This Specification Agreement (this "Agreement") is a legal agreement between Advanced Micro Devices, Inc. ("AMD") and "You" as the recipient of the attached AMD Specification (the "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. 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.

4. 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.

5. 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.

6. 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/.

7. 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.

8. 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 contained herein is for informational purposes only, and is subject to change without notice. While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this information. Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein. No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. Terms and limitations applicable to the purchase or use of AMD’s products are as set forth in a signed agreement between the parties or in AMD’s Standard Terms and Conditions of Sale.

AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

© 2018-2019 Advanced Micro Devices, Inc. All rights reserved.
12.5. SOPP Instructions ............................................................... 110
  12.5.1. Send Message ......................................................... 114
12.6. SMEM Instructions ........................................................... 114
12.7. VOP2 Instructions ........................................................... 122
  12.7.1. VOP2 using VOP3 encoding ......................................... 127
12.8. VOP1 Instructions ........................................................... 127
  12.8.1. VOP1 using VOP3 encoding ......................................... 141
12.9. VOPC Instructions ........................................................... 142
  12.9.1. VOPC using VOP3A encoding ....................................... 154
12.10. VOP3P Instructions ......................................................... 154
12.11. VINTERP Instructions ...................................................... 156
  12.11.1. VINTERP using VOP3 encoding .................................... 157
12.12. VOP3A & VOP3B Instructions ........................................... 157
12.13. LDS & GDS Instructions ................................................... 176
  12.13.1. DS_SWIZZLE_B32 Details .......................................... 197
  12.13.2. LDS Instruction Limitations ....................................... 199
12.14. MUBUF Instructions ....................................................... 200
12.15. MTBUF Instructions ....................................................... 205
12.16. MIMG Instructions ......................................................... 206
12.17. EXPORT Instructions ...................................................... 211
12.18. FLAT, Scratch and Global Instructions ................................ 212
  12.18.1. Flat Instructions .................................................... 212
  12.18.2. Scratch Instructions ............................................... 216
  12.18.3. Global Instructions ............................................... 217
12.19. Instruction Limitations .................................................. 221
  12.19.1. DPP ................................................................. 221
  12.19.2. SDWA ............................................................... 222
13. Microcode Formats ............................................................. 223
  13.1. Scalar ALU and Control Formats ....................................... 224
    13.1.1. SOP2 ............................................................. 225
    13.1.2. SOPK ............................................................. 228
    13.1.3. SOP1 ............................................................. 230
    13.1.4. SOPC ............................................................. 233
    13.1.5. SOPP ............................................................. 235
  13.2. Scalar Memory Format ................................................... 237
    13.2.1. SMEM ............................................................. 237
  13.3. Vector ALU Formats ........................................................ 240
    13.3.1. VOP2 ............................................................. 240
    13.3.2. VOP1 ............................................................. 243
    13.3.3. VOPC ............................................................. 247
    13.3.4. VOP3A ............................................................. 256
    13.3.5. VOP3B ............................................................. 261
Preface

About This Document

This document describes the environment, organization and program state of AMD GCN "Vega" 7nm Generation 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 (include 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 GCN processors' hardware and programming environment (Chapter 1).
Chapter 2 describes the organization of GCN programs.
Chapter 3 describes the program state that is maintained.
Chapter 4 describes the program flow.
Chapter 5 describes the scalar ALU operations.
Chapter 6 describes the vector ALU operations.
Chapter 7 describes the scalar memory operations.
Chapter 8 describes the vector memory operations.
Chapter 9 provides information about the flat memory instructions.
Chapter 10 describes the data share operations.
Chapter 11 describes exporting the parameters of pixel color and vertex shaders.
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:

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

Related Documents

- Intermediate Language (IL) Reference Manual. Published by AMD.
- AMD Accelerated Parallel Processing OpenCL Programming Guide. Published by AMD.
- The OpenCL Specification. Published by Khronos Group. Aaftab Munshi, editor.

New Features of "Vega" 7nm Devices

Summary of kernel instruction changes in Vega GPUs:

- New packed 16-bit math instructions.
TMA and TBA registers are stored one per VM-ID, not per draw or dispatch.

- Added Image operations support 16-bit address and data.
- Added Global and Scratch memory read/write operations.
  - Also added Scratch load/store to scalar memory.
- Added Scalar memory atomic instructions.
- MIMG Microcode format: removed the R128 bit.
- FLAT Microcode format: added an offset field.
- Removed V_MOVEREL instructions.
- Added control over arithmetic overflow for FP16 VALU operations.
- Modified bit packing of surface descriptors and samplers:
  - T#: removed heap, elem_size, last_array, interlaced, uservm_mode bits.
  - V#: removed mtype.
  - S#: removed astc_hdr field.

**New Instructions**

Vega 7nm includes the additional instructions listed below:

- V_FMAC_F32
- V_XNOR_B32
- V_DOT2_F32_F16
- V_DOT2_I32_I16
- V_DOT2_U32_U16
- V_DOT4_I32_I8
- V_DOT4_U32_U8
- V_DOT8_I32_I4
- V_DOT8_U32_U4

**Contact Information**

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

For information about developing with AMD Accelerated Parallel Processing, please see: developer.amd.com/appsdk.
We also have a growing community of AMD Accelerated Parallel Processing users. Come visit us at the AMD Accelerated Parallel Processing Developer Forum (http://developer.amd.com/openclforum) to find out what applications other users are trying on their AMD Accelerated Parallel Processing products.
Chapter 1. Introduction

The AMD GCN processor implements a parallel micro-architecture that provides an excellent platform not only for computer graphics applications but also for general-purpose data parallel applications. Data-intensive applications that require high bandwidth or are computationally intensive may be run on an AMD GCN processor.

The figure below shows a block diagram of the AMD GCN Vega Generation series processors.

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

**Figure 1. AMD GCN Vega Generation Series Block Diagram**

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

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

The GCN programs are controlled by host commands that

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

The GCN driver program runs on the host.

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

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

The AMD GCN devices can detect floating point exceptions and can generate interrupts. In particular, they 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 GCN processor hides memory latency by keeping track of potentially hundreds of work-items in different stages of execution, and by overlapping compute operations with memory-access operations.

### 1.1. Terminology

**Table 1. Basic Terms**

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCN Processor</td>
<td>The Graphics Core Next shader processor is a scalar and vector ALU designed to run complex programs on behalf of a wavefront.</td>
</tr>
<tr>
<td>Dispatch</td>
<td>A dispatch launches a 1D, 2D, or 3D grid of work to the GCN processor array.</td>
</tr>
<tr>
<td>Workgroup</td>
<td>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.</td>
</tr>
<tr>
<td>Wavefront</td>
<td>A collection of 64 work-items that execute in parallel on a single GCN processor.</td>
</tr>
<tr>
<td>Work-item</td>
<td>A single element of work: one element from the dispatch grid, or in graphics a pixel or vertex.</td>
</tr>
<tr>
<td>Literal Constant</td>
<td>A 32-bit integer or float constant that is placed in the instruction stream.</td>
</tr>
<tr>
<td>Scalar ALU (SALU)</td>
<td>The scalar ALU operates on one value per wavefront and manages all control flow.</td>
</tr>
<tr>
<td>Term</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Vector ALU (VALU)</td>
<td>The vector ALU maintains Vector GPRs that are unique for each work item and execute arithmetic operations uniquely on each work-item.</td>
</tr>
<tr>
<td>Microcode format</td>
<td>The microcode format describes the bit patterns used to encode instructions. Each instruction is either 32 or 64 bits.</td>
</tr>
<tr>
<td>Instruction</td>
<td>An instruction is the basic unit of the kernel. Instructions include: vector ALU, scalar ALU, memory transfer, and control flow operations.</td>
</tr>
<tr>
<td>Quad</td>
<td>A quad is a 2x2 group of screen-aligned pixels. This is relevant for sampling texture maps.</td>
</tr>
<tr>
<td>Texture Sampler (S#)</td>
<td>A texture sampler is a 128-bit entity that describes how the vector memory system reads and samples (filters) a texture map.</td>
</tr>
<tr>
<td>Texture Resource (T#)</td>
<td>A texture resource descriptor describes an image in memory: address, data format, stride, etc.</td>
</tr>
<tr>
<td>Buffer Resource (V#)</td>
<td>A buffer resource descriptor describes a buffer in memory: address, data format, stride, etc.</td>
</tr>
</tbody>
</table>
Chapter 2. Program Organization

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

The GCN 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 GCN processor, taking data from memory, processing it, and writing results back to memory. Compute kernels are created by a dispatch, which causes the GCN processors to run the kernel over all of the work-items in a 1D, 2D, or 3D grid of data. The GCN 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 GCN stream processors are designed to share data between different work-items. Data sharing can boost performance. The figure below shows the memory hierarchy that is available to each work-item.

![Shared Memory Hierarchy](image)

*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 AMD GCN processors use a 64 kB local data share (LDS) memory for each compute unit; this enables 64 kB of low-latency bandwidth to the processing elements. The shared memory contains 32 integer atomic units to enable fast, unordered atomic operations. This memory can be used as a software cache for predictable re-use of data, a data exchange machine for the work-items of a work-group, or as a cooperative way to enable efficient access to off-chip memory.
2.2.2. Global Data Share (GDS)

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

2.3. Device Memory

The AMD GCN 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-only 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>
<thead>
<tr>
<th>Abbrev.</th>
<th>Name</th>
<th>Size (bits)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>Program Counter</td>
<td>48</td>
<td>Points to the memory address of the next shader instruction to execute.</td>
</tr>
<tr>
<td>V0-V255</td>
<td>VGPR</td>
<td>32</td>
<td>Vector general-purpose register.</td>
</tr>
<tr>
<td>S0-S103</td>
<td>SGPR</td>
<td>32</td>
<td>Scalar general-purpose register.</td>
</tr>
<tr>
<td>LDS</td>
<td>Local Data Share</td>
<td>64kB</td>
<td>Local data share is a scratch RAM with built-in arithmetic capabilities that allow data to be shared between threads in a workgroup.</td>
</tr>
<tr>
<td>EXEC</td>
<td>Execute Mask</td>
<td>64</td>
<td>A bit mask with one bit per thread, which is applied to vector instructions and controls that threads execute and that ignore the instruction.</td>
</tr>
<tr>
<td>EXECZ</td>
<td>EXEC is zero</td>
<td>1</td>
<td>A single bit flag indicating that the EXEC mask is all zeros.</td>
</tr>
<tr>
<td>VCC</td>
<td>Vector Condition Code</td>
<td>64</td>
<td>A bit mask with one bit per thread; it holds the result of a vector compare operation.</td>
</tr>
<tr>
<td>VCCZ</td>
<td>VCC is zero</td>
<td>1</td>
<td>A single bit-flag indicating that the VCC mask is all zeros.</td>
</tr>
<tr>
<td>SCC</td>
<td>Scalar Condition Code</td>
<td>1</td>
<td>Result from a scalar ALU comparison instruction.</td>
</tr>
<tr>
<td>FLAT_SCRATCH</td>
<td>Flat scratch address</td>
<td>64</td>
<td>The base address of scratch memory.</td>
</tr>
<tr>
<td>XNACK_MASK</td>
<td>Address translation failure</td>
<td>64</td>
<td>Bit mask of threads that have failed their address translation.</td>
</tr>
<tr>
<td>STATUS</td>
<td>Status</td>
<td>32</td>
<td>Read-only shader status bits.</td>
</tr>
<tr>
<td>MODE</td>
<td>Mode</td>
<td>32</td>
<td>Writable shader mode bits.</td>
</tr>
<tr>
<td>M0</td>
<td>Memory Reg</td>
<td>32</td>
<td>A temporary register that has various uses, including GPR indexing and bounds checking.</td>
</tr>
<tr>
<td>TRAPSTS</td>
<td>Trap Status</td>
<td>32</td>
<td>Holds information about exceptions and pending traps.</td>
</tr>
<tr>
<td>TBA</td>
<td>Trap Base Address</td>
<td>64</td>
<td>Holds the pointer to the current trap handler program.</td>
</tr>
</tbody>
</table>
3.2. Program Counter (PC)

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

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

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

3.3. EXECute Mask

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

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

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit Position</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCC</td>
<td>1</td>
<td>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.</td>
</tr>
<tr>
<td>SPI_PRIO</td>
<td>2:1</td>
<td>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.</td>
</tr>
<tr>
<td>WAVE_PRIO</td>
<td>4:3</td>
<td>Wavefront priority set by the shader program. See the S_SETPRIO instruction (page 12-49) for details.</td>
</tr>
<tr>
<td>PRIV</td>
<td>5</td>
<td>Privileged mode. Can only be active when in the trap handler. Gives write access to the TTMP, TMA, and TBA registers.</td>
</tr>
<tr>
<td>TRAP_EN</td>
<td>6</td>
<td>Indicates that a trap handler is present. When set to zero, traps are not taken.</td>
</tr>
<tr>
<td>TTRACE_EN</td>
<td>7</td>
<td>Indicates whether thread trace is enabled for this wavefront. If zero, also ignore any shader-generated (instruction) thread-trace data.</td>
</tr>
<tr>
<td>EXPORT_RDY</td>
<td>8</td>
<td>This status bit indicates if export buffer space has been allocated. The shader stalls any export instruction until this bit becomes 1. It is set to 1 when export buffer space has been allocated. Before a Pixel or Vertex shader can export, the hardware checks the state of this bit. If the bit is 1, export can be issued. If the bit is zero, the wavefront sleeps until space becomes available in the export buffer. Then, this bit is set to 1, and the wavefront resumes.</td>
</tr>
<tr>
<td>EXECZ</td>
<td>9</td>
<td>Exec mask is zero.</td>
</tr>
<tr>
<td>VCCZ</td>
<td>10</td>
<td>Vector condition code is zero.</td>
</tr>
<tr>
<td>IN_TG</td>
<td>11</td>
<td>Wavefront is a member of a work-group of more than one wavefront.</td>
</tr>
<tr>
<td>IN_BARRIER</td>
<td>12</td>
<td>Wavefront is waiting at a barrier.</td>
</tr>
<tr>
<td>HALT</td>
<td>13</td>
<td>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).</td>
</tr>
</tbody>
</table>
### 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>
<thead>
<tr>
<th>Field</th>
<th>Bit Position</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Round Modes: 0=nearest even, 1= +infinity, 2= -infinity, 3= toward zero.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mode. Denorm modes:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = flush input and output denorms.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = allow input denorms, flush output denorms.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = flush input denorms, allow output denorms.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 = allow input and output denorms.</td>
</tr>
<tr>
<td>DX10_CLAMP</td>
<td>8</td>
<td>Used by the vector ALU to force DX10-style treatment of NaNs: when set,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>clamp NaN to zero; otherwise, pass NaN through.</td>
</tr>
<tr>
<td>Field</td>
<td>Bit Position</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>IEEE</td>
<td>9</td>
<td>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.</td>
</tr>
<tr>
<td>LOD_CLAMPED</td>
<td>10</td>
<td>Sticky bit indicating that one or more texture accesses had their LOD clamped.</td>
</tr>
<tr>
<td>DEBUG</td>
<td>11</td>
<td>Forces the wavefront to jump to the exception handler after each instruction is executed (but not after ENDPGM). Only works if TRAP_EN = 1.</td>
</tr>
<tr>
<td>FP16_OVFL</td>
<td>23</td>
<td>If set, an overflowed FP16 result is clamped to +/- MAX_FP16, regardless of round mode, while still preserving true INF values.</td>
</tr>
<tr>
<td>POPS_PACKER0</td>
<td>24</td>
<td>1 = this wave is associated with packer 0. User shader must set this to !PackerID from the POPS initialized SGPR (load_collision_waveID), or zero if not using POPS.</td>
</tr>
<tr>
<td>POPS_PACKER1</td>
<td>25</td>
<td>1 = this wave is associated with packer 1. User shader must set this to PackerID from the POPS initialized SGPR (load_collision_waveID), or zero if not using POPS.</td>
</tr>
<tr>
<td>DISABLE_PERF</td>
<td>26</td>
<td>1 = disable performance counting for this wave</td>
</tr>
<tr>
<td>GPR_IDX_EN</td>
<td>27</td>
<td>GPR index enable.</td>
</tr>
<tr>
<td>VSKIP</td>
<td>28</td>
<td>0 = normal operation. 1 = skip (do not execute) any vector instructions: valu, vmem, export, lds, gds. &quot;Skipping&quot; instructions occurs at high-speed (10 wavefronts per clock cycle can skip one instruction). This is much faster than issuing and discarding instructions.</td>
</tr>
<tr>
<td>CSP</td>
<td>31:29</td>
<td>Conditional branch stack pointer.</td>
</tr>
</tbody>
</table>

### 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 normally used as VCC, the remaining 14 are available to the shader. Shader hardware does not prevent use of all 16 SGPRs.

3.6.3. SGPR Alignment

Even-aligned SGPRs are required in the following cases.

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

Quad-alignment is required for the data-GPR when a scalar memory read returns four or more Dwords. When a 64-bit quantity is stored in SGPRs, the LSBs are in SGPR[n], and the MSBs are in SGPR[n+1].

3.6.4. VGPR Allocation and Alignment

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

3.6.5. LDS Allocation and Clamping

LDS is allocated per work-group or per-wavefront when work-groups are not in use. LDS space is allocated to a work-group or wavefront in contiguous blocks of 128 Dwords on 128-Dword alignment. LDS allocations do not wrap around the LDS storage. All accesses to LDS are restricted to the space allocated to that wavefront/work-group.

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

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

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

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.

\[
V_{\text{CMP} \ _*} \Rightarrow VCC[n] = \text{EXEC}[n] \& (\text{test passed for thread}[n])
\]

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:

\[\{\text{TTMP}1, \text{TTMP}0\} = \{3'h0, \text{pc\_rewind}[3:0], \text{HT}[0], \text{trapID}[7:0], \text{PC}[47:0]\}\]

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 will be: \((\text{PC} - \text{PC\_rewind}*4)\).

**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.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Invalid</td>
</tr>
<tr>
<td>1</td>
<td>Input Denormal</td>
</tr>
<tr>
<td>2</td>
<td>Divide by zero</td>
</tr>
<tr>
<td>3</td>
<td>Overflow</td>
</tr>
<tr>
<td>4</td>
<td>Underflow</td>
</tr>
<tr>
<td>5</td>
<td>Inexact</td>
</tr>
<tr>
<td>6</td>
<td>Integer divide by zero</td>
</tr>
<tr>
<td>7</td>
<td>Address Watch - TC (L1) has witnessed a thread access to an 'address of interest'</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXCP</td>
<td>8:0</td>
<td>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</td>
</tr>
<tr>
<td>SAVECTX</td>
<td>10</td>
<td>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.</td>
</tr>
<tr>
<td>ILLEGAL_INST</td>
<td>11</td>
<td>An illegal instruction has been detected.</td>
</tr>
<tr>
<td>ADDR_WATCH1-3</td>
<td>14:12</td>
<td>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.</td>
</tr>
</tbody>
</table>
**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 surface.
- GDS alignment or address range error.
- GWS operation aborted (semaphore or barrier not executed).

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

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

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

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

Memory violations are not precise. The violation is reported when the LDS or TC processes the address; during this time, the wave may have processed many more instructions. When a mem_viol is reported, the Program Counter saved is that of the next instruction to execute; it has no relationship the faulting instruction.
Chapter 4. Program Flow Control

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

4.1. Program Control

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

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_ENDPGM</td>
<td>Terminates the wavefront. It can appear anywhere in the kernel and can appear multiple times.</td>
</tr>
<tr>
<td>S_ENDPGM_SAVED</td>
<td>Terminates the wavefront due to context save. It can appear anywhere in the kernel and can appear multiple times.</td>
</tr>
<tr>
<td>S_NOP</td>
<td>Does nothing; it can be repeated in hardware up to eight times.</td>
</tr>
<tr>
<td>S_TRAP</td>
<td>Jumps to the trap handler.</td>
</tr>
<tr>
<td>S_RFE</td>
<td>Returns from the trap handler.</td>
</tr>
<tr>
<td>S_SETPRIO</td>
<td>Modifies the priority of this wavefront: 0=lowest, 3 = highest.</td>
</tr>
<tr>
<td>S_SLEEP</td>
<td>Causes the wavefront to sleep for 64 - 960 clock cycles.</td>
</tr>
<tr>
<td>S_SENDMSG</td>
<td>Sends a message (typically an interrupt) to the host CPU.</td>
</tr>
</tbody>
</table>

4.2. Branching

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

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_BRANCH</td>
<td>Unconditional branch.</td>
</tr>
<tr>
<td>S_CBRANCH_&lt;test&gt;</td>
<td>Conditional branch. Branch only if &lt;test&gt; is true. Tests are VCCZ, VCCNZ, EXECZ, EXECONZ, SCCZ, and SCCNZ.</td>
</tr>
<tr>
<td>S_CBRANCH_CDBGSYS</td>
<td>Conditional branch, taken if the COND_DBG_SYS status bit is set.</td>
</tr>
<tr>
<td>S_CBRANCH_CDBGUSER</td>
<td>Conditional branch, taken if the COND_DBG_USER status bit is set.</td>
</tr>
<tr>
<td>S_CBRANCH_CDBGSYS_AND_US ER</td>
<td>Conditional branch, taken only if both COND_DBG_SYS and COND_DBG_USER are set.</td>
</tr>
<tr>
<td>S_SETPC</td>
<td>Directly set the PC from an SGPR pair.</td>
</tr>
<tr>
<td>Instructions</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>--------------------------------------------------------------</td>
</tr>
<tr>
<td>S_SWAPPC</td>
<td>Swap the current PC with an address in an SGPR pair.</td>
</tr>
<tr>
<td>S_GETPC</td>
<td>Retrieve the current PC value (does not cause a branch).</td>
</tr>
<tr>
<td>S_CBRANCH_FORK and</td>
<td>Conditional branch for complex branching.</td>
</tr>
<tr>
<td>S_CBRANCH_JOIN</td>
<td></td>
</tr>
<tr>
<td>S_SETVSKIP</td>
<td>Set a bit that causes all vector instructions to be ignored. Useful alternative to branching.</td>
</tr>
<tr>
<td>S_CALL_B64</td>
<td>Jump to a subroutine, and save return address. SGPR_pair = PC+4; PC = PC+4+SIMM16*4.</td>
</tr>
</tbody>
</table>

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 GDS and LDS instructions use LGKM_cnt, but they can return out-of-order.

- VM_CNT: Vector memory count.
  Determines when memory reads have returned data to VGPRs, or memory writes have

---

4.3. Workgroups
completed.

- Incremented every time a vector-memory read or write (MIMG, MUBUF, or MTBUF format) instruction is issued.
- Decremented for reads when the data has been written back to the VGPRs, and for writes when the data has been written to the L2 cache. Ordering: Memory reads and writes return in the order they were issued, including mixing reads and writes.

