATA[63 : 32].b32)
```

BUFFER_STORE_FORMAT_XYZ 6

Convert 96 bits of data from vector input registers into 3-component formatted data and store the data into a

buffer surface. The instruction specifies the data format of the surface, overriding the resource descriptor.

```
MEM[TADDR.X] = ConvertToFormat(VDATA[31 : 0].b32);
// Mem access size depends on format
MEM[TADDR.Y] = ConvertToFormat(VDATA[63 : 32].b32);
MEM[TADDR.Z] = ConvertToFormat(VDATA[95 : 64].b32)
```

BUFFER_STORE_FORMAT_XYZW 7

Convert 128 bits of data from vector input registers into 4-component formatted data and store the data into a buffer surface. The instruction specifies the data format of the surface, overriding the resource descriptor.

```
MEM[TADDR.X] = ConvertToFormat(VDATA[31 : 0].b32);
// Mem access size depends on format
MEM[TADDR.Y] = ConvertToFormat(VDATA[63 : 32].b32);
MEM[TADDR.Z] = ConvertToFormat(VDATA[95 : 64].b32);
MEM[TADDR.W] = ConvertToFormat(VDATA[127 : 96].b32)
```

BUFFER_LOAD_FORMAT_D16_X 8

Load 1-component formatted data from a buffer surface, convert the data to packed 16 bit integral or floating point format, then store the result into the low 16 bits of a 32-bit vector register. The resource descriptor specifies the data format of the surface.

```
VDATA[15 : 0].b16 = 16'B(ConvertFromFormat(MEM[TADDR.X]));
// Mem access size depends on format
// VDATA[31:16].b16 is preserved.
```

BUFFER_LOAD_FORMAT_D16_XY 9

Load 2-component formatted data from a buffer surface, convert the data to packed 16 bit integral or floating point format, then store the result into a vector register. The resource descriptor specifies the data format of the surface.

```
VDATA[15 : 0].b16 = 16'B(ConvertFromFormat(MEM[TADDR.X]));
// Mem access size depends on format
VDATA[31 : 16].b16 = 16'B(ConvertFromFormat(MEM[TADDR.Y]))
```

BUFFER_LOAD_FORMAT_D16_XYZ 10

Load 3-component formatted data from a buffer surface, convert the data to packed 16 bit integral or floating point format, then store the result into a vector register. The resource descriptor specifies the data format of the surface.

```
VDATA[15 : 0].b16 = 16'B(ConvertFromFormat(MEM[TADDR.X]));
// Mem access size depends on format
VDATA[31 : 16].b16 = 16'B(ConvertFromFormat(MEM[TADDR.Y]));
VDATA[47 : 32].b16 = 16'B(ConvertFromFormat(MEM[TADDR.Z]));
// VDATA[63:48].b16 is preserved.
```

BUFFER_LOAD_FORMAT_D16_XYZW 11

Load 4-component formatted data from a buffer surface, convert the data to packed 16 bit integral or floating point format, then store the result into a vector register. The resource descriptor specifies the data format of the surface.

```
VDATA[15 : 0].b16 = 16'B(ConvertFromFormat(MEM[TADDR.X]));
// Mem access size depends on format
VDATA[31 : 16].b16 = 16'B(ConvertFromFormat(MEM[TADDR.Y]));
VDATA[47 : 32].b16 = 16'B(ConvertFromFormat(MEM[TADDR.Z]));
VDATA[63 : 48].b16 = 16'B(ConvertFromFormat(MEM[TADDR.W]))
```

BUFFER_STORE_FORMAT_D16_X 12

Convert 16 bits of data from the low 16 bits of a 32-bit vector input register into 1-component formatted data and store the data into a buffer surface. The instruction specifies the data format of the surface, overriding the resource descriptor.

```
MEM[TADDR.X] = ConvertToFormat(32'B(VDATA[15 : 0].b16));
// Mem access size depends on format
```

BUFFER_STORE_FORMAT_D16_XY 13

Convert 32 bits of data from vector input registers into 2-component formatted data and store the data into a buffer surface. The instruction specifies the data format of the surface, overriding the resource descriptor.

```
MEM[TADDR.X] = ConvertToFormat(32'B(VDATA[15 : 0].b16));
// Mem access size depends on format
MEM[TADDR.Y] = ConvertToFormat(32'B(VDATA[31 : 16].b16))
```

BUFFER_STORE_FORMAT_D16_XYZ 14

Convert 48 bits of data from vector input registers into 3-component formatted data and store the data into a buffer surface. The instruction specifies the data format of the surface, overriding the resource descriptor.

```
MEM[TADDR.X] = ConvertToFormat(32'B(VDATA[15 : 0].b16));
// Mem access size depends on format
MEM[TADDR.Y] = ConvertToFormat(32'B(VDATA[31 : 16].b16));
MEM[TADDR.Z] = ConvertToFormat(32'B(VDATA[47 : 32].b16))
```

BUFFER_STORE_FORMAT_D16_XYZW 15

Convert 64 bits of data from vector input registers into 4-component formatted data and store the data into a buffer surface. The instruction specifies the data format of the surface, overriding the resource descriptor.

```
MEM[TADDR.X] = ConvertToFormat(32'B(VDATA[15 : 0].b16));
// Mem access size depends on format
MEM[TADDR.Y] = ConvertToFormat(32'B(VDATA[31 : 16].b16));
MEM[TADDR.Z] = ConvertToFormat(32'B(VDATA[47 : 32].b16));
MEM[TADDR.W] = ConvertToFormat(32'B(VDATA[63 : 48].b16))
```

BUFFER_LOAD_UBYTE 16

Load 8 bits of unsigned data from a buffer surface, zero extend to 32 bits and store the result into a vector register.

```
VDATA.u32 = 32'U({ 24'0U, MEM[ADDR].u8 })
```

BUFFER_LOAD_SBYTE 17

Load 8 bits of signed data from a buffer surface, sign extend to 32 bits and store the result into a vector register.

```
VDATA.i32 = 32'I(signext(MEM[ADDR].i8))
```

BUFFER_LOAD_USHORT 18

Load 16 bits of unsigned data from a buffer surface, zero extend to 32 bits and store the result into a vector register.

```
VDATA.u32 = 32'U({ 16'0U, MEM[ADDR].u16 })
```

BUFFER_LOAD_SSHORT 19

Load 16 bits of signed data from a buffer surface, sign extend to 32 bits and store the result into a vector register.

```
VDATA.i32 = 32'I(signext(MEM[ADDR].i16))
```

BUFFER_LOAD_DWORD 20

Load 32 bits of data from a buffer surface into a vector register.

```
VDATA[31 : 0] = MEM[ADDR].b32
```

BUFFER_LOAD_DWORDX2 21

Load 64 bits of data from a buffer surface into a vector register.

```
VDATA[31 : 0] = MEM[ADDR].b32;
VDATA[63 : 32] = MEM[ADDR + 4U].b32
```

BUFFER_LOAD_DWORDX3 22

Load 96 bits of data from a buffer surface into a vector register.

```
VDATA[31 : 0] = MEM[ADDR].b32;
VDATA[63 : 32] = MEM[ADDR + 4U].b32;
VDATA[95 : 64] = MEM[ADDR + 8U].b32
```

BUFFER_LOAD_DWORDX4 23

Load 128 bits of data from a buffer surface into a vector register.

```
VDATA[31 : 0] = MEM[ADDR].b32;
```

```
VDATA[63 : 32] = MEM[ADDR + 4U].b32;
VDATA[95 : 64] = MEM[ADDR + 8U].b32;
VDATA[127 : 96] = MEM[ADDR + 12U].b32
```

BUFFER_STORE_BYTE 24

Store 8 bits of data from a vector register into a buffer surface.

```
MEM[ADDR].b8 = VDATA[7 : 0]
```

BUFFER_STORE_BYTE_D16_HI 25

Store 8 bits of data from the high 16 bits of a 32-bit vector register into a buffer surface.

```
MEM[ADDR].b8 = VDATA[23 : 16]
```

BUFFER_STORE_SHORT 26

Store 16 bits of data from a vector register into a buffer surface.

```
MEM[ADDR].b16 = VDATA[15 : 0]
```

BUFFER_STORE_SHORT_D16_HI 27

Store 16 bits of data from the high 16 bits of a 32-bit vector register into a buffer surface.

```
MEM[ADDR].b16 = VDATA[31 : 16]
```

BUFFER_STORE_DWORD 28

Store 32 bits of data from vector input registers into a buffer surface.

```
MEM[ADDR].b32 = VDATA[31 : 0]
```

BUFFER_STORE_DWORDX2 29

Store 64 bits of data from vector input registers into a buffer surface.

```
MEM[ADDR].b32 = VDATA[31 : 0];
MEM[ADDR + 4U].b32 = VDATA[63 : 32]
```

BUFFER_STORE_DWORDX3 30

Store 96 bits of data from vector input registers into a buffer surface.

```
MEM[ADDR].b32 = VDATA[31 : 0];
MEM[ADDR + 4U].b32 = VDATA[63 : 32];
MEM[ADDR + 8U].b32 = VDATA[95 : 64]
```

BUFFER_STORE_DWORDX4 31

Store 128 bits of data from vector input registers into a buffer surface.

```
MEM[ADDR].b32 = VDATA[31 : 0];
MEM[ADDR + 4U].b32 = VDATA[63 : 32];
MEM[ADDR + 8U].b32 = VDATA[95 : 64];
MEM[ADDR + 12U].b32 = VDATA[127 : 96]
```

BUFFER_LOAD_UBYTE_D16 32

Load 8 bits of unsigned data from a buffer surface, zero extend to 16 bits and store the result into the low 16 bits of a 32-bit vector register.

```
VDATA[15 : 0].u16 = 16'U({ 8'0U, MEM[ADDR].u8 });
// VDATA[31:16] is preserved.
```

BUFFER_LOAD_UBYTE_D16_HI 33

Load 8 bits of unsigned data from a buffer surface, zero extend to 16 bits and store the result into the high 16 bits of a 32-bit vector register.

```
VDATA[31 : 16].u16 = 16'U({ 8'0U, MEM[ADDR].u8 });
```

```
// VDATA[15:0] is preserved.
```

BUFFER_LOAD_SBYTE_D16 34

Load 8 bits of signed data from a buffer surface, sign extend to 16 bits and store the result into the low 16 bits of a 32-bit vector register.

```
VDATA[15 : 0].i16 = 16'I(signext(MEM[ADDR].i8));
// VDATA[31:16] is preserved.
```

BUFFER_LOAD_SBYTE_D16_HI 35

Load 8 bits of signed data from a buffer surface, sign extend to 16 bits and store the result into the high 16 bits of a 32-bit vector register.

```
VDATA[31 : 16].i16 = 16'I(signext(MEM[ADDR].i8));
// VDATA[15:0] is preserved.
```

BUFFER_LOAD_SHORT_D16 36

Load 16 bits of unsigned data from a buffer surface and store the result into the low 16 bits of a 32-bit vector register.

```
VDATA[15 : 0].b16 = MEM[ADDR].b16;
// VDATA[31:16] is preserved.
```

BUFFER_LOAD_SHORT_D16_HI 37

Load 16 bits of unsigned data from a buffer surface and store the result into the high 16 bits of a 32-bit vector register.

```
VDATA[31 : 16].b16 = MEM[ADDR].b16;
// VDATA[15:0] is preserved.
```

BUFFER_LOAD_FORMAT_D16_HI_X 38

Load 1-component formatted data from a buffer surface, convert the data to packed 16 bit integral or floating

point format, then store the result into the high 16 bits of a 32-bit vector register. The resource descriptor specifies the data format of the surface.

```
VDATA[31 : 16].b16 = 16'B(ConvertFromFormat(MEM[TADDR.X]));
// Mem access size depends on format
// VDATA[15:0].b16 is preserved.
```

BUFFER_STORE_FORMAT_D16_HI_X 39

Convert 16 bits of data from the high 16 bits of a 32-bit vector input register into 1-component formatted data and store the data into a buffer surface. The instruction specifies the data format of the surface, overriding the resource descriptor.

```
MEM[TADDR.X] = ConvertToFormat(32'B(VDATA[31 : 16].b16));
// Mem access size depends on format
```

BUFFER_WBL2 40

Write back L2 cache. Returns ACK to shader.

BUFFER_INV 41

Invalidate CU and/or L2 cache depending on sc0 and sc1 bits. Returns ACK to shader.

BUFFER_ATOMIC_SWAP 64

Swap an unsigned 32-bit integer value in the data register with a location in a buffer surface. Store the original value from buffer surface into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].b32;
MEM[ADDR].b32 = DATA.b32;
RETURN_DATA.b32 = tmp
```

BUFFER_ATOMIC_CMPSWAP 65

Compare two unsigned 32-bit integer values stored in the data comparison register and a location in a buffer surface. Modify the memory location with a value in the data source register iff the comparison is equal. Store the original value from buffer surface into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u32;
src = DATA[31 : 0].u32;
cmp = DATA[63 : 32].u32;
MEM[ADDR].u32 = tmp == cmp ? src : tmp;
RETURN_DATA.u32 = tmp
```

BUFFER_ATOMIC_ADD 66

Add two unsigned 32-bit integer values stored in the data register and a location in a buffer surface. Store the original value from buffer surface into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u32;
MEM[ADDR].u32 += DATA.u32;
RETURN_DATA.u32 = tmp
```

BUFFER_ATOMIC_SUB 67

Subtract an unsigned 32-bit integer value stored in the data register from a value stored in a location in a buffer surface. Store the original value from buffer surface into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u32;
MEM[ADDR].u32 -= DATA.u32;
RETURN_DATA.u32 = tmp
```

BUFFER_ATOMIC_SMIN 68

Select the minimum of two signed 32-bit integer inputs, given two values stored in the data register and a location in a buffer surface. Store the original value from buffer surface into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].i32;
src = DATA.i32;
MEM[ADDR].i32 = src < tmp ? src : tmp;
RETURN_DATA.i32 = tmp
```

BUFFER_ATOMIC_UMIN 69

Select the minimum of two unsigned 32-bit integer inputs, given two values stored in the data register and a location in a buffer surface. Store the original value from buffer surface into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u32;
src = DATA.u32;
MEM[ADDR].u32 = src < tmp ? src : tmp;
RETURN_DATA.u32 = tmp
```

BUFFER_ATOMIC_SMAX 70

Select the maximum of two signed 32-bit integer inputs, given two values stored in the data register and a location in a buffer surface. Store the original value from buffer surface into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].i32;
src = DATA.i32;
MEM[ADDR].i32 = src >= tmp ? src : tmp;
RETURN_DATA.i32 = tmp
```

BUFFER_ATOMIC_UMAX 71

Select the maximum of two unsigned 32-bit integer inputs, given two values stored in the data register and a location in a buffer surface. Store the original value from buffer surface into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u32;
src = DATA.u32;
MEM[ADDR].u32 = src >= tmp ? src : tmp;
RETURN_DATA.u32 = tmp
```

BUFFER_ATOMIC_AND 72

Calculate bitwise AND given two unsigned 32-bit integer values stored in the data register and a location in a buffer surface. Store the original value from buffer surface into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].b32;
MEM[ADDR].b32 = (tmp & DATA.b32);
RETURN_DATA.b32 = tmp
```

BUFFER_ATOMIC_OR 73

Calculate bitwise OR given two unsigned 32-bit integer values stored in the data register and a location in a buffer surface. Store the original value from buffer surface into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].b32;
MEM[ADDR].b32 = (tmp | DATA.b32);
RETURN_DATA.b32 = tmp
```

BUFFER_ATOMIC_XOR 74

Calculate bitwise XOR given two unsigned 32-bit integer values stored in the data register and a location in a buffer surface. Store the original value from buffer surface into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].b32;
MEM[ADDR].b32 = (tmp ^ DATA.b32);
RETURN_DATA.b32 = tmp
```

BUFFER_ATOMIC_INC 75

Increment an unsigned 32-bit integer value from a location in a buffer surface with wraparound to 0 if the value exceeds a value in the data register. Store the original value from buffer surface into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u32;
src = DATA.u32;
MEM[ADDR].u32 = tmp >= src ? 0U : tmp + 1U;
RETURN_DATA.u32 = tmp
```

BUFFER_ATOMIC_DEC 76

Decrement an unsigned 32-bit integer value from a location in a buffer surface with wraparound to a value in the data register if the decrement yields a negative value. Store the original value from buffer surface into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u32;
src = DATA.u32;
MEM[ADDR].u32 = ((tmp == 0U) || (tmp > src)) ? src : tmp - 1U;
RETURN_DATA.u32 = tmp
```

BUFFER_ATOMIC_ADD_F32 77

Add a single-precision float value in the data register to a location in a buffer surface. Store the original value from buffer surface into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].f32;
MEM[ADDR].f32 += DATA.f32;
RETURN_DATA = tmp
```

Notes

Floating-point addition handles NAN/INF/denorm.

BUFFER_ATOMIC_PK_ADD_F16 78

Add a packed 2-component half-precision float value in the data register to a location in a buffer surface. Store the original value from buffer surface into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR];
src = DATA;
dst[31 : 16].f16 = tmp[31 : 16].f16 + src[31 : 16].f16;
dst[15 : 0].f16 = tmp[15 : 0].f16 + src[15 : 0].f16;
MEM[ADDR] = dst.b32;
RETURN_DATA = tmp
```

Notes

Floating-point addition handles NAN/INF/denorm.

BUFFER_ATOMIC_ADD_F64 79

Add a double-precision float value in the data register to a location in a buffer surface. Store the original value from buffer surface into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].f64;
MEM[ADDR].f64 += DATA.f64;
RETURN_DATA = tmp
```

Notes

Floating-point addition handles NAN/INF/denorm.

BUFFER_ATOMIC_MIN_F64 80

Select the minimum signed integer value given the data register and a location in a buffer surface. Store the original value from buffer surface into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].f64;
```

```
src = DATA.f64;
MEM[ADDR].f64 = src < tmp ? src : tmp;
RETURN_DATA.f64 = tmp
```

BUFFER_ATOMIC_MAX_F64 81

Select the maximum signed integer value given the data register and a location in a buffer surface. Store the original value from buffer surface into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].f64;
src = DATA.f64;
MEM[ADDR].f64 = src > tmp ? src : tmp;
RETURN_DATA.f64 = tmp
```

BUFFER_ATOMIC_SWAP_X2 96

Swap an unsigned 64-bit integer value in the data register with a location in a buffer surface. Store the original value from buffer surface into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].b64;
MEM[ADDR].b64 = DATA.b64;
RETURN_DATA.b64 = tmp
```

BUFFER_ATOMIC_CMPSWAP_X2 97

Compare two unsigned 64-bit integer values stored in the data comparison register and a location in a buffer surface. Modify the memory location with a value in the data source register iff the comparison is equal. Store the original value from buffer surface into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u64;
src = DATA[63 : 0].u64;
cmp = DATA[127 : 64].u64;
MEM[ADDR].u64 = tmp == cmp ? src : tmp;
RETURN_DATA.u64 = tmp
```

BUFFER_ATOMIC_ADD_X2 98

Add two unsigned 64-bit integer values stored in the data register and a location in a buffer surface. Store the original value from buffer surface into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u64;
MEM[ADDR].u64 += DATA.u64;
RETURN_DATA.u64 = tmp
```

BUFFER_ATOMIC_SUB_X2 99

Subtract an unsigned 64-bit integer value stored in the data register from a value stored in a location in a buffer surface. Store the original value from buffer surface into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u64;
MEM[ADDR].u64 -= DATA.u64;
RETURN_DATA.u64 = tmp
```

BUFFER_ATOMIC_SMIN_X2 100

Select the minimum of two signed 64-bit integer inputs, given two values stored in the data register and a location in a buffer surface. Store the original value from buffer surface into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].i64;
src = DATA.i64;
MEM[ADDR].i64 = src < tmp ? src : tmp;
RETURN_DATA.i64 = tmp
```

BUFFER_ATOMIC_UMIN_X2 101

Select the minimum of two unsigned 64-bit integer inputs, given two values stored in the data register and a location in a buffer surface. Store the original value from buffer surface into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u64;
src = DATA.u64;
MEM[ADDR].u64 = src < tmp ? src : tmp;
RETURN_DATA.u64 = tmp
```

BUFFER_ATOMIC_SMAX_X2 102

Select the maximum of two signed 64-bit integer inputs, given two values stored in the data register and a location in a buffer surface. Store the original value from buffer surface into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].i64;
src = DATA.i64;
MEM[ADDR].i64 = src >= tmp ? src : tmp;
RETURN_DATA.i64 = tmp
```

BUFFER_ATOMIC_UMAX_X2 103

Select the maximum of two unsigned 64-bit integer inputs, given two values stored in the data register and a location in a buffer surface. Store the original value from buffer surface into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u64;
src = DATA.u64;
MEM[ADDR].u64 = src >= tmp ? src : tmp;
RETURN_DATA.u64 = tmp
```

BUFFER_ATOMIC_AND_X2 104

Calculate bitwise AND given two unsigned 64-bit integer values stored in the data register and a location in a buffer surface. Store the original value from buffer surface into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].b64;
MEM[ADDR].b64 = (tmp & DATA.b64);
RETURN_DATA.b64 = tmp
```

BUFFER_ATOMIC_OR_X2 105

Calculate bitwise OR given two unsigned 64-bit integer values stored in the data register and a location in a buffer surface. Store the original value from buffer surface into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].b64;
MEM[ADDR].b64 = (tmp | DATA.b64);
RETURN_DATA.b64 = tmp
```

BUFFER_ATOMIC_XOR_X2 106

Calculate bitwise XOR given two unsigned 64-bit integer values stored in the data register and a location in a buffer surface. Store the original value from buffer surface into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].b64;
MEM[ADDR].b64 = (tmp ^ DATA.b64);
RETURN_DATA.b64 = tmp
```