- LGKM_CNT: (LDS, GDS, (K)constant, (M)essage) Determines when one of these low-latency instructions have completed.
  - Incremented by 1 for every LDS or GDS instruction issued, as well as by Dword-count for scalar-memory reads. For example, s_memtime counts the same as an s_load_dwordx2.
  - Decremented by 1 for LDS/GDS reads or atomic-with-return when the data has been returned to VGPRs.
  - Incremented by 1 for each S_SENDMSG issued. Decremented by 1 when message is sent out.
  - Decremented by 1 for LDS/GDS writes when the data has been written to LDS/GDS.
  - Decremented by 1 for each Dword returned from the data-cache (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 GDS, at which time it is safe to overwrite the contents of that VGPR.
  - Incremented when an Export/GDS instruction is issued from the wavefront buffer.
  - Decremented for exports/GDS when the last cycle of the export instruction is granted and executed (VGPRs read out). Ordering
    - Exports are kept in order only within each export type (color/null, position, parameter cache).

### Table 8. Required Software-inserted Wait States

<table>
<thead>
<tr>
<th>First Instruction</th>
<th>Second Instruction</th>
<th>Wait</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_SETREG &lt;&gt;</td>
<td>S_GETREG &lt;same reg&gt;</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>S_SETREG &lt;&gt;</td>
<td>S_SETREG &lt;same reg&gt;</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>SET_VSKIP</td>
<td>S_GETREG MODE</td>
<td>2</td>
<td>Reads VSKIP from MODE.</td>
</tr>
</tbody>
</table>

## 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>
<thead>
<tr>
<th>First Instruction</th>
<th>Second Instruction</th>
<th>Wait</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_SETREG MODE.vskip</td>
<td>any vector op</td>
<td>2</td>
<td>Requires two nops or non-vector instructions.</td>
</tr>
<tr>
<td>VALU that sets VCC or EXEC</td>
<td>VALU that uses EXECZ or VCCZ as a data source</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>VALU writes SGPR/VCC (readlane, cmp, add/sub, div_scale)</td>
<td>V_(READ,WRITE)LANE using that SGPR/VCC as the lane select</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>VALU writes VCC (including v_div_scale)</td>
<td>V_DIV_FMAS</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>FLAT_STORE_X3, FLAT_STORE_X4, FLAT_ATOMIC_(F)CMPSWAP_X2, BUFFER_STORE_DWORD_X3, BUFFER_STORE_DWORD_X4, BUFFER_STORE_FORMAT_XYZ, BUFFER_STORE_FORMAT_XYZW, BUFFER_ATOMIC_(F)CMPSWAP_X2, IMAGE_STORE_* &gt; 64 bits, IMAGE_ATOMIC_(F)CMPSWAP &gt; + 64bits</td>
<td>Write VGPRs holding writedata from those instructions.</td>
<td>1</td>
<td>BUFFER_STORE_* operations that use an SGPR for &quot;offset&quot; do not require any wait states. IMAGE_STORE_* and IMAGE_(F)CMPSWAP* ops with more than two DMASK bits set require this one wait state. Ops that use a 256-bit T# do not need a wait state.</td>
</tr>
<tr>
<td>VALU writes SGPR</td>
<td>VMEM reads that SGPR</td>
<td>5</td>
<td>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.</td>
</tr>
<tr>
<td>SALU writes M0</td>
<td>GDS, S_SENDMSG or S_TTRACE_DATA</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>VALU writes VGPR</td>
<td>VALU DPP reads that VGPR</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>VALU writes EXEC</td>
<td>VALU DPP op</td>
<td>5</td>
<td>ALU does not forward EXEC to DPP.</td>
</tr>
<tr>
<td>Mixed use of VCC: alias vs SGPR# v_readlane, v_readfirstlane v_cmp v_add*/u v_sub*/_u v_div_scale* (writes vcc)</td>
<td>VALU which reads VCC as a constant (not as a carry-in which is 0 wait states).</td>
<td>1</td>
<td>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.</td>
</tr>
<tr>
<td>S_SETREG TRAPSTS</td>
<td>RFE, RFE_restore</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SALU writes M0</td>
<td>LDS &quot;add-TID&quot; instruction, buffer_store_LDS_dword, scratch or global with LDS = 1, VINTERP or LDS_direct</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SALU writes M0</td>
<td>S_MOVEREL</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
4.6. Arbitrary Divergent Control Flow

In the GCN 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](image)

The register requirements per wavefront are:

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

This method compares how many of the 64 threads go down the PASS path instead of the FAIL path; then, it selects the path with the fewer number of threads first. This means at most 50% of
the threads are active, and this limits the necessary stack depth to \( \text{Log}_{2}64 = 6 \).