BUFFER_ATOMIC_INC_X2 107

Increment an unsigned 64-bit integer value from a location in a buffer surface with wraparound to 0 if the value exceeds a value in the data register. Store the original value from buffer surface into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u64;
src = DATA.u64;
MEM[ADDR].u64 = tmp >= src ? 0ULL : tmp + 1ULL;
RETURN_DATA.u64 = tmp
```

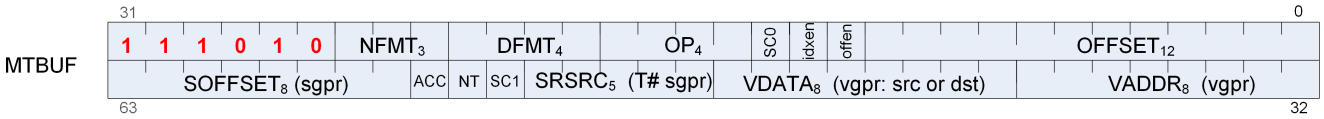
BUFFER_ATOMIC_DEC_X2 108

Decrement an unsigned 64-bit integer value from a location in a buffer surface with wraparound to a value in the data register if the decrement yields a negative value. Store the original value from buffer surface into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u64;
src = DATA.u64;
MEM[ADDR].u64 = ((tmp == 0ULL) || (tmp > src)) ? src : tmp - 1ULL;
RETURN_DATA.u64 = tmp
```

12.14. MTBUF Instructions

The bitfield map of the MTBUF format is:




```
where:

OFFSET  = Unsigned immediate byte offset.
OFFEN   = Send offset either as VADDR or as zero.
IDXEN   = Send index either as VADDR or as zero.
LDS     = Data is transferred between LDS and Memory, not VGPRs.
OP      = Instruction Opcode.
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.
SOFFSET = Unsigned byte offset from an SGPR.
SC      = Scope
NT      = Non-Temporal
ACC     = Return to ACC VGPRs
```

TBUFFER_LOAD_FORMAT_X 0

Load 1-component formatted data from a buffer surface, convert the data to 32 bit integral or floating point format, then store the result into a vector register. The instruction specifies the data format of the surface, overriding the resource descriptor.

```
VDATA[31 : 0].b32 = ConvertFromFormat(MEM[TADDR.X]);
// Mem access size depends on format
```

TBUFFER_LOAD_FORMAT_XY 1

Load 2-component formatted data from a buffer surface, convert the data to 32 bit integral or floating point format, then store the result into a vector register. The instruction specifies the data format of the surface, overriding the resource descriptor.

```
VDATA[31 : 0].b32 = ConvertFromFormat(MEM[TADDR.X]);
// Mem access size depends on format
VDATA[63 : 32].b32 = ConvertFromFormat(MEM[TADDR.Y])
```

TBUFFER_LOAD_FORMAT_XYZ 2

Load 3-component formatted data from a buffer surface, convert the data to 32 bit integral or floating point format, then store the result into a vector register. The instruction specifies the data format of the surface, overriding the resource descriptor.

```
VDATA[31 : 0].b32 = ConvertFromFormat(MEM[TADDR.X]);
// Mem access size depends on format
VDATA[63 : 32].b32 = ConvertFromFormat(MEM[TADDR.Y]);
VDATA[95 : 64].b32 = ConvertFromFormat(MEM[TADDR.Z])
```

TBUFFER_LOAD_FORMAT_XYZW 3

Load 4-component formatted data from a buffer surface, convert the data to 32 bit integral or floating point format, then store the result into a vector register. The instruction specifies the data format of the surface, overriding the resource descriptor.

```
VDATA[31 : 0].b32 = ConvertFromFormat(MEM[TADDR.X]);
// Mem access size depends on format
VDATA[63 : 32].b32 = ConvertFromFormat(MEM[TADDR.Y]);
VDATA[95 : 64].b32 = ConvertFromFormat(MEM[TADDR.Z]);
VDATA[127 : 96].b32 = ConvertFromFormat(MEM[TADDR.W])
```

TBUFFER_STORE_FORMAT_X 4

Convert 32 bits of data from vector input registers into 1-component formatted data and store the data into a buffer surface. The instruction specifies the data format of the surface, overriding the resource descriptor.

```
MEM[TADDR.X] = ConvertToFormat(VDATA[31 : 0].b32);
// Mem access size depends on format
```

TBUFFER_STORE_FORMAT_XY 5

Convert 64 bits of data from vector input registers into 2-component formatted data and store the data into a buffer surface. The instruction specifies the data format of the surface, overriding the resource descriptor.

```
MEM[TADDR.X] = ConvertToFormat(VDATA[31 : 0].b32);
// Mem access size depends on format
MEM[TADDR.Y] = ConvertToFormat(VDATA[63 : 32].b32)
```

TBUFFER_STORE_FORMAT_XYZ 6

Convert 96 bits of data from vector input registers into 3-component formatted data and store the data into a buffer surface. The instruction specifies the data format of the surface, overriding the resource descriptor.

```
MEM[TADDR.X] = ConvertToFormat(VDATA[31 : 0].b32);
// Mem access size depends on format
MEM[TADDR.Y] = ConvertToFormat(VDATA[63 : 32].b32);
MEM[TADDR.Z] = ConvertToFormat(VDATA[95 : 64].b32)
```

TBUFFER_STORE_FORMAT_XYZW 7

Convert 128 bits of data from vector input registers into 4-component formatted data and store the data into a buffer surface. The instruction specifies the data format of the surface, overriding the resource descriptor.

```
MEM[TADDR.X] = ConvertToFormat(VDATA[31 : 0].b32);
// Mem access size depends on format
MEM[TADDR.Y] = ConvertToFormat(VDATA[63 : 32].b32);
MEM[TADDR.Z] = ConvertToFormat(VDATA[95 : 64].b32);
MEM[TADDR.W] = ConvertToFormat(VDATA[127 : 96].b32)
```

TBUFFER_LOAD_FORMAT_D16_X 8

Load 1-component formatted data from a buffer surface, convert the data to packed 16 bit integral or floating point format, then store the result into a vector register. The instruction specifies the data format of the surface, overriding the resource descriptor.

```
VDATA[15 : 0].b16 = 16'B(ConvertFromFormat(MEM[TADDR.X]));
// Mem access size depends on format
// VDATA[31:16].b16 is preserved.
```

TBUFFER_LOAD_FORMAT_D16_XY 9

Load 2-component formatted data from a buffer surface, convert the data to packed 16 bit integral or floating point format, then store the result into a vector register. The instruction specifies the data format of the surface, overriding the resource descriptor.

```
VDATA[15 : 0].b16 = 16'B(ConvertFromFormat(MEM[TADDR.X]));
// Mem access size depends on format
VDATA[31 : 16].b16 = 16'B(ConvertFromFormat(MEM[TADDR.Y]))
```

TBUFFER_LOAD_FORMAT_D16_XYZ 10

Load 3-component formatted data from a buffer surface, convert the data to packed 16 bit integral or floating point format, then store the result into a vector register. The instruction specifies the data format of the surface, overriding the resource descriptor.

```
VDATA[15 : 0].b16 = 16'B(ConvertFromFormat(MEM[TADDR.X]));
// Mem access size depends on format
VDATA[31 : 16].b16 = 16'B(ConvertFromFormat(MEM[TADDR.Y]));
VDATA[47 : 32].b16 = 16'B(ConvertFromFormat(MEM[TADDR.Z]));
// VDATA[63:48].b16 is preserved.
```

TBUFFER_LOAD_FORMAT_D16_XYZW 11

Load 4-component formatted data from a buffer surface, convert the data to packed 16 bit integral or floating point format, then store the result into a vector register. The instruction specifies the data format of the surface, overriding the resource descriptor.

```
VDATA[15 : 0].b16 = 16'B(ConvertFromFormat(MEM[TADDR.X]));
// Mem access size depends on format
VDATA[31 : 16].b16 = 16'B(ConvertFromFormat(MEM[TADDR.Y]));
VDATA[47 : 32].b16 = 16'B(ConvertFromFormat(MEM[TADDR.Z]));
VDATA[63 : 48].b16 = 16'B(ConvertFromFormat(MEM[TADDR.W]))
```

TBUFFER_STORE_FORMAT_D16_X 12

Convert 16 bits of data from vector input registers into 1-component formatted data and store the data into a buffer surface. The instruction specifies the data format of the surface, overriding the resource descriptor.

```
MEM[TADDR.X] = ConvertToFormat(32'B(VDATA[15 : 0].b16));
// Mem access size depends on format
```

TBUFFER_STORE_FORMAT_D16_XY 13

Convert 32 bits of data from vector input registers into 2-component formatted data and store the data into a buffer surface. The instruction specifies the data format of the surface, overriding the resource descriptor.

```
MEM[TADDR.X] = ConvertToFormat(32'B(VDATA[15 : 0].b16));
// Mem access size depends on format
MEM[TADDR.Y] = ConvertToFormat(32'B(VDATA[31 : 16].b16))
```

TBUFFER_STORE_FORMAT_D16_XYZ 14

Convert 48 bits of data from vector input registers into 3-component formatted data and store the data into a buffer surface. The instruction specifies the data format of the surface, overriding the resource descriptor.

```
MEM[TADDR.X] = ConvertToFormat(32'B(VDATA[15 : 0].b16));
// Mem access size depends on format
MEM[TADDR.Y] = ConvertToFormat(32'B(VDATA[31 : 16].b16));
MEM[TADDR.Z] = ConvertToFormat(32'B(VDATA[47 : 32].b16))
```

TBUFFER_STORE_FORMAT_D16_XYZW 15

Convert 64 bits of data from vector input registers into 4-component formatted data and store the data into a buffer surface. The instruction specifies the data format of the surface, overriding the resource descriptor.

```
MEM[TADDR.X] = ConvertToFormat(32'B(VDATA[15 : 0].b16));
// Mem access size depends on format
MEM[TADDR.Y] = ConvertToFormat(32'B(VDATA[31 : 16].b16));
MEM[TADDR.Z] = ConvertToFormat(32'B(VDATA[47 : 32].b16));
MEM[TADDR.W] = ConvertToFormat(32'B(VDATA[63 : 48].b16))
```

12.15. FLAT, Scratch and Global Instructions

The bitfield map of the FLAT format is:


```
where:

OP     = Instruction Opcode.
ADDR   = Source of flat address VGPR.
DATA   = Source data.
VDST   = Destination VGPR.
NV     = Access to non-volatile memory.
SADDR  = SGPR holding address or offset
SEG    = Instruction type: Flat, Scratch, or Global
LDS    = Data is transferred between LDS and Memory, not VGPRs.
OFFSET = Immediate address byte-offset.
SC      = Scope
NT      = Non-Temporal
```

12.15.1. Flat Instructions

Flat instructions look at the per-workitem address and determine for each work item if the target memory address is in global, private or scratch memory.

FLAT_LOAD_UBYTE **16**

Load 8 bits of unsigned data from the flat aperture, zero extend to 32 bits and store the result into a vector register.

```
VDATA.u32 = 32'U({ 24'0U, MEM[ADDR].u8 })
```

FLAT_LOAD_SBYTE **17**

Load 8 bits of signed data from the flat aperture, sign extend to 32 bits and store the result into a vector register.

```
VDATA.i32 = 32'I(signext(MEM[ADDR].i8))
```

FLAT_LOAD_USHORT **18**

Load 16 bits of unsigned data from the flat aperture, zero extend to 32 bits and store the result into a vector register.

```
VDATA.u32 = 32'U({ 16'0U, MEM[ADDR].u16 })
```

FLAT_LOAD_SSHORT 19

Load 16 bits of signed data from the flat aperture, sign extend to 32 bits and store the result into a vector register.

```
VDATA.i32 = 32'I(signext(MEM[ADDR].i16))
```

FLAT_LOAD_DWORD 20

Load 32 bits of data from the flat aperture into a vector register.

```
VDATA[31 : 0] = MEM[ADDR].b32
```

FLAT_LOAD_DWORDX2 21

Load 64 bits of data from the flat aperture into a vector register.

```
VDATA[31 : 0] = MEM[ADDR].b32;
VDATA[63 : 32] = MEM[ADDR + 4U].b32
```

FLAT_LOAD_DWORDX3 22

Load 96 bits of data from the flat aperture into a vector register.

```
VDATA[31 : 0] = MEM[ADDR].b32;
VDATA[63 : 32] = MEM[ADDR + 4U].b32;
VDATA[95 : 64] = MEM[ADDR + 8U].b32
```

FLAT_LOAD_DWORDX4 23

Load 128 bits of data from the flat aperture into a vector register.

```
VDATA[31 : 0] = MEM[ADDR].b32;
VDATA[63 : 32] = MEM[ADDR + 4U].b32;
VDATA[95 : 64] = MEM[ADDR + 8U].b32;
VDATA[127 : 96] = MEM[ADDR + 12U].b32
```

FLAT_STORE_BYTE 24

Store 8 bits of data from a vector register into the flat aperture.

```
MEM[ADDR].b8 = VDATA[7 : 0]
```

FLAT_STORE_BYTE_D16_HI 25

Store 8 bits of data from the high 16 bits of a 32-bit vector register into the flat aperture.

```
MEM[ADDR].b8 = VDATA[23 : 16]
```

FLAT_STORE_SHORT 26

Store 16 bits of data from a vector register into the flat aperture.

```
MEM[ADDR].b16 = VDATA[15 : 0]
```

FLAT_STORE_SHORT_D16_HI 27

Store 16 bits of data from the high 16 bits of a 32-bit vector register into the flat aperture.

```
MEM[ADDR].b16 = VDATA[31 : 16]
```

FLAT_STORE_DWORD 28

Store 32 bits of data from vector input registers into the flat aperture.

```
MEM[ADDR].b32 = VDATA[31 : 0]
```

FLAT_STORE_DWORDX2 29

Store 64 bits of data from vector input registers into the flat aperture.

```
MEM[ADDR].b32 = VDATA[31 : 0];
MEM[ADDR + 4U].b32 = VDATA[63 : 32]
```

FLAT_STORE_DWORDX3 30

Store 96 bits of data from vector input registers into the flat aperture.

```
MEM[ADDR].b32 = VDATA[31 : 0];
MEM[ADDR + 4U].b32 = VDATA[63 : 32];
MEM[ADDR + 8U].b32 = VDATA[95 : 64]
```

FLAT_STORE_DWORDX4 31

Store 128 bits of data from vector input registers into the flat aperture.

```
MEM[ADDR].b32 = VDATA[31 : 0];
MEM[ADDR + 4U].b32 = VDATA[63 : 32];
MEM[ADDR + 8U].b32 = VDATA[95 : 64];
MEM[ADDR + 12U].b32 = VDATA[127 : 96]
```

FLAT_LOAD_UBYTE_D16 32

Load 8 bits of unsigned data from the flat aperture, zero extend to 16 bits and store the result into the low 16 bits of a 32-bit vector register.

```
VDATA[15 : 0].u16 = 16'U({ 8'0U, MEM[ADDR].u8 });
// VDATA[31:16] is preserved.
```

FLAT_LOAD_UBYTE_D16_HI 33

Load 8 bits of unsigned data from the flat aperture, zero extend to 16 bits and store the result into the high 16 bits of a 32-bit vector register.

```
VDATA[31 : 16].u16 = 16'U({ 8'0U, MEM[ADDR].u8 });
// VDATA[15:0] is preserved.
```

FLAT_LOAD_SBYTE_D16 34

Load 8 bits of signed data from the flat aperture, sign extend to 16 bits and store the result into the low 16 bits of a 32-bit vector register.

```
VDATA[15 : 0].i16 = 16'I(signext(MEM[ADDR].i8));
// VDATA[31:16] is preserved.
```

FLAT_LOAD_SBYTE_D16_HI 35

Load 8 bits of signed data from the flat aperture, sign extend to 16 bits and store the result into the high 16 bits of a 32-bit vector register.

```
VDATA[31 : 16].i16 = 16'I(signext(MEM[ADDR].i8));
// VDATA[15:0] is preserved.
```

FLAT_LOAD_SHORT_D16 36

Load 16 bits of unsigned data from the flat aperture and store the result into the low 16 bits of a 32-bit vector register.

```
VDATA[15 : 0].b16 = MEM[ADDR].b16;
// VDATA[31:16] is preserved.
```

FLAT_LOAD_SHORT_D16_HI 37

Load 16 bits of unsigned data from the flat aperture and store the result into the high 16 bits of a 32-bit vector register.

```
VDATA[31 : 16].b16 = MEM[ADDR].b16;
// VDATA[15:0] is preserved.
```

FLAT_ATOMIC_SWAP 64

Swap an unsigned 32-bit integer value in the data register with a location in the flat aperture. Store the original value from flat aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].b32;
MEM[ADDR].b32 = DATA.b32;
RETURN_DATA.b32 = tmp
```

FLAT_ATOMIC_CMPSWAP 65

Compare two unsigned 32-bit integer values stored in the data comparison register and a location in the flat aperture. Modify the memory location with a value in the data source register iff the comparison is equal. Store the original value from flat aperture into a vector register iff the GLC bit is set.

NOTE: RETURN_DATA[1] is not modified.

```
tmp = MEM[ADDR].u32;
src = DATA[31 : 0].u32;
cmp = DATA[63 : 32].u32;
MEM[ADDR].u32 = tmp == cmp ? src : tmp;
RETURN_DATA.u32 = tmp
```

FLAT_ATOMIC_ADD 66

Add two unsigned 32-bit integer values stored in the data register and a location in the flat aperture. Store the original value from flat aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u32;
MEM[ADDR].u32 += DATA.u32;
RETURN_DATA.u32 = tmp
```

FLAT_ATOMIC_SUB 67

Subtract an unsigned 32-bit integer value stored in the data register from a value stored in a location in the flat aperture. Store the original value from flat aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u32;
MEM[ADDR].u32 -= DATA.u32;
RETURN_DATA.u32 = tmp
```

FLAT_ATOMIC_SMIN 68

Select the minimum of two signed 32-bit integer inputs, given two values stored in the data register and a location in the flat aperture. Store the original value from flat aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].i32;
src = DATA.i32;
MEM[ADDR].i32 = src < tmp ? src : tmp;
RETURN_DATA.i32 = tmp
```

FLAT_ATOMIC_UMIN 69

Select the minimum of two unsigned 32-bit integer inputs, given two values stored in the data register and a location in the flat aperture. Store the original value from flat aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u32;
src = DATA.u32;
MEM[ADDR].u32 = src < tmp ? src : tmp;
RETURN_DATA.u32 = tmp
```

FLAT_ATOMIC_SMAX 70

Select the maximum of two signed 32-bit integer inputs, given two values stored in the data register and a location in the flat aperture. Store the original value from flat aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].i32;
src = DATA.i32;
MEM[ADDR].i32 = src >= tmp ? src : tmp;
RETURN_DATA.i32 = tmp
```

FLAT_ATOMIC_UMAX 71

Select the maximum of two unsigned 32-bit integer inputs, given two values stored in the data register and a location in the flat aperture. Store the original value from flat aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u32;
src = DATA.u32;
MEM[ADDR].u32 = src >= tmp ? src : tmp;
RETURN_DATA.u32 = tmp
```

FLAT_ATOMIC_AND 72

Calculate bitwise AND given two unsigned 32-bit integer values stored in the data register and a location in the flat aperture. Store the original value from flat aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].b32;
MEM[ADDR].b32 = (tmp & DATA.b32);
RETURN_DATA.b32 = tmp
```

FLAT_ATOMIC_OR 73

Calculate bitwise OR given two unsigned 32-bit integer values stored in the data register and a location in the flat aperture. Store the original value from flat aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].b32;
MEM[ADDR].b32 = (tmp | DATA.b32);
RETURN_DATA.b32 = tmp
```

FLAT_ATOMIC_XOR 74

Calculate bitwise XOR given two unsigned 32-bit integer values stored in the data register and a location in the flat aperture. Store the original value from flat aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].b32;
MEM[ADDR].b32 = (tmp ^ DATA.b32);
RETURN_DATA.b32 = tmp
```

FLAT_ATOMIC_INC 75

Increment an unsigned 32-bit integer value from a location in the flat aperture with wraparound to 0 if the value exceeds a value in the data register. Store the original value from flat aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u32;
src = DATA.u32;
MEM[ADDR].u32 = tmp >= src ? 0U : tmp + 1U;
RETURN_DATA.u32 = tmp
```

FLAT_ATOMIC_DEC 76

Decrement an unsigned 32-bit integer value from a location in the flat aperture with wraparound to a value in the data register if the decrement yields a negative value. Store the original value from flat aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u32;
src = DATA.u32;
MEM[ADDR].u32 = ((tmp == 0U) || (tmp > src)) ? src : tmp - 1U;
RETURN_DATA.u32 = tmp
```

FLAT_ATOMIC_ADD_F32 77

Add a single-precision float value in the data register to a location in the flat aperture. Store the original value from flat aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].f32;
MEM[ADDR].f32 += DATA.f32;
RETURN_DATA = tmp
```

Notes

Floating-point addition handles NAN/INF/denorm.

FLAT_ATOMIC_PK_ADD_F16 78

Add a packed 2-component half-precision float value in the data register to a location in the flat aperture. Store the original value from flat aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR];
src = DATA;
dst[31 : 16].f16 = tmp[31 : 16].f16 + src[31 : 16].f16;
dst[15 : 0].f16 = tmp[15 : 0].f16 + src[15 : 0].f16;
MEM[ADDR] = dst.b32;
RETURN_DATA = tmp
```

Notes

Floating-point addition handles NAN/INF/denorm.

FLAT_ATOMIC_ADD_F64 79

Add a double-precision float value in the data register to a location in the flat aperture. Store the original value

from flat aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].f64;
MEM[ADDR].f64 += DATA.f64;
RETURN_DATA = tmp
```

Notes

Floating-point addition handles NAN/INF/denorm.

FLAT_ATOMIC_MIN_F64 80

Select the minimum signed integer value given the data register and a location in the flat aperture. Store the original value from flat aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].f64;
src = DATA.f64;
MEM[ADDR].f64 = src < tmp ? src : tmp;
RETURN_DATA.f64 = tmp
```

FLAT_ATOMIC_MAX_F64 81

Select the maximum signed integer value given the data register and a location in the flat aperture. Store the original value from flat aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].f64;
src = DATA.f64;
MEM[ADDR].f64 = src > tmp ? src : tmp;
RETURN_DATA.f64 = tmp
```

FLAT_ATOMIC_PK_ADD_BF16 82

Add a packed 2-component BF16 float value in the data register to a location in the flat aperture. Store the original value from flat aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR];
src = DATA;
dst[31 : 16].bf16 = tmp[31 : 16].bf16 + src[31 : 16].bf16;
dst[15 : 0].bf16 = tmp[15 : 0].bf16 + src[15 : 0].bf16;
MEM[ADDR] = dst.b32;
RETURN_DATA = tmp
```

Notes

Floating-point addition handles NAN/INF/denorm.