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

```
S_CBRANCH_G_FORK  arg0, arg1
    // arg1 is an sgpr-pair which holds 64bit (48bit) target address
S_CBRANCH_I_FORK  arg0, #target_addr_offset[17:2]
    // target_addr_offset: 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:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP</td>
<td>Opcode: instruction to be executed.</td>
</tr>
<tr>
<td>SDST</td>
<td>Destination SGPR.</td>
</tr>
<tr>
<td>SSRC0</td>
<td>First source operand.</td>
</tr>
<tr>
<td>SSRC1</td>
<td>Second source operand.</td>
</tr>
<tr>
<td>SIMM16</td>
<td>Signed immediate 16-bit integer constant.</td>
</tr>
</tbody>
</table>

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>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar Dest (7 bits)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 - 101</td>
<td>SGPR 0 to 101</td>
<td>Scalar GPRs</td>
</tr>
<tr>
<td>102</td>
<td>FLAT_SCR_LO</td>
<td>Holds the low Dword of the flat-scratch memory descriptor</td>
</tr>
<tr>
<td>103</td>
<td>FLAT_SCR_HI</td>
<td>Holds the high Dword of the flat-scratch memory descriptor</td>
</tr>
<tr>
<td>104</td>
<td>XNACK_MASK_LO</td>
<td>Holds the low Dword of the XNACK mask.</td>
</tr>
<tr>
<td>105</td>
<td>XNACK_MASK_HI</td>
<td>Holds the high Dword of the XNACK mask.</td>
</tr>
<tr>
<td>106</td>
<td>VCC_LO</td>
<td>Holds the low Dword of the vector condition code</td>
</tr>
<tr>
<td>107</td>
<td>VCC_HI</td>
<td>Holds the high Dword of the vector condition code</td>
</tr>
<tr>
<td>108-123</td>
<td>TTMP0 to TTMP15</td>
<td>Trap temps (privileged)</td>
</tr>
<tr>
<td>124</td>
<td>M0</td>
<td>Holds the low Dword of the flat-scratch memory descriptor</td>
</tr>
<tr>
<td>125</td>
<td>reserved</td>
<td>reserved</td>
</tr>
<tr>
<td>126</td>
<td>EXEC_LO</td>
<td>Execute mask, low Dword</td>
</tr>
<tr>
<td>127</td>
<td>EXEC_HI</td>
<td>Execute mask, high Dword</td>
</tr>
<tr>
<td>128</td>
<td>0</td>
<td>zero</td>
</tr>
<tr>
<td>129-192</td>
<td>int 1 to 64</td>
<td>Positive integer values.</td>
</tr>
<tr>
<td>193-208</td>
<td>int -1 to -16</td>
<td>Negative integer values.</td>
</tr>
<tr>
<td>209-234</td>
<td>reserved</td>
<td>Unused.</td>
</tr>
<tr>
<td>Code</td>
<td>Meaning</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>--------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>235</td>
<td>SHARED_BASE</td>
<td>Memory Aperture definition.</td>
</tr>
<tr>
<td>236</td>
<td>SHARED_LIMIT</td>
<td></td>
</tr>
<tr>
<td>237</td>
<td>PRIVATE_BASE</td>
<td></td>
</tr>
<tr>
<td>238</td>
<td>PRIVATE_LIMIT</td>
<td></td>
</tr>
<tr>
<td>239</td>
<td>POPS_EXITING_WAVE_ID</td>
<td>Primitive Ordered Pixel Shading wave ID.</td>
</tr>
<tr>
<td>240</td>
<td>0.5</td>
<td>single or double floats</td>
</tr>
<tr>
<td>241</td>
<td>-0.5</td>
<td></td>
</tr>
<tr>
<td>242</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>243</td>
<td>-1.0</td>
<td></td>
</tr>
<tr>
<td>244</td>
<td>2.0</td>
<td></td>
</tr>
<tr>
<td>245</td>
<td>-2.0</td>
<td></td>
</tr>
<tr>
<td>246</td>
<td>4.0</td>
<td></td>
</tr>
<tr>
<td>247</td>
<td>-4.0</td>
<td></td>
</tr>
<tr>
<td>248</td>
<td>1.0 / (2 * PI)</td>
<td></td>
</tr>
<tr>
<td>249-250</td>
<td>reserved</td>
<td>unused</td>
</tr>
<tr>
<td>251</td>
<td>VCCZ</td>
<td>{ zeros, VCCZ }</td>
</tr>
<tr>
<td>252</td>
<td>EXECZ</td>
<td>{ zeros, EXECZ }</td>
</tr>
<tr>
<td>253</td>
<td>SCC</td>
<td>{ zeros, SCC }</td>
</tr>
<tr>
<td>254</td>
<td>reserved</td>
<td>unused</td>
</tr>
<tr>
<td>255</td>
<td>Literal</td>
<td>constant 32-bit constant from instruction stream.</td>
</tr>
</tbody>
</table>

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 possibly EXEC (if saveexec) will still be 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>
<thead>
<tr>
<th>Instruction</th>
<th>Encoding</th>
<th>Sets SCC?</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_ADD_I32</td>
<td>SOP2</td>
<td>y</td>
<td>D = S0 + S1, SCC = overflow.</td>
</tr>
<tr>
<td>S_ADD_U32</td>
<td>SOP2</td>
<td>y</td>
<td>D = S0 + S1, SCC = carry out.</td>
</tr>
<tr>
<td>S_ADDC_U32</td>
<td>SOP2</td>
<td>y</td>
<td>D = S0 + S1 + SCC = overflow.</td>
</tr>
<tr>
<td>S_SUB_I32</td>
<td>SOP2</td>
<td>y</td>
<td>D = S0 - S1, SCC = overflow.</td>
</tr>
<tr>
<td>S_SUB_U32</td>
<td>SOP2</td>
<td>y</td>
<td>D = S0 - S1, SCC = carry out.</td>
</tr>
<tr>
<td>S_SUBB_U32</td>
<td>SOP2</td>
<td>y</td>
<td>D = S0 - S1 - SCC = carry out.</td>
</tr>
<tr>
<td>S_ABSDIFF_I32</td>
<td>SOP2</td>
<td>y</td>
<td>D = abs (s1 - s2), SCC = result not zero.</td>
</tr>
<tr>
<td>S_MIN_I32</td>
<td>SOP2</td>
<td>y</td>
<td>D = (S0 &lt; S1) ? S0 : S1. SCC = 1 if S0 was min.</td>
</tr>
<tr>
<td>S_MAX_I32</td>
<td>SOP2</td>
<td>y</td>
<td>D = (S0 &gt; S1) ? S0 : S1. SCC = 1 if S0 was max.</td>
</tr>
<tr>
<td>S_MUL_I32</td>
<td>SOP2</td>
<td>n</td>
<td>D = S0 * S1. Low 32 bits of result.</td>
</tr>
<tr>
<td>S_ADDK_I32</td>
<td>SOPK</td>
<td>y</td>
<td>D = D + simm16, SCC = overflow. Sign extended version of simm16.</td>
</tr>
<tr>
<td>S_ABS_I32</td>
<td>SOP1</td>
<td>y</td>
<td>D.i = abs (S0.i). SCC=result not zero.</td>
</tr>
<tr>
<td>S_SEXT_I32_I8</td>
<td>SOP1</td>
<td>n</td>
<td>D = { 24[S0[7]], S0[7:0] }.</td>
</tr>
</tbody>
</table>
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 11. Conditional Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Encoding</th>
<th>Sets SCC?</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_SEXT_I32_I16</td>
<td>SOP1</td>
<td>n</td>
<td>[D = { 16[S0[15]], S0[15:0] } ].</td>
</tr>
<tr>
<td>S_CSELECT_{B32, B64}</td>
<td>SOP2</td>
<td>n</td>
<td>[D = \text{SCC} ? S0 : S1.]</td>
</tr>
<tr>
<td>S_CMOVK_I32</td>
<td>SOPK</td>
<td>n</td>
<td>[\text{if (SCC)} D = \text{signext(simm16)}.]</td>
</tr>
<tr>
<td>S_CMOV_{B32,B64}</td>
<td>SOP1</td>
<td>n</td>
<td>[\text{if (SCC)} D = S0, else NOP.]</td>
</tr>
</tbody>
</table>

5.6. Comparison Instructions

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

### Table 12. Conditional Instructions

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

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 13. Bit-Wise Instructions

5.5. Conditional Instructions
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Encoding</th>
<th>Sets SCC?</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_MOV_([B32,B64])</td>
<td>SOP1</td>
<td>n</td>
<td>D = S0</td>
</tr>
<tr>
<td>S_MOVK_[i32]</td>
<td>SOPK</td>
<td>n</td>
<td>D = signext(simm16)</td>
</tr>
<tr>
<td>{S_AND,S_OR,S_XOR}_([B32,B64])</td>
<td>SOP2</td>
<td>y</td>
<td>D = S0 &amp; S1, S0 OR S1, S0 XOR S1</td>
</tr>
<tr>
<td>{S_ANDN2,S_ORN2}_([B32,B64])</td>
<td>SOP2</td>
<td>y</td>
<td>D = S0 &amp; ~S1, S0 OR ~S1, S0 XOR ~S1</td>
</tr>
<tr>
<td>{S_NAND,S_NOR,S_XNOR}_([B32,B64])</td>
<td>SOP2</td>
<td>y</td>
<td>D = ~(S0 &amp; S1), ~(S0 OR S1), ~(S0 XOR S1)</td>
</tr>
<tr>
<td>S_LSHL_([B32,B64])</td>
<td>SOP2</td>
<td>y</td>
<td>D = S0 &lt;&lt; S1[4:0], S0 &lt;&lt; S1[5:0] for B64.</td>
</tr>
<tr>
<td>S_LSHR_([B32,B64])</td>
<td>SOP2</td>
<td>y</td>
<td>D = S0 &gt;&gt; S1[4:0], S0 &gt;&gt; S1[5:0] for B64.</td>
</tr>
<tr>
<td>S_ASHR_([i32,i64])</td>
<td>SOP2</td>
<td>y</td>
<td>D = sext(S0 &gt;&gt; S1[4:0]) ((S0 &gt;&gt; S1[5:0]) for i64).</td>
</tr>
<tr>
<td>S_BFM_([B32,B64])</td>
<td>SOP2</td>
<td>n</td>
<td>Bit field mask. D = ((1 &lt;&lt; S0[4:0]) - 1) &lt;&lt; S1[4:0].</td>
</tr>
<tr>
<td>S_BFE_U32_S_BFE_U64_S_BFE_I32_S_BFE_I64 (signed/unsigned)</td>
<td>SOP2</td>
<td>n</td>
<td>Bit Field Extract, then sign-extend result for i32/64 instructions.</td>
</tr>
<tr>
<td>S_NOT_([B32,B64])</td>
<td>SOP1</td>
<td>y</td>
<td>D = ~S0.</td>
</tr>
<tr>
<td>S_WQM_([B32,B64])</td>
<td>SOP1</td>
<td>y</td>
<td>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.</td>
</tr>
<tr>
<td>S_QUADMASK_([B32,B64])</td>
<td>SOP1</td>
<td>y</td>
<td>D[0] = OR(S0[3:0]), D[1]=OR(S0[7:4]), etc.</td>
</tr>
<tr>
<td>S_BREV_([B32,B64])</td>
<td>SOP1</td>
<td>n</td>
<td>D = S0[0:31] are reverse bits.</td>
</tr>
<tr>
<td>S_BCNT0_[i32]_([B32,B64])</td>
<td>SOP1</td>
<td>y</td>
<td>D = CountZeroBits(S0).</td>
</tr>
<tr>
<td>S_BCNT1_[i32]_([B32,B64])</td>
<td>SOP1</td>
<td>y</td>
<td>D = CountOneBits(S0).</td>
</tr>
<tr>
<td>S_FF0_[i32]_([B32,B64])</td>
<td>SOP1</td>
<td>n</td>
<td>D = Bit position of first zero in S0 starting from LSB. -1 if not found.</td>
</tr>
<tr>
<td>S_FF1_[i32]_([B32,B64])</td>
<td>SOP1</td>
<td>n</td>
<td>D = Bit position of first one in S0 starting from LSB. -1 if not found.</td>
</tr>
<tr>
<td>S_FLBIT_[i32]_([B32,B64])</td>
<td>SOP1</td>
<td>n</td>
<td>Find last bit. D = the number of zeros before the first one starting from the MSB. Returns -1 if none.</td>
</tr>
<tr>
<td>S_FLBIT_I32</td>
<td>SOP1</td>
<td>n</td>
<td>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</td>
</tr>
<tr>
<td>S_BITSET0_([B32,B64])</td>
<td>SOP1</td>
<td>n</td>
<td>D[S0[4:0], S0[5:0] for B64] = 0</td>
</tr>
</tbody>
</table>
### 5.8. Access Instructions

These instructions access hardware internal registers.

#### Table 14. Hardware Internal Registers

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Encoding</th>
<th>Sets SCC?</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_GETREG_B32</td>
<td>SOPK*</td>
<td>n</td>
<td>Read a hardware register into the LSBs of D.</td>
</tr>
<tr>
<td>S_SETREG_B32</td>
<td>SOPK*</td>
<td>n</td>
<td>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.</td>
</tr>
<tr>
<td>S_SETREG_IMM32_B32</td>
<td>SOPK*</td>
<td>n</td>
<td>S_SETREG where 32-bit data comes from a literal constant (so this is a 64-bit instruction format).</td>
</tr>
</tbody>
</table>

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 specific register to read/write:

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

#### Table 15. Hardware Register Values

<table>
<thead>
<tr>
<th>Code</th>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>MODE</td>
<td>R/W.</td>
</tr>
</tbody>
</table>

---

"Vega" 7nm Instruction Set Architecture

5.8. Access Instructions
<table>
<thead>
<tr>
<th>Code</th>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>STATUS</td>
<td>Read only.</td>
</tr>
<tr>
<td>3</td>
<td>TRAPSTS</td>
<td>R/W.</td>
</tr>
<tr>
<td>4</td>
<td>HW_ID</td>
<td>Read only. Debug only.</td>
</tr>
<tr>
<td>5</td>
<td>GPR_ALLOC</td>
<td>Read only. {sgpr_size, sgpr_base, vgpr_size, vgpr_base }.</td>
</tr>
<tr>
<td>6</td>
<td>LDS_ALLOC</td>
<td>Read only. {lds_size, lds_base}.</td>
</tr>
<tr>
<td>7</td>
<td>IB_STS</td>
<td>Read only. {valu_cnt, lgkm_cnt, exp_cnt, vm_cnt}.</td>
</tr>
<tr>
<td>8 - 15</td>
<td>reserved.</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>TBA_LO</td>
<td>Trap base address register [31:0].</td>
</tr>
<tr>
<td>17</td>
<td>TBA_HI</td>
<td>Trap base address register [47:32].</td>
</tr>
<tr>
<td>18</td>
<td>TMA_LO</td>
<td>Trap memory address register [31:0].</td>
</tr>
<tr>
<td>19</td>
<td>TMA_HI</td>
<td>Trap memory address register [47:32].</td>
</tr>
</tbody>
</table>

**Table 16. IB_STS**

<table>
<thead>
<tr>
<th>Code</th>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM_CNT</td>
<td>23:22, 3:0</td>
<td>Number of VMEM instructions issued but not yet returned.</td>
</tr>
<tr>
<td>EXP_CNT</td>
<td>6:4</td>
<td>Number of Exports issued but have not yet read their data from VGPRs.</td>
</tr>
<tr>
<td>LGKM_CNT</td>
<td>11:8</td>
<td>LDS, GDS, Constant-memory and Message instructions issued-but-not-completed count.</td>
</tr>
<tr>
<td>VALU_CNT</td>
<td>14:12</td>
<td>Number of VALU instructions outstanding for this wavefront.</td>
</tr>
</tbody>
</table>

**Table 17. GPR_ALLOC**

<table>
<thead>
<tr>
<th>Code</th>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VGPR_BASE</td>
<td>5:0</td>
<td>Physical address of first VGPR assigned to this wavefront, as [7:2]</td>
</tr>
<tr>
<td>VGPR_SIZE</td>
<td>13:8</td>
<td>Number of VGPRs assigned to this wavefront, as [7:2]. 0=4 VGPRs, 1=8 VGPRs, etc.</td>
</tr>
<tr>
<td>SGPR_BASE</td>
<td>21:16</td>
<td>Physical address of first SGPR assigned to this wavefront, as [7:3].</td>
</tr>
<tr>
<td>SGPR_SIZE</td>
<td>27:24</td>
<td>Number of SGPRs assigned to this wave, as [7:3]. 0=8 SGPRs, 1=16 SGPRs, etc.</td>
</tr>
</tbody>
</table>

**Table 18. LDS_ALLOC**

<table>
<thead>
<tr>
<th>Code</th>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDS_BASE</td>
<td>7:0</td>
<td>Physical address of first LDS location assigned to this wavefront, in units of 64 Dwords.</td>
</tr>
<tr>
<td>LDS_SIZE</td>
<td>20:12</td>
<td>Amount of LDS space assigned to this wavefront, in units of 64 Dwords.</td>
</tr>
</tbody>
</table>
Chapter 6. Vector ALU Operations

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

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

6.1. Microcode Encodings

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

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

The microcode encodings are shown below.

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

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

VOPC is for comparison instructions.

VINTRP is for parameter interpolation 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.

6.2. Operands

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

6.2.1. Instruction Inputs

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

- VGPRs.
- SGPRs.
- Inline constants - 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.
- LDS direct data read.
- M0.
- EXEC mask.

Limitations

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

Specific Cases 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.

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].

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>
<thead>
<tr>
<th>Value</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-101</td>
<td>SGPR</td>
<td>0 .. 101</td>
</tr>
<tr>
<td>102</td>
<td>FLATSCR_LO</td>
<td>Flat Scratch[31:0].</td>
</tr>
</tbody>
</table>

Table 19. Instruction Operands
<table>
<thead>
<tr>
<th>Value</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>103</td>
<td>FLATSCR_HI</td>
<td>Flat Scratch[63:32].</td>
</tr>
<tr>
<td>104</td>
<td>XNACK_MASK_LO</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td>XNACK_MASK_HI</td>
<td></td>
</tr>
<tr>
<td>106</td>
<td>VCC_LO</td>
<td>vcc[31:0].</td>
</tr>
<tr>
<td>107</td>
<td>VCC_HI</td>
<td>vcc[63:32].</td>
</tr>
<tr>
<td>108-123</td>
<td>TTMP0 to TTMP 15</td>
<td>Trap handler temps (privileged).</td>
</tr>
<tr>
<td>124</td>
<td>M0</td>
<td></td>
</tr>
<tr>
<td>125</td>
<td>reserved</td>
<td></td>
</tr>
<tr>
<td>126</td>
<td>EXEC_LO</td>
<td>exec[31:0].</td>
</tr>
<tr>
<td>127</td>
<td>EXEC_HI</td>
<td>exec[63:32].</td>
</tr>
<tr>
<td>128</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>129-192</td>
<td>int 1.. 64</td>
<td>Integer inline constants.</td>
</tr>
<tr>
<td>193-208</td>
<td>int -1 .. -16</td>
<td></td>
</tr>
<tr>
<td>209-234</td>
<td>reserved</td>
<td>Unused.</td>
</tr>
<tr>
<td>235</td>
<td>SHARED_BASE</td>
<td>Memory Aperture definition.</td>
</tr>
<tr>
<td>236</td>
<td>SHARED_LIMIT</td>
<td></td>
</tr>
<tr>
<td>237</td>
<td>PRIVATE_BASE</td>
<td></td>
</tr>
<tr>
<td>238</td>
<td>PRIVATE_LIMIT</td>
<td></td>
</tr>
<tr>
<td>239</td>
<td>POPS_EXITING_WAVE_ID</td>
<td>Primitive Ordered Pixel Shading wave ID.</td>
</tr>
<tr>
<td>240</td>
<td>0.5</td>
<td>Single, double, or half-precision inline floats.</td>
</tr>
<tr>
<td>241</td>
<td>-0.5</td>
<td>1/(2*PI) is 0.15915494.</td>
</tr>
<tr>
<td>242</td>
<td>1.0</td>
<td>The exact value used is:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>half: 0x3118</td>
</tr>
<tr>
<td></td>
<td></td>
<td>single: 0x3e22f983</td>
</tr>
<tr>
<td></td>
<td></td>
<td>double: 0x3fc45f306dc9c882</td>
</tr>
<tr>
<td>243</td>
<td>-1.0</td>
<td></td>
</tr>
<tr>
<td>244</td>
<td>2.0</td>
<td></td>
</tr>
<tr>
<td>245</td>
<td>-2.0</td>
<td></td>
</tr>
<tr>
<td>246</td>
<td>4.0</td>
<td></td>
</tr>
<tr>
<td>247</td>
<td>-4.0</td>
<td></td>
</tr>
<tr>
<td>248</td>
<td>1/(2*PI)</td>
<td></td>
</tr>
<tr>
<td>249</td>
<td>SDWA</td>
<td>Sub Dword Address (only valid as Source-0)</td>
</tr>
<tr>
<td>250</td>
<td>DPP</td>
<td>DPP over 16 lanes (only valid as Source-0)</td>
</tr>
<tr>
<td>251</td>
<td>VCCZ</td>
<td>{ zeros, VCCZ }</td>
</tr>
<tr>
<td>252</td>
<td>EXECZ</td>
<td>{ zeros, EXECZ }</td>
</tr>
<tr>
<td>253</td>
<td>SCC</td>
<td>{ zeros, SCC }</td>
</tr>
</tbody>
</table>
### 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 20. VALU Instruction Set*

<table>
<thead>
<tr>
<th>Value</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>254</td>
<td>LDS direct</td>
<td>Use LDS direct read to supply 32-bit value Vector-alu instructions only.</td>
</tr>
<tr>
<td>255</td>
<td>Literal</td>
<td>constant 32-bit constant from instruction stream.</td>
</tr>
<tr>
<td>256-511</td>
<td>VGPR</td>
<td>0 .. 255</td>
</tr>
</tbody>
</table>

### 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 20. VALU Instruction Set*

<table>
<thead>
<tr>
<th>VOP3</th>
<th>VOP3 - 1-2 operand opcodes</th>
<th>VOP2</th>
<th>VOP1</th>
</tr>
</thead>
<tbody>
<tr>
<td>V_MAD_LEGACY_F32</td>
<td>V_ADD_F64</td>
<td>V_ADD_{ F16,F32, U16,U32}</td>
<td>V_NOP</td>
</tr>
<tr>
<td>V_MAD_{ F16,I16,U16,F32}</td>
<td>V_MUL_F64</td>
<td>V_SUB_{ F16,F32,U16, U32}</td>
<td>V_MOV_B32</td>
</tr>
<tr>
<td>V_MAD_LEGACY_{F16,U16, I16}</td>
<td>V_MIN_F64</td>
<td>V_SUBREV_{ F16,F32, U16, U32}</td>
<td></td>
</tr>
<tr>
<td>V_MAD_I32_I24</td>
<td>V_MAX_F64</td>
<td>V_ADDCO_U32</td>
<td>V_READFIRSTLANE_B32</td>
</tr>
<tr>
<td>V_MAD_U32_U24</td>
<td>V_LDEXP_F64</td>
<td>V_SUBCO_U32</td>
<td>V_CVT_F32_{I32,U32,F16,F64}</td>
</tr>
<tr>
<td>V_CUBEID_F32</td>
<td>V_MUL_LO_U32</td>
<td>V_SUBREVCO_U32</td>
<td>V_CVT_{I32,U32,F16,F64}_F32</td>
</tr>
<tr>
<td>V_CUBESC_F32</td>
<td>V_MUL_HI_{I32,U32}</td>
<td>V_ADDC_U32</td>
<td>V_CVT_{I32,U32}_F64</td>
</tr>
<tr>
<td>V_CUBETC_F32</td>
<td>V_LSHLREV_B64</td>
<td>V_SUBB_U32</td>
<td>V_CVT_F64_{I32,U32}</td>
</tr>
<tr>
<td>V_CUBEMA_F32</td>
<td>V_LSHREV_B64</td>
<td>V_SUBBREV_U32</td>
<td>V_CVT_F32_{U32,F16,F64}_F32</td>
</tr>
<tr>
<td>V_BFE_{U32, I32}</td>
<td>V_ASHREV_I64</td>
<td>V_MUL_LEGACY_F32</td>
<td>V_CVF16_{U16, I16}</td>
</tr>
<tr>
<td>V_FMA_{ F16, F32, F64}</td>
<td>V_LDEXP_F32</td>
<td>V_MUL_{F16, F32}</td>
<td>V_CVT_RPI_I32_F32</td>
</tr>
<tr>
<td>V_FMA_LEGACY_F16</td>
<td>V_READLANE_B32</td>
<td>V_MULI32_I24</td>
<td>V_CVT_FLR_I32_F32</td>
</tr>
<tr>
<td>VOP3</td>
<td>VOP3 - 1-2 operand opcodes</td>
<td>VOP2</td>
<td>VOP1</td>
</tr>
<tr>
<td>-----------</td>
<td>---------------------------</td>
<td>-----------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>V_BFI_B32</td>
<td>V_WRITELANE_B32</td>
<td>V_MUL_HI_I32_I24</td>
<td>V_CVT_OFF_F32_I4</td>
</tr>
<tr>
<td>V.Lerp_U8</td>
<td>V_BCNT_U32_B32</td>
<td>V_MUL_U32_U24</td>
<td>V_FRAC_T_ (F16,F32,F64)</td>
</tr>
<tr>
<td>V_ALIGNBIT_B32</td>
<td>V_MBCNT_LO_U32_B32</td>
<td>V_MUL_HI_U32_U24</td>
<td>V_TRUNC_T_ (F16,F32, F64)</td>
</tr>
<tr>
<td>V_ALIGNBYTE_B32</td>
<td>V_MBCNT_HI_U32_B32</td>
<td>V_MIN_T_ (F16,U16, I16,F32,I32,U32)</td>
<td>V_CEIL_T_ (F16,F32, F64)</td>
</tr>
<tr>
<td>V_MIN3_(F32,I32,U32)</td>
<td>V_CVT_PKACCUM_U8_F32</td>
<td>V_MAX_T_ (F16,U16, I16,F32,I32,U32)</td>
<td>V_RNDNE_T_ (F16,F32, F64)</td>
</tr>
<tr>
<td>V_MAX3_(F32,I32,U32)</td>
<td>V_CVT_PKNORM_I16_F32</td>
<td>V_LSHREV_T_ (B16,B32)</td>
<td>V_FLOOR_T_ (F16,F32, F64)</td>
</tr>
<tr>
<td>V_MED3_(F32,I32,U32)</td>
<td>V_CVT_PKNORM_U16_F32</td>
<td>V_ASHRREV_T_ (I16,I32)</td>
<td>V_EXP_T_ (F16,F32)</td>
</tr>
<tr>
<td>V_SAD_(U8, HI_U8, U16, U32)</td>
<td>V_CVT_PKRTZ_F16_F32</td>
<td>V_LSHREV_T_ (B16,B32)</td>
<td>V_LOG_T_ (F16,F32)</td>
</tr>
<tr>
<td>V_CVT_PK_U8_F32</td>
<td>V_CVT_PK_U16_U32</td>
<td>V_AND_B32</td>
<td>V_RCP_T_ (F16,F32,F64)</td>
</tr>
<tr>
<td>V_DIV_FIXUP_(F16,F32,F64)</td>
<td>V_CVT_PK_I16_I32</td>
<td>V_OR_B32</td>
<td>V_RCP_IFLAG_F32</td>
</tr>
<tr>
<td>V_DIV_FIXUP_LEGACY_F16</td>
<td>V_MAC_LEGACY_F32</td>
<td>V_XOR_B32</td>
<td>V_RSQ_T_ (F16,F32, F64)</td>
</tr>
<tr>
<td>V_DIV_SCALE_(F32,F64)</td>
<td>V_BFM_B32</td>
<td>V_MAC_T_ (F16,F32)</td>
<td>V_SQRT_T_ (F16,F32,F64)</td>
</tr>
<tr>
<td>V_DIV_FMAS_(F32,F64)</td>
<td>V_INTERP_P1_F32</td>
<td>V_MADMK_T_ (F16,F32)</td>
<td>V_SIN_T_ (F16,F32)</td>
</tr>
<tr>
<td>V_MSAD_U8</td>
<td>V_INTERP_P2_F32</td>
<td>V_MADAK_T_ (F16,F32)</td>
<td>V_COS_T_ (F16,F32)</td>
</tr>
<tr>
<td>V_QSAD_PK_U16_U8</td>
<td>V_INTERP_MOV_F32</td>
<td>V_CNDMASK_B32</td>
<td>V_NOT_B32</td>
</tr>
<tr>
<td>V_MQASD_PK_U16_U8</td>
<td>V_INTERP_P1LL_F16</td>
<td>V_LDEXP_F16</td>
<td>V_BFREV_B32</td>
</tr>
<tr>
<td>V_MQASD_PK_U32_U8</td>
<td>V_INTERP_P1LV_F16</td>
<td>_MUL_LO_U16</td>
<td>V_FFBH_(U32, I32)</td>
</tr>
<tr>
<td>V_TRIG_PREOP_F64</td>
<td>V_INTERP_P2_F16</td>
<td>V_FBBL_B32</td>
<td></td>
</tr>
<tr>
<td>V_MAD_(U64_U32, I64_I32)</td>
<td>V_INTERP_P2_LEGACY_F16</td>
<td></td>
<td>V_FREXP_EXP_I32_F64</td>
</tr>
<tr>
<td>V_CVT_PKNORM_I16_F16</td>
<td></td>
<td></td>
<td>V_FREXP_MANT_T_ (F16,F32,F64)</td>
</tr>
<tr>
<td>V_CVT_PKNORM_U16_F16</td>
<td></td>
<td></td>
<td>V_FREXP_EXP_I32_F32</td>
</tr>
<tr>
<td>V_MAD_U32_U16</td>
<td>V_FREXP_EXP_I16_F16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V_MAD_I32_I16</td>
<td>V_CLR_EXCP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V_XAD_U32</td>
<td>V_MOV_FED_B32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V_MIN_(F16,I16,U16)</td>
<td>V_CVT_NORM_I16_F16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>V_MAX_(F16,I16, U32)</td>
<td>V_CVT_NORM_I16_F32</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

"Vega" 7nm Instruction Set Architecture
The next table lists the compare instructions.

### Table 21. VALU Instruction Set

<table>
<thead>
<tr>
<th>Op</th>
<th>Formats</th>
<th>Functions</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>V_CMP</td>
<td>I16, I32, I64, U16, U32, U64</td>
<td>F, LT, EQ, LE, GT, LG, GE, T</td>
<td>Write VCC..</td>
</tr>
<tr>
<td>V_CMPX</td>
<td>F16, F32, F64</td>
<td>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)</td>
<td>Write VCC and exec.</td>
</tr>
<tr>
<td>V_CMP_CL</td>
<td>F16, F32, F64</td>
<td>Test for one of: signaling-NaN, quiet-NaN, positive or negative: infinity, normal, subnormal, zero.</td>
<td>Write VCC.</td>
</tr>
<tr>
<td>V_CMPX_CL</td>
<td></td>
<td></td>
<td>Write VCC and exec.</td>
</tr>
</tbody>
</table>

### 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.

### Table 22. Round and Denormal Modes

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit Position</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP_ROUND</td>
<td>3.0</td>
<td>[1:0] Single-precision round mode.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Round Modes: 0=nearest even; 1= +infinity; 2= -infinity, 3= toward zero.</td>
</tr>
</tbody>
</table>
### 6.5. ALU Clamp Bit Usage

In GCN Vega Generation, the meaning of the "Clamp" bit in the VALU instructions has changed. For 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>
<thead>
<tr>
<th>Instruction</th>
<th>Encoding</th>
<th>Sets SCC?</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_SET_GPR_IDX_OFF</td>
<td>SOPP</td>
<td>N</td>
<td>Disable VGPR indexing mode. Sets: mode.gpr_idx_en = 0.</td>
</tr>
</tbody>
</table>
| 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[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. Specific Cases

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 instructions.

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

where:
src = vector source
SS = scalar source
dst = vector destination
sdst = scalar destination
6.7. Packed Math

Vega adds support for **packed math**, which performs operations on two 16-bit values within a Dword as if they were separate threads. 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:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Instruction</th>
<th>Instruction</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>V_PK_MAD_I16</td>
<td>V_PK_MUL_LO_U16</td>
<td>V_PK_ADD_I16</td>
<td>V_PK_SUB_I16</td>
</tr>
<tr>
<td>V_PK_LSHLREV_B16</td>
<td>V_PK_LSHRREV_B16</td>
<td>V_PK_ASHRREV_I16</td>
<td>V_PK_MAX_I16</td>
</tr>
<tr>
<td>V_PK_MIN_I16</td>
<td>V_PK_MAD_U16</td>
<td>V_PK_ADD_U16</td>
<td>V_PK_SUB_U16</td>
</tr>
<tr>
<td>V_PK_MAX_U16</td>
<td>V_PK_MIN_U16</td>
<td>V_PK_FMA_F16</td>
<td>V_PK_ADD_F16</td>
</tr>
<tr>
<td>V_PK_MUL_F16</td>
<td>V_PK_MIN_F16</td>
<td>V_PK_MAX_F16</td>
<td>V_MAD_MIX_F32</td>
</tr>
</tbody>
</table>

V\_MAD\_MIX\_* are not packed math, but perform a single MAD operation on a mixture of 16- and 32-bit inputs. They are listed here because they use the VOP3P encoding.
Chapter 7. 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.

7.1. 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>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP</td>
<td>8</td>
<td>Opcode.</td>
</tr>
<tr>
<td>IMM</td>
<td>1</td>
<td>Determines how the OFFSET field is interpreted. IMM=1: Offset is a 20-bit unsigned byte offset to the address. IMM=0: Offset[6:0] specifies an SGPR or M0 which provides an unsigned byte offset. STORE and ATOMIC instructions cannot use an SGPR: only imm or M0.</td>
</tr>
<tr>
<td>GLC</td>
<td>1</td>
<td>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.</td>
</tr>
<tr>
<td>SDATA</td>
<td>7</td>
<td>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.</td>
</tr>
<tr>
<td>SBASE</td>
<td>6</td>
<td>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.</td>
</tr>
<tr>
<td>OFFSET</td>
<td>20</td>
<td>An unsigned byte offset, or the address of an SGPR holding the offset. Writes and atomics: M0 or immediate only, not SGPR.</td>
</tr>
<tr>
<td>NV</td>
<td>1</td>
<td>Non-volatile.</td>
</tr>
</tbody>
</table>
### 7.2. Operations

#### 7.2.1. 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.

**Scalar Memory Addressing**

**S_LOAD / S_STORE / S_DACHE_DISCARD:**

\[
\text{ADDR} = \text{SGPR}[\text{base}] + \text{inst\_offset} + \{ \text{M0 or SGPR}[\text{offset}] \text{ or zero } \}
\]

**S_SCRATCH_LOAD / S_SCRATCH_STORE:**

\[
\text{ADDR} = \text{SGPR}[\text{base}] + \text{inst\_offset} + \{ \text{M0 or SGPR}[\text{offset}] \text{ or zero } \} * 64
\]

Use of offset fields:

<table>
<thead>
<tr>
<th>IMM</th>
<th>SOFFSET_EN (SOE)</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>\text{SGPR}[\text{base}] + (\text{SGPR}[\text{offset}] \text{ or M0})</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>\text{SGPR}[\text{base}] + (\text{SGPR}[\text{soffset}] \text{ or M0})</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>\text{SGPR}[\text{base}] + \text{inst_offset}</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>\text{SGPR}[\text{base}] + \text{inst_offset} + (\text{SGPR}[\text{soffset}] \text{ or M0})</td>
</tr>
</tbody>
</table>

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 (\text{inst\_offset} + (\text{M0 or SGPR}[\text{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.

Atomics are a different case 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#):

\[
\begin{align*}
\text{offset} & = \text{IMM} ? \text{OFFSET} : \text{SGPR}[\text{OFFSET}] \\
\text{m_base} & = \{ \text{SGPR}[\text{SBASE} \times 2 +1][15:0], \text{SGPR}[\text{SBASE}] \} \\
\text{m_stride} & = \text{SGPR}[\text{SBASE} \times 2 +1][31:16] \\
\text{m_num_records} & = \text{SGPR}[\text{SBASE} \times 2 + 2] \\
\text{m_size} & = (\text{m_stride} == 0) ? 1 : \text{m_num_records} \\
\text{m_addr} & = (\text{SGPR}[\text{SBASE} \times 2] + \text{offset}) & ~0x3 \\
\text{SGPR}[\text{SDST}] & = \text{read_Dword_from_dcache} (\text{m_base}, \text{offset}, \text{m_size})
\end{align*}
\]

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.

7.2.2. 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.

7.2.3. S_DCACHE_INV, S_DCACHE_WB

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

7.2.4. S_MEMTIME

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

7.2.5. 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 clock for which the frequency is constant (not affected by power modes or core clock frequency changes).

7.3. 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.

7.4. Alignment and Bounds Checking

SDST

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

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

OFFSET

The value of OFFSET has no alignment restrictions.

Memory Address : If the memory address is out-of-range (clamped), the operation is not performed for any Dwords that are out-of-range.
Chapter 8. Vector Memory Operations

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

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

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

The instruction defines which VGPR(s) supply the addresses for the operation, which VGPRs supply or receive data from the operation, and a series of SGPRs that contain the memory buffer descriptor (V# or T#). Also, MIMG operations supply a texture sampler from a series of four SGPRs; this sampler defines texel filtering operations to be performed on data read from the image.

8.1. Vector Memory Buffer Instructions

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

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

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

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

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

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

All VM operations use a buffer resource constant (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.

### 8.1.1. Simplified Buffer Addressing

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

\[
ADDR = V# + baseOffset + Inst_offset + Voffset + Stride * (Vindex + TID)
\]

- `Voffset` is ignored when instruction bit "OFFEN" == 0
- `Vindex` is ignored when instruction 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

### 8.1.2. Buffer Instructions

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

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

### Table 25. Buffer Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MTBUF Instructions</strong></td>
<td></td>
</tr>
<tr>
<td>TBUFFER_LOAD_FORMAT_{x,xy,xyz,xyzw}</td>
<td>Read from, or write to, a typed buffer object. Also used for a vertex fetch.</td>
</tr>
<tr>
<td>TBUFFER_STORE_FORMAT_{x,xy,xyz,xyzw}</td>
<td></td>
</tr>
<tr>
<td><strong>MUBUF Instructions</strong></td>
<td></td>
</tr>
<tr>
<td>BUFFER_LOAD_FORMAT_{x,xy,xyz,xyzw}</td>
<td>Read to, or write from, an untyped buffer object.</td>
</tr>
<tr>
<td>BUFFER_STORE_FORMAT_{x,xy,xyz,xyzw}</td>
<td></td>
</tr>
<tr>
<td>BUFFER_LOAD_{&lt;size&gt;}</td>
<td></td>
</tr>
<tr>
<td>BUFFER_STORE_{&lt;size&gt;}</td>
<td></td>
</tr>
</tbody>
</table>

### Table 26. Microcode Formats
<table>
<thead>
<tr>
<th>Field</th>
<th>Bit Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP</td>
<td>4</td>
<td>MTBUF: Opcode for Typed buffer instructions. MUBUF: Opcode for Untyped buffer instructions.</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>VADDR</td>
<td>8</td>
<td>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.</td>
</tr>
<tr>
<td>VDATA</td>
<td>8</td>
<td>Address of VGPR to supply first component of write data or receive first component of read-data.</td>
</tr>
<tr>
<td>SOFFSET</td>
<td>8</td>
<td>SGPR to supply unsigned byte offset. Must be an SGPR, M0, or inline constant.</td>
</tr>
<tr>
<td>SR SRC</td>
<td>5</td>
<td>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.</td>
</tr>
<tr>
<td>DFMT</td>
<td>4</td>
<td>Data Format of data in memory buffer:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 invalid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 8_8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 16_16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6 10_11_11</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7 11_11_10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 10_10_10_2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9 2_10_10_10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10 8_8_8_8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11 32_32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12 16_16_16_16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13 32_32_32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>14 32_32_32_32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15 reserved</td>
</tr>
<tr>
<td>NFMT</td>
<td>3</td>
<td>Numeric format of data in memory:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 unorm</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 snorm</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 uscaled</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 sscaled</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 uint</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 sint</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6 reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7 float</td>
</tr>
<tr>
<td>OFFSET</td>
<td>12</td>
<td>Unsigned byte offset.</td>
</tr>
<tr>
<td>OFFEN</td>
<td>1</td>
<td>1 = Supply an offset from VGPR (VADDR). 0 = Do not (offset = 0).</td>
</tr>
<tr>
<td>IDXEN</td>
<td>1</td>
<td>1 = Supply an index from VGPR (VADDR). 0 = Do not (index = 0).</td>
</tr>
</tbody>
</table>
8.1.3. VGPR Usage

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

**Address**

Zero, one or two VGPRs are used, depending of the offset-enable (OFFEN) and index-enable (IDXEN) in the instruction word, as shown in the table below:

<table>
<thead>
<tr>
<th>IDXEN</th>
<th>OFFEN</th>
<th>VGPRn</th>
<th>VGPRn+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>nothing</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>uint</td>
<td>offset</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>uint</td>
<td>index</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>uint</td>
<td>index</td>
</tr>
</tbody>
</table>

**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.

### 8.1.4. Buffer Data

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

**Table 28. Buffer Instructions**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Data Format</th>
<th>Num Format</th>
<th>DST SEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBUFFER_LOAD_FORMAT_*</td>
<td>instruction</td>
<td>instruction</td>
<td>identity</td>
</tr>
<tr>
<td>TBUFFER_STORE_FORMAT_*</td>
<td>instruction</td>
<td>instruction</td>
<td>identity</td>
</tr>
<tr>
<td>BUFFER_LOAD_&lt;type&gt;</td>
<td>derived</td>
<td>derived</td>
<td>identity</td>
</tr>
<tr>
<td>BUFFER_STORE_&lt;type&gt;</td>
<td>derived</td>
<td>derived</td>
<td>identity</td>
</tr>
<tr>
<td>BUFFER_LOAD_FORMAT_*</td>
<td>resource</td>
<td>resource</td>
<td>resource</td>
</tr>
<tr>
<td>BUFFER_STORE_FORMAT_*</td>
<td>resource</td>
<td>resource</td>
<td>resource</td>
</tr>
<tr>
<td>BUFFER_ATOMIC_*</td>
<td>derived</td>
<td>derived</td>
<td>identity</td>
</tr>
</tbody>
</table>

**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.

### 8.1.5. 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 29. BUFFER Instruction Fields for Addressing**

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inst_offset</td>
<td>12</td>
<td>Literal byte offset from the instruction.</td>
</tr>
<tr>
<td>inst_idxen</td>
<td>1</td>
<td>Boolean: get index from VGPR when true, or no index when false.</td>
</tr>
<tr>
<td>inst_offen</td>
<td>1</td>
<td>Boolean: get offset from VGPR when true, or no offset when false. Note that inst_offset is present, regardless of this bit.</td>
</tr>
</tbody>
</table>

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 30. V# Buffer Resource Constant Fields for Addressing**

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>const_base</td>
<td>48</td>
<td>Base address, in bytes, of the buffer resource.</td>
</tr>
</tbody>
</table>
| 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]}
```

<table>
<thead>
<tr>
<th>const_num_records</th>
<th>32</th>
<th>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</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field</td>
<td>Size</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>const_add_tid_enable</td>
<td>1</td>
<td>Boolean. Add thread_ID within the wavefront to the index when true.</td>
</tr>
<tr>
<td>const_swizzle_enable</td>
<td>1</td>
<td>Boolean. Indicates that the surface is swizzled when true.</td>
</tr>
<tr>
<td>const_element_size</td>
<td>2</td>
<td>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 &gt;= the maximum element size in the structure. const_stride must be an integer multiple of const_element_size.</td>
</tr>
<tr>
<td>const_index_stride</td>
<td>2</td>
<td>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.</td>
</tr>
</tbody>
</table>

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.

**Address Calculation for a Linear Buffer**

ADDRESS = const_base + sgpr_offset + buffer_offset

\[
\text{Buffer_Offset} = (\text{inst_offset} + \text{vgpr_offset}) + \text{const_stride} \times (\text{vgpr_index} + \text{ThreadId})
\]

Full equations:

\[
\text{Index} = \text{inst_idxen} \, \text{vgpr_index} + (\text{const_addtid_enable} \, \text{thread_id}[5:0])
\]

\[
\text{Offset} = \text{inst_offen} \, \text{vgpr_offset} + \text{inst_offset}
\]

**Figure 4. Address Calculation for a Linear Buffer**
Range Checking

Addresses can be checked to see if they are in or out of range. When an address is out of range, reads will return zero, and writes and atomics will be dropped. The address range check algorithm 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.

Swizzled Buffer Addressing

Swizzled addressing rearranges the data in the buffer and can 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_idxn ? 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.
Figure 5. Example of Buffer Swizzling

Proposed Use Cases for Swizzled Addressing

Here are few proposed uses of swizzled addressing in common graphics buffers.
Table 32. Swizzled Buffer Use Cases

<table>
<thead>
<tr>
<th></th>
<th>DX11 Raw Uav OpenCL Buffer Object</th>
<th>Dx11 Structured (literal offset)</th>
<th>Dx11 Structured (gpr offset)</th>
<th>Scratch</th>
<th>Ring / stream-out</th>
<th>Const Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>inst_vgpr_offset_enabled</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>inst_vgpr_index_enabled</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>const_stride</td>
<td>na</td>
<td>&lt;api&gt;</td>
<td>&lt;api&gt;</td>
<td>scratchSize</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>const_add_tid_enable</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>const_buffer_swizzle</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>const_elem_size</td>
<td>na</td>
<td>4</td>
<td>4</td>
<td>4 or 16</td>
<td>na</td>
<td>4</td>
</tr>
<tr>
<td>const_index_stride</td>
<td>na</td>
<td>16</td>
<td>16</td>
<td>64</td>
<td>na</td>
<td>64</td>
</tr>
</tbody>
</table>

8.1.6. 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.

8.1.7. Alignment

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

8.1.8. 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 33. Buffer Resource Descriptor
<table>
<thead>
<tr>
<th>Bits</th>
<th>Size</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>47:0</td>
<td>48</td>
<td>Base address</td>
<td>Byte address.</td>
</tr>
<tr>
<td>61:48</td>
<td>14</td>
<td>Stride</td>
<td>Bytes 0 to 16383</td>
</tr>
<tr>
<td>62</td>
<td>1</td>
<td>Cache swizzle</td>
<td>Buffer access. Optionally, swizzle texture cache TC L1 cache banks.</td>
</tr>
<tr>
<td>63</td>
<td>1</td>
<td>Swizzle enable</td>
<td>Swizzle AOS according to stride, index_stride, and element_size, else linear (stride * index + offset).</td>
</tr>
<tr>
<td>95:64</td>
<td>32</td>
<td>Num_records</td>
<td>In units of stride or bytes.</td>
</tr>
<tr>
<td>98:96</td>
<td>3</td>
<td>Dst_sel_x</td>
<td>Destination channel select: 0=0, 1=1, 4=R, 5=G, 6=B, 7=A</td>
</tr>
<tr>
<td>101:99</td>
<td>3</td>
<td>Dst_sel_y</td>
<td></td>
</tr>
<tr>
<td>104:102</td>
<td>3</td>
<td>Dst_sel_z</td>
<td></td>
</tr>
<tr>
<td>107:105</td>
<td>3</td>
<td>Dst_sel_w</td>
<td></td>
</tr>
<tr>
<td>110:108</td>
<td>3</td>
<td>Num format</td>
<td>Numeric data type (float, int, …). See instruction encoding for values.</td>
</tr>
<tr>
<td>114:111</td>
<td>4</td>
<td>Data format</td>
<td>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].</td>
</tr>
<tr>
<td>115</td>
<td>1</td>
<td>User VM Enable</td>
<td>Resource is mapped via tiled pool / heap.</td>
</tr>
<tr>
<td>116</td>
<td>1</td>
<td>User VM mode</td>
<td>Unmapped behavior: 0: null (return 0 / drop write); 1:invalid (results in error)</td>
</tr>
<tr>
<td>118:117</td>
<td>2</td>
<td>Index stride</td>
<td>8, 16, 32, or 64. Used for swizzled buffer addressing.</td>
</tr>
<tr>
<td>119</td>
<td>1</td>
<td>Add tid enable</td>
<td>Add thread ID to the index for to calculate the address.</td>
</tr>
<tr>
<td>122:120</td>
<td>3</td>
<td>RSVD</td>
<td>Reserved. Must be set to zero.</td>
</tr>
<tr>
<td>123</td>
<td>1</td>
<td>NV</td>
<td>Non-volatile (0=volatile)</td>
</tr>
<tr>
<td>125:124</td>
<td>2</td>
<td>RSVD</td>
<td>Reserved. Must be set to zero.</td>
</tr>
<tr>
<td>127:126</td>
<td>2</td>
<td>Type</td>
<td>Value == 0 for buffer. Overlaps upper two bits of four-bit TYPE field in 128-bit T# resource.</td>
</tr>
</tbody>
</table>

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

### 8.1.9. 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}.
- It is illegal to set the instruction’s TFE bit for loads to LDS.
LDS_offset = 16-bit unsigned byte offset from M0[15:0].
Mem_offset = 32-bit unsigned byte offset from an SGPR (the SOFFSET SGPR).
idx_vgpr = index value from a VGPR (located at VADDR). (Zero if idxen=0.)
off_vgpr = offset value from a VGPR (located at VADDR or VADDR+1). (Zero if offen=0.)

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

\[
\text{LDS}\_\text{ADDR} = \text{LDSbase} + \text{LDS}\_\text{offset} + \text{inst}_{\text{offset}} + (\text{TIDinWave} \times 4)
\]

\[
\text{MEM}\_\text{ADDR} = \text{Base} + \text{mem}\_\text{offset} + \text{inst}_{\text{offset}} + \text{off}_{\text{vgpr}} + \text{stride} \times (\text{idx}_{\text{vgpr}} + \text{TIDinWave})
\]

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

Clamping Rules

Memory address clamping follows the same rules as any other buffer fetch. LDS address clamping: the return data must not be written outside the LDS space allocated to this wave.

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

8.1.10. GLC Bit Explained

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

GLC Meaning for Loads

- For GLC==0
  - The load can read data from the GPU L1.
  - Typically, all loads (except load-acquire) use GLC==0.
- For GLC==1
  - The load intentionally misses the GPU L1 and reads from L2. If there was a line in the GPU L1 that matched, it is invalidated; L2 is reread.
NOTE: L2 is not re-read for every work-item in the same wave-front for a single load instruction. For example: b=uav[N+tid] // assume this is a byte read w/ glc==1 and N is aligned to 64B In the above op, the first Tid of the wavefront brings in the line from L2 or beyond, and all 63 of the other Tids read from same 64 B cache line in the L1.

GLC Meaning for Stores

• For GLC==0 This causes a write-combine across work-items of the wavefront store op; dirtied lines are written to the L2 automatically.
  ◦ If the store operation dirtied all bytes of the 64 B line, it is left clean and valid in the L1; subsequent accesses to the cache are allowed to hit on this cache line.
  ◦ Else do not leave write-combined lines in L1.
• For GLC==1 Same as GLC==0, except the write-combined lines are not left in the line, even if all bytes are dirtied.

Atomics

• For GLC == 0 No return data (this is “write-only” atomic op).
• For GLC == 1 Returns previous value in memory (before the atomic operation).

8.2. Vector Memory (VM) Image Instructions

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

Examples of image objects are texture maps and typed surfaces.

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

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

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

### 8.2.1. Image Instructions

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

#### Table 34. Image Instructions

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

#### Table 35. Instruction Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP</td>
<td>7</td>
<td>Opcode.</td>
</tr>
<tr>
<td>VADDR</td>
<td>8</td>
<td>Address of VGPR to supply first component of address.</td>
</tr>
<tr>
<td>VDATA</td>
<td>8</td>
<td>Address of VGPR to supply first component of write data or receive first component of read-data.</td>
</tr>
<tr>
<td>SSAMP</td>
<td>5</td>
<td>SGPR to supply S# (sampler constant) in four consecutive SGPRs. Missing two LSBs of SGPR-address since must be aligned to a multiple of four SGPRs.</td>
</tr>
<tr>
<td>SRSRC</td>
<td>5</td>
<td>SGPR to supply T# (resource constant) in four or eight consecutive SGPRs. Missing two LSBs of SGPR-address since must be aligned to a multiple of four SGPRs.</td>
</tr>
<tr>
<td>UNRM</td>
<td>1</td>
<td>Force address to be un-normalized regardless of T#. Must be set to 1 for image stores and atomics.</td>
</tr>
<tr>
<td>DA</td>
<td>1</td>
<td>Shader declared an array resource to be used with this fetch. When 1, the shader provides an array-index with the instruction. When 0, no array index is provided.</td>
</tr>
<tr>
<td>DMASK</td>
<td>4</td>
<td>Data VGPR enable mask: one to four consecutive VGPRs. Reads: defines which components are returned. 0 = red, 1 = green, 2 = blue, 3 = alpha Writes: defines which components are written with data from VGPRs (missing components get 0). Enabled components come from consecutive VGPRs. For example: DMASK=1001: Red is in VGPRn and alpha in VGPRn+1. For D16 writes, DMASK is used only as a word count: each bit represents 16 bits of data to be written, starting at the LSBs of VADDR, the MSBs, VADDR+1, etc. Bit position is ignored.</td>
</tr>
</tbody>
</table>
### Field | Bit Size | Description
--- | --- | ---
GLC | 1 | Globally Coherent. Controls how reads and writes are handled by the L1 texture cache.  
**READ:**  
GLC = 0 Reads can hit on the L1 and persist across waves.  
GLC = 1 Reads miss the L1 and force fetch to L2. No L1 persistence across waves.  
**WRITE:**  
GLC = 0 Writes miss the L1, write through to L2, and persist in L1 across wavefronts.  
GLC = 1 Writes miss the L1, write through to L2. No persistence across wavefronts.  
**ATOMIC:**  
GLC = 0 Previous data value is not returned. No L1 persistence across wavefronts.  
GLC = 1 Previous data value is returned. No L1 persistence across wavefronts.
SLC | 1 | System Level Coherent. When set, accesses are forced to miss in level 2 texture cache and are coherent with system memory.
TFE | 1 | Texel Fail Enable for PRT (partially resident textures). When set, a fetch can return a NACK, which causes a VGPR write into DST+1 (first GPR after all fetch-dest GPRs).
LWE | 1 | LOD Warning Enable. When set to 1, a texture fetch may return "LOD_CLAMPED = 1".
A16 | 1 | Address components are 16-bits (instead of the usual 32 bits). When set, all address components are 16 bits (packed into two per Dword), except:  
Texel offsets (three 6-bit uint packed into one Dword).  
PCF reference (for _C instructions).  
Address components are 16-bit uint for image ops without sampler; 16-bit float with sampler.
D16 | 1 | VGPR-Data-16bit. On loads, convert data in memory to 16-bit format before storing it in VGPRs. For stores, convert 16-bit data in VGPRs to 32 bits before going to memory. Whether the data is treated as float or int is decided by NFMT. Allowed only with these opcodes:  
IMAGE_SAMPLE*  
IMAGE_GATHER4*, but not GATHER4H_PCK  
IMAGE_LOAD  
IMAGE_LOAD_MIP  
IMAGE_STORE  
IMAGE_STORE_MIP

### 8.3. Image Opcodes with No Sampler

For image opcodes with no sampler, all VGPR address values are taken as uint. For cubemaps, face_id = slice * 6 + face.

The table below shows the contents of address VGPRs for the various image opcodes.

**Table 36. Image Opcodes with No Sampler**

<table>
<thead>
<tr>
<th>Image Opcode (Resource w/o Sampler)</th>
<th>Acnt</th>
<th>dim</th>
<th>VGPRn</th>
<th>VGPRn+1</th>
<th>VGPRn+2</th>
<th>VGPRn+3</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_resinfo</td>
<td>0</td>
<td>Any</td>
<td>mipid</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
8.4. Image Opcodes with a Sampler

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

Certain sample and gather opcodes require additional values from VGPRs beyond what is shown. These values are: offset, bias, z-compare, and gradients.

### Table 37. Image Opcodes with Sampler

<table>
<thead>
<tr>
<th>Image Opcode (w/ Sampler)</th>
<th>Acnt</th>
<th>dim</th>
<th>VGPRn</th>
<th>VGPRn+1</th>
<th>VGPRn+2</th>
<th>VGPRn+3</th>
</tr>
</thead>
<tbody>
<tr>
<td>sample</td>
<td>0</td>
<td>1D</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1D Array</td>
<td>x</td>
<td>slice</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2D</td>
<td>x</td>
<td>y</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2D MSAA</td>
<td>x</td>
<td>y</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2D Array</td>
<td>x</td>
<td>y</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2D Array MSAA</td>
<td>x</td>
<td>y</td>
<td>slice</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3D</td>
<td>x</td>
<td>y</td>
<td>z</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Cube</td>
<td>x</td>
<td>y</td>
<td>face_id</td>
<td></td>
</tr>
<tr>
<td>load_mip / store_mip</td>
<td>1</td>
<td>1D</td>
<td>x</td>
<td>mipid</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1D Array</td>
<td>x</td>
<td>slice</td>
<td>mipid</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2D</td>
<td>x</td>
<td>y</td>
<td>mipid</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2D Array</td>
<td>x</td>
<td>y</td>
<td>slice</td>
<td>mipid</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3D</td>
<td>x</td>
<td>y</td>
<td>z</td>
<td>mipid</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Cube</td>
<td>x</td>
<td>y</td>
<td>face_id</td>
<td>mipid</td>
</tr>
<tr>
<td>Image Opcode (w/ Sampler)</td>
<td>Acnt</td>
<td>dim</td>
<td>VGPRn</td>
<td>VGPRn+1</td>
<td>VGPRn+2</td>
<td>VGPRn+3</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------</td>
<td>-------</td>
<td>-------</td>
<td>---------</td>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>sample_l</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1D</td>
<td>x</td>
<td>lod</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1D Array</td>
<td>x</td>
<td>slice</td>
<td>lod</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2D</td>
<td>x</td>
<td>y</td>
<td>lod</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2D interlaced</td>
<td>x</td>
<td>y</td>
<td>field</td>
<td>lod</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2D Array</td>
<td>x</td>
<td>y</td>
<td>slice</td>
<td>lod</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3D</td>
<td>x</td>
<td>y</td>
<td>z</td>
<td>lod</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Cube</td>
<td>x</td>
<td>y</td>
<td>face_id</td>
<td>lod</td>
</tr>
<tr>
<td>sample_cl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1D</td>
<td>x</td>
<td>clamp</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1D Array</td>
<td>x</td>
<td>slice</td>
<td>clamp</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2D</td>
<td>x</td>
<td>y</td>
<td>clamp</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2D interlaced</td>
<td>x</td>
<td>y</td>
<td>field</td>
<td>clamp</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2D Array</td>
<td>x</td>
<td>y</td>
<td>slice</td>
<td>clamp</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3D</td>
<td>x</td>
<td>y</td>
<td>z</td>
<td>clamp</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Cube</td>
<td>x</td>
<td>y</td>
<td>face_id</td>
<td>clamp</td>
</tr>
<tr>
<td>gather4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2D</td>
<td>x</td>
<td>y</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2D interlaced</td>
<td>x</td>
<td>y</td>
<td>field</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2D Array</td>
<td>x</td>
<td>y</td>
<td>slice</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Cube</td>
<td>x</td>
<td>y</td>
<td>face_id</td>
<td></td>
</tr>
<tr>
<td>gather4_l</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2D</td>
<td>x</td>
<td>y</td>
<td>lod</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2D interlaced</td>
<td>x</td>
<td>y</td>
<td>field</td>
<td>lod</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2D Array</td>
<td>x</td>
<td>y</td>
<td>slice</td>
<td>lod</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Cube</td>
<td>x</td>
<td>y</td>
<td>face_id</td>
<td>lod</td>
</tr>
<tr>
<td>gather4_cl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2D</td>
<td>x</td>
<td>y</td>
<td>clamp</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2D interlaced</td>
<td>x</td>
<td>y</td>
<td>field</td>
<td>clamp</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2D Array</td>
<td>x</td>
<td>y</td>
<td>slice</td>
<td>clamp</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Cube</td>
<td>x</td>
<td>y</td>
<td>face_id</td>
<td>clamp</td>
</tr>
</tbody>
</table>

1. Sample includes sample, sample_d, sample_b, sample_lz, sample_c, sample_c_d, sample_c_b, sample_c_lz, and getlod.
2. Sample_l includes sample_l and sample_c_l.
3. Sample_cl includes sample_cl, sample_d_cl, sample_b_cl, sample_c_cl, sample_c_d_cl, and sample_c_b_cl.
4. Gather4 includes gather4, gather4_lz, gather4_c, and gather4_c_lz.
The table below lists and briefly describes the legal suffixes for image instructions:

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

8.4.1. VGPR Usage

Address: The address consists of up to four parts:

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

These are all packed into consecutive VGPRs.

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

<table>
<thead>
<tr>
<th>Image Dim</th>
<th>Vgpr N</th>
<th>N+1</th>
<th>N+2</th>
<th>N+3</th>
<th>N+4</th>
<th>N+5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1D</td>
<td>DX/DH</td>
<td>DX/DV</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2D</td>
<td>DX/DH</td>
<td>DY/DH</td>
<td>DX/DV</td>
<td>DY/DV</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3D</td>
<td>DX/DH</td>
<td>DY/DH</td>
<td>DZ/DH</td>
<td>DX/DV</td>
<td>DY/DV</td>
<td>DZ/DV</td>
</tr>
</tbody>
</table>

- Body: One to four Dwords, as defined by the table: [Image Opcodes with Sampler] Address components are X,Y,Z,W with X in VGPR_M, Y in VGPR_M+1, etc. The number of components in "body" is the value of the ACNT field in the table, plus one.
- Data: Written from, or returned to, one to four consecutive VGPRs. The amount of data read
or written is determined by the DMASK field of the instruction.

- **Reads:** DMASK specifies which elements of the resource are returned to consecutive VGPRs. The texture system reads data from memory and based on the data format expands it to a canonical RGBA form, filling in zero or one for missing components. Then, DMASK is applied, and only those components selected are returned to the shader.

- **Writes:** When writing an image object, it is only possible to write an entire element (all components), not just individual components. The components come from consecutive VGPRs, and the texture system fills in the value zero for any missing components of the image’s data format; it ignores any values that are not part of the stored data format. For example, if the DMASK=1001, the shader sends Red from VGPR_N, and Alpha from VGPR_N+1, to the texture unit. If the image object is RGB, the texel is overwritten with Red from the VGPR_N, Green and Blue set to zero, and Alpha from the shader ignored.

- **Atomics:** Image atomic operations are supported only on 32- and 64-bit-per pixel surfaces. The surface data format is specified in the resource constant. Atomic operations treat the element as a single component of 32- or 64-bits. For atomic operations, DMASK is set to the number of VGPRs (Dwords) to send to the texture unit. DMASK legal values for atomic image operations: no other values of DMASK are legal.
  - 0x1 = 32-bit atomics except cmpswap.
  - 0x3 = 32-bit atomic cmpswap.
  - 0x3 = 64-bit atomics except cmpswap.
  - 0xf = 64-bit atomic cmpswap.