FLAT_ATOMIC_SWAP_X2 96

Swap an unsigned 64-bit integer value in the data register with a location in the flat aperture. Store the original value from flat aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].b64;
MEM[ADDR].b64 = DATA.b64;
RETURN_DATA.b64 = tmp
```

FLAT_ATOMIC_CMPSWAP_X2 97

Compare two unsigned 64-bit integer values stored in the data comparison register and a location in the flat aperture. Modify the memory location with a value in the data source register iff the comparison is equal. Store the original value from flat aperture into a vector register iff the GLC bit is set.

NOTE: RETURN_DATA[2:3] is not modified.

```
tmp = MEM[ADDR].u64;
src = DATA[63 : 0].u64;
cmp = DATA[127 : 64].u64;
MEM[ADDR].u64 = tmp == cmp ? src : tmp;
RETURN_DATA.u64 = tmp
```

FLAT_ATOMIC_ADD_X2 98

Add two unsigned 64-bit integer values stored in the data register and a location in the flat aperture. Store the original value from flat aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u64;
MEM[ADDR].u64 += DATA.u64;
RETURN_DATA.u64 = tmp
```

FLAT_ATOMIC_SUB_X2 99

Subtract an unsigned 64-bit integer value stored in the data register from a value stored in a location in the flat aperture. Store the original value from flat aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u64;
MEM[ADDR].u64 -= DATA.u64;
RETURN_DATA.u64 = tmp
```

FLAT_ATOMIC_SMIN_X2 100

Select the minimum of two signed 64-bit integer inputs, given two values stored in the data register and a location in the flat aperture. Store the original value from flat aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].i64;
src = DATA.i64;
MEM[ADDR].i64 = src < tmp ? src : tmp;
RETURN_DATA.i64 = tmp
```

FLAT_ATOMIC_UMIN_X2 101

Select the minimum of two unsigned 64-bit integer inputs, given two values stored in the data register and a location in the flat aperture. Store the original value from flat aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u64;
src = DATA.u64;
MEM[ADDR].u64 = src < tmp ? src : tmp;
RETURN_DATA.u64 = tmp
```

FLAT_ATOMIC_SMAX_X2 102

Select the maximum of two signed 64-bit integer inputs, given two values stored in the data register and a location in the flat aperture. Store the original value from flat aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].i64;
src = DATA.i64;
MEM[ADDR].i64 = src >= tmp ? src : tmp;
RETURN_DATA.i64 = tmp
```

FLAT_ATOMIC_UMAX_X2 103

Select the maximum of two unsigned 64-bit integer inputs, given two values stored in the data register and a

location in the flat aperture. Store the original value from flat aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u64;
src = DATA.u64;
MEM[ADDR].u64 = src >= tmp ? src : tmp;
RETURN_DATA.u64 = tmp
```

FLAT_ATOMIC_AND_X2 104

Calculate bitwise AND given two unsigned 64-bit integer values stored in the data register and a location in the flat aperture. Store the original value from flat aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].b64;
MEM[ADDR].b64 = (tmp & DATA.b64);
RETURN_DATA.b64 = tmp
```

FLAT_ATOMIC_OR_X2 105

Calculate bitwise OR given two unsigned 64-bit integer values stored in the data register and a location in the flat aperture. Store the original value from flat aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].b64;
MEM[ADDR].b64 = (tmp | DATA.b64);
RETURN_DATA.b64 = tmp
```

FLAT_ATOMIC_XOR_X2 106

Calculate bitwise XOR given two unsigned 64-bit integer values stored in the data register and a location in the flat aperture. Store the original value from flat aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].b64;
MEM[ADDR].b64 = (tmp ^ DATA.b64);
RETURN_DATA.b64 = tmp
```

FLAT_ATOMIC_INC_X2 107

Increment an unsigned 64-bit integer value from a location in the flat aperture with wraparound to 0 if the value exceeds a value in the data register. Store the original value from flat aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u64;
src = DATA.u64;
MEM[ADDR].u64 = tmp >= src ? 0ULL : tmp + 1ULL;
RETURN_DATA.u64 = tmp
```

FLAT_ATOMIC_DEC_X2 108

Decrement an unsigned 64-bit integer value from a location in the flat aperture with wraparound to a value in the data register if the decrement yields a negative value. Store the original value from flat aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u64;
src = DATA.u64;
MEM[ADDR].u64 = ((tmp == 0ULL) || (tmp > src)) ? src : tmp - 1ULL;
RETURN_DATA.u64 = tmp
```

12.15.2. Scratch Instructions

Scratch instructions are like Flat, but assume all workitem addresses fall in scratch (private) space.

SCRATCH_LOAD_UBYTE 16

Load 8 bits of unsigned data from the scratch aperture, zero extend to 32 bits and store the result into a vector register.

```
VDATA.u32 = 32'U({ 24'0U, MEM[ADDR].u8 })
```

SCRATCH_LOAD_SBYTE 17

Load 8 bits of signed data from the scratch aperture, sign extend to 32 bits and store the result into a vector register.

```
VDATA.i32 = 32'I(signext(MEM[ADDR].i8))
```

SCRATCH_LOAD_USHORT 18

Load 16 bits of unsigned data from the scratch aperture, zero extend to 32 bits and store the result into a vector register.

```
VDATA.u32 = 32'U({ 16'0U, MEM[ADDR].u16 })
```

SCRATCH_LOAD_SSHORT 19

Load 16 bits of signed data from the scratch aperture, sign extend to 32 bits and store the result into a vector register.

```
VDATA.i32 = 32'I(signext(MEM[ADDR].i16))
```

SCRATCH_LOAD_DWORD 20

Load 32 bits of data from the scratch aperture into a vector register.

```
VDATA[31 : 0] = MEM[ADDR].b32
```

SCRATCH_LOAD_DWORDX2 21

Load 64 bits of data from the scratch aperture into a vector register.

```
VDATA[31 : 0] = MEM[ADDR].b32;
VDATA[63 : 32] = MEM[ADDR + 4U].b32
```

SCRATCH_LOAD_DWORDX3 22

Load 96 bits of data from the scratch aperture into a vector register.

```
VDATA[31 : 0] = MEM[ADDR].b32;
VDATA[63 : 32] = MEM[ADDR + 4U].b32;
VDATA[95 : 64] = MEM[ADDR + 8U].b32
```

SCRATCH_LOAD_DWORDX4 23

Load 128 bits of data from the scratch aperture into a vector register.

```
VDATA[31 : 0] = MEM[ADDR].b32;
VDATA[63 : 32] = MEM[ADDR + 4U].b32;
VDATA[95 : 64] = MEM[ADDR + 8U].b32;
VDATA[127 : 96] = MEM[ADDR + 12U].b32
```

SCRATCH_STORE_BYTE 24

Store 8 bits of data from a vector register into the scratch aperture.

```
MEM[ADDR].b8 = VDATA[7 : 0]
```

SCRATCH_STORE_BYTE_D16_HI 25

Store 8 bits of data from the high 16 bits of a 32-bit vector register into the scratch aperture.

```
MEM[ADDR].b8 = VDATA[23 : 16]
```

SCRATCH_STORE_SHORT 26

Store 16 bits of data from a vector register into the scratch aperture.

```
MEM[ADDR].b16 = VDATA[15 : 0]
```

SCRATCH_STORE_SHORT_D16_HI 27

Store 16 bits of data from the high 16 bits of a 32-bit vector register into the scratch aperture.

```
MEM[ADDR].b16 = VDATA[31 : 16]
```

SCRATCH_STORE_DWORD 28

Store 32 bits of data from vector input registers into the scratch aperture.

```
MEM[ADDR].b32 = VDATA[31 : 0]
```

SCRATCH_STORE_DWORDX2 29

Store 64 bits of data from vector input registers into the scratch aperture.

```
MEM[ADDR].b32 = VDATA[31 : 0];
MEM[ADDR + 4U].b32 = VDATA[63 : 32]
```

SCRATCH_STORE_DWORDX3 30

Store 96 bits of data from vector input registers into the scratch aperture.

```
MEM[ADDR].b32 = VDATA[31 : 0];
MEM[ADDR + 4U].b32 = VDATA[63 : 32];
MEM[ADDR + 8U].b32 = VDATA[95 : 64]
```

SCRATCH_STORE_DWORDX4 31

Store 128 bits of data from vector input registers into the scratch aperture.

```
MEM[ADDR].b32 = VDATA[31 : 0];
MEM[ADDR + 4U].b32 = VDATA[63 : 32];
MEM[ADDR + 8U].b32 = VDATA[95 : 64];
MEM[ADDR + 12U].b32 = VDATA[127 : 96]
```

SCRATCH_LOAD_UBYTE_D16 32

Load 8 bits of unsigned data from the scratch aperture, zero extend to 16 bits and store the result into the low 16 bits of a 32-bit vector register.

```
VDATA[15 : 0].u16 = 16'U({ 8'0U, MEM[ADDR].u8 });
// VDATA[31:16] is preserved.
```

SCRATCH_LOAD_UBYTE_D16_HI 33

Load 8 bits of unsigned data from the scratch aperture, zero extend to 16 bits and store the result into the high 16 bits of a 32-bit vector register.

```
VDATA[31 : 16].u16 = 16'U({ 8'0U, MEM[ADDR].u8 });
// VDATA[15:0] is preserved.
```

SCRATCH_LOAD_SBYTE_D16 34

Load 8 bits of signed data from the scratch aperture, sign extend to 16 bits and store the result into the low 16 bits of a 32-bit vector register.

```
VDATA[15 : 0].i16 = 16'I(signext(MEM[ADDR].i8));
// VDATA[31:16] is preserved.
```

SCRATCH_LOAD_SBYTE_D16_HI 35

Load 8 bits of signed data from the scratch aperture, sign extend to 16 bits and store the result into the high 16 bits of a 32-bit vector register.

```
VDATA[31 : 16].i16 = 16'I(signext(MEM[ADDR].i8));
// VDATA[15:0] is preserved.
```

SCRATCH_LOAD_SHORT_D16 36

Load 16 bits of unsigned data from the scratch aperture and store the result into the low 16 bits of a 32-bit vector register.

```
VDATA[15 : 0].b16 = MEM[ADDR].b16;
// VDATA[31:16] is preserved.
```

SCRATCH_LOAD_SHORT_D16_HI 37

Load 16 bits of unsigned data from the scratch aperture and store the result into the high 16 bits of a 32-bit vector register.

```
VDATA[31 : 16].b16 = MEM[ADDR].b16;
// VDATA[15:0] is preserved.
```

SCRATCH_LOAD_LDS_UBYTE 38

Load 8 bits of untyped data from the scratch aperture, zero extend to 32 bits and store the result into a data share.

SCRATCH_LOAD_LDS_SBYTE 39

Load 8 bits of untyped data from the scratch aperture, sign extend to 32 bits and store the result into a data share.

SCRATCH_LOAD_LDS_USHORT 40

Load 16 bits of untyped data from the scratch aperture, zero extend to 32 bits and store the result into a data share.

SCRATCH_LOAD_LDS_SSHORT 41

Load 16 bits of untyped data from the scratch aperture, sign extend to 32 bits and store the result into a data share.

SCRATCH_LOAD_LDS_DWORD 42

Load 32 bits of untyped data from the scratch aperture and store the result into a data share.

12.15.3. Global Instructions

Global instructions are like Flat, but assume all workitem addresses fall in global memory space.

GLOBAL_LOAD_UBYTE 16

Load 8 bits of unsigned data from the global aperture, zero extend to 32 bits and store the result into a vector register.

```
VDATA.u32 = 32'U({ 24'0U, MEM[ADDR].u8 })
```

GLOBAL_LOAD_SBYTE 17

Load 8 bits of signed data from the global aperture, sign extend to 32 bits and store the result into a vector register.

```
VDATA.i32 = 32'I(signext(MEM[ADDR].i8))
```

GLOBAL_LOAD_USHORT 18

Load 16 bits of unsigned data from the global aperture, zero extend to 32 bits and store the result into a vector register.

```
VDATA.u32 = 32'U({ 16'0U, MEM[ADDR].u16 })
```

GLOBAL_LOAD_SSHORT 19

Load 16 bits of signed data from the global aperture, sign extend to 32 bits and store the result into a vector register.

```
VDATA.i32 = 32'I(signext(MEM[ADDR].i16))
```

GLOBAL_LOAD_DWORD 20

Load 32 bits of data from the global aperture into a vector register.

```
VDATA[31 : 0] = MEM[ADDR].b32
```

GLOBAL_LOAD_DWORDX2 21

Load 64 bits of data from the global aperture into a vector register.

```
VDATA[31 : 0] = MEM[ADDR].b32;
VDATA[63 : 32] = MEM[ADDR + 4U].b32
```

GLOBAL_LOAD_DWORDX3 22

Load 96 bits of data from the global aperture into a vector register.

```
VDATA[31 : 0] = MEM[ADDR].b32;
VDATA[63 : 32] = MEM[ADDR + 4U].b32;
VDATA[95 : 64] = MEM[ADDR + 8U].b32
```

GLOBAL_LOAD_DWORDX4 23

Load 128 bits of data from the global aperture into a vector register.

```
VDATA[31 : 0] = MEM[ADDR].b32;
VDATA[63 : 32] = MEM[ADDR + 4U].b32;
VDATA[95 : 64] = MEM[ADDR + 8U].b32;
VDATA[127 : 96] = MEM[ADDR + 12U].b32
```

GLOBAL_STORE_BYTE 24

Store 8 bits of data from a vector register into the global aperture.

```
MEM[ADDR].b8 = VDATA[7 : 0]
```

GLOBAL_STORE_BYTE_D16_HI 25

Store 8 bits of data from the high 16 bits of a 32-bit vector register into the global aperture.

```
MEM[ADDR].b8 = VDATA[23 : 16]
```

GLOBAL_STORE_SHORT 26

Store 16 bits of data from a vector register into the global aperture.

```
MEM[ADDR].b16 = VDATA[15 : 0]
```

GLOBAL_STORE_SHORT_D16_HI 27

Store 16 bits of data from the high 16 bits of a 32-bit vector register into the global aperture.

```
MEM[ADDR].b16 = VDATA[31 : 16]
```

GLOBAL_STORE_DWORD 28

Store 32 bits of data from vector input registers into the global aperture.

```
MEM[ADDR].b32 = VDATA[31 : 0]
```

GLOBAL_STORE_DWORDX2 29

Store 64 bits of data from vector input registers into the global aperture.

```
MEM[ADDR].b32 = VDATA[31 : 0];
MEM[ADDR + 4U].b32 = VDATA[63 : 32]
```

GLOBAL_STORE_DWORDX3 30

Store 96 bits of data from vector input registers into the global aperture.

```
MEM[ADDR].b32 = VDATA[31 : 0];
MEM[ADDR + 4U].b32 = VDATA[63 : 32];
MEM[ADDR + 8U].b32 = VDATA[95 : 64]
```

GLOBAL_STORE_DWORDX4 31

Store 128 bits of data from vector input registers into the global aperture.

```
MEM[ADDR].b32 = VDATA[31 : 0];
MEM[ADDR + 4U].b32 = VDATA[63 : 32];
MEM[ADDR + 8U].b32 = VDATA[95 : 64];
MEM[ADDR + 12U].b32 = VDATA[127 : 96]
```

GLOBAL_LOAD_UBYTE_D16 32

Load 8 bits of unsigned data from the global aperture, zero extend to 16 bits and store the result into the low 16 bits of a 32-bit vector register.

```
VDATA[15 : 0].u16 = 16'U({ 8'0U, MEM[ADDR].u8 });
// VDATA[31:16] is preserved.
```

GLOBAL_LOAD_UBYTE_D16_HI 33

Load 8 bits of unsigned data from the global aperture, zero extend to 16 bits and store the result into the high 16 bits of a 32-bit vector register.

```
VDATA[31 : 16].u16 = 16'U({ 8'0U, MEM[ADDR].u8 });
// VDATA[15:0] is preserved.
```

GLOBAL_LOAD_SBYTE_D16 34

Load 8 bits of signed data from the global aperture, sign extend to 16 bits and store the result into the low 16 bits of a 32-bit vector register.

```
VDATA[15 : 0].i16 = 16'I(signext(MEM[ADDR].i8));
// VDATA[31:16] is preserved.
```

GLOBAL_LOAD_SBYTE_D16_HI 35

Load 8 bits of signed data from the global aperture, sign extend to 16 bits and store the result into the high 16 bits of a 32-bit vector register.

```
VDATA[31 : 16].i16 = 16'I(signext(MEM[ADDR].i8));
// VDATA[15:0] is preserved.
```

GLOBAL_LOAD_SHORT_D16 36

Load 16 bits of unsigned data from the global aperture and store the result into the low 16 bits of a 32-bit vector register.

```
VDATA[15 : 0].b16 = MEM[ADDR].b16;
// VDATA[31:16] is preserved.
```

GLOBAL_LOAD_SHORT_D16_HI 37

Load 16 bits of unsigned data from the global aperture and store the result into the high 16 bits of a 32-bit vector register.

```
VDATA[31 : 16].b16 = MEM[ADDR].b16;
// VDATA[15:0] is preserved.
```

GLOBAL_LOAD_LDS_UBYTE 38

Load 8 bits of untyped data from the global aperture, zero extend to 32 bits and store the result into a data share.

GLOBAL_LOAD_LDS_SBYTE 39

Load 8 bits of untyped data from the global aperture, sign extend to 32 bits and store the result into a data share.

GLOBAL_LOAD_LDS_USHORT 40

Load 16 bits of untyped data from the global aperture, zero extend to 32 bits and store the result into a data share.

GLOBAL_LOAD_LDS_SSHORT 41

Load 16 bits of untyped data from the global aperture, sign extend to 32 bits and store the result into a data share.

GLOBAL_LOAD_LDS_DWORD 42

Load 32 bits of untyped data from the global aperture and store the result into a data share.

GLOBAL_ATOMIC_SWAP 64

Swap an unsigned 32-bit integer value in the data register with a location in the global aperture. Store the original value from global aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].b32;
MEM[ADDR].b32 = DATA.b32;
RETURN_DATA.b32 = tmp
```

GLOBAL_ATOMIC_CMPSWAP 65

Compare two unsigned 32-bit integer values stored in the data comparison register and a location in the global aperture. Modify the memory location with a value in the data source register iff the comparison is equal. Store the original value from global aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u32;
src = DATA[31 : 0].u32;
cmp = DATA[63 : 32].u32;
MEM[ADDR].u32 = tmp == cmp ? src : tmp;
RETURN_DATA.u32 = tmp
```

GLOBAL_ATOMIC_ADD 66

Add two unsigned 32-bit integer values stored in the data register and a location in the global aperture. Store the original value from global aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u32;
MEM[ADDR].u32 += DATA.u32;
RETURN_DATA.u32 = tmp
```

GLOBAL_ATOMIC_SUB 67

Subtract an unsigned 32-bit integer value stored in the data register from a value stored in a location in the global aperture. Store the original value from global aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u32;
```

```
MEM[ADDR].u32 -= DATA.u32;
RETURN_DATA.u32 = tmp
```

GLOBAL_ATOMIC_SMIN 68

Select the minimum of two signed 32-bit integer inputs, given two values stored in the data register and a location in the global aperture. Store the original value from global aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].i32;
src = DATA.i32;
MEM[ADDR].i32 = src < tmp ? src : tmp;
RETURN_DATA.i32 = tmp
```

GLOBAL_ATOMIC_UMIN 69

Select the minimum of two unsigned 32-bit integer inputs, given two values stored in the data register and a location in the global aperture. Store the original value from global aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u32;
src = DATA.u32;
MEM[ADDR].u32 = src < tmp ? src : tmp;
RETURN_DATA.u32 = tmp
```

GLOBAL_ATOMIC_SMAX 70

Select the maximum of two signed 32-bit integer inputs, given two values stored in the data register and a location in the global aperture. Store the original value from global aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].i32;
src = DATA.i32;
MEM[ADDR].i32 = src >= tmp ? src : tmp;
RETURN_DATA.i32 = tmp
```

GLOBAL_ATOMIC_UMAX 71

Select the maximum of two unsigned 32-bit integer inputs, given two values stored in the data register and a location in the global aperture. Store the original value from global aperture into a vector register iff the GLC

bit is set.

```
tmp = MEM[ADDR].u32;
src = DATA.u32;
MEM[ADDR].u32 = src >= tmp ? src : tmp;
RETURN_DATA.u32 = tmp
```

GLOBAL_ATOMIC_AND 72

Calculate bitwise AND given two unsigned 32-bit integer values stored in the data register and a location in the global aperture. Store the original value from global aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].b32;
MEM[ADDR].b32 = (tmp & DATA.b32);
RETURN_DATA.b32 = tmp
```

GLOBAL_ATOMIC_OR 73

Calculate bitwise OR given two unsigned 32-bit integer values stored in the data register and a location in the global aperture. Store the original value from global aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].b32;
MEM[ADDR].b32 = (tmp | DATA.b32);
RETURN_DATA.b32 = tmp
```

GLOBAL_ATOMIC_XOR 74

Calculate bitwise XOR given two unsigned 32-bit integer values stored in the data register and a location in the global aperture. Store the original value from global aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].b32;
MEM[ADDR].b32 = (tmp ^ DATA.b32);
RETURN_DATA.b32 = tmp
```

GLOBAL_ATOMIC_INC 75

Increment an unsigned 32-bit integer value from a location in the global aperture with wraparound to 0 if the value exceeds a value in the data register. Store the original value from global aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u32;
src = DATA.u32;
MEM[ADDR].u32 = tmp >= src ? 0U : tmp + 1U;
RETURN_DATA.u32 = tmp
```

GLOBAL_ATOMIC_DEC 76

Decrement an unsigned 32-bit integer value from a location in the global aperture with wraparound to a value in the data register if the decrement yields a negative value. Store the original value from global aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u32;
src = DATA.u32;
MEM[ADDR].u32 = ((tmp == 0U) || (tmp > src)) ? src : tmp - 1U;
RETURN_DATA.u32 = tmp
```

GLOBAL_ATOMIC_ADD_F32 77

Add a single-precision float value in the data register to a location in the global aperture. Store the original value from global aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].f32;
MEM[ADDR].f32 += DATA.f32;
RETURN_DATA = tmp
```

Notes

Floating-point addition handles NAN/INF/denorm.

GLOBAL_ATOMIC_PK_ADD_F16 78

Add a packed 2-component half-precision float value in the data register to a location in the global aperture. Store the original value from global aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR];
src = DATA;
dst[31 : 16].f16 = tmp[31 : 16].f16 + src[31 : 16].f16;
dst[15 : 0].f16 = tmp[15 : 0].f16 + src[15 : 0].f16;
MEM[ADDR] = dst.b32;
RETURN_DATA = tmp
```

Notes

Floating-point addition handles NAN/INF/denorm.

GLOBAL_ATOMIC_ADD_F64 79

Add a double-precision float value in the data register to a location in the global aperture. Store the original value from global aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].f64;
MEM[ADDR].f64 += DATA.f64;
RETURN_DATA = tmp
```

Notes

Floating-point addition handles NAN/INF/denorm.

GLOBAL_ATOMIC_MIN_F64 80

Select the minimum signed integer value given the data register and a location in the global aperture. Store the original value from global aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].f64;
src = DATA.f64;
MEM[ADDR].f64 = src < tmp ? src : tmp;
RETURN_DATA.f64 = tmp
```

GLOBAL_ATOMIC_MAX_F64 81

Select the maximum signed integer value given the data register and a location in the global aperture. Store the original value from global aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].f64;
src = DATA.f64;
MEM[ADDR].f64 = src > tmp ? src : tmp;
RETURN_DATA.f64 = tmp
```

GLOBAL_ATOMIC_PK_ADD_BF16 82

Add a packed 2-component BF16 float value in the data register to a location in the global aperture. Store the original value from global aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR];
src = DATA;
dst[31 : 16].bf16 = tmp[31 : 16].bf16 + src[31 : 16].bf16;
dst[15 : 0].bf16 = tmp[15 : 0].bf16 + src[15 : 0].bf16;
MEM[ADDR] = dst.b32;
RETURN_DATA = tmp
```

Notes

Floating-point addition handles NAN/INF/denorm.

GLOBAL_ATOMIC_SWAP_X2 96

Swap an unsigned 64-bit integer value in the data register with a location in the global aperture. Store the original value from global aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].b64;
MEM[ADDR].b64 = DATA.b64;
RETURN_DATA.b64 = tmp
```

GLOBAL_ATOMIC_CMPSWAP_X2 97

Compare two unsigned 64-bit integer values stored in the data comparison register and a location in the global aperture. Modify the memory location with a value in the data source register iff the comparison is equal. Store the original value from global aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u64;
src = DATA[63 : 0].u64;
cmp = DATA[127 : 64].u64;
MEM[ADDR].u64 = tmp == cmp ? src : tmp;
RETURN_DATA.u64 = tmp
```

GLOBAL_ATOMIC_ADD_X2 98

Add two unsigned 64-bit integer values stored in the data register and a location in the global aperture. Store the original value from global aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u64;
MEM[ADDR].u64 += DATA.u64;
RETURN_DATA.u64 = tmp
```

GLOBAL_ATOMIC_SUB_X2 99

Subtract an unsigned 64-bit integer value stored in the data register from a value stored in a location in the global aperture. Store the original value from global aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u64;
MEM[ADDR].u64 -= DATA.u64;
RETURN_DATA.u64 = tmp
```

GLOBAL_ATOMIC_SMIN_X2 100

Select the minimum of two signed 64-bit integer inputs, given two values stored in the data register and a location in the global aperture. Store the original value from global aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].i64;
src = DATA.i64;
MEM[ADDR].i64 = src < tmp ? src : tmp;
RETURN_DATA.i64 = tmp
```

GLOBAL_ATOMIC_UMIN_X2 101

Select the minimum of two unsigned 64-bit integer inputs, given two values stored in the data register and a location in the global aperture. Store the original value from global aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u64;
src = DATA.u64;
MEM[ADDR].u64 = src < tmp ? src : tmp;
RETURN_DATA.u64 = tmp
```

GLOBAL_ATOMIC_SMAX_X2 102

Select the maximum of two signed 64-bit integer inputs, given two values stored in the data register and a location in the global aperture. Store the original value from global aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].i64;
src = DATA.i64;
MEM[ADDR].i64 = src >= tmp ? src : tmp;
RETURN_DATA.i64 = tmp
```

GLOBAL_ATOMIC_UMAX_X2 103

Select the maximum of two unsigned 64-bit integer inputs, given two values stored in the data register and a location in the global aperture. Store the original value from global aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u64;
src = DATA.u64;
MEM[ADDR].u64 = src >= tmp ? src : tmp;
RETURN_DATA.u64 = tmp
```

GLOBAL_ATOMIC_AND_X2 104

Calculate bitwise AND given two unsigned 64-bit integer values stored in the data register and a location in the global aperture. Store the original value from global aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].b64;
MEM[ADDR].b64 = (tmp & DATA.b64);
RETURN_DATA.b64 = tmp
```

GLOBAL_ATOMIC_OR_X2 105

Calculate bitwise OR given two unsigned 64-bit integer values stored in the data register and a location in the global aperture. Store the original value from global aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].b64;
MEM[ADDR].b64 = (tmp | DATA.b64);
RETURN_DATA.b64 = tmp
```

GLOBAL_ATOMIC_XOR_X2 106

Calculate bitwise XOR given two unsigned 64-bit integer values stored in the data register and a location in the global aperture. Store the original value from global aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].b64;
MEM[ADDR].b64 = (tmp ^ DATA.b64);
RETURN_DATA.b64 = tmp
```

GLOBAL_ATOMIC_INC_X2 107

Increment an unsigned 64-bit integer value from a location in the global aperture with wraparound to 0 if the value exceeds a value in the data register. Store the original value from global aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u64;
src = DATA.u64;
MEM[ADDR].u64 = tmp >= src ? 0ULL : tmp + 1ULL;
RETURN_DATA.u64 = tmp
```

GLOBAL_ATOMIC_DEC_X2 108

Decrement an unsigned 64-bit integer value from a location in the global aperture with wraparound to a value in the data register if the decrement yields a negative value. Store the original value from global aperture into a vector register iff the GLC bit is set.

```
tmp = MEM[ADDR].u64;
src = DATA.u64;
MEM[ADDR].u64 = ((tmp == 0ULL) || (tmp > src)) ? src : tmp - 1ULL;
RETURN_DATA.u64 = tmp
```

12.16. Instruction Limitations

12.16.1. DPP

The following instructions cannot use DPP:

- V_MADMK_F32
- V_MADAK_F32
- V_MADMK_F16
- V_MADAK_F16
- V_READFIRSTLANE_B32
- V_CVT_I32_F64
- V_CVT_F64_I32
- V_CVT_F32_F64
- V_CVT_F64_F32
- V_CVT_U32_F64
- V_CVT_F64_U32
- V_TRUNC_F64
- V_CEIL_F64
- V_RNDNE_F64
- V_FLOOR_F64
- V_RCP_F64
- V_RSQ_F64
- V_SQRT_F64
- V_FREXP_EXP_I32_F64
- V_FREXP_MANT_F64
- V_FRACT_F64
- V_CLREXCP
- V_SWAP_B32
- V_CMP_CLASS_F64
- V_CMPX_CLASS_F64
- V_CMP_*_F64
- V_CMPX_*_F64
- V_CMP_*_I64
- V_CMP_*_U64
- V_CMPX_*_I64
- V_CMPX_*_U64

12.16.2. SDWA

The following instructions cannot use SDWA:

- V_MAC_F32
- V_MADMK_F32

- V_MADAK_F32
- V_MAC_F16
- V_MADMK_F16
- V_MADAK_F16
- V_FMAC_F32
- V_READFIRSTLANE_B32
- V_CLREXCP
- V_SWAP_B32

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 CDNA 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.

The table below summarizes the microcode formats and their widths. The sections that follow provide details

Table 63. Summary of Microcode Formats

Microcode Formats	Reference	Width (bits)
Scalar ALU and Control Formats		
SOP2	SOP2	32
SOP1	SOP1	
SOPK	SOPK	
SOPP	SOPP	
SOPC	SOPC	
Scalar Memory Format		
SMEM	SMEM	64
Vector ALU Format		
VOP1	VOP1	32
VOP2	VOP2	32
VOPC	VOPC	32
VOP3A	VOP3A	64
VOP3B	VOP3B	64
VOP3P	VOP3P	64
VOP3P-MAI	VOP3P-MAI	64
DPP	DPP	32
SDWA	VOP2	32
LDS/GWS Format		
DS	DS	64
Vector Memory Buffer Formats		
MTBUF	MTBUF	64
MUBUF	MUBUF	64
Flat Formats		
FLAT	FLAT	64
GLOBAL	GLOBAL	64
SCRATCH	SCRATCH	64

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 27 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.

Instruction Suffixes

Most instructions include a suffix which indicates the data type the instruction handles. This suffix may also include a number which indicate the size of the data.

For example: "F32" indicates "32-bit floating point data", or "B16" is "16-bit binary data".

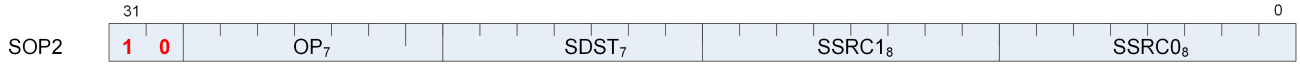
- B = binary
- F = floating point
- U = unsigned integer
- S = signed integer

When more than one data-type specifier occurs in an instruction, the last one is the result type and size, and the earlier one(s) is/are input data type and size.

13.1. Scalar ALU and Control Formats

13.1.1. SOP2

Scalar format with 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.

Table 64. SOP2 Fields

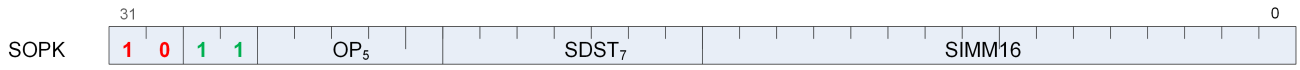
Field Name	Bits	Format or Description
SSRC0	[7:0]	Source 0. First operand for the instruction.
	0 - 101	SGPR0 to SGPR101: Scalar general-purpose registers.
	102	FLAT_SCRATCH_LO.
	103	FLAT_SCRATCH_HI.
	104	XNACK_MASK_LO.
	105	XNACK_MASK_HI.
	106	VCC_LO: vcc[31:0].
	107	VCC_HI: vcc[63:32].
	108-123	TTMP0 - TTMP15: Trap handler temporary register.
	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.
	209-234	Reserved.
	235	SHARED_BASE (Memory Aperture definition).
	236	SHARED_LIMIT (Memory Aperture definition).
	237	PRIVATE_BASE (Memory Aperture definition).
	238	PRIVATE_LIMIT (Memory Aperture definition).
	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	1/(2*PI).
	249 - 250	Reserved.
	251	VCCZ.
	252	EXECZ.
	253	SCC.
	254	Reserved.
	255	Literal constant.
SSRC1	[15:8]	Second scalar source operand. Same codes as SSRC0, above.
SDST	[22:16]	Scalar destination. Same codes as SSRC0, above except only codes 0-127 are valid.
OP	[29:23]	See Opcode table below.
ENCODING	[31:30]	Must be: 10

Table 65. SOP2 Opcodes

Opcode #	Name	Opcode #	Name
0	S_ADD_U32	26	S_XNOR_B32
1	S_SUB_U32	27	S_XNOR_B64
2	S_ADD_I32	28	S_LSHL_B32
3	S_SUB_I32	29	S_LSHL_B64
4	S_ADDC_U32	30	S_LSHR_B32
5	S_SUBB_U32	31	S_LSHR_B64
6	S_MIN_I32	32	S_ASHR_I32

Opcode #	Name	Opcode #	Name
7	S_MIN_U32	33	S_ASHR_I64
8	S_MAX_I32	34	S_BFM_B32
9	S_MAX_U32	35	S_BFM_B64
10	S_CSELECT_B32	36	S_MUL_I32
11	S_CSELECT_B64	37	S_BFE_U32
12	S_AND_B32	38	S_BFE_I32
13	S_AND_B64	39	S_BFE_U64
14	S_OR_B32	40	S_BFE_I64
15	S_OR_B64	41	S_CBRANCH_G_FORK
16	S_XOR_B32	42	S_ABSDIFF_I32
17	S_XOR_B64	44	S_MUL_HI_U32
18	S_ANDN2_B32	45	S_MUL_HI_I32
19	S_ANDN2_B64	46	S_LSHL1_ADD_U32
20	S_ORN2_B32	47	S_LSHL2_ADD_U32
21	S_ORN2_B64	48	S_LSHL3_ADD_U32
22	S_NAND_B32	49	S_LSHL4_ADD_U32
23	S_NAND_B64	50	S_PACK_LL_B32_B16
24	S_NOR_B32	51	S_PACK_LH_B32_B16
25	S_NOR_B64	52	S_PACK_HH_B32_B16

13.1.2. SOPK



Format SOPK

Description This is a scalar instruction with one 16-bit signed immediate (SIMM16) input and a single destination. Instructions which take 2 inputs use the destination as the second input.

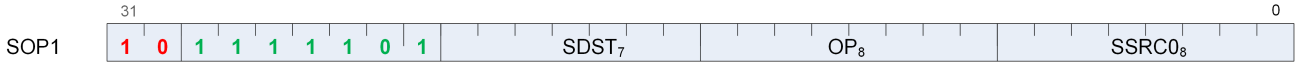
Table 66. SOPK Fields

Field Name	Bits	Format or Description
SIMM16	[15:0]	Signed immediate 16-bit value.
SDST	[22:16]	Scalar destination, and can provide second source operand.
	0 - 101	SGPR0 to SGPR101: Scalar general-purpose registers.
	102	FLAT_SCRATCH_LO.
	103	FLAT_SCRATCH_HI.
	104	XNACK_MASK_LO.
	105	XNACK_MASK_HI.
	106	VCC_LO: vcc[31:0].
	107	VCC_HI: vcc[63:32].
	108-123	TTMP0 - TTMP15: Trap handler temporary register.
	124	M0. Memory register 0.
	125	Reserved
	126	EXEC_LO: exec[31:0].
	127	EXEC_HI: exec[63:32].
OP	[27:23]	See Opcode table below.
ENCODING	[31:28]	Must be: 1011

Table 67. SOPK Opcodes

Opcode #	Name	Opcode #	Name
0	S_MOVK_I32	11	S_CMPK_GE_U32
1	S_CMOVK_I32	12	S_CMPK_LT_U32
2	S_CMPK_EQ_I32	13	S_CMPK_LE_U32
3	S_CMPK_LG_I32	14	S_ADDK_I32
4	S_CMPK_GT_I32	15	S_MULK_I32
5	S_CMPK_GE_I32	16	S_CBRANCH_I_FORK
6	S_CMPK_LT_I32	17	S_GETREG_B32
7	S_CMPK_LE_I32	18	S_SETREG_B32
8	S_CMPK_EQ_U32	20	S_SETREG_IMM32_B32
9	S_CMPK_LG_U32	21	S_CALL_B64
10	S_CMPK_GT_U32		

13.1.3. SOP1



Format SOP1

Description This is a scalar instruction with two inputs and one output. Can be followed by a 32-bit literal constant.

Table 68. SOP1 Fields

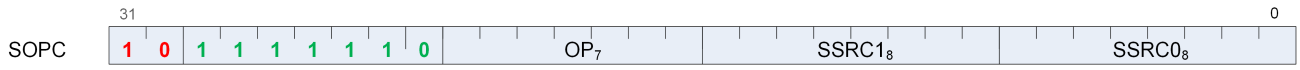
Field Name	Bits	Format or Description
SSRC0	[7:0]	Source 0. First operand for the instruction.
	0 - 101	SGPR0 to SGPR101: Scalar general-purpose registers.
	102	FLAT_SCRATCH_LO.
	103	FLAT_SCRATCH_HI.
	104	XNACK_MASK_LO.
	105	XNACK_MASK_HI.
	106	VCC_LO: vcc[31:0].
	107	VCC_HI: vcc[63:32].
	108-123	TTMP0 - TTMP15: Trap handler temporary register.
	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.
	209-234	Reserved.
	235	SHARED_BASE (Memory Aperture definition).
	236	SHARED_LIMIT (Memory Aperture definition).
	237	PRIVATE_BASE (Memory Aperture definition).
	238	PRIVATE_LIMIT (Memory Aperture definition).
	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	1/(2*PI).
	249 - 250	Reserved.
	251	VCCZ.
	252	EXECZ.
	253	SCC.
	254	Reserved.
	255	Literal constant.
OP	[15:8]	See Opcode table below.
SDST	[22:16]	Scalar destination. Same codes as SSRC0, above except only codes 0-127 are valid.
ENCODING	[31:23]	Must be: 10_1111101

Table 69. SOP1 Opcodes

Opcode #	Name	Opcode #	Name
0	S_MOV_B32	27	S_BITSET1_B64
1	S_MOV_B64	28	S_GETPC_B64
2	S_CMOV_B32	29	S_SETPC_B64
3	S_CMOV_B64	30	S_SWAPPC_B64
4	S_NOT_B32	31	S_RFE_B64
5	S_NOT_B64	32	S_AND_SAVEEXEC_B64
6	S_WQM_B32	33	S_OR_SAVEEXEC_B64
7	S_WQM_B64	34	S_XOR_SAVEEXEC_B64

Opcode #	Name	Opcode #	Name
8	S_BREV_B32	35	S_ANDN2_SAVEEXEC_B64
9	S_BREV_B64	36	S_ORN2_SAVEEXEC_B64
10	S_BCNT0_I32_B32	37	S_NAND_SAVEEXEC_B64
11	S_BCNT0_I32_B64	38	S_NOR_SAVEEXEC_B64
12	S_BCNT1_I32_B32	39	S_XNOR_SAVEEXEC_B64
13	S_BCNT1_I32_B64	40	S_QUADMASK_B32
14	S_FF0_I32_B32	41	S_QUADMASK_B64
15	S_FF0_I32_B64	42	S_MOVRELS_B32
16	S_FF1_I32_B32	43	S_MOVRELS_B64
17	S_FF1_I32_B64	44	S_MOVRELD_B32
18	S_FLBIT_I32_B32	45	S_MOVRELD_B64
19	S_FLBIT_I32_B64	46	S_CBRANCH_JOIN
20	S_FLBIT_I32	48	S_ABS_I32
21	S_FLBIT_I32_I64	50	S_SET_GPR_IDX_IDX
22	S_SEXT_I32_I8	51	S_ANDN1_SAVEEXEC_B64
23	S_SEXT_I32_I16	52	S_ORN1_SAVEEXEC_B64
24	S_BITSET0_B32	53	S_ANDN1_WREXEC_B64
25	S_BITSET0_B64	54	S_ANDN2_WREXEC_B64
26	S_BITSET1_B32	55	S_BITREPLICATE_B64_B32

13.1.4. SOPC



Format SOPC

Description This is a scalar instruction with two inputs which are compared and produces SCC as a result. Can be followed by a 32-bit literal constant.

Table 70. SOPC Fields

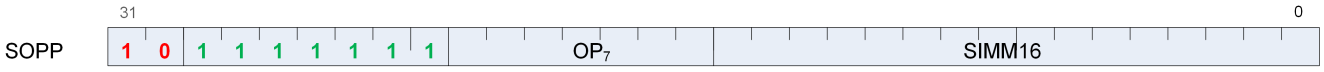
Field Name	Bits	Format or Description
SSRC0	[7:0]	Source 0. First operand for the instruction.
	0 - 101	SGPR0 to SGPR101: Scalar general-purpose registers.
	102	FLAT_SCRATCH_LO.
	103	FLAT_SCRATCH_HI.
	104	XNACK_MASK_LO.
	105	XNACK_MASK_HI.
	106	VCC_LO: vcc[31:0].
	107	VCC_HI: vcc[63:32].
	108-123	TTMP0 - TTMP15: Trap handler temporary register.
	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.
	209-234	Reserved.
	235	SHARED_BASE (Memory Aperture definition).
	236	SHARED_LIMIT (Memory Aperture definition).
	237	PRIVATE_BASE (Memory Aperture definition).
	238	PRIVATE_LIMIT (Memory Aperture definition).
	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	1/(2*PI).
	249 - 250	Reserved.
	251	VCCZ.
	252	EXECZ.
	253	SCC.
	254	Reserved.
	255	Literal constant.
SSRC1	[15:8]	Second scalar source operand. Same codes as SSRC0, above.
OP	[22:16]	See Opcode table below.
ENCODING	[31:23]	Must be: 10_1111110

Table 71. SOPC Opcodes

Opcode #	Name	Opcode #	Name
0	S_CMP_EQ_I32	10	S_CMP_LT_U32
1	S_CMP_LG_I32	11	S_CMP_LE_U32
2	S_CMP_GT_I32	12	S_BITCMP0_B32
3	S_CMP_GE_I32	13	S_BITCMP1_B32
4	S_CMP_LT_I32	14	S_BITCMP0_B64
5	S_CMP_LE_I32	15	S_BITCMP1_B64
6	S_CMP_EQ_U32	16	S_SETVSKIP
7	S_CMP_LG_U32	17	S_SET_GPR_IDX_ON

Opcode #	Name	Opcode #	Name
8	S_CMP_GT_U32	18	S_CMP_EQ_U64
9	S_CMP_GE_U32	19	S_CMP_LG_U64

13.1.5. SOPP



Format SOPP

Description This is a scalar instruction with one 16-bit signed immediate (SIMM16) input.