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

- **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. The 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). The DMASK bit represents individual 16-bit elements; so, when DMASK=0011 for an image-load, two 16-bit components are loaded into a single 32-bit VGPR.

### 8.4.2. Image Resource

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

<table>
<thead>
<tr>
<th>Bits</th>
<th>Size</th>
<th>Name</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>128-bit Resource: 1D-tex, 2d-tex, 2d-msaa (multi-sample auto-aliasing)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>39:0</td>
<td>40</td>
<td>base address</td>
<td>256-byte aligned. Also used for fmask-ptr.</td>
</tr>
<tr>
<td>51:40</td>
<td>12</td>
<td>min lod</td>
<td>4.8 (four uint bits, eight fraction bits) format.</td>
</tr>
<tr>
<td>Bits</td>
<td>Size</td>
<td>Name</td>
<td>Comments</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
<td>----------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>57:52</td>
<td>6</td>
<td>data format</td>
<td>Number of comps, number of bits/comp.</td>
</tr>
<tr>
<td>61:58</td>
<td>4</td>
<td>num format</td>
<td>Numeric format.</td>
</tr>
<tr>
<td>62</td>
<td>1</td>
<td>NV</td>
<td>Non-volatile (0=volatile)</td>
</tr>
<tr>
<td>77:64</td>
<td>14</td>
<td>width</td>
<td>width-1 of mip0 in texels</td>
</tr>
<tr>
<td>91:78</td>
<td>14</td>
<td>height</td>
<td>height-1 of mip0 in texels</td>
</tr>
<tr>
<td>94:92</td>
<td>3</td>
<td>perf modulation</td>
<td>Scales sampler's perf_z, perf_mip, aniso_bias, lod_bias_sec.</td>
</tr>
<tr>
<td>98:96</td>
<td>3</td>
<td>dst_sel_x</td>
<td>0 = 0, 1 = 1, 4 = R, 5 = G, 6 = B, 7 = A.</td>
</tr>
<tr>
<td>101:99</td>
<td>3</td>
<td>dst_sel_y</td>
<td></td>
</tr>
<tr>
<td>104:102</td>
<td>3</td>
<td>dst_sel_z</td>
<td></td>
</tr>
<tr>
<td>107:105</td>
<td>3</td>
<td>dst_sel_w</td>
<td></td>
</tr>
<tr>
<td>111:108</td>
<td>4</td>
<td>base level</td>
<td>largest mip level in the resource view. For msaa, set to zero.</td>
</tr>
<tr>
<td>115:112</td>
<td>4</td>
<td>last level</td>
<td>For msaa, holds number of samples</td>
</tr>
<tr>
<td>120:116</td>
<td>5</td>
<td>Tiling index</td>
<td>Lookuptable: 32 x 16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bank_width[2], bank_height[2], num_banks[2], tile_split[2],</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>macro_tile_aspect[2], micro_tile_mode[2], array_mode[4].</td>
</tr>
<tr>
<td>127:124</td>
<td>4</td>
<td>type</td>
<td>0 = buf, 8 = 1d, 9 = 2d, 10 = 3d, 11 = cube, 12 = 1d-array, 13 = 2d-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>array, 14 = 2d-msaa, 15 = 2d-msaa-array. 1-7 are reserved.</td>
</tr>
</tbody>
</table>

**256-bit Resource: 1d-array, 2d-array, 3d, cubemap, MSAA**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Size</th>
<th>Name</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>140:128</td>
<td>13</td>
<td>depth</td>
<td>depth-1 of mip0 for 3d map</td>
</tr>
<tr>
<td>156:141</td>
<td>16</td>
<td>pitch</td>
<td>In texel units.</td>
</tr>
<tr>
<td>159:157</td>
<td>3</td>
<td>border color swizzle</td>
<td>Specifies the channel ordering for border color independent of the T# dst_sel fields. 0=xyzw, 1=xwyz, 2=wqyx, 3=wxyz, 4=zyxw, 5=xywz</td>
</tr>
<tr>
<td>176:173</td>
<td>4</td>
<td>Array Pitch</td>
<td>array pitch for quilts, encoded as: trunc(log2(array_pitch))+1</td>
</tr>
<tr>
<td>184:177</td>
<td>8</td>
<td>meta data address</td>
<td>bits[47:40]</td>
</tr>
<tr>
<td>185</td>
<td>1</td>
<td>meta_linear</td>
<td>forces metadata surface to be linear</td>
</tr>
<tr>
<td>186</td>
<td>1</td>
<td>meta_pipe_aligned</td>
<td>maintain pipe alignment in metadata addressing</td>
</tr>
<tr>
<td>187</td>
<td>1</td>
<td>meta_rb_aligned</td>
<td>maintain RB alignment in metadata addressing</td>
</tr>
<tr>
<td>191:188</td>
<td>4</td>
<td>Max Mip</td>
<td>Resource mipLevel-1. Describes the resource, as opposed to base_level and last_level, which describes the resource view. For MSAA, holds log2(number of samples).</td>
</tr>
<tr>
<td>203:192</td>
<td>12</td>
<td>min LOD warn</td>
<td>Feedback trigger for LOD, in U4.8 format.</td>
</tr>
<tr>
<td>211:204</td>
<td>8</td>
<td>counter bank ID</td>
<td>PRT counter ID</td>
</tr>
<tr>
<td>212</td>
<td>1</td>
<td>LOD hardware count enable</td>
<td>PRT hardware counter enable</td>
</tr>
<tr>
<td>213</td>
<td>1</td>
<td>Compression Enable</td>
<td>enable delta color compression</td>
</tr>
</tbody>
</table>
All image resource view descriptors (T#'s) are written by the driver as 256 bits.

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

### 8.4.3. Image Sampler

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

#### Table 40. Image Sampler Definition

<table>
<thead>
<tr>
<th>Bits</th>
<th>Size</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2:0</td>
<td>3</td>
<td>clamp x</td>
<td>Clamp/wrap mode.</td>
</tr>
<tr>
<td>5:3</td>
<td>3</td>
<td>clamp y</td>
<td></td>
</tr>
<tr>
<td>8:6</td>
<td>3</td>
<td>clamp z</td>
<td></td>
</tr>
<tr>
<td>11:9</td>
<td>3</td>
<td>max aniso ratio</td>
<td></td>
</tr>
<tr>
<td>14:12</td>
<td>3</td>
<td>depth compare func</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>force unnormalized</td>
<td>Force address cords to be unorm.</td>
</tr>
<tr>
<td>18:16</td>
<td>3</td>
<td>aniso threshold</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>mc coord trunc</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>force degamma</td>
<td></td>
</tr>
<tr>
<td>26:21</td>
<td>6</td>
<td>aniso bias</td>
<td>u1.5.</td>
</tr>
<tr>
<td>27</td>
<td>1</td>
<td>trunc coord</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>1</td>
<td>disable cube wrap</td>
<td></td>
</tr>
<tr>
<td>30:29</td>
<td>2</td>
<td>filter_mode</td>
<td>Normal lerp, min, or max filter.</td>
</tr>
<tr>
<td>31</td>
<td>1</td>
<td>compat_mode</td>
<td>1 = new mode; 0 = legacy</td>
</tr>
<tr>
<td>43:32</td>
<td>12</td>
<td>min lod</td>
<td>u4.8.</td>
</tr>
<tr>
<td>55:44</td>
<td>12</td>
<td>max lod</td>
<td>u4.8.</td>
</tr>
</tbody>
</table>
8.4.4. Data Formats

Data formats 0-15 are available to buffer resources, and all formats are available to image formats. The table below details all the data formats that can be used by image and buffer resources.
### 8.4.5. Vector Memory Instruction Data Dependencies

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

The shader developer’s responsibility to avoid data hazards associated with VMEM instructions include waiting for VMEM read instruction completion before reading data fetched from the TC.
(VMCNT).

This is explained in the section: Data Dependency Resolution
Chapter 9. Flat Memory Instructions

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

9.1. Flat Memory 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>
<thead>
<tr>
<th>Field</th>
<th>Bit Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP</td>
<td>7</td>
<td>Opcode. Can be Flat, Scratch or Global instruction. See next table.</td>
</tr>
<tr>
<td>ADDR</td>
<td>8</td>
<td>VGPR which holds the address. For 64-bit addresses, ADDR has the LSBs, and ADDR+1 has the MSBs.</td>
</tr>
<tr>
<td>DATA</td>
<td>8</td>
<td>VGPR which holds the first Dword of data. Instructions can use 0-4 Dwords.</td>
</tr>
<tr>
<td>VDST</td>
<td>8</td>
<td>VGPR destination for data returned to the kernel, either from LOADs or Atomics with GLC=1 (return pre-op value).</td>
</tr>
<tr>
<td>SLC</td>
<td>1</td>
<td>System Level Coherent. Used in conjunction with GLC to determine cache policies.</td>
</tr>
<tr>
<td>GLC</td>
<td>1</td>
<td>Global Level Coherent. For Atomics, GLC: 1 means return pre-op value, 0 means do not return pre-op value.</td>
</tr>
<tr>
<td>SEG</td>
<td>2</td>
<td>Memory Segment: 0=FLAT, 1=SCRATCH, 2=GLOBAL, 3=reserved.</td>
</tr>
<tr>
<td>LDS</td>
<td>1</td>
<td>When set, data is moved between LDS and memory instead of VGPRs and memory. For Global and Scratch only; must be zero for Flat.</td>
</tr>
<tr>
<td>NV</td>
<td>1</td>
<td>Non-volatile. When set, the read/write is operating on non-volatile memory.</td>
</tr>
<tr>
<td>OFFSET</td>
<td>13</td>
<td>Address offset. Scratch, Global: 13-bit signed byte offset. Flat: 12-bit unsigned offset (MSB is ignored).</td>
</tr>
<tr>
<td>Field</td>
<td>Bit Size</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>SADDR</td>
<td>7</td>
<td>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.</td>
</tr>
<tr>
<td>M0</td>
<td>16</td>
<td>Implied use of M0 for SCRATCH and GLOBAL only when LDS=1. Provides the LDS address-offset.</td>
</tr>
</tbody>
</table>

### Table 42. Flat, Global and Scratch Opcodes

<table>
<thead>
<tr>
<th>Flat Opcodes</th>
<th>Global Opcodes</th>
<th>Scratch Opcodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLAT</td>
<td>GLOBAL</td>
<td>SCATCH</td>
</tr>
<tr>
<td>FLAT_LOAD_UBYTE</td>
<td>GLOBAL_LOAD_UBYTE</td>
<td>SCRATCH_LOAD_UBYTE</td>
</tr>
<tr>
<td>FLAT_LOAD_UBYTE_D16</td>
<td>GLOBAL_LOAD_UBYTE_D16</td>
<td>SCRATCH_LOAD_UBYTE_D16</td>
</tr>
<tr>
<td>FLAT_LOAD_UBYTE_D16_HI</td>
<td>GLOBAL_LOAD_UBYTE_D16_HI</td>
<td>SCRATCH_LOAD_UBYTE_D16_HI</td>
</tr>
<tr>
<td>FLAT_LOAD_SBYTE</td>
<td>GLOBAL_LOAD_SBYTE</td>
<td>SCRATCH_LOAD_SBYTE</td>
</tr>
<tr>
<td>FLAT_LOAD_SBYTE_D16</td>
<td>GLOBAL_LOAD_SBYTE_D16</td>
<td>SCRATCH_LOAD_SBYTE_D16</td>
</tr>
<tr>
<td>FLAT_LOAD_SBYTE_D16_HI</td>
<td>GLOBAL_LOAD_SBYTE_D16_HI</td>
<td>SCRATCH_LOAD_SBYTE_D16_HI</td>
</tr>
<tr>
<td>FLAT_LOAD_USHORT</td>
<td>GLOBAL_LOAD_USHORT</td>
<td>SCRATCH_LOAD_USHORT</td>
</tr>
<tr>
<td>FLAT_LOAD_SSHORT</td>
<td>GLOBAL_LOAD_SSHORT</td>
<td>SCRATCH_LOAD_SSHORT</td>
</tr>
<tr>
<td>FLAT_LOAD_SHORT_D16</td>
<td>GLOBAL_LOAD_SHORT_D16</td>
<td>SCRATCH_LOAD_SHORT_D16</td>
</tr>
<tr>
<td>FLAT_LOAD_SHORT_D16_HI</td>
<td>GLOBAL_LOAD_SHORT_D16_HI</td>
<td>SCRATCH_LOAD_SHORT_D16_HI</td>
</tr>
<tr>
<td>FLAT_LOAD_DWORD</td>
<td>GLOBAL_LOAD_DWORD</td>
<td>SCRATCH_LOAD_DWORD</td>
</tr>
<tr>
<td>FLAT_LOAD_DWORDX2</td>
<td>GLOBAL_LOAD_DWORDX2</td>
<td>SCRATCH_LOAD_DWORDX2</td>
</tr>
<tr>
<td>FLAT_LOAD_DWORDX3</td>
<td>GLOBAL_LOAD_DWORDX3</td>
<td>SCRATCH_LOAD_DWORDX3</td>
</tr>
<tr>
<td>FLAT_LOAD_DWORDX4</td>
<td>GLOBAL_LOAD_DWORDX4</td>
<td>SCRATCH_LOAD_DWORDX4</td>
</tr>
<tr>
<td>FLAT_STORE_BYTE</td>
<td>GLOBAL_STORE_BYTE</td>
<td>SCRATCH_STORE_BYTE</td>
</tr>
<tr>
<td>FLAT_STORE_BYTE_D16_HI</td>
<td>GLOBAL_STORE_BYTE_D16_HI</td>
<td>SCRATCH_STORE_BYTE_D16_HI</td>
</tr>
<tr>
<td>FLAT_STORE_SHORT</td>
<td>GLOBAL_STORE_SHORT</td>
<td>SCRATCH_STORE_SHORT</td>
</tr>
<tr>
<td>FLAT_STORE_SHORT_D16_HI</td>
<td>GLOBAL_STORE_SHORT_D16_HI</td>
<td>SCRATCH_STORE_SHORT_D16_HI</td>
</tr>
<tr>
<td>FLATSTORE_DWORD</td>
<td>GLOBAL_STORE_DWORD</td>
<td>SCRATCH_STORE_DWORD</td>
</tr>
<tr>
<td>FLAT_STORE_DWORDX2</td>
<td>GLOBAL_STORE_DWORDX2</td>
<td>SCRATCH_STORE_DWORDX2</td>
</tr>
<tr>
<td>FLAT_STORE_DWORDX3</td>
<td>GLOBAL_STORE_DWORDX3</td>
<td>SCRATCH_STORE_DWORDX3</td>
</tr>
<tr>
<td>FLAT_STORE_DWORDX4</td>
<td>GLOBAL_STORE_DWORDX4</td>
<td>SCRATCH_STORE_DWORDX4</td>
</tr>
<tr>
<td>FLAT_ATOMIC_SWAP</td>
<td>GLOBAL_ATOMIC_SWAP</td>
<td>none</td>
</tr>
<tr>
<td>FLAT_ATOMIC_CMPSWAP</td>
<td>GLOBAL_ATOMIC_CMPSWAP</td>
<td>none</td>
</tr>
</tbody>
</table>
9.2. Instructions

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

FLAT instructions do not use a resource constant (V#) or sampler (S#); however, they do require a 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.

9.2.1. Ordering

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

9.2.2. Important Timing Consideration

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

### 9.3. Addressing

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

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

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

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

\[
\text{Address} = \text{VGPR}[\text{addr}] + \text{TID}_{\text{in}} \cdot \text{Size} \\
\quad - \text{private aperture base (in SH_MEM_BASES)} \\
\quad + \text{offset (from flat_scratch)}
\]

### 9.4. 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.

### 9.5. 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.

**9.6. Memory Error Checking**

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

- invalid address (outside any aperture)
- write to read-only surface
- misaligned data
- out-of-range address:
  - LDS access with an address outside the range: [0, \( \min(M0, \text{LDS\_SIZE})-1 \)]
  - Scratch access with an address outside the range: [0, scratch-size -1 ]

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

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

**9.7. Data**

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

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

10.1. Overview

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

![Figure 6. High-Level Memory Configuration](image)

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

There are 64 kB memory per compute unit, segmented into 32 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 decreases the effective bandwidth of the LDS. For maximum throughput (optimal efficiency), therefore, it is important to avoid bank conflicts. A knowledge of request scheduling and address mapping is key to achieving this.

10.2. Dataflow in Memory Hierarchy

The figure below is a conceptual diagram of the dataflow withing 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, an algorithm must perform many operations on what is transferred between global memory and LDS. It also is possible to load data from a memory buffer directly into LDS, bypassing VGPRs.

LDS atomics are performed in the LDS hardware. (Thus, although ALUs are not directly used for these operations, latency is incurred by the LDS executing this function.)

10.3. LDS Access

The LDS is accessed in one of three ways:

- Direct Read
- Parameter Read
• Indexed or Atomic

The following subsections describe these methods.

### 10.3.1. LDS Direct Reads

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

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

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

- **LDS_addr = M0[15:0]** (byte address and must be Dword aligned)
- **DataType = M0[18:16]**
  - 0: unsigned byte
  - 1: unsigned short
  - 2: Dword
  - 3: unused
  - 4: signed byte
  - 5: signed short

### 10.3.2. LDS Parameter Reads

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

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

- **V_INTERP_P1_F32 D = P10 * S + P0** Parameter interpolation, first step.
- **V_INTERP_P2_F32D = P20 * S + D** Parameter interpolation, second step.
- **V_INTERP_MOV_F32D = {P10,P20,P0}[S]** Parameter load.

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

\[
\text{Final value} = P0 + P10 * I + P20 * J
\]

Parameter interpolation instructions indicate the parameter attribute number (0 to 32) and the component number (0=x, 1=y, 2=z and 3=w).
Table 43. Parameter Instruction Fields

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

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

10.3.3. Data Share Indexed and Atomic Access

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

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

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

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

The table below lists and briefly describes the LDS instruction fields.
### Table 44. LDS Instruction Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP</td>
<td>7</td>
<td>LDS opcode.</td>
</tr>
<tr>
<td>GDS</td>
<td>1</td>
<td>0 = LDS, 1 = GDS.</td>
</tr>
<tr>
<td>OFFSET0</td>
<td>8</td>
<td>Immediate offset, in bytes. Instructions with one address combine the offset fields into a single 16-bit unsigned offset: {offset1, offset0}. Instructions with two addresses (for example: READ2) use the offsets separately as two 8-bit unsigned offsets. DS_<em><em>SRC2</em></em> ops treat the offset as a 16-bit signed Dword offset.</td>
</tr>
<tr>
<td>OFFSET1</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>VDST</td>
<td>8</td>
<td>VGPR to which result is written: either from LDS-load or atomic return value.</td>
</tr>
<tr>
<td>ADDR</td>
<td>8</td>
<td>VGPR that supplies the byte address offset.</td>
</tr>
<tr>
<td>DATA0</td>
<td>8</td>
<td>VGPR that supplies first data source.</td>
</tr>
<tr>
<td>DATA1</td>
<td>8</td>
<td>VGPR that supplies second data source.</td>
</tr>
</tbody>
</table>

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 45. LDS Indexed Load/Store

<table>
<thead>
<tr>
<th>Load / Store</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS_READ_{B32,B64,B96,B128,U8,I8,U16,I16}</td>
<td>Read one value per thread; sign extend to Dword, if signed.</td>
</tr>
<tr>
<td>DS_READ2_{B32,B64}</td>
<td>Read two values at unique addresses.</td>
</tr>
<tr>
<td>DS_READ2ST64_{B32,B64}</td>
<td>Read 2 values at unique addresses; offset *= 64.</td>
</tr>
<tr>
<td>DS_WRITE_{B32,B64,B96,B128,B8,B16}</td>
<td>Write one value.</td>
</tr>
<tr>
<td>DS_WRITE2_{B32,B64}</td>
<td>Write two values.</td>
</tr>
<tr>
<td>DS_WRITE2ST64_{B32,B64}</td>
<td>Write two values, offset *= 64.</td>
</tr>
<tr>
<td>DS_WRXCHG2_RTN_{B32,B64}</td>
<td>Exchange GPR with LDS-memory.</td>
</tr>
<tr>
<td>DS_WRXCHG2ST64_RTN_{B32,B64}</td>
<td>Exchange GPR with LDS-memory; offset *= 64.</td>
</tr>
<tr>
<td>DS_PERMUTE_B32</td>
<td>Forward permute. Does not write any LDS memory. LDS[dst] = src0 returnVal = LDS[thread_id] where thread_id is 0..63.</td>
</tr>
<tr>
<td>DS_BPERMUTE_B32</td>
<td>Backward permute. Does not actually write any LDS memory. LDS[thread_id] = src0 where thread_id is 0..63, and returnVal = LDS[dst].</td>
</tr>
</tbody>
</table>

### Single Address Instructions
**Double Address Instructions**

\[
\text{LDS\_Addr} = \text{LDS\_BASE} + \text{VGPR}[\text{ADDR}] + (\text{InstrOffset1}, \text{InstrOffset0})
\]

Note that LDS\_ADDR1 is used only for READ2*, WRITE2*, and WREXCHG2*.

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

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

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

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

**SRC2 Ops** The ds\_<op>_src2_<type> opcodes are different. These operands perform an atomic operation on 2 operands from the LDS memory: one is viewed as the data and the other is the second source operand and the final destination. The addressing for these can operate in two different modes depending on the MSB of offset1[7]: If it is 0, the offset for the data term is derived by the offset fields as a SIGNED dword offset:

\[
\begin{align*}
\text{LDS\_Addr} & = \text{LDS\_BASE} + \text{VGPR}(\text{ADDR}) + \text{SIGNEXTEND}(\text{InstrOffset1}[6:0], \text{InstrOffset0}) << 2 \quad \text{// data term} \\
\text{LDS\_Addr} & = \text{LDS\_BASE} + \text{VGPR}(\text{ADDR}) \quad \text{// second source and final destination address}
\end{align*}
\]

If the bit is 1, the offset for the data term becomes per thread and is a SIGNED dword offset derived from the msbs read from the VGPR for the index. The addressing becomes:

\[
\begin{align*}
\text{LDS\_Addr} & = \text{LDS\_BASE} + \text{VGPR}(\text{ADDR}[16:0]) + \text{SIGNEXTEND}(\text{VGPR}(\text{ADDR}[31:17])) << 2 \quad \text{// data term} \\
\text{LDS\_Addr} & = \text{LDS\_BASE} + \text{VGPR}(\text{ADDR}[16:0]) \quad \text{// second source and final destination address}
\end{align*}
\]

**LDS Atomic Ops** DS\_<atomicOp> OP, GDS=0, OFFSET0, OFFSET1, VDST, ADDR, Data0, Data1

Data size is encoded in atomicOp: byte, word, Dword, or double.
LDS.Addr0 = LDS_BASE + VGPR[ADDR] + {InstrOffset1, InstrOffset0}

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

VGPR data sources can only be VGPRs or constant values, not SGPRs.
Chapter 11. Exporting Pixel and Vertex Data

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

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

11.1. Microcode Encoding

The export instruction uses the EXP microcode format.