Table 72. SOPP Fields

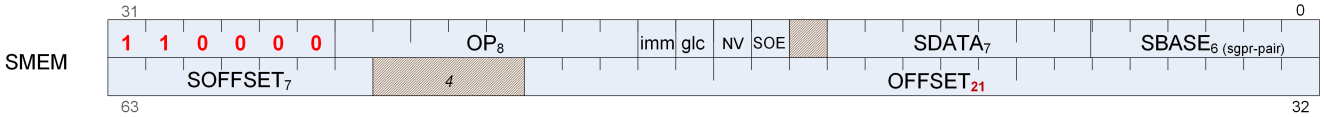
Field Name	Bits	Format or Description
SIMM16	[15:0]	Signed immediate 16-bit value.
OP	[22:16]	See Opcode table below.
ENCODING	[31:23]	Must be: 10_1111111

Table 73. SOPP Opcodes

Opcode #	Name	Opcode #	Name
0	S_NOP	15	S_SETPRIO
1	S_ENDPGM	16	S_SENDMSG
2	S_BRANCH	17	S_SENDMSGHALT
3	S_WAKEUP	18	S_TRAP
4	S_CBRANCH_SCC0	19	S_ICACHE_INV
5	S_CBRANCH_SCC1	20	S_INCPERFLEVEL
6	S_CBRANCH_VCCZ	21	S_DECPERFLEVEL
7	S_CBRANCH_VCCNZ	23	S_CBRANCH_CDBGSYS
8	S_CBRANCH_EXECZ	24	S_CBRANCH_CDBGUSER
9	S_CBRANCH_EXECNZ	25	S_CBRANCH_CDBGSYS_OR_USER
10	S_BARRIER	26	S_CBRANCH_CDBGSYS_AND_USER
11	S_SETKILL	27	S_ENDPGM_SAVED
12	S_WAITCNT	28	S_SET_GPR_IDX_OFF
13	S_SETHALT	29	S_SET_GPR_IDX_MODE
14	S_SLEEP		

13.2. Scalar Memory Format

13.2.1. SMEM



Format SMEM

Description Scalar Memory data load/store

Table 74. SMEM Fields

Field Name	Bits	Format or Description
SBASE	[5:0]	SGPR-pair which provides base address or SGPR-quad which provides V#. (LSB of SGPR address is omitted).
SDATA	[12:6]	SGPR which provides write data or accepts return data.
SOE	[14]	Scalar offset enable.
NV	[15]	Non-volatile
GLC	[16]	Globally memory Coherent. Force bypass of L1 and L2 cache, or for atomics, cause pre-op value to be returned.
IMM	[17]	Immediate enable.
OP	[25:18]	See Opcode table below.
ENCODING	[31:26]	Must be: 110000
OFFSET	[52:32]	An immediate signed byte offset, or the address of an SGPR holding the unsigned byte offset. Signed offsets only work with S_LOAD/STORE.
SOFFSET	[63:57]	SGPR offset. Used only when SOFFSET_EN = 1 May only specify an SGPR or M0.

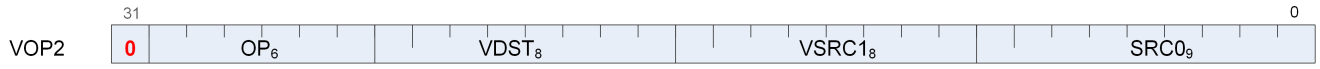
Table 75. SMEM Opcodes

Opcode #	Name	Opcode #	Name
0	S_LOAD_DWORD	75	S_BUFFER_ATOMIC_INC
1	S_LOAD_DWORDX2	76	S_BUFFER_ATOMIC_DEC
2	S_LOAD_DWORDX4	96	S_BUFFER_ATOMIC_SWAP_X2
3	S_LOAD_DWORDX8	97	S_BUFFER_ATOMIC_CMPSWAP_X2
4	S_LOAD_DWORDX16	98	S_BUFFER_ATOMIC_ADD_X2
5	S_SCRATCH_LOAD_DWORD	99	S_BUFFER_ATOMIC_SUB_X2
6	S_SCRATCH_LOAD_DWORDX2	100	S_BUFFER_ATOMIC_SMIN_X2
7	S_SCRATCH_LOAD_DWORDX4	101	S_BUFFER_ATOMIC_UMIN_X2
8	S_BUFFER_LOAD_DWORD	102	S_BUFFER_ATOMIC_SMAX_X2
9	S_BUFFER_LOAD_DWORDX2	103	S_BUFFER_ATOMIC_UMAX_X2
10	S_BUFFER_LOAD_DWORDX4	104	S_BUFFER_ATOMIC_AND_X2
11	S_BUFFER_LOAD_DWORDX8	105	S_BUFFER_ATOMIC_OR_X2
12	S_BUFFER_LOAD_DWORDX16	106	S_BUFFER_ATOMIC_XOR_X2
16	S_STORE_DWORD	107	S_BUFFER_ATOMIC_INC_X2
17	S_STORE_DWORDX2	108	S_BUFFER_ATOMIC_DEC_X2
18	S_STORE_DWORDX4	128	S_ATOMIC_SWAP
21	S_SCRATCH_STORE_DWORD	129	S_ATOMIC_CMPSWAP
22	S_SCRATCH_STORE_DWORDX2	130	S_ATOMIC_ADD
23	S_SCRATCH_STORE_DWORDX4	131	S_ATOMIC_SUB
24	S_BUFFER_STORE_DWORD	132	S_ATOMIC_SMIN
25	S_BUFFER_STORE_DWORDX2	133	S_ATOMIC_UMIN
26	S_BUFFER_STORE_DWORDX4	134	S_ATOMIC_SMAX
32	S_DCACHE_INV	135	S_ATOMIC_UMAX
33	S_DCACHE_WB	136	S_ATOMIC_AND
34	S_DCACHE_INV_VOL	137	S_ATOMIC_OR
35	S_DCACHE_WB_VOL	138	S_ATOMIC_XOR
36	S_MEMTIME	139	S_ATOMIC_INC

Opcode #	Name	Opcode #	Name
37	S_MEMREALTIME	140	S_ATOMIC_DEC
40	S_DCACHE_DISCARD	160	S_ATOMIC_SWAP_X2
41	S_DCACHE_DISCARD_X2	161	S_ATOMIC_CMPSWAP_X2
64	S_BUFFER_ATOMIC_SWAP	162	S_ATOMIC_ADD_X2
65	S_BUFFER_ATOMIC_CMPSWAP	163	S_ATOMIC_SUB_X2
66	S_BUFFER_ATOMIC_ADD	164	S_ATOMIC_SMIN_X2
67	S_BUFFER_ATOMIC_SUB	165	S_ATOMIC_UMIN_X2
68	S_BUFFER_ATOMIC_SMIN	166	S_ATOMIC_SMAX_X2
69	S_BUFFER_ATOMIC_UMIN	167	S_ATOMIC_UMAX_X2
70	S_BUFFER_ATOMIC_SMAX	168	S_ATOMIC_AND_X2
71	S_BUFFER_ATOMIC_UMAX	169	S_ATOMIC_OR_X2
72	S_BUFFER_ATOMIC_AND	170	S_ATOMIC_XOR_X2
73	S_BUFFER_ATOMIC_OR	171	S_ATOMIC_INC_X2
74	S_BUFFER_ATOMIC_XOR	172	S_ATOMIC_DEC_X2

13.3. Vector ALU Formats

13.3.1. VOP2



Format VOP2

Description Vector ALU format with two operands

Table 76. VOP2 Fields

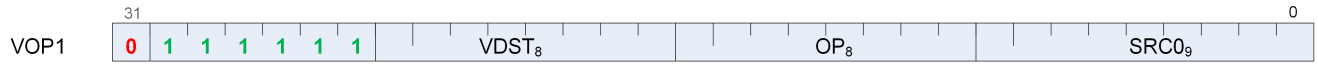
Field Name	Bits	Format or Description
SRC0	[8:0]	Source 0. First operand for the instruction.
	0 - 101	SGPR0 to SGPR101: Scalar general-purpose registers.
	102	FLAT_SCRATCH_LO.
	103	FLAT_SCRATCH_HI.
	104	XNACK_MASK_LO.
	105	XNACK_MASK_HI.
	106	VCC_LO: vcc[31:0].
	107	VCC_HI: vcc[63:32].
	108-123	TTMP0 - TTMP15: Trap handler temporary register.
	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.
	209-234	Reserved.
	235	SHARED_BASE (Memory Aperture definition).
	236	SHARED_LIMIT (Memory Aperture definition).
	237	PRIVATE_BASE (Memory Aperture definition).
	238	PRIVATE_LIMIT (Memory Aperture definition).
	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	1/(2*PI).
	249	SDWA
	250	DPP
	251	VCCZ.
	252	EXECZ.
	253	SCC.
	254	Reserved.
	255	Literal constant.
	256 - 511	VGPR 0 - 255
VSRC1	[16:9]	VGPR which provides the second operand.
VDST	[24:17]	Destination VGPR.
OP	[30:25]	See Opcode table below.
ENCODING	[31]	Must be: 0

Table 77. VOP2 Opcodes

Opcode #	Name	Opcode #	Name
0	V_CNDMASK_B32	32	V_SUB_F16
1	V_ADD_F32	33	V_SUBREV_F16
2	V_SUB_F32	34	V_MUL_F16
3	V_SUBREV_F32	35	V_MAC_F16
4	V_FMAC_F64	36	V_MADMK_F16
5	V_MUL_F32	37	V_MADAK_F16
6	V_MUL_I32_I24	38	V_ADD_U16

Opcode #	Name	Opcode #	Name
7	V_MUL_HI_I32_I24	39	V_SUB_U16
8	V_MUL_U32_U24	40	V_SUBREV_U16
9	V_MUL_HI_U32_U24	41	V_MUL_LO_U16
10	V_MIN_F32	42	V_LSHLREV_B16
11	V_MAX_F32	43	V_LSHRREV_B16
12	V_MIN_I32	44	V_ASHRREV_I16
13	V_MAX_I32	45	V_MAX_F16
14	V_MIN_U32	46	V_MIN_F16
15	V_MAX_U32	47	V_MAX_U16
16	V_LSHRREV_B32	48	V_MAX_I16
17	V_ASHRREV_I32	49	V_MIN_U16
18	V_LSHLREV_B32	50	V_MIN_I16
19	V_AND_B32	51	V_LDEXP_F16
20	V_OR_B32	52	V_ADD_U32
21	V_XOR_B32	53	V_SUB_U32
23	V_FMAMK_F32	54	V_SUBREV_U32
24	V_FMAAK_F32	55	V_DOT2C_F32_F16
25	V_ADD_CO_U32	56	V_DOT2C_I32_I16
26	V_SUB_CO_U32	57	V_DOT4C_I32_I8
27	V_SUBREV_CO_U32	58	V_DOT8C_I32_I4
28	V_ADDC_CO_U32	59	V_FMAC_F32
29	V_SUBB_CO_U32	60	V_PK_FMAC_F16
30	V_SUBBREV_CO_U32	61	V_XNOR_B32
31	V_ADD_F16		

13.3.2. VOP1



Format VOP1

Description Vector ALU format with one operand

Table 78. VOP1 Fields

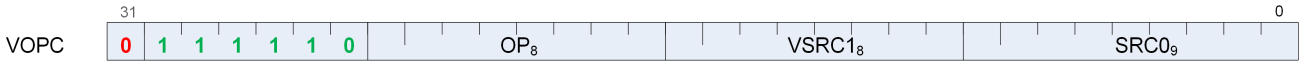
Field Name	Bits	Format or Description
SRC0	[8:0]	Source 0. First operand for the instruction.
	0 - 101	SGPR0 to SGPR101: Scalar general-purpose registers.
	102	FLAT_SCRATCH_LO.
	103	FLAT_SCRATCH_HI.
	104	XNACK_MASK_LO.
	105	XNACK_MASK_HI.
	106	VCC_LO: vcc[31:0].
	107	VCC_HI: vcc[63:32].
	108-123	TTMP0 - TTMP15: Trap handler temporary register.
	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.
	209-234	Reserved.
	235	SHARED_BASE (Memory Aperture definition).
	236	SHARED_LIMIT (Memory Aperture definition).
	237	PRIVATE_BASE (Memory Aperture definition).
	238	PRIVATE_LIMIT (Memory Aperture definition).
	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	1/(2*PI).
	249	SDWA
	250	DPP
	251	VCCZ.
	252	EXECZ.
	253	SCC.
	254	Reserved.
	255	Literal constant.
	256 - 511	VGPR 0 - 255
OP	[16:9]	See Opcode table below.
VDST	[24:17]	Destination VGPR.
ENCODING	[31:25]	Must be: 0_111111

Table 79. VOP1 Opcodes

Opcode #	Name	Opcode #	Name
0	V_NOP	42	V_COS_F32
1	V_MOV_B32	43	V_NOT_B32
2	V_READFIRSTLANE_B32	44	V_BFREV_B32
3	V_CVT_I32_F64	45	V_FFBH_U32
4	V_CVT_F64_I32	46	V_FFBL_B32
5	V_CVT_F32_I32	47	V_FFBH_I32
6	V_CVT_F32_U32	48	V_FREXP_EXP_I32_F64
7	V_CVT_U32_F32	49	V_FREXP_MANT_F64

Opcode #	Name	Opcode #	Name
8	V_CVT_I32_F32	50	V_FRACT_F64
10	V_CVT_F16_F32	51	V_FREXP_EXP_I32_F32
11	V_CVT_F32_F16	52	V_FREXP_MANT_F32
12	V_CVT_RPI_I32_F32	53	V_CLREXCP
13	V_CVT_FLR_I32_F32	56	V_MOV_B64
14	V_CVT_OFF_F32_I4	57	V_CVT_F16_U16
15	V_CVT_F32_F64	58	V_CVT_F16_I16
16	V_CVT_F64_F32	59	V_CVT_U16_F16
17	V_CVT_F32_UBYTE0	60	V_CVT_I16_F16
18	V_CVT_F32_UBYTE1	61	V_RCP_F16
19	V_CVT_F32_UBYTE2	62	V_SQRT_F16
20	V_CVT_F32_UBYTE3	63	V_RSQ_F16
21	V_CVT_U32_F64	64	V_LOG_F16
22	V_CVT_F64_U32	65	V_EXP_F16
23	V_TRUNC_F64	66	V_FREXP_MANT_F16
24	V_CEIL_F64	67	V_FREXP_EXP_I16_F16
25	V_RNDNE_F64	68	V_FLOOR_F16
26	V_FLOOR_F64	69	V_CEIL_F16
27	V_FRACT_F32	70	V_TRUNC_F16
28	V_TRUNC_F32	71	V_RNDNE_F16
29	V_CEIL_F32	72	V_FRACT_F16
30	V_RNDNE_F32	73	V_SIN_F16
31	V_FLOOR_F32	74	V_COS_F16
32	V_EXP_F32	77	V_CVT_NORM_I16_F16
33	V_LOG_F32	78	V_CVT_NORM_U16_F16
34	V_RCP_F32	79	V_SAT_PK_U8_I16
35	V_RCP_IFLAG_F32	81	V_SWAP_B32
36	V_RSQ_F32	82	V_ACCVGPR_MOV_B32
37	V_RCP_F64	84	V_CVT_F32_FP8
38	V_RSQ_F64	85	V_CVT_F32_BF8
39	V_SQRT_F32	86	V_CVT_PK_F32_FP8
40	V_SQRT_F64	87	V_CVT_PK_F32_BF8
41	V_SIN_F32		

13.3.3. VOPC



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 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, compare instruction have variants that also writes to the EXEC mask. The destination of the compare result is VCC when encoded using the VOPC format, and can be an arbitrary SGPR when encoded in the VOP3 format.

Comparison Operations

Table 80. Comparison Operations

Compare Operation	Opcode Offset	Description
Sixteen Compare Operations (OP16)		
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)		
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 81. VOPC Fields

Field Name	Bits	Format or Description
SRC0	[8:0]	Source 0. First operand for the instruction.
	0 - 101	SGPR0 to SGPR101: Scalar general-purpose registers.
	102	FLAT_SCRATCH_LO.
	103	FLAT_SCRATCH_HI.
	104	XNACK_MASK_LO.
	105	XNACK_MASK_HI.
	106	VCC_LO: vcc[31:0].
	107	VCC_HI: vcc[63:32].
	108-123	TTMP0 - TTMP15: Trap handler temporary register.
	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.
	209-234	Reserved.
	235	SHARED_BASE (Memory Aperture definition).
	236	SHARED_LIMIT (Memory Aperture definition).
	237	PRIVATE_BASE (Memory Aperture definition).
	238	PRIVATE_LIMIT (Memory Aperture definition).
	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	1/(2*PI).
	249	SDWA
	250	DPP
	251	VCCZ.
	252	EXECZ.
	253	SCC.
	254	Reserved.
	255	Literal constant.
	256 - 511	VGPR 0 - 255
VSRC1	[16:9]	VGPR which provides the second operand.
OP	[24:17]	See Opcode table below.
ENCODING	[31:25]	Must be: 0_111110

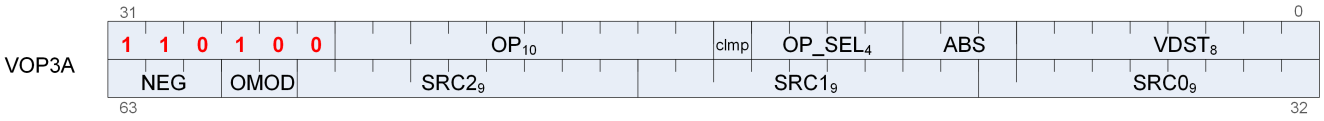
Table 82. VOPC Opcodes

Opcode #	Name	Opcode #	Name
16	V_CMP_CLASS_F32	125	V_CMPX_NEQ_F64
17	V_CMPX_CLASS_F32	126	V_CMPX_NLT_F64
18	V_CMP_CLASS_F64	127	V_CMPX_TRU_F64
19	V_CMPX_CLASS_F64	160	V_CMP_F_I16
20	V_CMP_CLASS_F16	161	V_CMP_LT_I16
21	V_CMPX_CLASS_F16	162	V_CMP_EQ_I16
32	V_CMP_F_F16	163	V_CMP_LE_I16
33	V_CMP_LT_F16	164	V_CMP_GT_I16

Opcode #	Name	Opcode #	Name
34	V_CMP_EQ_F16	165	V_CMP_NE_I16
35	V_CMP_LE_F16	166	V_CMP_GE_I16
36	V_CMP_GT_F16	167	V_CMP_T_I16
37	V_CMP_LG_F16	168	V_CMP_F_U16
38	V_CMP_GE_F16	169	V_CMP_LT_U16
39	V_CMP_O_F16	170	V_CMP_EQ_U16
40	V_CMP_U_F16	171	V_CMP_LE_U16
41	V_CMP_NGE_F16	172	V_CMP_GT_U16
42	V_CMP_NLG_F16	173	V_CMP_NE_U16
43	V_CMP_NGT_F16	174	V_CMP_GE_U16
44	V_CMP_NLE_F16	175	V_CMP_T_U16
45	V_CMP_NEQ_F16	176	V_CMPX_F_I16
46	V_CMP_NLT_F16	177	V_CMPX_LT_I16
47	V_CMP_TRU_F16	178	V_CMPX_EQ_I16
48	V_CMPX_F_F16	179	V_CMPX_LE_I16
49	V_CMPX_LT_F16	180	V_CMPX_GT_I16
50	V_CMPX_EQ_F16	181	V_CMPX_NE_I16
51	V_CMPX_LE_F16	182	V_CMPX_GE_I16
52	V_CMPX_GT_F16	183	V_CMPX_T_I16
53	V_CMPX_LG_F16	184	V_CMPX_F_U16
54	V_CMPX_GE_F16	185	V_CMPX_LT_U16
55	V_CMPX_O_F16	186	V_CMPX_EQ_U16
56	V_CMPX_U_F16	187	V_CMPX_LE_U16
57	V_CMPX_NGE_F16	188	V_CMPX_GT_U16
58	V_CMPX_NLG_F16	189	V_CMPX_NE_U16
59	V_CMPX_NGT_F16	190	V_CMPX_GE_U16
60	V_CMPX_NLE_F16	191	V_CMPX_T_U16
61	V_CMPX_NEQ_F16	192	V_CMP_F_I32
62	V_CMPX_NLT_F16	193	V_CMP_LT_I32
63	V_CMPX_TRU_F16	194	V_CMP_EQ_I32
64	V_CMP_F_F32	195	V_CMP_LE_I32
65	V_CMP_LT_F32	196	V_CMP_GT_I32
66	V_CMP_EQ_F32	197	V_CMP_NE_I32
67	V_CMP_LE_F32	198	V_CMP_GE_I32
68	V_CMP_GT_F32	199	V_CMP_T_I32
69	V_CMP_LG_F32	200	V_CMP_F_U32
70	V_CMP_GE_F32	201	V_CMP_LT_U32
71	V_CMP_O_F32	202	V_CMP_EQ_U32
72	V_CMP_U_F32	203	V_CMP_LE_U32
73	V_CMP_NGE_F32	204	V_CMP_GT_U32
74	V_CMP_NLG_F32	205	V_CMP_NE_U32
75	V_CMP_NGT_F32	206	V_CMP_GE_U32
76	V_CMP_NLE_F32	207	V_CMP_T_U32
77	V_CMP_NEQ_F32	208	V_CMPX_F_I32
78	V_CMP_NLT_F32	209	V_CMPX_LT_I32
79	V_CMP_TRU_F32	210	V_CMPX_EQ_I32
80	V_CMPX_F_F32	211	V_CMPX_LE_I32

Opcode #	Name	Opcode #	Name
81	V_CMPX_LT_F32	212	V_CMPX_GT_I32
82	V_CMPX_EQ_F32	213	V_CMPX_NE_I32
83	V_CMPX_LE_F32	214	V_CMPX_GE_I32
84	V_CMPX_GT_F32	215	V_CMPX_T_I32
85	V_CMPX_LG_F32	216	V_CMPX_F_U32
86	V_CMPX_GE_F32	217	V_CMPX_LT_U32
87	V_CMPX_O_F32	218	V_CMPX_EQ_U32
88	V_CMPX_U_F32	219	V_CMPX_LE_U32
89	V_CMPX_NGE_F32	220	V_CMPX_GT_U32
90	V_CMPX_NLG_F32	221	V_CMPX_NE_U32
91	V_CMPX_NGT_F32	222	V_CMPX_GE_U32
92	V_CMPX_NLE_F32	223	V_CMPX_T_U32
93	V_CMPX_NEQ_F32	224	V_CMP_F_I64
94	V_CMPX_NLT_F32	225	V_CMP_LT_I64
95	V_CMPX_TRU_F32	226	V_CMP_EQ_I64
96	V_CMP_F_F64	227	V_CMP_LE_I64
97	V_CMP_LT_F64	228	V_CMP_GT_I64
98	V_CMP_EQ_F64	229	V_CMP_NE_I64
99	V_CMP_LE_F64	230	V_CMP_GE_I64
100	V_CMP_GT_F64	231	V_CMP_T_I64
101	V_CMP_LG_F64	232	V_CMP_F_U64
102	V_CMP_GE_F64	233	V_CMP_LT_U64
103	V_CMP_O_F64	234	V_CMP_EQ_U64
104	V_CMP_U_F64	235	V_CMP_LE_U64
105	V_CMP_NGE_F64	236	V_CMP_GT_U64
106	V_CMP_NLG_F64	237	V_CMP_NE_U64
107	V_CMP_NGT_F64	238	V_CMP_GE_U64
108	V_CMP_NLE_F64	239	V_CMP_T_U64
109	V_CMP_NEQ_F64	240	V_CMPX_F_I64
110	V_CMP_NLT_F64	241	V_CMPX_LT_I64
111	V_CMP_TRU_F64	242	V_CMPX_EQ_I64
112	V_CMPX_F_F64	243	V_CMPX_LE_I64
113	V_CMPX_LT_F64	244	V_CMPX_GT_I64
114	V_CMPX_EQ_F64	245	V_CMPX_NE_I64
115	V_CMPX_LE_F64	246	V_CMPX_GE_I64
116	V_CMPX_GT_F64	247	V_CMPX_T_I64
117	V_CMPX_LG_F64	248	V_CMPX_F_U64
118	V_CMPX_GE_F64	249	V_CMPX_LT_U64
119	V_CMPX_O_F64	250	V_CMPX_EQ_U64
120	V_CMPX_U_F64	251	V_CMPX_LE_U64
121	V_CMPX_NGE_F64	252	V_CMPX_GT_U64
122	V_CMPX_NLG_F64	253	V_CMPX_NE_U64
123	V_CMPX_NGT_F64	254	V_CMPX_GE_U64
124	V_CMPX_NLE_F64	255	V_CMPX_T_U64

13.3.4. VOP3A



Format VOP3A

Description Vector ALU format with three operands

Table 83. VOP3A Fields

Field Name	Bits	Format or Description
VDST	[7:0]	Destination VGPR
ABS	[10:8]	Absolute value of input. [8] = src0, [9] = src1, [10] = src2
OPSEL	[14:11]	Operand select for 16-bit data. 0 = select low half, 1 = select high half. [11] = src0, [12] = src1, [13] = src2, [14] = dest.
CLMP	[15]	Clamp output
OP	[25:16]	Opcode. See next table.
ENCODING	[31:26]	Must be: 110100

Field Name	Bits	Format or Description
SRC0	[40:32]	Source 0. First operand for the instruction.
	0 - 101	SGPR0 to SGPR101: Scalar general-purpose registers.
	102	FLAT_SCRATCH_LO.
	103	FLAT_SCRATCH_HI.
	104	XNACK_MASK_LO.
	105	XNACK_MASK_HI.
	106	VCC_LO: vcc[31:0].
	107	VCC_HI: vcc[63:32].
	108-123	TTMP0 - TTMP15: Trap handler temporary register.
	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.
	209-234	Reserved.
	235	SHARED_BASE (Memory Aperture definition).
	236	SHARED_LIMIT (Memory Aperture definition).
	237	PRIVATE_BASE (Memory Aperture definition).
	238	PRIVATE_LIMIT (Memory Aperture definition).
	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	1/(2*PI).
	249	SDWA
	250	DPP
	251	VCCZ.
	252	EXECZ.
	253	SCC.
	254	Reserved.
	255	Literal constant.
	256 - 511	VGPR 0 - 255
SRC1	[49:41]	Second input operand. Same options as SRC0.
SRC2	[58:50]	Third input operand. Same options as SRC0.
OMOD	[60:59]	Output Modifier: 0=none, 1=*2, 2=*4, 3=div-2
NEG	[63:61]	Negate input. [61] = src0, [62] = src1, [63] = src2

Table 84. VOP3A Opcodes

Opcode #	Name	Opcode #	Name
384	V_NOP	659	V_BFM_B32
385	V_MOV_B32	660	V_CVT_PKNORM_I16_F32
386	V_READFIRSTLANE_B32	661	V_CVT_PKNORM_U16_F32
387	V_CVT_I32_F64	662	V_CVT_PKRTZ_F16_F32
388	V_CVT_F64_I32	663	V_CVT_PK_U16_U32
389	V_CVT_F32_I32	664	V_CVT_PK_I16_I32
390	V_CVT_F32_U32	665	V_CVT_PKNORM_I16_F16

Opcode #	Name	Opcode #	Name
391	V_CVT_U32_F32	666	V_CVT_PKNORM_U16_F16
392	V_CVT_I32_F32	668	V_ADD_I32
394	V_CVT_F16_F32	669	V_SUB_I32
395	V_CVT_F32_F16	670	V_ADD_I16
396	V_CVT_RPI_I32_F32	671	V_SUB_I16
397	V_CVT_FLR_I32_F32	672	V_PACK_B32_F16
398	V_CVT_OFF_F32_I4	673	V_MUL_LEGACY_F32
399	V_CVT_F32_F64	674	V_CVT_PK_FP8_F32
400	V_CVT_F64_F32	675	V_CVT_PK_BF8_F32
401	V_CVT_F32_UBYTE0	676	V_CVT_SR_FP8_F32
402	V_CVT_F32_UBYTE1	677	V_CVT_SR_BF8_F32
403	V_CVT_F32_UBYTE2	16	V_CMP_CLASS_F32
404	V_CVT_F32_UBYTE3	17	V_CMPX_CLASS_F32
405	V_CVT_U32_F64	18	V_CMP_CLASS_F64
406	V_CVT_F64_U32	19	V_CMPX_CLASS_F64
407	V_TRUNC_F64	20	V_CMP_CLASS_F16
408	V_CEIL_F64	21	V_CMPX_CLASS_F16
409	V_RNDNE_F64	32	V_CMP_F_F16
410	V_FLOOR_F64	33	V_CMP_LT_F16
411	V_FRACT_F32	34	V_CMP_EQ_F16
412	V_TRUNC_F32	35	V_CMP_LE_F16
413	V_CEIL_F32	36	V_CMP_GT_F16
414	V_RNDNE_F32	37	V_CMP_LG_F16
415	V_FLOOR_F32	38	V_CMP_GE_F16
416	V_EXP_F32	39	V_CMP_O_F16
417	V_LOG_F32	40	V_CMP_U_F16
418	V_RCP_F32	41	V_CMP_NGE_F16
419	V_RCP_IFLAG_F32	42	V_CMP_NLG_F16
420	V_RSQ_F32	43	V_CMP_NGT_F16
421	V_RCP_F64	44	V_CMP_NLE_F16
422	V_RSQ_F64	45	V_CMP_NEQ_F16
423	V_SQRT_F32	46	V_CMP_NLT_F16
424	V_SQRT_F64	47	V_CMP_TRU_F16
425	V_SIN_F32	48	V_CMPX_F_F16
426	V_COS_F32	49	V_CMPX_LT_F16
427	V_NOT_B32	50	V_CMPX_EQ_F16
428	V_BFREV_B32	51	V_CMPX_LE_F16
429	V_FFBH_U32	52	V_CMPX_GT_F16
430	V_FFBL_B32	53	V_CMPX_LG_F16
431	V_FFBH_I32	54	V_CMPX_GE_F16
432	V_FREXP_EXP_I32_F64	55	V_CMPX_O_F16
433	V_FREXP_MANT_F64	56	V_CMPX_U_F16
434	V_FRACT_F64	57	V_CMPX_NGE_F16
435	V_FREXP_EXP_I32_F32	58	V_CMPX_NLG_F16
436	V_FREXP_MANT_F32	59	V_CMPX_NGT_F16
437	V_CLREXCP	60	V_CMPX_NLE_F16
440	V_MOV_B64	61	V_CMPX_NEQ_F16

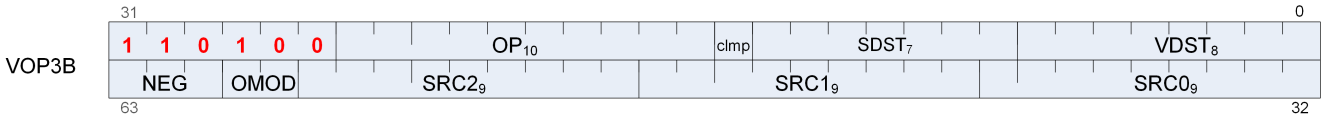
Opcode #	Name	Opcode #	Name
441	V_CVT_F16_U16	62	V_CMPX_NLT_F16
442	V_CVT_F16_I16	63	V_CMPX_TRU_F16
443	V_CVT_U16_F16	64	V_CMP_F_F32
444	V_CVT_I16_F16	65	V_CMP_LT_F32
445	V_RCP_F16	66	V_CMP_EQ_F32
446	V_SQRT_F16	67	V_CMP_LE_F32
447	V_RSQ_F16	68	V_CMP_GT_F32
448	V_LOG_F16	69	V_CMP_LG_F32
449	V_EXP_F16	70	V_CMP_GE_F32
450	V_FREXP_MANT_F16	71	V_CMP_O_F32
451	V_FREXP_EXP_I16_F16	72	V_CMP_U_F32
452	V_FLOOR_F16	73	V_CMP_NGE_F32
453	V_CEIL_F16	74	V_CMP_NLG_F32
454	V_TRUNC_F16	75	V_CMP_NGT_F32
455	V_RNDNE_F16	76	V_CMP_NLE_F32
456	V_FRACT_F16	77	V_CMP_NEQ_F32
457	V_SIN_F16	78	V_CMP_NLT_F32
458	V_COS_F16	79	V_CMP_TRU_F32
461	V_CVT_NORM_I16_F16	80	V_CMPX_F_F32
462	V_CVT_NORM_U16_F16	81	V_CMPX_LT_F32
463	V_SAT_PK_U8_I16	82	V_CMPX_EQ_F32
465	V_SWAP_B32	83	V_CMPX_LE_F32
466	V_ACCVGPR_MOV_B32	84	V_CMPX_GT_F32
468	V_CVT_F32_FP8	85	V_CMPX_LG_F32
469	V_CVT_F32_BF8	86	V_CMPX_GE_F32
470	V_CVT_PK_F32_FP8	87	V_CMPX_O_F32
471	V_CVT_PK_F32_BF8	88	V_CMPX_U_F32
256	V_CNDMASK_B32	89	V_CMPX_NGE_F32
257	V_ADD_F32	90	V_CMPX_NLG_F32
258	V_SUB_F32	91	V_CMPX_NGT_F32
259	V_SUBREV_F32	92	V_CMPX_NLE_F32
260	V_FMAC_F64	93	V_CMPX_NEQ_F32
261	V_MUL_F32	94	V_CMPX_NLT_F32
262	V_MUL_I32_I24	95	V_CMPX_TRU_F32
263	V_MUL_HI_I32_I24	96	V_CMP_F_F64
264	V_MUL_U32_U24	97	V_CMP_LT_F64
265	V_MUL_HI_U32_U24	98	V_CMP_EQ_F64
266	V_MIN_F32	99	V_CMP_LE_F64
267	V_MAX_F32	100	V_CMP_GT_F64
268	V_MIN_I32	101	V_CMP_LG_F64
269	V_MAX_I32	102	V_CMP_GE_F64
270	V_MIN_U32	103	V_CMP_O_F64
271	V_MAX_U32	104	V_CMP_U_F64
272	V_LSHRREV_B32	105	V_CMP_NGE_F64
273	V_ASHRREV_I32	106	V_CMP_NLG_F64
274	V_LSHLREV_B32	107	V_CMP_NGT_F64
275	V_AND_B32	108	V_CMP_NLE_F64

Opcode #	Name	Opcode #	Name
276	V_OR_B32	109	V_CMP_NEQ_F64
277	V_XOR_B32	110	V_CMP_NLT_F64
287	V_ADD_F16	111	V_CMP_TRU_F64
288	V_SUB_F16	112	V_CMPX_F_F64
289	V_SUBREV_F16	113	V_CMPX_LT_F64
290	V_MUL_F16	114	V_CMPX_EQ_F64
291	V_MAC_F16	115	V_CMPX_LE_F64
294	V_ADD_U16	116	V_CMPX_GT_F64
295	V_SUB_U16	117	V_CMPX_LG_F64
296	V_SUBREV_U16	118	V_CMPX_GE_F64
297	V_MUL_LO_U16	119	V_CMPX_O_F64
298	V_LSHLREV_B16	120	V_CMPX_U_F64
299	V_LSHRREV_B16	121	V_CMPX_NGE_F64
300	V_ASHRREV_I16	122	V_CMPX_NLG_F64
301	V_MAX_F16	123	V_CMPX_NGT_F64
302	V_MIN_F16	124	V_CMPX_NLE_F64
303	V_MAX_U16	125	V_CMPX_NEQ_F64
304	V_MAX_I16	126	V_CMPX_NLT_F64
305	V_MIN_U16	127	V_CMPX_TRU_F64
306	V_MIN_I16	160	V_CMP_F_I16
307	V_LDEXP_F16	161	V_CMP_LT_I16
308	V_ADD_U32	162	V_CMP_EQ_I16
309	V_SUB_U32	163	V_CMP_LE_I16
310	V_SUBREV_U32	164	V_CMP_GT_I16
311	V_DOT2C_F32_F16	165	V_CMP_NE_I16
312	V_DOT2C_I32_I16	166	V_CMP_GE_I16
313	V_DOT4C_I32_I8	167	V_CMP_T_I16
314	V_DOT8C_I32_I4	168	V_CMP_F_U16
315	V_FMAC_F32	169	V_CMP_LT_U16
316	V_PK_FMAC_F16	170	V_CMP_EQ_U16
317	V_XNOR_B32	171	V_CMP_LE_U16
450	V_MAD_I32_I24	172	V_CMP_GT_U16
451	V_MAD_U32_U24	173	V_CMP_NE_U16
452	V_CUBEID_F32	174	V_CMP_GE_U16
453	V_CUBESC_F32	175	V_CMP_T_U16
454	V_CUBETC_F32	176	V_CMPX_F_I16
455	V_CUBEMA_F32	177	V_CMPX_LT_I16
456	V_BFE_U32	178	V_CMPX_EQ_I16
457	V_BFE_I32	179	V_CMPX_LE_I16
458	V_BFI_B32	180	V_CMPX_GT_I16
459	V_FMA_F32	181	V_CMPX_NE_I16
460	V_FMA_F64	182	V_CMPX_GE_I16
461	V_LERP_U8	183	V_CMPX_T_I16
462	V_ALIGNBIT_B32	184	V_CMPX_F_U16
463	V_ALIGNBYTE_B32	185	V_CMPX_LT_U16
464	V_MIN3_F32	186	V_CMPX_EQ_U16
465	V_MIN3_I32	187	V_CMPX_LE_U16

Opcode #	Name	Opcode #	Name
466	V_MIN3_U32	188	V_CMPX_GT_U16
467	V_MAX3_F32	189	V_CMPX_NE_U16
468	V_MAX3_I32	190	V_CMPX_GE_U16
469	V_MAX3_U32	191	V_CMPX_T_U16
470	V_MED3_F32	192	V_CMP_F_I32
471	V_MED3_I32	193	V_CMP_LT_I32
472	V_MED3_U32	194	V_CMP_EQ_I32
473	V_SAD_U8	195	V_CMP_LE_I32
474	V_SAD_HI_U8	196	V_CMP_GT_I32
475	V_SAD_U16	197	V_CMP_NE_I32
476	V_SAD_U32	198	V_CMP_GE_I32
477	V_CVT_PK_U8_F32	199	V_CMP_T_I32
478	V_DIV_FIXUP_F32	200	V_CMP_F_U32
479	V_DIV_FIXUP_F64	201	V_CMP_LT_U32
482	V_DIV_FMAS_F32	202	V_CMP_EQ_U32
483	V_DIV_FMAS_F64	203	V_CMP_LE_U32
484	V_MSAD_U8	204	V_CMP_GT_U32
485	V_QSAD_PK_U16_U8	205	V_CMP_NE_U32
486	V_MQSAD_PK_U16_U8	206	V_CMP_GE_U32
487	V_MQSAD_U32_U8	207	V_CMP_T_U32
490	V_MAD_LEGACY_F16	208	V_CMPX_F_I32
491	V_MAD_LEGACY_U16	209	V_CMPX_LT_I32
492	V_MAD_LEGACY_I16	210	V_CMPX_EQ_I32
493	V_PERM_B32	211	V_CMPX_LE_I32
494	V_FMA_LEGACY_F16	212	V_CMPX_GT_I32
495	V_DIV_FIXUP_LEGACY_F16	213	V_CMPX_NE_I32
496	V_CVT_PKACCUM_U8_F32	214	V_CMPX_GE_I32
497	V_MAD_U32_U16	215	V_CMPX_T_I32
498	V_MAD_I32_I16	216	V_CMPX_F_U32
499	V_XAD_U32	217	V_CMPX_LT_U32
500	V_MIN3_F16	218	V_CMPX_EQ_U32
501	V_MIN3_I16	219	V_CMPX_LE_U32
502	V_MIN3_U16	220	V_CMPX_GT_U32
503	V_MAX3_F16	221	V_CMPX_NE_U32
504	V_MAX3_I16	222	V_CMPX_GE_U32
505	V_MAX3_U16	223	V_CMPX_T_U32
506	V_MED3_F16	224	V_CMP_F_I64
507	V_MED3_I16	225	V_CMP_LT_I64
508	V_MED3_U16	226	V_CMP_EQ_I64
509	V_LSHL_ADD_U32	227	V_CMP_LE_I64
510	V_ADD_LSHL_U32	228	V_CMP_GT_I64
511	V_ADD3_U32	229	V_CMP_NE_I64
512	V_LSHL_OR_B32	230	V_CMP_GE_I64
513	V_AND_OR_B32	231	V_CMP_T_I64
514	V_OR3_B32	232	V_CMP_F_U64
515	V_MAD_F16	233	V_CMP_LT_U64
516	V_MAD_U16	234	V_CMP_EQ_U64

Opcode #	Name	Opcode #	Name
517	V_MAD_I16	235	V_CMP_LE_U64
518	V_FMA_F16	236	V_CMP_GT_U64
519	V_DIV_FIXUP_F16	237	V_CMP_NE_U64
520	V_LSHL_ADD_U64	238	V_CMP_GE_U64
640	V_ADD_F64	239	V_CMP_T_U64
641	V_MUL_F64	240	V_CMPX_F_I64
642	V_MIN_F64	241	V_CMPX_LT_I64
643	V_MAX_F64	242	V_CMPX_EQ_I64
644	V_LDEXP_F64	243	V_CMPX_LE_I64
645	V_MUL_LO_U32	244	V_CMPX_GT_I64
646	V_MUL_HI_U32	245	V_CMPX_NE_I64
647	V_MUL_HI_I32	246	V_CMPX_GE_I64
648	V_LDEXP_F32	247	V_CMPX_T_I64
649	V_READLANE_B32	248	V_CMPX_F_U64
650	V_WRITELANE_B32	249	V_CMPX_LT_U64
651	V_BCNT_U32_B32	250	V_CMPX_EQ_U64
652	V_MBCNT_LO_U32_B32	251	V_CMPX_LE_U64
653	V_MBCNT_HI_U32_B32	252	V_CMPX_GT_U64
655	V_LSHLREV_B64	253	V_CMPX_NE_U64
656	V_LSHRREV_B64	254	V_CMPX_GE_U64
657	V_ASHRREV_I64	255	V_CMPX_T_U64
658	V_TRIG_PREOP_F64		

13.3.5. VOP3B



Format VOP3B

Description Vector ALU format with three operands and a scalar result. This encoding is used only for a few opcodes.

This encoding allows specifying a unique scalar destination, and is used only for the opcodes listed below. All other opcodes use VOP3A.