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM</td>
<td>1</td>
<td>Valid Mask. When set to 1, this indicates that the EXEC mask represents the valid-mask for this wavefront. It can be sent multiple times per shader (the final value is used), but must be sent at least once per pixel shader.</td>
</tr>
<tr>
<td>DONE</td>
<td>1</td>
<td>This is the final pixel shader or vertex-position export of the program. Used only for pixel and position exports. Set to zero for parameters.</td>
</tr>
<tr>
<td>COMPR</td>
<td>1</td>
<td>Compressed data. When set, indicates that the data being exported is 16-bits per component rather than the usual 32-bit.</td>
</tr>
<tr>
<td>TARGET</td>
<td>6</td>
<td>Indicates type of data exported. 0..7 MRT 0..7 8 Z 9 Null (no data) 12-15 Position 0..3 32-63 Param 0..31</td>
</tr>
<tr>
<td>EN</td>
<td>4</td>
<td>COMPR==1: export half-Dword enable. Valid values are: 0x0,3,C,F. [0] enables VSRC0 : R,G from one VGPR [2] enables VSRC1 : B,A from one VGPR COMPR==0: [0-3] = enables for VSRC0..3. EN can be zero (used when exporting only valid mask to NULL target).</td>
</tr>
<tr>
<td>Field</td>
<td>Size</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>VSRC3</td>
<td>8</td>
<td>VGPR from which to read data.</td>
</tr>
<tr>
<td>VSRC2</td>
<td>8</td>
<td>Pos &amp; Param: vsrc0=X, 1=Y, 2=Z, 3=W MRT: vsrc0=R, 1=G, 2=B, 3=A</td>
</tr>
<tr>
<td>VSRC1</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>VSRC0</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

**11.2. Operations**

**11.2.1. Pixel Shader Exports**

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

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

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

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

**11.2.2. Vertex Shader Exports**

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

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

**11.3. Dependency Checking**

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

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

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

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

If the STATUS register’s SKIP_EXPORT bit is set to one, the hardware treats all EXPORT instructions as if they were NOPs.
Chapter 12. Instructions

This chapter lists, and provides descriptions for, all instructions in the GCN Vega 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
- F16 floating-point 16-bit
- F32 floating-point 32-bit (IEEE 754 single-precision float)
- F64 floating-point 64-bit (IEEE 754 double-precision float)
- I8 signed 8-bit integer
- I16 signed 16-bit integer
- I32 signed 32-bit integer
- I64 signed 64-bit integer
- U16 unsigned 16-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

<table>
<thead>
<tr>
<th>SOP2</th>
<th>31</th>
<th>OP7</th>
<th>SDST7</th>
<th>SSRC17</th>
<th>SSRC08</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

12.1. SOP2 Instructions
Instructions in this format may use a 32-bit literal constant which occurs immediately after the instruction.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S_ADD_U32</td>
<td>D.u = S0.u + S1.u; SCC = (S0.u + S1.u &gt;= 0x100000000ULL ? 1 : 0). // unsigned overflow/carry-out, S_ADDC_U32</td>
</tr>
<tr>
<td>1</td>
<td>S_SUB_U32</td>
<td>D.u = S0.u - S1.u; SCC = (S1.u &gt; S0.u ? 1 : 0). // unsigned overflow or carry-out for S_SUBB_U32.</td>
</tr>
<tr>
<td>2</td>
<td>S_ADD_I32</td>
<td>D.i = S0.i + S1.i; SCC = (S0.u[31] == S1.u[31] &amp;&amp; S0.u[31] != D.u[31]). // signed overflow. This opcode is not suitable for use with S_ADDC_U32 for implementing 64-bit operations.</td>
</tr>
<tr>
<td>3</td>
<td>S_SUB_I32</td>
<td>D.i = S0.i - S1.i; SCC = (S0.u[31] != S1.u[31] &amp;&amp; S0.u[31] != D.u[31]). // signed overflow. This opcode is not suitable for use with S_SUBB_U32 for implementing 64-bit operations.</td>
</tr>
<tr>
<td>4</td>
<td>S_ADDC_U32</td>
<td>D.u = S0.u + S1.u + SCC; SCC = (S0.u + S1.u + SCC &gt;= 0x100000000ULL ? 1 : 0). // unsigned overflow.</td>
</tr>
<tr>
<td>5</td>
<td>S_SUBB_U32</td>
<td>D.u = S0.u - S1.u - SCC; SCC = (S1.u + SCC &gt; S0.u ? 1 : 0). // unsigned overflow.</td>
</tr>
<tr>
<td>6</td>
<td>S_MIN_I32</td>
<td>D.i = (S0.i &lt; S1.i) ? S0.i : S1.i; SCC = (S0.i &lt; S1.i).</td>
</tr>
<tr>
<td>7</td>
<td>S_MIN_U32</td>
<td>D.u = (S0.u &lt; S1.u) ? S0.u : S1.u; SCC = (S0.u &lt; S1.u).</td>
</tr>
<tr>
<td>8</td>
<td>S_MAX_I32</td>
<td>D.i = (S0.i &gt; S1.i) ? S0.i : S1.i; SCC = (S0.i &gt; S1.i).</td>
</tr>
<tr>
<td>9</td>
<td>S_MAX_U32</td>
<td>D.u = (S0.u &gt; S1.u) ? S0.u : S1.u; SCC = (S0.u &gt; S1.u).</td>
</tr>
<tr>
<td>10</td>
<td>S_CSELECT_B32</td>
<td>D.u = SCC ? S0.u : S1.u. Conditional select.</td>
</tr>
<tr>
<td>11</td>
<td>S_CSELECT_B64</td>
<td>D.u64 = SCC ? S0.u64 : S1.u64. Conditional select.</td>
</tr>
<tr>
<td>12</td>
<td>S_AND_B32</td>
<td>D = S0 &amp; S1; SCC = (D != 0).</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>---------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>13</td>
<td>S_AND_B64</td>
<td>D = S0 &amp; S1; SCC = (D != 0).</td>
</tr>
<tr>
<td>14</td>
<td>S_OR_B32</td>
<td>D = S0</td>
</tr>
<tr>
<td>15</td>
<td>S_OR_B64</td>
<td>D = S0</td>
</tr>
<tr>
<td>16</td>
<td>S_XOR_B32</td>
<td>D = S0 ^ S1; SCC = (D != 0).</td>
</tr>
<tr>
<td>17</td>
<td>S_XOR_B64</td>
<td>D = S0 ^ S1; SCC = (D != 0).</td>
</tr>
<tr>
<td>18</td>
<td>S_ANDN2_B32</td>
<td>D = S0 &amp; ~S1; SCC = (D != 0).</td>
</tr>
<tr>
<td>19</td>
<td>S_ANDN2_B64</td>
<td>D = S0 &amp; ~S1; SCC = (D != 0).</td>
</tr>
<tr>
<td>20</td>
<td>S_ORN2_B32</td>
<td>D = S0</td>
</tr>
<tr>
<td>21</td>
<td>S_ORN2_B64</td>
<td>D = S0</td>
</tr>
<tr>
<td>22</td>
<td>S_NAND_B32</td>
<td>D = ~(S0 &amp; S1); SCC = (D != 0).</td>
</tr>
<tr>
<td>23</td>
<td>S_NAND_B64</td>
<td>D = ~(S0 &amp; S1); SCC = (D != 0).</td>
</tr>
<tr>
<td>24</td>
<td>S_NOR_B32</td>
<td>D = ~(S0</td>
</tr>
<tr>
<td>25</td>
<td>S_NOR_B64</td>
<td>D = ~(S0</td>
</tr>
<tr>
<td>26</td>
<td>S_XNOR_B32</td>
<td>D = ~(S0 ^ S1); SCC = (D != 0).</td>
</tr>
<tr>
<td>27</td>
<td>S_XNOR_B64</td>
<td>D = ~(S0 ^ S1); SCC = (D != 0).</td>
</tr>
<tr>
<td>28</td>
<td>S_LSHL_B32</td>
<td>D.u = S0.u &lt;&lt; S1.u[4:0]; SCC = (D.u != 0).</td>
</tr>
<tr>
<td>29</td>
<td>S_LSHL_B64</td>
<td>D.u64 = S0.u64 &lt;&lt; S1.u[5:0]; SCC = (D.u64 != 0).</td>
</tr>
<tr>
<td>30</td>
<td>S_LSHR_B32</td>
<td>D.u = S0.u &gt;&gt; S1.u[4:0]; SCC = (D.u != 0).</td>
</tr>
<tr>
<td>31</td>
<td>S_LSHR_B64</td>
<td>D.u64 = S0.u64 &gt;&gt; S1.u[5:0]; SCC = (D.u64 != 0).</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>32</td>
<td>S_ASHR_I32</td>
<td>D.i = signext(S0.i) &gt;&gt; S1.u[4:0]; SCC = (D.i != 0).</td>
</tr>
<tr>
<td>33</td>
<td>S_ASHR_I64</td>
<td>D.i64 = signext(S0.i64) &gt;&gt; S1.u[5:0]; SCC = (D.i64 != 0).</td>
</tr>
<tr>
<td>34</td>
<td>S_BFM_B32</td>
<td>D.u = ((1 &lt;&lt; S0.u[4:0]) - 1) &lt;&lt; S1.u[4:0]. Bitfield mask.</td>
</tr>
<tr>
<td>35</td>
<td>S_BFM_B64</td>
<td>D.u64 = ((1ULL &lt;&lt; S0.u[5:0]) - 1) &lt;&lt; S1.u[5:0]. Bitfield mask.</td>
</tr>
<tr>
<td>36</td>
<td>S_MUL_I32</td>
<td>D.i = S0.i * S1.i.</td>
</tr>
<tr>
<td>37</td>
<td>S_BFE_U32</td>
<td>D.u = (S0.u &gt;&gt; S1.u[4:0]) &amp; ((1 &lt;&lt; S1.u[22:16]) - 1); SCC = (D.u != 0). Bit field extract. S0 is Data, S1[4:0] is field offset, S1[22:16] is field width.</td>
</tr>
<tr>
<td>38</td>
<td>S_BFE_I32</td>
<td>D.i = signext((S0.i &gt;&gt; S1.u[4:0]) &amp; ((1 &lt;&lt; S1.u[22:16]) - 1)); SCC = (D.i != 0). Bit field extract. S0 is Data, S1[4:0] is field offset, S1[22:16] is field width.</td>
</tr>
<tr>
<td>39</td>
<td>S_BFE_U64</td>
<td>D.u64 = (S0.u64 &gt;&gt; S1.u[5:0]) &amp; ((1 &lt;&lt; S1.u[22:16]) - 1); SCC = (D.u64 != 0). Bit field extract. S0 is Data, S1[5:0] is field offset, S1[22:16] is field width.</td>
</tr>
<tr>
<td>40</td>
<td>S_BFE_I64</td>
<td>D.i64 = signext((S0.i64 &gt;&gt; S1.u[5:0]) &amp; ((1 &lt;&lt; S1.u[22:16]) - 1)); SCC = (D.i64 != 0). Bit field extract. S0 is Data, S1[5:0] is field offset, S1[22:16] is field width.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| 41     | S_CBRANCH_G_FOR_K | mask_pass = S0.u64 & EXEC;
|        |      | mask_fail = ~S0.u64 & EXEC;
|        |      | if(mask_pass == EXEC) then
|        |      |    PC = S1.u64;
|        |      | elsif(mask_fail == EXEC) then
|        |      |    PC += 4;
|        |      | elsif(bitcount(mask_fail) < bitcount(mask_pass))
|        |      |    EXEC = mask_fail;
|        |      |    SGPR[CSP*4] = { S1.u64, mask_pass };
|        |      |    CSP += 1;
|        |      |    PC += 4;
|        |      | else
|        |      |    EXEC = mask_pass;
|        |      |    SGPR[CSP*4] = { PC + 4, mask_fail };
|        |      |    CSP += 1;
|        |      |    PC = S1.u64;
|        |      | endif. |
|        |      | 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. |
| 42     | S_ABSDIFF_I32 | D.i = S0.i - S1.i;
|        |      | if(D.i < 0) then
|        |      |    D.i = -D.i;
|        |      | endif;
|        |      | SCC = (D.i != 0). |
|        |      | Compute the absolute value of difference between two values. |
|        |      | 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 |
| 43     | S_RFE_RESTORE_B64 | PRIV = 0;
|        |      | PC = S0.u64. |
|        |      | Return from exception handler and continue. This instruction may only be used within a trap handler. |
|        |      | This instruction is provided for compatibility with older ASICs. New shader code must use S_RFE_B64. The second argument is ignored. |
| 44     | S_MUL_HI_U32 | D.u = (S0.u * S1.u) >> 32. |
| 45     | S_MUL_HI_I32 | D.i = (S0.i * S1.i) >> 32. |
### 12.2. SOPK Instructions

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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S_MOVK_I32</td>
<td>D.i = signext(SIMM16).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sign extension from a 16-bit constant.</td>
</tr>
<tr>
<td>1</td>
<td>S_CMOVK_I32</td>
<td>if(SCC) then</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D.i = signext(SIMM16);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>endif.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Conditional move with sign extension.</td>
</tr>
<tr>
<td>2</td>
<td>S_CMPK_EQ_I32</td>
<td>SCC = (S0.i == signext(SIMM16)).</td>
</tr>
<tr>
<td>3</td>
<td>S_CMPK_LG_I32</td>
<td>SCC = (S0.i != signext(SIMM16)).</td>
</tr>
<tr>
<td>4</td>
<td>S_CMPK_GT_I32</td>
<td>SCC = (S0.i &gt; signext(SIMM16)).</td>
</tr>
<tr>
<td>5</td>
<td>S_CMPK_GE_I32</td>
<td>SCC = (S0.i &gt;= signext(SIMM16)).</td>
</tr>
<tr>
<td>6</td>
<td>S_CMPK_LT_I32</td>
<td>SCC = (S0.i &lt; signext(SIMM16)).</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>7</td>
<td>S_CMPK_LE_I32</td>
<td>$SCC = (S0.i &lt;= \text{signext}(SIMM16)).$</td>
</tr>
<tr>
<td>8</td>
<td>S_CMPK_EQ_U32</td>
<td>$SCC = (S0.u == SIMM16).$</td>
</tr>
<tr>
<td>9</td>
<td>S_CMPK_LG_U32</td>
<td>$SCC = (S0.u != SIMM16).$</td>
</tr>
<tr>
<td>10</td>
<td>S_CMPK_GT_U32</td>
<td>$SCC = (S0.u &gt; SIMM16).$</td>
</tr>
<tr>
<td>11</td>
<td>S_CMPK_GE_U32</td>
<td>$SCC = (S0.u &gt;= SIMM16).$</td>
</tr>
<tr>
<td>12</td>
<td>S_CMPK_LT_U32</td>
<td>$SCC = (S0.u &lt; SIMM16).$</td>
</tr>
<tr>
<td>13</td>
<td>S_CMPK_LE_U32</td>
<td>$SCC = (S0.u &lt;= SIMM16).$</td>
</tr>
<tr>
<td>14</td>
<td>S_ADDK_I32</td>
<td>$\text{tmp} = D.i; // save value so we can check sign bits for overflow later. \ D.i = D.i + \text{signext}(SIMM16); \ SCC = (\text{tmp}[31] == SIMM16[15] &amp;&amp; \text{tmp}[31] != D.i[31]). // signed overflow.$</td>
</tr>
<tr>
<td>15</td>
<td>S_MULK_I32</td>
<td>$D.i = D.i * \text{signext}(SIMM16).$</td>
</tr>
<tr>
<td>16</td>
<td>S_CBRANCH_I_FOR K</td>
<td>$\text{mask_pass} = S0.u64 &amp;&amp; \text{EXEC}; \ \text{mask_fail} = \sim S0.u64 &amp;&amp; \text{EXEC}; \ \text{target_addr} = PC + \text{signext}(SIMM16 * 4) + 4; \ \text{if}(\text{mask_pass} == \text{EXEC}) \ PC = \text{target_addr}; \ \text{elsif}(\text{mask_fail} == \text{EXEC}) \ \ PC += 4; \ \ text{elsif}(\text{bitcount}(\text{mask_fail}) &lt; \text{bitcount}(\text{mask_pass})) \ EXEC = \text{mask_fail}; \ \ SGPR[\text{CSP}*4] = { \text{target_addr}, \text{mask_pass} }; \ \text{CSP} += 1; \ \ PC += 4; \ \text{else} \ EXEC = \text{mask_pass}; \ \ SGPR[\text{CSP}*4] = { \text{PC} + 4, \text{mask_fail} }; \ \text{CSP} += 1; \ \ PC = \text{target_addr}; \ \text{endif}.$</td>
</tr>
<tr>
<td>17</td>
<td>S_GETREG_B32</td>
<td>$D.u = \text{hardware-reg}. \text{Read some or all of a hardware register into the LSBs of D.}$</td>
</tr>
</tbody>
</table>

SIMM16 = \{\text{size}[4:0], \text{offset}[4:0], \text{hwRegId}[5:0]\}; \text{offset is 0..31, size is 1..32.}$
### 12.3. SOP1 Instructions

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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S_MOV_B32</td>
<td>D.u = S0.u.</td>
</tr>
<tr>
<td>1</td>
<td>S_MOV_B64</td>
<td>D.u64 = S0.u64.</td>
</tr>
</tbody>
</table>
| 2      | S_CMOV_B32    | if(SCC) then
              D.u = S0.u;
              endif.
              Conditional move. |
| 3      | S_CMOV_B64    | if(SCC) then
              D.u64 = S0.u64;
              endif.
              Conditional move. |
| 4      | S_NOT_B32     | D = ~S0;
              SCC = (D != 0).
              Bitwise negation. |
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>S_NOT_B64</td>
<td>D = ~S0;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SCC = (D != 0).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bitwise negation.</td>
</tr>
<tr>
<td>6</td>
<td>S_WQM_B32</td>
<td>for i in 0 ... opcode_size_in_bits - 1 do</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D[i] = (S0[(i &amp; ~3):(i</td>
</tr>
<tr>
<td></td>
<td></td>
<td>endfor;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SCC = (D != 0).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Computes whole quad mode for an active/valid mask. If any pixel in a quad is active, all pixels of the quad are marked active.</td>
</tr>
<tr>
<td>7</td>
<td>S_WQM_B64</td>
<td>for i in 0 ... opcode_size_in_bits - 1 do</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D[i] = (S0[(i &amp; ~3):(i</td>
</tr>
<tr>
<td></td>
<td></td>
<td>endfor;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SCC = (D != 0).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Computes whole quad mode for an active/valid mask. If any pixel in a quad is active, all pixels of the quad are marked active.</td>
</tr>
<tr>
<td>8</td>
<td>S_BREV_B32</td>
<td>D.u[31:0] = S0.u[0:31].</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reverse bits.</td>
</tr>
<tr>
<td>9</td>
<td>S_BREV_B64</td>
<td>D.u64[63:0] = S0.u64[0:63].</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reverse bits.</td>
</tr>
<tr>
<td>10</td>
<td>S_BCNT0_I32_B32</td>
<td>D = 0;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for i in 0 ... opcode_size_in_bits - 1 do</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D += (S0[i] == 0 ? 1 : 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>endfor;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SCC = (D != 0).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Examples:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S_BCNT0_I32_B32(0x00000000) =&gt; 32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S_BCNT0_I32_B32(0xffffffff) =&gt; 16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S_BCNT0_I32_B32(0xffffffff) =&gt; 0</td>
</tr>
<tr>
<td>11</td>
<td>S_BCNT0_I32_B64</td>
<td>D = 0;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for i in 0 ... opcode_size_in_bits - 1 do</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D += (S0[i] == 0 ? 1 : 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>endfor;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SCC = (D != 0).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Examples:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S_BCNT0_I32_B32(0x00000000) =&gt; 32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S_BCNT0_I32_B32(0xffffffff) =&gt; 16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S_BCNT0_I32_B32(0xffffffff) =&gt; 0</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>12</td>
<td>S_BCNT1_I32_B32</td>
<td>D = 0; for i in 0 ... opcode_size_in_bits - 1 do D += (S0[i] == 1 ? 1 : 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SCC = (D != 0).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Examples:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S_BCNT1_I32_B32(0x00000000) =&gt; 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S_BCNT1_I32_B32(0xcccccccc) =&gt; 16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S_BCNT1_I32_B32(0xffffffff) =&gt; 32</td>
</tr>
<tr>
<td>13</td>
<td>S_BCNT1_I32_B64</td>
<td>D = 0; for i in 0 ... opcode_size_in_bits - 1 do D += (S0[i] == 1 ? 1 : 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SCC = (D != 0).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Examples:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S_BCNT1_I32_B32(0x00000000) =&gt; 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S_BCNT1_I32_B32(0xcccccccc) =&gt; 16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S_BCNT1_I32_B32(0xffffffff) =&gt; 32</td>
</tr>
<tr>
<td>14</td>
<td>S_FF0_I32_B32</td>
<td>D.i = -1; // Set if no zeros are found</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for i in 0 ... opcode_size_in_bits - 1 do // Search from LSB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if S0[i] == 0 then</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D.i = i; break for;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>endif;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>endfor.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Returns the bit position of the first zero from the LSB, or -1 if</td>
</tr>
<tr>
<td></td>
<td></td>
<td>there are no zeros.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Examples:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S_FF0_I32_B32(0xaa000000) =&gt; 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S_FF0_I32_B32(0x55555555) =&gt; 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S_FF0_I32_B32(0x00000000) =&gt; 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S_FF0_I32_B32(0xffffffff) =&gt; 0xffffffff</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S_FF0_I32_B32(0xffffffff) =&gt; 0xffffffff</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>---------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>15</td>
<td>S_FF0_I32_B64</td>
<td>D.i = -1; // Set if no zeros are found &lt;br&gt;for i in 0 ... opcode_size_in_bits - 1 do // Search from LSB &lt;br&gt;if S0[i] == 0 then &lt;br&gt;  D.i = i; &lt;br&gt;        break for; &lt;br&gt;      endif; &lt;br&gt;    endfor. &lt;br&gt; &lt;br&gt;Returns the bit position of the first zero from the LSB, or -1 if there are no zeros. &lt;br&gt;Examples:&lt;br&gt;S_FF0_I32_B32(0xaaaaaaaa) =&gt; 0 &lt;br&gt;S_FF0_I32_B32(0x55555555) =&gt; 1 &lt;br&gt;S_FF0_I32_B32(0x00000000) =&gt; 0 &lt;br&gt;S_FF0_I32_B32(0xffffffff) =&gt; 0xffffffff &lt;br&gt;S_FF0_I32_B32(0xfffefffff) =&gt; 16</td>
</tr>
<tr>
<td>16</td>
<td>S_FF1_I32_B32</td>
<td>D.i = -1; // Set if no ones are found &lt;br&gt;for i in 0 ... opcode_size_in_bits - 1 do // Search from LSB &lt;br&gt;if S0[i] == 1 then &lt;br&gt;  D.i = i; &lt;br&gt;        break for; &lt;br&gt;      endif; &lt;br&gt;    endfor. &lt;br&gt; &lt;br&gt;Returns the bit position of the first one from the LSB, or -1 if there are no ones. &lt;br&gt;Examples:&lt;br&gt;S_FF1_I32_B32(0xaaaaaaaa) =&gt; 1 &lt;br&gt;S_FF1_I32_B32(0x55555555) =&gt; 0 &lt;br&gt;S_FF1_I32_B32(0x00000000) =&gt; 0xffffffff &lt;br&gt;S_FF1_I32_B32(0xffffffff) =&gt; 0 &lt;br&gt;S_FF1_I32_B32(0x00010000) =&gt; 16</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| 17     | S_FF1_I32_B64    | D.i = -1; // Set if no ones are found  
for i in 0 ... opcode_size_in_bits - 1 do // Search from LSB  
  if S0[i] == 1 then  
    D.i = i;  
    break for;  
  endif;  
endfor.  