- V_ADD_CO_U32
- V_SUB_CO_U32
- V_SUBREV_CO_U32
- V_ADDC_CO_U32
- V_SUBB_CO_U32
- V_SUBBREV_CO_U32
- V_DIV_SCALE_F32
- V_DIV_SCALE_F64
- V_MAD_U64_U32

- V_MAD_I64_I32

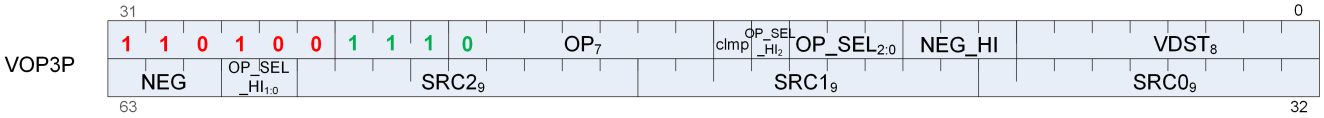
Table 85. VOP3B Fields

Field Name	Bits	Format or Description
VDST	[7:0]	Destination VGPR
SDST	[14:8]	Scalar destination
CLMP	[15]	Clamp result
OP	[25:16]	Opcode. see next table.
ENCODING	[31:26]	Must be: 110100
SRC0	[40:32]	Source 0. First operand for the instruction.
	0 - 101	SGPR0 to SGPR101: Scalar general-purpose registers.
	102	FLAT_SCRATCH_LO.
	103	FLAT_SCRATCH_HI.
	104	XNACK_MASK_LO.
	105	XNACK_MASK_HI.
	106	VCC_LO: vcc[31:0].
	107	VCC_HI: vcc[63:32].
	108-123	TTMP0 - TTMP15: Trap handler temporary register.
	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.
	209-234	Reserved.
	235	SHARED_BASE (Memory Aperture definition).
	236	SHARED_LIMIT (Memory Aperture definition).
	237	PRIVATE_BASE (Memory Aperture definition).
	238	PRIVATE_LIMIT (Memory Aperture definition).
	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	1/(2*PI).
	249	SDWA
	250	DPP
	251	VCCZ.
	252	EXECZ.
	253	SCC.
	254	Reserved.
	255	Literal constant.
	256 - 511	VGPR 0 - 255
SRC1	[49:41]	Second input operand. Same options as SRC0.
SRC2	[58:50]	Third input operand. Same options as SRC0.
OMOD	[60:59]	Output Modifier: 0=none, 1=*2, 2=*4, 3=div-2
NEG	[63:61]	Negate input. [61] = src0, [62] = src1, [63] = src2

Table 86. VOP3B Opcodes

Opcode #	Name	Opcode #	Name
281	V_ADD_CO_U32	286	V_SUBBREV_CO_U32
282	V_SUB_CO_U32	480	V_DIV_SCALE_F32
283	V_SUBREV_CO_U32	481	V_DIV_SCALE_F64
284	V_ADDC_CO_U32	488	V_MAD_U64_U32
285	V_SUBB_CO_U32	489	V_MAD_I64_I32

13.3.6. VOP3P



Format VOP3P

Description Vector ALU format taking one, two or three pairs of 16 bit inputs and producing two 16-bit outputs (packed into 1 dword).

Table 87. VOP3P Fields

Field Name	Bits	Format or Description
VDST	[7:0]	Destination VGPR
NEG_HI	[10:8]	Negate sources 0,1,2 of the high 16-bits.
OPSEL	[13:11]	Select low or high for low sources 0=[11], 1=[12], 2=[13].
OPSEL_HI2	[14]	Select low or high for high sources 0=[14], 1=[60], 2=[59].
CLMP	[15]	1 = clamp result.
OP	[22:16]	Opcode. see next table.
ENCODING	[31:24]	Must be: 110100111

Field Name	Bits	Format or Description
SRC0	[40:32]	Source 0. First operand for the instruction.
	0 - 101	SGPR0 to SGPR101: Scalar general-purpose registers.
	102	FLAT_SCRATCH_LO.
	103	FLAT_SCRATCH_HI.
	104	XNACK_MASK_LO.
	105	XNACK_MASK_HI.
	106	VCC_LO: vcc[31:0].
	107	VCC_HI: vcc[63:32].
	108-123	TTMP0 - TTMP15: Trap handler temporary register.
	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.
	209-234	Reserved.
	235	SHARED_BASE (Memory Aperture definition).
	236	SHARED_LIMIT (Memory Aperture definition).
	237	PRIVATE_BASE (Memory Aperture definition).
	238	PRIVATE_LIMIT (Memory Aperture definition).
	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	1/(2*PI).
	249	SDWA
	250	DPP
	251	VCCZ.
	252	EXECZ.
	253	SCC.
	254	Reserved.
	255	Literal constant.
	256 - 511	VGPR 0 - 255
SRC1	[49:41]	Second input operand. Same options as SRC0.
SRC2	[58:50]	Third input operand. Same options as SRC0.
OPSEL_HI	[60:59]	See OP_SEL_HI2.
NEG	[63:61]	Negate input for low 16-bits of sources. [61] = src0, [62] = src1, [63] = src2

13.3.6.1. VOP3P-MAI

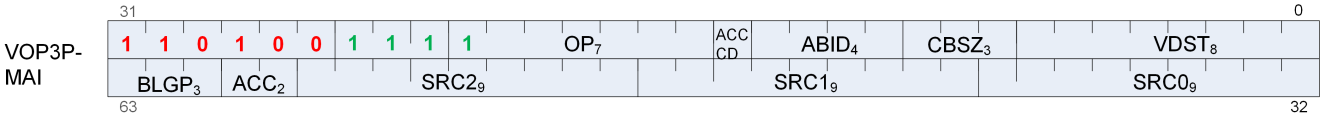


Table 88. VOP3P-MAI Fields

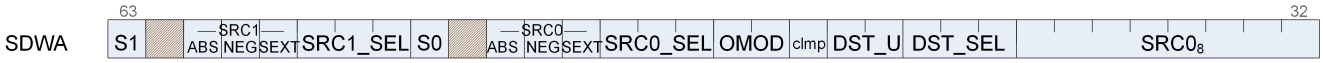
Field Name	Bits	Format or Description
VDST	[7:0]	Destination VGPR
CBSZ	[10:8]	Control Broadcast Size: Broadcast one chosen block of the A matrix to the input of 2^{CBSZ} other blocks of matrix multiplication. Legal values = 0-4, but must not be greater than log2(blocks) for any MFMA instruction. The block ID to broadcast comes from ABID. Defines the number of blocks that can do a broadcast within a group. Legal values = 0-4. The block ID of this group comes from ABID.
ABID	[14:11]	A-matrix Broadcast Identifier: When CBSZ is set to a non-zero value, within each contiguous set of 2^{CBSZ} blocks, this chooses which block of A to broadcast to the matrix multiplication inputs of the others.
ACC_CD	[15]	Indicates that SRC-C and VDST use ACC VGPRs. For SMFMAC ops, ACC_CD affects only the D matrix, and the compression indices, held in SRC2, come from Arch VGPRs.
OP	[22:16]	Opcode. see next table.
ENCODING	[31:24]	Must be: 110100111
SRC0	[40:32] 0 - 107 128 129-192 193-208 209-239 240 241 242 243 244 245 246 247 248 249 - 255 256 - 511	Source 0. First operand for the instruction. Reserved. 0. Signed integer 1 to 64. Signed integer -1 to -16. Reserved. 0.5. (float32) -0.5.(float32) 1.0. (float32) -1.0. (float32) 2.0. (float32) -2.0. (float32) 4.0. (float32) -4.0. (float32) 1/(2*PI). (float32) Reserved VGPR 0 - 255
SRC1	[49:41]	Second input operand. Same options as SRC0.
SRC2	[58:50]	Third input operand. Same options as SRC0.
ACC	[60:59]	ACC[0] : 0 = read SRC-A from Arch VGPR; 1 = read SRC-A from Acc VGPR. ACC[1] : 0 = read SRC-B from Arch VGPR; 1 = read SRC-B from Acc VGPR.
BLGP	[63:61]	“B”-Matrix Lane-Group Pattern. Controls how to swizzle the matrix lane groups (LG) in VGPRs when doing matrix multiplication by controlling the swizzle muxes. For V_MFMA_F64_4X4X4F64 and V_MFMA_F64_16X16X4F64 this field specifies the NEG modifier instead of BLGP.

Table 89. VOP3P Opcodes

Opcode #	Name	Opcode #	Name
0	V_PK_MAD_I16	73	V_MFMA_F32_16X16X4_4B_F16
1	V_PK_MUL_LO_U16	74	V_MFMA_F32_4X4X4_16B_F16
2	V_PK_ADD_I16	76	V_MFMA_F32_32X32X8_F16
3	V_PK_SUB_I16	77	V_MFMA_F32_16X16X16_F16
4	V_PK_LSHLREV_B16	80	V_MFMA_I32_32X32X4_2B_I8
5	V_PK_LSHRREV_B16	81	V_MFMA_I32_16X16X4_4B_I8
6	V_PK_ASHRREV_I16	82	V_MFMA_I32_4X4X4_16B_I8
7	V_PK_MAX_I16	86	V_MFMA_I32_32X32X16_I8
8	V_PK_MIN_I16	87	V_MFMA_I32_16X16X32_I8

Opcode #	Name	Opcode #	Name
9	V_PK_MAD_U16	88	V_ACCVGPR_READ
10	V_PK_ADD_U16	89	V_ACCVGPR_WRITE
11	V_PK_SUB_U16	93	V_MFMA_F32_32X32X4_2B_BF16
12	V_PK_MAX_U16	94	V_MFMA_F32_16X16X4_4B_BF16
13	V_PK_MIN_U16	95	V_MFMA_F32_4X4X4_16B_BF16
14	V_PK_FMA_F16	96	V_MFMA_F32_32X32X8_BF16
15	V_PK_ADD_F16	97	V_MFMA_F32_16X16X16_BF16
16	V_PK_MUL_F16	98	V_SMFMAC_F32_16X16X32_F16
17	V_PK_MIN_F16	100	V_SMFMAC_F32_32X32X16_F16
18	V_PK_MAX_F16	102	V_SMFMAC_F32_16X16X32_BF16
32	V_MAD_MIX_F32	104	V_SMFMAC_F32_32X32X16_BF16
33	V_MAD_MIXLO_F16	106	V_SMFMAC_I32_16X16X64_I8
34	V_MAD_MIXHI_F16	108	V_SMFMAC_I32_32X32X32_I8
35	V_DOT2_F32_F16	110	V_MFMA_F64_16X16X4_F64
38	V_DOT2_I32_I16	111	V_MFMA_F64_4X4X4_4B_F64
39	V_DOT2_U32_U16	112	V_MFMA_F32_16X16X32_BF8_BF8
40	V_DOT4_I32_I8	113	V_MFMA_F32_16X16X32_BF8_FP8
41	V_DOT4_U32_U8	114	V_MFMA_F32_16X16X32_FP8_BF8
42	V_DOT8_I32_I4	115	V_MFMA_F32_16X16X32_FP8_FP8
43	V_DOT8_U32_U4	116	V_MFMA_F32_32X32X16_BF8_BF8
48	V_PK_FMA_F32	117	V_MFMA_F32_32X32X16_BF8_FP8
49	V_PK_MUL_F32	118	V_MFMA_F32_32X32X16_FP8_BF8
50	V_PK_ADD_F32	119	V_MFMA_F32_32X32X16_FP8_FP8
51	V_PK_MOV_B32	120	V_SMFMAC_F32_16X16X64_BF8_BF8
62	V_MFMA_F32_16X16X8_XF32	121	V_SMFMAC_F32_16X16X64_BF8_FP8
63	V_MFMA_F32_32X32X4_XF32	122	V_SMFMAC_F32_16X16X64_FP8_BF8
64	V_MFMA_F32_32X32X1_2B_F32	123	V_SMFMAC_F32_16X16X64_FP8_FP8
65	V_MFMA_F32_16X16X1_4B_F32	124	V_SMFMAC_F32_32X32X32_BF8_BF8
66	V_MFMA_F32_4X4X1_16B_F32	125	V_SMFMAC_F32_32X32X32_BF8_FP8
68	V_MFMA_F32_32X32X2_F32	126	V_SMFMAC_F32_32X32X32_FP8_BF8
69	V_MFMA_F32_16X16X4_F32	127	V_SMFMAC_F32_32X32X32_FP8_FP8
72	V_MFMA_F32_32X32X4_2B_F16		

13.3.7. SDWA



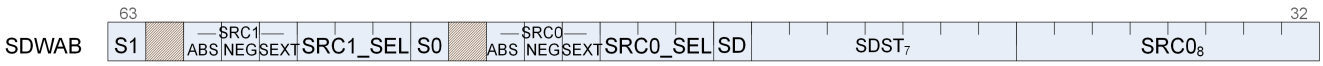
Format SDWA

Description Sub-Dword Addressing. This is a second dword which can follow VOP1 or VOP2 instructions (in place of a literal constant) to control selection of sub-dword (16-bit) operands. Use of SDWA is indicated by assigning the SRC0 field to SDWA, and then the actual VGPR used as source-zero is determined in SDWA instruction word.

Table 90. SDWA Fields

Field Name	Bits	Format or Description
SRC0	[39:32]	Real SRC0 operand (VGPR).
DST_SEL	[42:40]	Select the data destination: 0-3 = reserved 4 = data[15:0] 5 = data[31:16] 6 = data[31:0] 7 = reserved
DST_U	[44:43]	Destination format: what do with the bits in the VGPR that are not selected by DST_SEL: 0 = pad with zeros + 1 = sign extend upper / zero lower 2 = preserve (don't modify) 3 = reserved
CLMP	[45]	1 = clamp result
OMOD	[47:46]	Output modifiers (see VOP3). [46] = low half, [47] = high half
SRC0_SEL	[50:48]	Source 0 select. Same options as DST_SEL.
SRC0_SEXT	[51]	Sign extend modifier for source 0.
SRC0_NEG	[52]	1 = negate source 0.
SRC0_ABS	[53]	1 = Absolute value of source 0.
S0	[55]	0 = source 0 is VGPR, 1 = is SGPR.
SRC1_SEL	[58:56]	Same options as SRC0_SEL.
SRC1_SEXT	[59]	Sign extend modifier for source 1.
SRC1_NEG	[60]	1 = negate source 1.
SRC1_ABS	[61]	1 = Absolute value of source 1.
S1	[63]	0 = source 1 is VGPR, 1 = is SGPR.

13.3.8. SDWAB



Format SDWAB

Description Sub-Dword Addressing. This is a second dword which can follow VOPC instructions (in place of a literal constant) to control selection of sub-dword (16-bit) operands. Use of SDWA is indicated by assigning the SRC0 field to SDWA, and then the actual VGPR used as source-zero is determined in SDWA instruction word. This version has a scalar destination.

Table 91. SDWAB Fields

Field Name	Bits	Format or Description
SRC0	[39:32]	Real SRC0 operand (VGPR).
SDST	[46:40]	Scalar GPR destination.
SD	[47]	Scalar destination type: 0 = VCC, 1 = normal SGPR.
SRC0_SEL	[50:48]	Source 0 select. Same options as DST_SEL.
SRC0_SEXT	[51]	Sign extend modifier for source 0.
SRC0_NEG	[52]	1 = negate source 0.
SRC0_ABS	[53]	1 = Absolute value of source 0.
S0	[55]	0 = source 0 is VGPR, 1 = is SGPR.

Field Name	Bits	Format or Description
SRC1_SEL	[58:56]	Same options as SRC0_SEL.
SRC1_SEXT	[59]	Sign extend modifier for source 1.
SRC1_NEG	[60]	1 = negate source 1.
SRC1_ABS	[61]	1 = Absolute value of source 1.
S1	[63]	0 = source 1 is VGPR, 1 = is SGPR.

13.3.9. DPP

Format DPP

Description Data Parallel Primitives. This is a second dword which can follow VOP1, VOP2 or VOPC instructions (in place of a literal constant) to control selection of data from other lanes.

Table 92. DPP Fields

Field Name	Bits	Format or Description
SRC0	[39:32]	Real SRC0 operand (VGPR).
DPP_CTRL	[48:40]	See next table: "DPP_CTRL Enumeration"
BC	[51]	Bounds Control: 0 = do not write when source is out of range, 1 = write.
SRC0_NEG	[52]	1 = negate source 0.
SRC0_ABS	[53]	1 = Absolute value of source 0.
SRC1_NEG	[54]	1 = negate source 1.
SRC1_ABS	[55]	1 = Absolute value of source 1.
BANK_MASK	[59:56]	Bank Mask Applies to the VGPR destination write only, does not impact the thread mask when fetching source VGPR data. 27==0: lanes[12:15, 28:31, 44:47, 60:63] are disabled 26==0: lanes[8:11, 24:27, 40:43, 56:59] are disabled 25==0: lanes[4:7, 20:23, 36:39, 52:55] are disabled 24==0: lanes[0:3, 16:19, 32:35, 48:51] are disabled Notice: the term "bank" here is not the same as we used for the VGPR bank.
ROW_MASK	[63:60]	Row Mask Applies to the VGPR destination write only, does not impact the thread mask when fetching source VGPR data. 31==0: lanes[63:48] are disabled (wave 64 only) 30==0: lanes[47:32] are disabled (wave 64 only) 29==0: lanes[31:16] are disabled 28==0: lanes[15:0] are disabled

Table 93. DPP_CTRL Enumeration

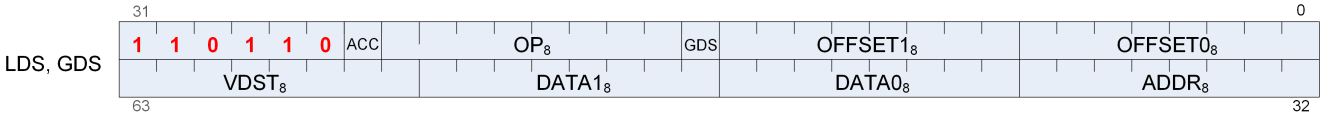
DPP_Cntl Enumeration	Hex Value	Function	Description
DPP_QUAD_PERM*	000-0FF	pix[n].srca = pix[(n&0x3c)+ dpp_cntl[n%4*2+1 : n%4*2]].srca	Full permute of four threads.
DPP_UNUSED	100	Undefined	Reserved.
DPP_ROW_SL*	101-10F	if ((n & 0xf) < (16-cntl[3:0])) pix[n].srca = pix[n + cntl[3:0]].srca else use bound_cntl	Row shift left by 1-15 threads.
DPP_ROW_SR*	111-11F	if ((n&0xf) >= cntl[3:0]) pix[n].srca = pix[n - cntl[3:0]].srca else use bound_cntl	Row shift right by 1-15 threads.

DPP_Cntl Enumeration	Hex Value	Function	Description
DPP_ROW_RR*	121-12F	if ((n&0xf) >= cnt[3:0]) pix[n].srca = pix[n - cntl[3:0]].srca else pix[n].srca = pix[n + 16 - cntl[3:0]].srca	Row rotate right by 1-15 threads.
DPP_WF_SL1*	130	if (n<63) pix[n].srca = pix[n+1].srca else use bound_cntl	Wavefront left shift by 1 thread.
DPP_WF_RL1*	134	if (n<63) pix[n].srca = pix[n+1].srca else pix[n].srca = pix[0].srca	Wavefront left rotate by 1 thread.
DPP_WF_SR1*	138	if (n>0) pix[n].srca = pix[n-1].srca else use bound_cntl	Wavefront right shift by 1 thread.
DPP_WF_RR1*	13C	if (n>0) pix[n].srca = pix[n-1].srca else pix[n].srca = pix[63].srca	Wavefront right rotate by 1 thread.
DPP_ROW_MIRROR*	140	pix[n].srca = pix[15-(n&f)].srca	Mirror threads within row.
DPP_ROW_HALF_MIRROR*	141	pix[n].srca = pix[7-(n&7)].srca	Mirror threads within row (8 threads).
DPP_ROW_BCAST15*	142	if (n>15) pix[n].srca = pix[n & 0x30 - 1].srca	Broadcast 15th thread of each row to next row.
DPP_ROW_BCAST31*	143	if (n>31) pix[n].srca = pix[n & 0x20 - 1].srca	Broadcast thread 31 to rows 2 and 3.
DPP_ROW*	150 - 15F	pix[n].srca = pix[(n & 0xfffffff0)+count].srca;	Broadcast thread 0-15 within a row to the whole row.

Note that for 64-bit input data the only legal DPP type is "DPP_ROW*".

13.4. LDS and GWS format

13.4.1. DS



Format LDS and GDS

Description Local and Global Data Sharing instructions

Table 94. DS Fields

Field Name	Bits	Format or Description
OFFSET0	[7:0]	First address offset
OFFSET1	[15:8]	Second address offset. For some opcodes this is concatenated with OFFSET0.
GDS	[16]	1=GWS, 0=LDS operation.
OP	[24:17]	See Opcode table below.
ACC	[25]	VDST is Accumulation VGPR
ENCODING	[31:26]	Must be: 110110
ADDR	[39:32]	VGPR which supplies the address.
DATA0	[47:40]	First data VGPR.
DATA1	[55:48]	Second data VGPR.
VDST	[63:56]	Destination VGPR when results returned to VGPRs.