Returns the bit position of the first one from the LSB, or -1 if there are no ones.  
Examples:  
S_FF1_I32_B32(0xffffffff) => 0  
S_FF1_I32_B32(0x00000000) => 0xffffffff  
S_FF1_I32_B32(0x00000001) => 0  
S_FF1_I32_B32(0x00000002) => 16 |
| 18     | S_FLBIT_I32_B32  | D.i = -1; // Set if no ones are found  
for i in 0 ... opcode_size_in_bits - 1 do // Note: search is from the MSB  
  if S0[opcode_size_in_bits - 1 - i] == 1 then  
    D.i = i;  
    break for;  
  endif;  
endfor.  

Counts how many zeros before the first one starting from the MSB. Returns -1 if there are no ones.  
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 |
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| 19     | S_FLBIT_I32_B64    | **D.i = -1;** // Set if no ones are found  
**for** i **in** 0 ... opcode_size_in_bits - 1 **do**  
  // Note: search is from the MSB  
  if S0[opcode_size_in_bits - 1 - i] == 1 then  
    D.i = i;  
    break for;  
  endif;  
endfor.  
Counts how many zeros before the first one starting from the MSB. Returns -1 if there are no ones.  
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 |
| 20     | S_FLBIT_I32       | **D.i = -1;** // Set if all bits are the same  
**for** i **in** 1 ... opcode_size_in_bits - 1 **do**  
  // Note: search is from the MSB  
  if S0[opcode_size_in_bits - 1 - i] != S0[opcode_size_in_bits - 1] then  
    D.i = i;  
    break for;  
  endif;  
endfor.  
Counts how many bits in a row (from MSB to LSB) are the same as the sign bit. Returns -1 if all bits are the same.  
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 |
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| 21     | S_FLBIT_I32_I64 | D.i = -1; // Set if all bits are the same for i in 1 ... opcode_size_in_bits - 1 do // Note: search is from the MSB if S0[opcode_size_in_bits - 1 - i] != S0[opcode_size_in_bits - 1] then D.i = i; break for; endif; endfor. Counts how many bits in a row (from MSB to LSB) are the same as the sign bit. Returns -1 if all bits are the same. Examples:
  S_FLBIT_I32(0x00000000) => 0xffffffff
  S_FLBIT_I32(0x0000cccc) => 16
  S_FLBIT_I32(0x0000ccc0) => 16
  S_FLBIT_I32(0x7fffffff) => 1
  S_FLBIT_I32(0x80000000) => 1
  S_FLBIT_I32(0xffffffff) => 0xffffffff |
<p>| 22     | S_SEXT_I32_I8 | D.i = signext(S0.i[7:0]). Sign extension. |
| 23     | S_SEXT_I32_I16 | D.i = signext(S0.i[15:0]). Sign extension. |
| 24     | S_BITSET0_B32 | D.u[S0.u[4:0]] = 0. |
| 25     | S_BITSET0_B64 | D.u64[S0.u[5:0]] = 0. |
| 26     | S_BITSET1_B32 | D.u[S0.u[4:0]] = 1. |
| 27     | S_BITSET1_B64 | D.u64[S0.u[5:0]] = 1. |
| 28     | S_GETPC_B64 | D.u64 = PC + 4. Destination receives the byte address of the next instruction. Note that this instruction is always 4 bytes. |
| 29     | S_SETPC_B64 | PC = S0.u64. S0.u64 is a byte address of the instruction to jump to. |
| 30     | S_SWAPPC_B64 | D.u64 = PC + 4; PC = S0.u64. S0.u64 is a byte address of the instruction to jump to. Destination receives the byte address of the instruction immediately following the SWAPPC instruction. Note that this instruction is always 4 bytes. |</p>
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>S_RFE_B64</td>
<td>PRIV = 0; PC = S0.u64. Return from exception handler and continue. This instruction may only be used within a trap handler.</td>
</tr>
<tr>
<td>32</td>
<td>S_AND_SAVEEXEC_B64</td>
<td>D.u64 = EXEC; EXEC = S0.u64 &amp; EXEC; SCC = (EXEC != 0).</td>
</tr>
<tr>
<td>33</td>
<td>S_OR_SAVEEXEC_B64</td>
<td>D.u64 = EXEC; EXEC = S0.u64</td>
</tr>
<tr>
<td>34</td>
<td>S_XOR_SAVEEXEC_B64</td>
<td>D.u64 = EXEC; EXEC = S0.u64 ^ EXEC; SCC = (EXEC != 0).</td>
</tr>
<tr>
<td>35</td>
<td>S_ANDN2_SAVEEXEC_B64</td>
<td>D.u64 = EXEC; EXEC = S0.u64 &amp; ~EXEC; SCC = (EXEC != 0).</td>
</tr>
<tr>
<td>36</td>
<td>S_ORN2_SAVEEXEC_B64</td>
<td>D.u64 = EXEC; EXEC = S0.u64</td>
</tr>
<tr>
<td>37</td>
<td>S_NAND_SAVEEXEC_B64</td>
<td>D.u64 = EXEC; EXEC = ~(S0.u64 &amp; EXEC); SCC = (EXEC != 0).</td>
</tr>
<tr>
<td>38</td>
<td>S_NOR_SAVEEXEC_B64</td>
<td>D.u64 = EXEC; EXEC = ~(S0.u64</td>
</tr>
<tr>
<td>39</td>
<td>S_XNOR_SAVEEXEC_B64</td>
<td>D.u64 = EXEC; EXEC = ~(S0.u64 ^ EXEC); SCC = (EXEC != 0).</td>
</tr>
<tr>
<td>40</td>
<td>S_QUADMASK_B32</td>
<td>D = 0; for i in 0 ... (opcode_size_in_bits / 4) - 1 do D[i] = (S0[i * 4 + 3:i * 4] != 0); endfor; SCC = (D != 0). Reduce a pixel mask to a quad mask. To perform the inverse operation see S_BITREPLICATE_B64_B32.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>41</td>
<td>S_QUADMASK_B64</td>
<td></td>
</tr>
</tbody>
</table>
D = 0;  
for i in 0 ... (opcode.size_in_bits / 4) - 1 do  
    D[i] = (S0[i * 4 + 3:i * 4] != 0);  
endfor;  
SCC = (D != 0).  
Reduce a pixel mask to a quad mask. To perform the inverse operation see S_BITREPLICATE_B64_B32. |
| 42     | S_MOVRELS_B32         | addr = SGPR address appearing in instruction SRC0 field;  
addr += M0.u;  
D.u = SGPR[addr].u.  
Move from a relative source address. For example, the following instruction sequence will perform a move s5 <= s17:  
s_mov_b32 m0, 10  
s_movrels_b32 s5, s7 |
| 43     | S_MOVRELS_B64         | addr = SGPR address appearing in instruction SRC0 field;  
addr += M0.u;  
D.u64 = SGPR[addr].u64.  
Move from a relative source address. The index in M0.u must be even for this operation. |
| 44     | S_MOVRELD_B32         | addr = SGPR address appearing in instruction DST field;  
addr += M0.u;  
SGPR[addr].u = S0.u.  
Move to a relative destination address. For example, the following instruction sequence will perform a move s15 <= s7:  
s_mov_b32 m0, 10  
s_movreld_b32 s5, s7 |
| 45     | S_MOVRELD_B64         | addr = SGPR address appearing in instruction DST field;  
addr += M0.u;  
SGPR[addr].u64 = S0.u64.  
Move to a relative destination address. The index in M0.u must be even for this operation. |
| 46     | S_CBRANCH_JOIN        | saved_csp = S0.u;  
if(CSP == saved_csp) then  
    PC += 4; // Second time to JOIN: continue with program.  
else  
    CSP -= 1; // First time to JOIN: jump to other FORK path.  
    {PC, EXEC} = SGPR[CSP * 4]; // Read 128 bits from 4 consecutive SGPRs.  
endif.  
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. |
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| 48     | S_ABS_I32                   | \[D.i = (S.i < 0 ? -S.i : S.i); \]
|        |                             | \[SCC = (D.i != 0).\]                                                                          |
|        |                             | Integer absolute value.                                                                         |
|        |                             | Examples:                                                                                       |
|        |                             | \[S_ABS_I32(0x00000001) => 0x00000001\]                                                          |
|        |                             | \[S_ABS_I32(0x7fffffff) => 0x7fffffff\]                                                            |
|        |                             | \[S_ABS_I32(0x80000000) => 0x80000000 \] Note this is negative!                               |
|        |                             | \[S_ABS_I32(0x80000001) => 0x7fffffff\]                                                           |
|        |                             | \[S_ABS_I32(0x80000002) => 0x7ffffffe\]                                                          |
|        |                             | \[S_ABS_I32(0xffffffff) => 0x00000001\]                                                          |
| 50     | S_SET_GPR_IDX_IDX           | \[M0[7:0] = S0.u[7:0].\]                                                                       |
|        |                             | Modify the index used in vector GPR indexing.                                                    |
|        |                             | \[S_SET_GPR_IDX_ON, S_SET_GPR_IDX_OFF, S_SET_GPR_IDX_MODE and S_SET_GPR_IDX_IDX are related instructions.\] |
| 51     | S_ANDN1_SAVEEXEC_C_B64      | \[D.u64 = EXEC;\] \[EXEC = ~S0.u64 & EXEC;\] \[SCC = (EXEC != 0).\]                          |
| 52     | S_ORN1_SAVEEXEC_B64         | \[D.u64 = EXEC;\] \[EXEC = ~S0.u64 | EXEC;\] \[SCC = (EXEC != 0).\]                          |
| 53     | S_ANDN1_WREEXEC_B64         | \[EXEC = ~S0.u64 & EXEC;\] \[D.u64 = EXEC;\] \[SCC = (EXEC != 0).\]                           |
| 54     | S_ANDN2_WREEXEC_B64         | \[EXEC = S0.u64 & ~EXEC;\] \[D.u64 = EXEC;\] \[SCC = (EXEC != 0).\]                           |
| 55     | S_BITREPLICATE_B64_B32      | \[for i in 0 ... 31 do\] \[D.u64[i * 2 + 0] = S0.u32[i]\] \[D.u64[i * 2 + 1] = S0.u32[i]\] \[endfor.\] |
|        |                             | Replicate the low 32 bits of S0 by 'doubling' each bit.                                        |
|        |                             | This opcode can be used to convert a quad mask into a pixel mask; given quad mask in s0, the following sequence will produce a pixel mask in s1: |
|        |                             | \[s_bitreplicate_b64 s1, s0\] \[s_bitreplicate_b64 s1, s1\]                                     |
|        |                             | 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.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| 0      | S_CMP_EQ_I32    | SCC = (S0 == S1).  
Note that S_CMP_EQ_I32 and S_CMP_EQ_U32 are identical opcodes, but both are provided for symmetry. |
| 1      | S_CMP_LT_I32    | SCC = (S0 != S1).  
Note that S_CMP_LT_I32 and S_CMP_LT_U32 are identical opcodes, but both are provided for symmetry. |
| 2      | S_CMP_EQ_U32    | SCC = (S0 == S1).  
Note that S_CMP_EQ_I32 and S_CMP_EQ_U32 are identical opcodes, but both are provided for symmetry. |
| 3      | S_CMP_LT_U32    | SCC = (S0.u > S1.u).                                                      |
| 4      | S_CMP_EQ_U32    | SCC = (S0.u == S1.u).                                                     |
| 5      | S_CMP_LT_U32    | SCC = (S0.u < S1.u).                                                      |
| 6      | S_CMP_EQ_U32    | SCC = (S0.u == S1.u).                                                     |
| 7      | S_CMP_LT_U32    | SCC = (S0.u > S1.u).                                                      |
| 8      | S_CMP_GE_U32    | SCC = (S0.u >= S1.u).                                                     |
| 9      | S_CMP_LE_U32    | SCC = (S0.u <= S1.u).                                                     |
| 10     | S_CMP_EQ_U32    | SCC = (S0.u == S1.u).                                                     |
| 11     | S_CMP_LT_U32    | SCC = (S0.u < S1.u).                                                      |
| 12     | S_BITCMP0_B32   | SCC = (S0.u[S1.u[4:0]] == 0).                                             |
| 13     | S_BITCMP1_B32   | SCC = (S0.u[S1.u[4:0]] == 1).                                             |
| 14     | S_BITCMP0_B64   | SCC = (S0.u64[S1.u[5:0]] == 0).                                           |
| 15     | S_BITCMP1_B64   | SCC = (S0.u64[S1.u[5:0]] == 1).                                           |
12.5. SOPP Instructions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S_NOP</td>
<td>Do nothing. Repeat NOP 1..16 times based on SIMM16[3:0] -- 0x0 = 1 time, 0xf = 16 times. This instruction may be used to introduce wait states to resolve hazards. Compare with S_SLEEP.</td>
</tr>
<tr>
<td>1</td>
<td>S_ENDPGM</td>
<td>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.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>2</td>
<td>S_BRANCH</td>
<td>PC = PC + signext(SIMM16 * 4) + 4. // short jump.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For a long jump, use S_SETPC_B64.</td>
</tr>
<tr>
<td>3</td>
<td>S_WAKEUP</td>
<td>Allow a wave to 'ping' all the other waves in its threadgroup to force them to wake up immediately 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 is useful for fBarrier implementations (speedup). This method is also safe from races because if any wave misses the ping, everything still works fine (waves which missed it just complete their normal S_SLEEP). If the wave executing S_WAKEUP is in a threadgroup (in_tg set), then it will wake up all waves associated with the same threadgroup ID. Otherwise, S_WAKEUP is treated as an S_NOP.</td>
</tr>
<tr>
<td>4</td>
<td>S_CBRANCH_SCC0</td>
<td>if(SCC == 0) then PC = PC + signext(SIMM16 * 4) + 4; endif.</td>
</tr>
<tr>
<td>5</td>
<td>S_CBRANCH_SCC1</td>
<td>if(SCC == 1) then PC = PC + signext(SIMM16 * 4) + 4; endif.</td>
</tr>
<tr>
<td>6</td>
<td>S_CBRANCH_VCCZ</td>
<td>if(VCC == 0) then PC = PC + signext(SIMM16 * 4) + 4; endif.</td>
</tr>
<tr>
<td>7</td>
<td>S_CBRANCH_VCCNZ</td>
<td>if(VCC != 0) then PC = PC + signext(SIMM16 * 4) + 4; endif.</td>
</tr>
<tr>
<td>8</td>
<td>S_CBRANCH_EXECZ</td>
<td>if(EXEC == 0) then PC = PC + signext(SIMM16 * 4) + 4; endif.</td>
</tr>
<tr>
<td>9</td>
<td>S_CBRANCH_EXECNZ</td>
<td>if(EXEC != 0) then PC = PC + signext(SIMM16 * 4) + 4; endif.</td>
</tr>
<tr>
<td>10</td>
<td>S_BARRIER</td>
<td>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.</td>
</tr>
<tr>
<td>11</td>
<td>S_SETKILL</td>
<td>Set KILL bit to value of SIMM16[0]. Used primarily for debugging kill wave host command behavior.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>12</td>
<td>S_WAITCNT</td>
<td>Wait for the counts of outstanding lds, vector-memory and export/vmem-write-data to be at or below the specified levels.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SIMM16[3:0] = vmcount (vector memory operations) lower bits [3:0],</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SIMM16[6:4] = export/mem-write-data count,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SIMM16[11:8] = LGKM_cnt (scalar-mem/GDS/LDS count),</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SIMM16[15:14] = vmcount (vector memory operations) upper bits [5:4],</td>
</tr>
<tr>
<td>13</td>
<td>S_SETHALT</td>
<td>Set HALT bit to value of SIMM16[0]; 1 = halt, 0 = resume. The halt flag is ignored while PRIV == 1 (inside trap handlers) but the shader will halt immediately after the handler returns if HALT is still set at that time.</td>
</tr>
<tr>
<td>14</td>
<td>S_SLEEP</td>
<td>Cause a wave to sleep for (64 * SIMM16[6:0] + 1..64) clocks. The exact amount of delay is approximate. Compare with S_NOP.</td>
</tr>
<tr>
<td>15</td>
<td>S_SETPRIO</td>
<td>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]}.</td>
</tr>
<tr>
<td>16</td>
<td>S_SENDMSG</td>
<td>Send a message upstream to VGT or the interrupt handler. SIMM16[9:0] contains the message type.</td>
</tr>
<tr>
<td>17</td>
<td>S_SENDMSGGHALT</td>
<td>Send a message and then HALT the wavefront; see S_SENDMSG for details.</td>
</tr>
<tr>
<td>18</td>
<td>S_TRAP</td>
<td>TrapID = SIMM16[7:0]; Wait for all instructions to complete; (TTMP1, TTMP0) = {3'h0, PCRewind[3:0], HT[0], TrapID[7:0], PC[47:0]}; PC = TBA; // trap base address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PRIV = 1. 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.</td>
</tr>
<tr>
<td>19</td>
<td>S_ICACHE_INV</td>
<td>Invalidate entire L1 instruction cache.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>You must have 16 separate S_NOP instructions or a jump/branch instruction after this instruction to ensure the SQ instruction buffer is purged.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NOTE: The number of S_NOPs required depends on the size of the shader instruction buffer, which in current generations is 16 DWORDs long. Older architectures had a 12 DWORD instruction buffer and in those architectures, 12 S_NOP instructions were sufficient.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>22</td>
<td>S_TTRACEDATA</td>
<td>Send M0 as user data to the thread trace stream.</td>
</tr>
<tr>
<td>23</td>
<td>S_CBRANCH_CDBGSYS/S</td>
<td>if(conditional_debug_system != 0) then PC = PC + signext(SIMM16 * 4) + 4; endif.</td>
</tr>
<tr>
<td>24</td>
<td>S_CBRANCH_CDBGUSER</td>
<td>if(conditional_debug_user != 0) then PC = PC + signext(SIMM16 * 4) + 4; endif.</td>
</tr>
<tr>
<td>25</td>
<td>S_CBRANCH_CDBGSYS/S_OR_USER</td>
<td>if(conditional_debug_system</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PC = PC + signext(SIMM16 * 4) + 4; endif.</td>
</tr>
<tr>
<td>26</td>
<td>S_CBRANCH_CDBGSYS/S_AND_USER</td>
<td>if(conditional_debug_system &amp;&amp; conditional_debug_user) then</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PC = PC + signext(SIMM16 * 4) + 4; endif.</td>
</tr>
<tr>
<td>27</td>
<td>S_ENDPGM_SAVED</td>
<td>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.</td>
</tr>
<tr>
<td>28</td>
<td>S_SET_GPRIDX_OFF</td>
<td>MODE.gpr_idx_en = 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Clear GPR indexing mode. Vector operations after this will not perform relative GPR addressing regardless of the contents of M0. This instruction does not modify M0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S_SET_GPRIDX_ON, S_SET_GPRIDX_OFF, S_SET_GPRIDX_MODE and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S_SET_GPRIDX_IDX are related instructions.</td>
</tr>
<tr>
<td>29</td>
<td>S_SET_GPRIDX_MODE</td>
<td>M0[15:12] = SIMM16[3:0].</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Modify the mode used for vector GPR indexing. The raw contents of the source field are read and used to set the enable bits.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SIMM16[0] = VSRC0_REL, SIMM16[1] = VSRC1_REL, SIMM16[2] =</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VSRC2_REL and SIMM16[3] = VDST_REL.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S_SET_GPRIDX_ON, S_SET_GPRIDX_OFF, S_SET_GPRIDX_MODE and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S_SET_GPRIDX_IDX are related instructions.</td>
</tr>
<tr>
<td>30</td>
<td>S_ENDPGM_ORDERED_PS_DONE</td>
<td>End of program; signal that a wave has exited its POPS critical section and terminate wavefront. The hardware implicitly executes S_WAITCNT 0 before executing this instruction. This instruction is an optimization that combines S_SENDMSG(MSG_ORDERED_PS_DONE) and S_ENDPGM; there may be cases where you still need to send the message separately, in which case you can end the shader with a normal S_ENDPGM instruction. See S_ENDPGM for additional variants.</td>
</tr>
</tbody>
</table>
### 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.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>0</td>
<td>-</td>
<td>illegal</td>
</tr>
<tr>
<td>GS</td>
<td>2</td>
<td>0=nop, 1=cut, 2=emit, 3=emit-cut</td>
<td>GS output. M0[4:0]=gs-waveID, SIMM[9:8] = stream-id</td>
</tr>
<tr>
<td>GS-done</td>
<td>3</td>
<td>-</td>
<td>used in context switching</td>
</tr>
<tr>
<td>save wave</td>
<td>4</td>
<td>-</td>
<td>used in context switching</td>
</tr>
<tr>
<td>Stall Wave Gen</td>
<td>5</td>
<td>-</td>
<td>stop new wave generation</td>
</tr>
<tr>
<td>Halt Waves</td>
<td>6</td>
<td>-</td>
<td>halt all running waves of this vmid</td>
</tr>
<tr>
<td>Ordered PS Done</td>
<td>7</td>
<td>-</td>
<td>POPS ordered section done</td>
</tr>
<tr>
<td>Early Prim Dealloc</td>
<td>8</td>
<td>-</td>
<td>Deallocate primitives. This message is optional. EXEC[N<em>12+10:N</em>12] = number of verts to deallocate from buffer N (N=0..3). Exec[58:48] = number of vertices to deallocate</td>
</tr>
<tr>
<td>GS alloc req</td>
<td>9</td>
<td>-</td>
<td>Request GS space in parameter cache. M0[9:0] = number of vertices</td>
</tr>
</tbody>
</table>