Table 95. DS Opcodes

Opcode #	Name	Opcode #	Name
0	DS_ADD_U32	69	DS_MIN_I64
1	DS_SUB_U32	70	DS_MAX_I64
2	DS_RSUB_U32	71	DS_MIN_U64
3	DS_INC_U32	72	DS_MAX_U64
4	DS_DEC_U32	73	DS_AND_B64
5	DS_MIN_I32	74	DS_OR_B64
6	DS_MAX_I32	75	DS_XOR_B64
7	DS_MIN_U32	76	DS_MSKOR_B64
8	DS_MAX_U32	77	DS_WRITE_B64
9	DS_AND_B32	78	DS_WRITE2_B64
10	DS_OR_B32	79	DS_WRITE2ST64_B64
11	DS_XOR_B32	80	DS_CMPST_B64
12	DS_MSKOR_B32	81	DS_CMPST_F64
13	DS_WRITE_B32	82	DS_MIN_F64
14	DS_WRITE2_B32	83	DS_MAX_F64
15	DS_WRITE2ST64_B32	84	DS_WRITE_B8_D16_HI
16	DS_CMPST_B32	85	DS_WRITE_B16_D16_HI
17	DS_CMPST_F32	86	DS_READ_U8_D16
18	DS_MIN_F32	87	DS_READ_U8_D16_HI
19	DS_MAX_F32	88	DS_READ_I8_D16
20	DS_NOP	89	DS_READ_I8_D16_HI
21	DS_ADD_F32	90	DS_READ_U16_D16
23	DS_PK_ADD_F16	91	DS_READ_U16_D16_HI
24	DS_PK_ADD_BF16	92	DS_ADD_F64
29	DS_WRITE_ADDTID_B32	96	DS_ADD_RTN_U64
30	DS_WRITE_B8	97	DS_SUB_RTN_U64
31	DS_WRITE_B16	98	DS_RSUB_RTN_U64
32	DS_ADD_RTN_U32	99	DS_INC_RTN_U64
33	DS_SUB_RTN_U32	100	DS_DEC_RTN_U64
34	DS_RSUB_RTN_U32	101	DS_MIN_RTN_I64
35	DS_INC_RTN_U32	102	DS_MAX_RTN_I64
36	DS_DEC_RTN_U32	103	DS_MIN_RTN_U64
37	DS_MIN_RTN_I32	104	DS_MAX_RTN_U64
38	DS_MAX_RTN_I32	105	DS_AND_RTN_B64
39	DS_MIN_RTN_U32	106	DS_OR_RTN_B64
40	DS_MAX_RTN_U32	107	DS_XOR_RTN_B64
41	DS_AND_RTN_B32	108	DS_MSKOR_RTN_B64
42	DS_OR_RTN_B32	109	DS_WRXCHG_RTN_B64
43	DS_XOR_RTN_B32	110	DS_WRXCHG2_RTN_B64
44	DS_MSKOR_RTN_B32	111	DS_WRXCHG2ST64_RTN_B64
45	DS_WRXCHG_RTN_B32	112	DS_CMPST_RTN_B64
46	DS_WRXCHG2_RTN_B32	113	DS_CMPST_RTN_F64
47	DS_WRXCHG2ST64_RTN_B32	114	DS_MIN_RTN_F64
48	DS_CMPST_RTN_B32	115	DS_MAX_RTN_F64
49	DS_CMPST_RTN_F32	118	DS_READ_B64
50	DS_MIN_RTN_F32	119	DS_READ2_B64

Opcode #	Name	Opcode #	Name
51	DS_MAX_RTN_F32	120	DS_READ2ST64_B64
52	DS_WRAP_RTN_B32	124	DS_ADD_RTN_F64
53	DS_ADD_RTN_F32	126	DS_CONDXCHG32_RTN_B64
54	DS_READ_B32	152	DS_GWS_SEMA_RELEASE_ALL
55	DS_READ2_B32	153	DS_GWS_INIT
56	DS_READ2ST64_B32	154	DS_GWS_SEMA_V
57	DS_READ_I8	155	DS_GWS_SEMA_BR
58	DS_READ_U8	156	DS_GWS_SEMA_P
59	DS_READ_I16	157	DS_GWS_BARRIER
60	DS_READ_U16	182	DS_READ_ADDTID_B32
61	DS_SWIZZLE_B32	183	DS_PK_ADD_RTN_F16
62	DS_PERMUTE_B32	184	DS_PK_ADD_RTN_BF16
63	DS_BPERMUTE_B32	189	DS_CONSUME
64	DS_ADD_U64	190	DS_APPEND
65	DS_SUB_U64	222	DS_WRITE_B96
66	DS_RSUB_U64	223	DS_WRITE_B128
67	DS_INC_U64	254	DS_READ_B96
68	DS_DEC_U64	255	DS_READ_B128

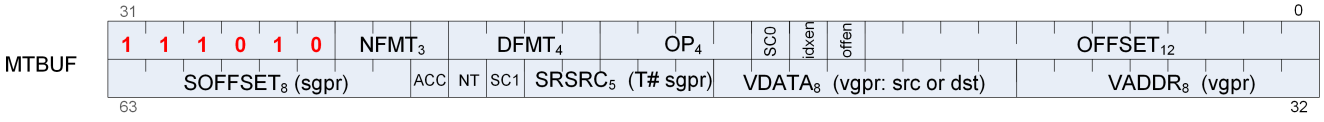
13.5. Vector Memory Buffer Formats

There are two memory buffer instruction formats:

MTBUF
typed buffer access (data type is defined by the instruction)

MUBUF
untyped buffer access (data type is defined by the buffer / resource-constant)

13.5.1. MTBUF



Format MTBUF

Description Memory Typed-Buffer Instructions

Table 96. MTBUF Fields

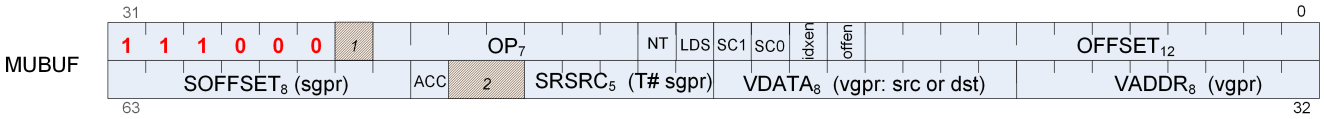
Field Name	Bits	Format or Description
OFFSET	[11:0]	Address offset, unsigned byte.
OFFEN	[12]	1 = enable offset VGPR, 0 = use zero for address offset
IDXEN	[13]	1 = enable index VGPR, 0 = use zero for address index
SC0	[14]	Scope bit 0

Field Name	Bits	Format or Description
OP	[18:15]	Opcode. See table below.
DFMT	22:19	Data Format of data in memory 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
NFMT	25:23	Numeric format of data in memory: 0 unorm 1 snorm 2 uscaled 3 sscaled 4 uint 5 sint 6 reserved 7 float
ENCODING	[31:26]	Must be: 111010
VADDR	[39:32]	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 and offset in the second.
VDATA	[47:40]	Address of VGPR to supply first component of write data or receive first component of read-data.
SRSRC	[52:48]	SGPR to supply V# (resource constant) in 4 or 8 consecutive SGPRs. It is missing 2 LSB's of SGPR-address since must be aligned to 4.
SC1	[53]	Scope bit 1
NT	[54]	Non-Temporal
ACC	[55]	VDATA is Accumulation VGPR
SOFFSET	[63:56]	Address offset, unsigned byte.

Table 97. MTBUF Opcodes

Opcode #	Name	Opcode #	Name
0	TBUFFER_LOAD_FORMAT_X	8	TBUFFER_LOAD_FORMAT_D16_X
1	TBUFFER_LOAD_FORMAT_XY	9	TBUFFER_LOAD_FORMAT_D16_XY
2	TBUFFER_LOAD_FORMAT_XYZ	10	TBUFFER_LOAD_FORMAT_D16_XYZ
3	TBUFFER_LOAD_FORMAT_XYZW	11	TBUFFER_LOAD_FORMAT_D16_XYZW
4	TBUFFER_STORE_FORMAT_X	12	TBUFFER_STORE_FORMAT_D16_X
5	TBUFFER_STORE_FORMAT_XY	13	TBUFFER_STORE_FORMAT_D16_XY
6	TBUFFER_STORE_FORMAT_XYZ	14	TBUFFER_STORE_FORMAT_D16_XYZ
7	TBUFFER_STORE_FORMAT_XYZW	15	TBUFFER_STORE_FORMAT_D16_XYZW

13.5.2. MUBUF



Format MUBUF

Description Memory Untyped-Buffer Instructions

Table 98. MUBUF Fields

Field Name	Bits	Format or Description
OFFSET	[11:0]	Address offset, unsigned byte.
OFFEN	[12]	1 = enable offset VGPR, 0 = use zero for address offset
IDXEN	[13]	1 = enable index VGPR, 0 = use zero for address index
SC0	[14]	Scope bit 0
SC1	[15]	Scope bit 1
LDS	[16]	0 = normal, 1 = transfer data between LDS and memory instead of VGPRs and memory.
NT	[17]	Non-Temporal
OP	[24:18]	Opcode. See table below.
ENCODING	[31:26]	Must be: 111000
VADDR	[39:32]	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 and offset in the second.
VDATA	[47:40]	Address of VGPR to supply first component of write data or receive first component of read-data.
SRSRC	[52:48]	SGPR to supply V# (resource constant) in 4 or 8 consecutive SGPRs. It is missing 2 LSB's of SGPR-address since must be aligned to 4.
ACC	[55]	VDATA is Accumulation VGPR
SOFFSET	[63:56]	Address offset, unsigned byte.

Table 99. MUBUF Opcodes

Opcode #	Name	Opcode #	Name
0	BUFFER_LOAD_FORMAT_X	37	BUFFER_LOAD_SHORT_D16_HI
1	BUFFER_LOAD_FORMAT_XY	38	BUFFER_LOAD_FORMAT_D16_HI_X
2	BUFFER_LOAD_FORMAT_XYZ	39	BUFFER_STORE_FORMAT_D16_HI_X
3	BUFFER_LOAD_FORMAT_XYZW	40	BUFFER_WBL2
4	BUFFER_STORE_FORMAT_X	41	BUFFER_INV
5	BUFFER_STORE_FORMAT_XY	64	BUFFER_ATOMIC_SWAP
6	BUFFER_STORE_FORMAT_XYZ	65	BUFFER_ATOMIC_CMPSWAP
7	BUFFER_STORE_FORMAT_XYZW	66	BUFFER_ATOMIC_ADD
8	BUFFER_LOAD_FORMAT_D16_X	67	BUFFER_ATOMIC_SUB
9	BUFFER_LOAD_FORMAT_D16_XY	68	BUFFER_ATOMIC_SMIN
10	BUFFER_LOAD_FORMAT_D16_XYZ	69	BUFFER_ATOMIC_UMIN
11	BUFFER_LOAD_FORMAT_D16_XYZW	70	BUFFER_ATOMIC_SMAX
12	BUFFER_STORE_FORMAT_D16_X	71	BUFFER_ATOMIC_UMAX
13	BUFFER_STORE_FORMAT_D16_XY	72	BUFFER_ATOMIC_AND
14	BUFFER_STORE_FORMAT_D16_XYZ	73	BUFFER_ATOMIC_OR
15	BUFFER_STORE_FORMAT_D16_XYZW	74	BUFFER_ATOMIC_XOR

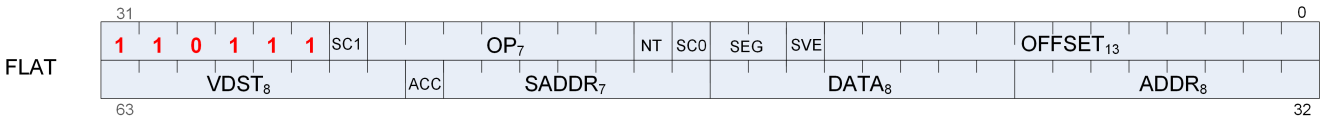
Opcode #	Name	Opcode #	Name
16	BUFFER_LOAD_UBYTE	75	BUFFER_ATOMIC_INC
17	BUFFER_LOAD_SBYTE	76	BUFFER_ATOMIC_DEC
18	BUFFER_LOAD_USHORT	77	BUFFER_ATOMIC_ADD_F32
19	BUFFER_LOAD_SSHORT	78	BUFFER_ATOMIC_PK_ADD_F16
20	BUFFER_LOAD_DWORD	79	BUFFER_ATOMIC_ADD_F64
21	BUFFER_LOAD_DWORDX2	80	BUFFER_ATOMIC_MIN_F64
22	BUFFER_LOAD_DWORDX3	81	BUFFER_ATOMIC_MAX_F64
23	BUFFER_LOAD_DWORDX4	96	BUFFER_ATOMIC_SWAP_X2
24	BUFFER_STORE_BYTE	97	BUFFER_ATOMIC_CMPSWAP_X2
25	BUFFER_STORE_BYTE_D16_HI	98	BUFFER_ATOMIC_ADD_X2
26	BUFFER_STORE_SHORT	99	BUFFER_ATOMIC_SUB_X2
27	BUFFER_STORE_SHORT_D16_HI	100	BUFFER_ATOMIC_SMIN_X2
28	BUFFER_STORE_DWORD	101	BUFFER_ATOMIC_UMIN_X2
29	BUFFER_STORE_DWORDX2	102	BUFFER_ATOMIC_SMAX_X2
30	BUFFER_STORE_DWORDX3	103	BUFFER_ATOMIC_UMAX_X2
31	BUFFER_STORE_DWORDX4	104	BUFFER_ATOMIC_AND_X2
32	BUFFER_LOAD_UBYTE_D16	105	BUFFER_ATOMIC_OR_X2
33	BUFFER_LOAD_UBYTE_D16_HI	106	BUFFER_ATOMIC_XOR_X2
34	BUFFER_LOAD_SBYTE_D16	107	BUFFER_ATOMIC_INC_X2
35	BUFFER_LOAD_SBYTE_D16_HI	108	BUFFER_ATOMIC_DEC_X2
36	BUFFER_LOAD_SHORT_D16		

13.6. Flat Formats

Flat memory instructions come in three versions: FLAT:: memory address (per work-item) may be in global memory, scratch (private) memory or shared memory (LDS) GLOBAL:: same as FLAT, but assumes all memory addresses are global memory. SCRATCH:: same as FLAT, but assumes all memory addresses are scratch (private) memory.

The microcode format is identical for each, and only the value of the SEG (segment) field differs.

13.6.1. FLAT



Format FLAT

Description FLAT Memory Access

Table 100. FLAT Fields

Field Name	Bits	Format or Description
OFFSET	[12:0]	Address offset Scratch, Global: 13-bit signed byte offset FLAT: 12-bit unsigned offset (MSB is ignored)

Field Name	Bits	Format or Description
LDS	[13]	0 = normal, 1 = transfer data between LDS and memory instead of VGPRs and memory.
SEG	[15:14]	Memory Segment (instruction type): 0 = flat, 1 = scratch, 2 = global.
SC0	[16]	Scope bit 0
NT	[17]	Non-Temporal
OP	[24:18]	Opcode. See tables below for FLAT, SCRATCH and GLOBAL opcodes.
SC1	[25]	Scope bit 1
ENCODING	[31:26]	Must be: 110111
ADDR	[39:32]	VGPR which holds address or offset. For 64-bit addresses, ADDR has the LSB's and ADDR+1 has the MSBs. For offset a single VGPR has a 32 bit unsigned offset. For FLAT_*: specifies an address. For GLOBAL_* and SCRATCH_* when SADDR is 0x7f: specifies an address. For GLOBAL_* and SCRATCH_* when SADDR is not 0x7f: specifies an offset.
DATA	[47:40]	VGPR which supplies data.
SADDR	[54:48]	Scalar SGPR which provides an address of offset (unsigned). Set this field to 0x7f to disable use. Meaning of this field is different for Scratch and Global: FLAT: Unused Scratch: use an SGPR for the address instead of a VGPR Global: use the SGPR to provide a base address and the VGPR provides a 32-bit byte offset.
ACC	[55]	VDATA is Accumulation VGPR
VDST	[63:56]	Destination VGPR for data returned from memory to VGPRs.

Table 101. FLAT Opcodes

Opcode #	Name	Opcode #	Name
16	FLAT_LOAD_UBYTE	69	FLAT_ATOMIC_UMIN
17	FLAT_LOAD_SBYTE	70	FLAT_ATOMIC_SMAX
18	FLAT_LOAD_USHORT	71	FLAT_ATOMIC_UMAX
19	FLAT_LOAD_SSHORT	72	FLAT_ATOMIC_AND
20	FLAT_LOAD_DWORD	73	FLAT_ATOMIC_OR
21	FLAT_LOAD_DWORDX2	74	FLAT_ATOMIC_XOR
22	FLAT_LOAD_DWORDX3	75	FLAT_ATOMIC_INC
23	FLAT_LOAD_DWORDX4	76	FLAT_ATOMIC_DEC
24	FLAT_STORE_BYTE	77	FLAT_ATOMIC_ADD_F32
25	FLAT_STORE_BYTE_D16_HI	78	FLAT_ATOMIC_PK_ADD_F16
26	FLAT_STORE_SHORT	79	FLAT_ATOMIC_ADD_F64
27	FLAT_STORE_SHORT_D16_HI	80	FLAT_ATOMIC_MIN_F64
28	FLAT_STORE_DWORD	81	FLAT_ATOMIC_MAX_F64
29	FLAT_STORE_DWORDX2	82	FLAT_ATOMIC_PK_ADD_BF16
30	FLAT_STORE_DWORDX3	96	FLAT_ATOMIC_SWAP_X2
31	FLAT_STORE_DWORDX4	97	FLAT_ATOMIC_CMPSWAP_X2
32	FLAT_LOAD_UBYTE_D16	98	FLAT_ATOMIC_ADD_X2
33	FLAT_LOAD_UBYTE_D16_HI	99	FLAT_ATOMIC_SUB_X2
34	FLAT_LOAD_SBYTE_D16	100	FLAT_ATOMIC_SMIN_X2
35	FLAT_LOAD_SBYTE_D16_HI	101	FLAT_ATOMIC_UMIN_X2
36	FLAT_LOAD_SHORT_D16	102	FLAT_ATOMIC_SMAX_X2
37	FLAT_LOAD_SHORT_D16_HI	103	FLAT_ATOMIC_UMAX_X2
64	FLAT_ATOMIC_SWAP	104	FLAT_ATOMIC_AND_X2

Opcode #	Name	Opcode #	Name
65	FLAT_ATOMIC_CMPSWAP	105	FLAT_ATOMIC_OR_X2
66	FLAT_ATOMIC_ADD	106	FLAT_ATOMIC_XOR_X2
67	FLAT_ATOMIC_SUB	107	FLAT_ATOMIC_INC_X2
68	FLAT_ATOMIC_SMIN	108	FLAT_ATOMIC_DEC_X2

13.6.2. GLOBAL

Table 102. GLOBAL Opcodes

Opcode #	Name	Opcode #	Name
16	GLOBAL_LOAD_UBYTE	67	GLOBAL_ATOMIC_SUB
17	GLOBAL_LOAD_SBYTE	68	GLOBAL_ATOMIC_SMIN
18	GLOBAL_LOAD_USHORT	69	GLOBAL_ATOMIC_UMIN
19	GLOBAL_LOAD_SSHORT	70	GLOBAL_ATOMIC_SMAX
20	GLOBAL_LOAD_DWORD	71	GLOBAL_ATOMIC_UMAX
21	GLOBAL_LOAD_DWORDX2	72	GLOBAL_ATOMIC_AND
22	GLOBAL_LOAD_DWORDX3	73	GLOBAL_ATOMIC_OR
23	GLOBAL_LOAD_DWORDX4	74	GLOBAL_ATOMIC_XOR
24	GLOBAL_STORE_BYTE	75	GLOBAL_ATOMIC_INC
25	GLOBAL_STORE_BYTE_D16_HI	76	GLOBAL_ATOMIC_DEC
26	GLOBAL_STORE_SHORT	77	GLOBAL_ATOMIC_ADD_F32
27	GLOBAL_STORE_SHORT_D16_HI	78	GLOBAL_ATOMIC_PK_ADD_F16
28	GLOBAL_STORE_DWORD	79	GLOBAL_ATOMIC_ADD_F64
29	GLOBAL_STORE_DWORDX2	80	GLOBAL_ATOMIC_MIN_F64
30	GLOBAL_STORE_DWORDX3	81	GLOBAL_ATOMIC_MAX_F64
31	GLOBAL_STORE_DWORDX4	82	GLOBAL_ATOMIC_PK_ADD_BF16
32	GLOBAL_LOAD_UBYTE_D16	96	GLOBAL_ATOMIC_SWAP_X2
33	GLOBAL_LOAD_UBYTE_D16_HI	97	GLOBAL_ATOMIC_CMPSWAP_X2
34	GLOBAL_LOAD_SBYTE_D16	98	GLOBAL_ATOMIC_ADD_X2
35	GLOBAL_LOAD_SBYTE_D16_HI	99	GLOBAL_ATOMIC_SUB_X2
36	GLOBAL_LOAD_SHORT_D16	100	GLOBAL_ATOMIC_SMIN_X2
37	GLOBAL_LOAD_SHORT_D16_HI	101	GLOBAL_ATOMIC_UMIN_X2
38	GLOBAL_LOAD_LDS_UBYTE	102	GLOBAL_ATOMIC_SMAX_X2
39	GLOBAL_LOAD_LDS_SBYTE	103	GLOBAL_ATOMIC_UMAX_X2
40	GLOBAL_LOAD_LDS_USHORT	104	GLOBAL_ATOMIC_AND_X2
41	GLOBAL_LOAD_LDS_SSHORT	105	GLOBAL_ATOMIC_OR_X2
42	GLOBAL_LOAD_LDS_DWORD	106	GLOBAL_ATOMIC_XOR_X2
64	GLOBAL_ATOMIC_SWAP	107	GLOBAL_ATOMIC_INC_X2
65	GLOBAL_ATOMIC_CMPSWAP	108	GLOBAL_ATOMIC_DEC_X2
66	GLOBAL_ATOMIC_ADD		

13.6.3. SCRATCH

Table 103. SCRATCH Opcodes

Opcode #	Name	Opcode #	Name
16	SCRATCH_LOAD_UBYTE	30	SCRATCH_STORE_DWORDX3

Opcode #	Name	Opcode #	Name
17	SCRATCH_LOAD_SBYTE	31	SCRATCH_STORE_DWORDX4
18	SCRATCH_LOAD_USHORT	32	SCRATCH_LOAD_UBYTE_D16
19	SCRATCH_LOAD_SSHORT	33	SCRATCH_LOAD_UBYTE_D16_HI
20	SCRATCH_LOAD_DWORD	34	SCRATCH_LOAD_SBYTE_D16
21	SCRATCH_LOAD_DWORDX2	35	SCRATCH_LOAD_SBYTE_D16_HI
22	SCRATCH_LOAD_DWORDX3	36	SCRATCH_LOAD_SHORT_D16
23	SCRATCH_LOAD_DWORDX4	37	SCRATCH_LOAD_SHORT_D16_HI
24	SCRATCH_STORE_BYTE	38	SCRATCH_LOAD_LDS_UBYTE
25	SCRATCH_STORE_BYTE_D16_HI	39	SCRATCH_LOAD_LDS_SBYTE
26	SCRATCH_STORE_SHORT	40	SCRATCH_LOAD_LDS_USHORT
27	SCRATCH_STORE_SHORT_D16_HI	41	SCRATCH_LOAD_LDS_SSHORT
28	SCRATCH_STORE_DWORD	42	SCRATCH_LOAD_LDS_DWORD
29	SCRATCH_STORE_DWORDX2		