### 12.6. SMEM Instructions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S_LOAD_DWORD</td>
<td>Read 1 dword from scalar data cache. 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.</td>
</tr>
<tr>
<td>1</td>
<td>S_LOAD_DWORDX2</td>
<td>Read 2 dwords from scalar data cache. See S_LOAD_DWORD for details on the offset input.</td>
</tr>
<tr>
<td>2</td>
<td>S_LOAD_DWORDX4</td>
<td>Read 4 dwords from scalar data cache. See S_LOAD_DWORD for details on the offset input.</td>
</tr>
<tr>
<td>3</td>
<td>S_LOAD_DWORDX8</td>
<td>Read 8 dwords from scalar data cache. See S_LOAD_DWORD for details on the offset input.</td>
</tr>
<tr>
<td>4</td>
<td>S_LOAD_DWORDX16</td>
<td>Read 16 dwords from scalar data cache. See S_LOAD_DWORD for details on the offset input.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>5</td>
<td>S_SCRATCH_LOAD_DWORD</td>
<td>Read 1 dword from scalar data cache. 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.</td>
</tr>
<tr>
<td>6</td>
<td>S_SCRATCH_LOAD_DWORD X2</td>
<td>Read 2 dwords from scalar data cache. See S_SCRATCH_LOAD_DWORD for details on the offset input.</td>
</tr>
<tr>
<td>7</td>
<td>S_SCRATCH_LOAD_DWORD X4</td>
<td>Read 4 dwords from scalar data cache. See S_SCRATCH_LOAD_DWORD for details on the offset input.</td>
</tr>
<tr>
<td>8</td>
<td>S_BUFFER_LOAD_DWORD</td>
<td>Read 1 dword from scalar data cache. See S_LOAD_DWORD for details on the offset input.</td>
</tr>
<tr>
<td>9</td>
<td>S_BUFFER_LOAD_DWORD X2</td>
<td>Read 2 dwords from scalar data cache. See S_LOAD_DWORD for details on the offset input.</td>
</tr>
<tr>
<td>10</td>
<td>S_BUFFER_LOAD_DWORD X4</td>
<td>Read 4 dwords from scalar data cache. See S_LOAD_DWORD for details on the offset input.</td>
</tr>
<tr>
<td>11</td>
<td>S_BUFFER_LOAD_DWORD X8</td>
<td>Read 8 dwords from scalar data cache. See S_LOAD_DWORD for details on the offset input.</td>
</tr>
<tr>
<td>12</td>
<td>S_BUFFER_LOAD_DWORD X16</td>
<td>Read 16 dwords from scalar data cache. See S_LOAD_DWORD for details on the offset input.</td>
</tr>
<tr>
<td>16</td>
<td>S_STORE_DWORD</td>
<td>Write 1 dword to scalar data cache. 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 an SIGNED BYTE offset.</td>
</tr>
<tr>
<td>17</td>
<td>S_STORE_DWORDX2</td>
<td>Write 2 dwords to scalar data cache. See S_STORE_DWORD for details on the offset input.</td>
</tr>
<tr>
<td>18</td>
<td>S_STORE_DWORDX4</td>
<td>Write 4 dwords to scalar data cache. See S_STORE_DWORD for details on the offset input.</td>
</tr>
<tr>
<td>21</td>
<td>S_SCRATCH_STORE_DWORD</td>
<td>Write 1 dword from scalar data cache. 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.</td>
</tr>
<tr>
<td>22</td>
<td>S_SCRATCH_STORE_DWORD X2</td>
<td>Write 2 dwords from scalar data cache. See S_SCRATCH_STORE_DWORD for details on the offset input.</td>
</tr>
<tr>
<td>23</td>
<td>S_SCRATCH_STORE_DWORD X4</td>
<td>Write 4 dwords from scalar data cache. See S_SCRATCH_STORE_DWORD for details on the offset input.</td>
</tr>
<tr>
<td>24</td>
<td>S_BUFFER_STORE_DWORD</td>
<td>Write 1 dword to scalar data cache. See S_STORE_DWORD for details on the offset input.</td>
</tr>
<tr>
<td>25</td>
<td>S_BUFFER_STORE_DWORD X2</td>
<td>Write 2 dwords to scalar data cache. See S_STORE_DWORD for details on the offset input.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>---------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>26</td>
<td>S_BUFFER_STORE_DWORD</td>
<td>Write 4 dwords to scalar data cache. See S_STORE_DWORD for details on the offset input.</td>
</tr>
<tr>
<td></td>
<td>S_BUFFER_STORE_DWORD X4</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>S_DCACHE_INV</td>
<td>Invalidate the scalar data cache.</td>
</tr>
<tr>
<td>33</td>
<td>S_DCACHE WB</td>
<td>Write back dirty data in the scalar data cache.</td>
</tr>
<tr>
<td>34</td>
<td>S_DCACHE_INV_VOL</td>
<td>Invalidate the scalar data cache volatile lines.</td>
</tr>
<tr>
<td>35</td>
<td>S_DCACHE WB_VOL</td>
<td>Write back dirty data in the scalar data cache volatile lines.</td>
</tr>
<tr>
<td>36</td>
<td>S_MEMTIME</td>
<td>Return current 64-bit timestamp.</td>
</tr>
<tr>
<td>37</td>
<td>S_MEMREALTIME</td>
<td>Return current 64-bit RTC.</td>
</tr>
<tr>
<td>38</td>
<td>S_ATC_PROBE</td>
<td>Probe or prefetch an address into the SQC data cache.</td>
</tr>
<tr>
<td>39</td>
<td>S_ATC_PROBE_BUFFER</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>S_DCACHE_DISCARD</td>
<td>Discard one dirty scalar data cache line. A cache line is 64 bytes. Normally, 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.</td>
</tr>
<tr>
<td>41</td>
<td>S_DCACHE_DISCARD X2</td>
<td>Discard two consecutive dirty scalar data cache lines. A cache line is 64 bytes. Normally, 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.</td>
</tr>
<tr>
<td>64</td>
<td>S_BUFFER_ATOMIC_SWAP</td>
<td>// 32bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR]; MEM[ADDR] = DATA; RETURN_DATA = tmp.</td>
</tr>
<tr>
<td>65</td>
<td>S_BUFFER_ATOMIC_CMPS</td>
<td>// 32bit</td>
</tr>
<tr>
<td></td>
<td>S_BUFFER_ATOMIC_CMPS WAP</td>
<td>src = DATA[0]; cmp = DATA[1]; MEM[ADDR] = (tmp == cmp) ? src : tmp; RETURN_DATA[0] = tmp.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------</td>
<td>--------------------------------------------------</td>
</tr>
</tbody>
</table>
| 66     | S_BUFFER_ATOMIC_ADD | // 32bit  
|        |                    | tmp = MEM[ADDR];  
|        |                    | MEM[ADDR] += DATA;  
|        |                    | RETURN_DATA = tmp.  |
| 67     | S_BUFFER_ATOMIC_SUB | // 32bit  
|        |                    | tmp = MEM[ADDR];  
|        |                    | MEM[ADDR] -= DATA;  
|        |                    | RETURN_DATA = tmp.  |
| 68     | S_BUFFER_ATOMIC_SMIN | // 32bit  
|        |                    | tmp = MEM[ADDR];  
|        |                    | MEM[ADDR] = (DATA < tmp) ? DATA : tmp; // signed compare  
|        |                    | RETURN_DATA = tmp.  |
| 69     | S_BUFFER_ATOMIC_UMIN | // 32bit  
|        |                    | tmp = MEM[ADDR];  
|        |                    | MEM[ADDR] = (DATA < tmp) ? DATA : tmp; // unsigned compare  
|        |                    | RETURN_DATA = tmp.  |
| 70     | S_BUFFER_ATOMIC_SMAX | // 32bit  
|        |                    | tmp = MEM[ADDR];  
|        |                    | MEM[ADDR] = (DATA > tmp) ? DATA : tmp; // signed compare  
|        |                    | RETURN_DATA = tmp.  |
| 71     | S_BUFFER_ATOMIC_UMAX | // 32bit  
|        |                    | tmp = MEM[ADDR];  
|        |                    | MEM[ADDR] = (DATA > tmp) ? DATA : tmp; // unsigned compare  
|        |                    | RETURN_DATA = tmp.  |
| 72     | S_BUFFER_ATOMIC_AND | // 32bit  
|        |                    | tmp = MEM[ADDR];  
|        |                    | MEM[ADDR] &= DATA;  
|        |                    | RETURN_DATA = tmp.  |
| 73     | S_BUFFER_ATOMIC_OR | // 32bit  
|        |                    | tmp = MEM[ADDR];  
|        |                    | MEM[ADDR] |= DATA;  
|        |                    | RETURN_DATA = tmp.  |
| 74     | S_BUFFER_ATOMIC_XOR | // 32bit  
|        |                    | tmp = MEM[ADDR];  
|        |                    | MEM[ADDR] ^= DATA;  
|        |                    | RETURN_DATA = tmp.  |
| 75     | S_BUFFER_ATOMIC_INC | // 32bit  
|        |                    | tmp = MEM[ADDR];  
|        |                    | MEM[ADDR] = (tmp >= DATA) ? 0 : tmp + 1; // unsigned compare  
<p>|        |                    | RETURN_DATA = tmp.  |</p>
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>S_BUFFER_ATOMIC_DEC</td>
<td>// 32bit&lt;br&gt;tmp = MEM[ADDR];&lt;br&gt;MEM[ADDR] = (tmp == 0</td>
</tr>
<tr>
<td>96</td>
<td>S_BUFFER_ATOMIC_SWAP_X2</td>
<td>// 64bit&lt;br&gt;tmp = MEM[ADDR];&lt;br&gt;MEM[ADDR] = DATA[0:1];&lt;br&gt;RETURN_DATA[0:1] = tmp.</td>
</tr>
<tr>
<td>97</td>
<td>S_BUFFER_ATOMIC_CMPS_WAP_X2</td>
<td>// 64bit&lt;br&gt;tmp = MEM[ADDR];&lt;br&gt;src = DATA[0:1];&lt;br&gt;cmp = DATA[2:3];&lt;br&gt;MEM[ADDR] = (tmp == cmp) ? src : tmp;&lt;br&gt;RETURN_DATA[0:1] = tmp.</td>
</tr>
<tr>
<td>98</td>
<td>S_BUFFER_ATOMIC_ADD_X2</td>
<td>// 64bit&lt;br&gt;tmp = MEM[ADDR];&lt;br&gt;MEM[ADDR] += DATA[0:1];&lt;br&gt;RETURN_DATA[0:1] = tmp.</td>
</tr>
<tr>
<td>99</td>
<td>S_BUFFER_ATOMIC_SUB_X2</td>
<td>// 64bit&lt;br&gt;tmp = MEM[ADDR];&lt;br&gt;MEM[ADDR] -= DATA[0:1];&lt;br&gt;RETURN_DATA[0:1] = tmp.</td>
</tr>
<tr>
<td>100</td>
<td>S_BUFFER_ATOMIC_SMIN_X2</td>
<td>// 64bit&lt;br&gt;tmp = MEM[ADDR];&lt;br&gt;MEM[ADDR] -= (DATA[0:1] &lt; tmp) ? DATA[0:1] : tmp; // signed compare&lt;br&gt;RETURN_DATA[0:1] = tmp.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| 104    | S_BUFFER_ATOMIC_AND_X2 | // 64bit  
|        |                     | tmp = MEM[ADDR];  
|        |                     | MEM[ADDR] &= DATA[0:1];  
|        |                     | RETURN_DATA[0:1] = tmp.  |
| 105    | S_BUFFER_ATOMIC_OR_X2 | // 64bit  
|        |                     | tmp = MEM[ADDR];  
|        |                     | MEM[ADDR] |= DATA[0:1];  
|        |                     | RETURN_DATA[0:1] = tmp.  |
| 106    | S_BUFFER_ATOMIC_XOR_X2 | // 64bit  
|        |                     | tmp = MEM[ADDR];  
|        |                     | MEM[ADDR] ^= DATA[0:1];  
|        |                     | RETURN_DATA[0:1] = tmp.  |
| 107    | S_BUFFER_ATOMIC_INC_X2 | // 64bit  
|        |                     | tmp = MEM[ADDR];  
|        |                     | MEM[ADDR] = (tmp >= DATA[0:1]) ? 0 : tmp + 1; // unsigned compare  
|        |                     | RETURN_DATA[0:1] = tmp.  |
| 108    | S_BUFFER_ATOMIC_DEC_X2 | // 64bit  
|        |                     | tmp = MEM[ADDR];  
|        |                     | MEM[ADDR] = (tmp == 0 || tmp > DATA[0:1]) ? DATA[0:1] : tmp - 1; // unsigned compare  
|        |                     | RETURN_DATA[0:1] = tmp.  |
| 128    | S_ATOMIC_SWAP       | // 32bit  
|        |                     | tmp = MEM[ADDR];  
|        |                     | MEM[ADDR] = DATA;  
|        |                     | RETURN_DATA = tmp.  |
| 129    | S_ATOMIC_CMPSWAP    | // 32bit  
|        |                     | tmp = MEM[ADDR];  
|        |                     | src = DATA[0];  
|        |                     | cmp = DATA[1];  
|        |                     | MEM[ADDR] = (tmp == cmp) ? src : tmp;  
|        |                     | RETURN_DATA[0] = tmp.  |
| 130    | S_ATOMIC_ADD        | // 32bit  
|        |                     | tmp = MEM[ADDR];  
|        |                     | MEM[ADDR] += DATA;  
|        |                     | RETURN_DATA = tmp.  |
| 131    | S_ATOMIC_SUB        | // 32bit  
|        |                     | tmp = MEM[ADDR];  
|        |                     | MEM[ADDR] -= DATA;  
|        |                     | RETURN_DATA = tmp.  |
| 132    | S_ATOMIC_SMIN       | // 32bit  
|        |                     | tmp = MEM[ADDR];  
|        |                     | MEM[ADDR] = (DATA < tmp) ? DATA : tmp; // signed compare  
<p>|        |                     | RETURN_DATA = tmp.  |</p>
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>133</td>
<td>S_ATOMIC_UMIN</td>
<td>// 32bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] = (DATA &lt; tmp) ? DATA : tmp; // unsigned compare</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA = tmp.</td>
</tr>
<tr>
<td>134</td>
<td>S_ATOMIC_SMAX</td>
<td>// 32bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] = (DATA &gt; tmp) ? DATA : tmp; // signed compare</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA = tmp.</td>
</tr>
<tr>
<td>135</td>
<td>S_ATOMIC_UMAX</td>
<td>// 32bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] = (DATA &gt; tmp) ? DATA : tmp; // unsigned compare</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA = tmp.</td>
</tr>
<tr>
<td>136</td>
<td>S_ATOMIC_AND</td>
<td>// 32bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] &amp;= DATA;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA = tmp.</td>
</tr>
<tr>
<td>137</td>
<td>S_ATOMIC_OR</td>
<td>// 32bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA = tmp.</td>
</tr>
<tr>
<td>138</td>
<td>S_ATOMIC_XOR</td>
<td>// 32bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] ^= DATA;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA = tmp.</td>
</tr>
<tr>
<td>139</td>
<td>S_ATOMIC_INC</td>
<td>// 32bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] = (tmp &gt;= DATA) ? 0 : tmp + 1; // unsigned compare</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA = tmp.</td>
</tr>
<tr>
<td>140</td>
<td>S_ATOMIC_DEC</td>
<td>// 32bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] = (tmp == 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA = tmp.</td>
</tr>
<tr>
<td>160</td>
<td>S_ATOMIC_SWAP_X2</td>
<td>// 64bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] = DATA[0:1];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA[0:1] = tmp.</td>
</tr>
<tr>
<td>161</td>
<td>S_ATOMIC_CMPSWAP_X2</td>
<td>// 64bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>src = DATA[0:1];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cmp = DATA[2:3];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] = (tmp == cmp) ? src : tmp;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA[0:1] = tmp.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>162</td>
<td>S_ATOMIC_ADD_X2</td>
<td>// 64bit&lt;br&gt;tmp = MEM[ADDR];&lt;br&gt;MEM[ADDR] += DATA[0:1];&lt;br&gt;RETURN_DATA[0:1] = tmp.</td>
</tr>
<tr>
<td>163</td>
<td>S_ATOMIC_SUB_X2</td>
<td>// 64bit&lt;br&gt;tmp = MEM[ADDR];&lt;br&gt;MEM[ADDR] -= DATA[0:1];&lt;br&gt;RETURN_DATA[0:1] = tmp.</td>
</tr>
<tr>
<td>168</td>
<td>S_ATOMIC_AND_X2</td>
<td>// 64bit&lt;br&gt;tmp = MEM[ADDR];&lt;br&gt;MEM[ADDR] &amp;= DATA[0:1];&lt;br&gt;RETURN_DATA[0:1] = tmp.</td>
</tr>
<tr>
<td>169</td>
<td>S_ATOMIC_OR_X2</td>
<td>// 64bit&lt;br&gt;tmp = MEM[ADDR];&lt;br&gt;MEM[ADDR]</td>
</tr>
<tr>
<td>170</td>
<td>S_ATOMIC_XOR_X2</td>
<td>// 64bit&lt;br&gt;tmp = MEM[ADDR];&lt;br&gt;MEM[ADDR] ^= DATA[0:1];&lt;br&gt;RETURN_DATA[0:1] = tmp.</td>
</tr>
<tr>
<td>171</td>
<td>S_ATOMIC_INC_X2</td>
<td>// 64bit&lt;br&gt;tmp = MEM[ADDR];&lt;br&gt;MEM[ADDR] = (tmp &gt;= DATA[0:1]) ? 0 : tmp + 1; // unsigned compare&lt;br&gt;RETURN_DATA[0:1] = tmp.</td>
</tr>
</tbody>
</table>
### Opcode Table

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>172</td>
<td>S_ATOMIC_DEC_X2</td>
<td>// 64bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] = (tmp == 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA[0:1] = tmp.</td>
</tr>
</tbody>
</table>

---

### 12.7. VOP2 Instructions

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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>V_CNDMASK_B32</td>
<td>D.u = (VCC[threadId] ? S1.u : S0.u).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Conditional mask on each thread. In VOP3 the VCC source may be a scalar GPR specified in S2.u.</td>
</tr>
<tr>
<td>1</td>
<td>V_ADD_F32</td>
<td>D.f = S0.f + S1.f.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.5ULP precision, denormals are supported.</td>
</tr>
<tr>
<td>2</td>
<td>V_SUB_F32</td>
<td>D.f = S0.f - S1.f.</td>
</tr>
<tr>
<td>3</td>
<td>V_SUBREV_F32</td>
<td>D.f = S1.f - S0.f.</td>
</tr>
<tr>
<td>4</td>
<td>V_MUL_LEGACY_F32</td>
<td>D.f = S0.f * S1.f. //DX9 rules; 0.0*x = 0.0</td>
</tr>
<tr>
<td>5</td>
<td>V_MUL_F32</td>
<td>D.f = S0.f * S1.f.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.5ULP precision, denormals are supported.</td>
</tr>
<tr>
<td>6</td>
<td>V_MUL_I32_I24</td>
<td>D.i = S0.i[23:0] * S1.i[23:0].</td>
</tr>
<tr>
<td>7</td>
<td>V_MUL_HI_I32_I24</td>
<td>D.i = (S0.i[23:0] * S1.i[23:0])&gt;&gt;32.</td>
</tr>
<tr>
<td>8</td>
<td>V_MUL_U32_U24</td>
<td>D.u = S0.u[23:0] * S1.u[23:0].</td>
</tr>
<tr>
<td>9</td>
<td>V_MUL_HI_U32_U24</td>
<td>D.i = (S0.u[23:0] * S1.u[23:0])&gt;&gt;32.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>----------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| 10     | V_MIN_F32      | if (IEEE_MODE && S0.f == sNaN)  
|        |                | D.f = Quiet(S0.f);  
|        |                | else if (IEEE_MODE && S1.f == sNaN)  
|        |                | D.f = Quiet(S1.f);  
|        |                | else if (S0.f == NaN)  
|        |                | D.f = S1.f;  
|        |                | else if (S1.f == NaN)  
|        |                | D.f = S0.f;  
|        |                | else if (S0.f == +0.0 && S1.f == -0.0)  
|        |                | D.f = S1.f;  
|        |                | else if (S0.f == -0.0 && S1.f == +0.0)  
|        |                | D.f = S0.f;  
|        |                | else  
|        |                | // Note: there's no IEEE special case here like there is for V_MAX_F32.  
|        |                | D.f = (S0.f < S1.f ? S0.f : S1.f);  
|        |                | endif.                                                                                                                                                                                                  |
| 11     | V_MAX_F32      | if (IEEE_MODE && S0.f == sNaN)  
|        |                | D.f = Quiet(S0.f);  
|        |                | else if (IEEE_MODE && S1.f == sNaN)  
|        |                | D.f = Quiet(S1.f);  
|        |                | else if (S0.f == NaN)  
|        |                | D.f = S1.f;  
|        |                | else if (S1.f == NaN)  
|        |                | D.f = S0.f;  
|        |                | else if (S0.f == +0.0 && S1.f == -0.0)  
|        |                | D.f = S1.f;  
|        |                | else if (S0.f == -0.0 && S1.f == +0.0)  
|        |                | D.f = S0.f;  
|        |                | else if (IEEE_MODE)  
|        |                | D.f = (S0.f >= S1.f ? S0.f : S1.f);  
|        |                | else  
|        |                | D.f = (S0.f > S1.f ? S0.f : S1.f);  
|        |                | endif.                                                                                                                                                                                                  |
| 12     | V_MIN_I32      | D.i = (S0.i < S1.i ? S0.i : S1.i).                                                                                                                                                                         |
| 13     | V_MAX_I32      | D.i = (S0.i >= S1.i ? S0.i : S1.i).                                                                                                                                                                         |
| 14     | V_MIN_U32      | D.u = (S0.u < S1.u ? S0.u : S1.u).                                                                                                                                                                         |
| 15     | V_MAX_U32      | D.u = (S0.u >= S1.u ? S0.u : S1.u).                                                                                                                                                                         |
| 16     | V_LSHRREV_B32  | D.u = S1.u >> S0.u[4:0].                                                                                                                                                                                  |
| 17     | V_ASHRREV_I32  | D.i = signext(S1.i) >> S0.i[4:0].                                                                                                                                                                          |
| 18     | V_LSHLREV_B32  | D.u = S1.u << S0.u[4:0].                                                                                                                                                                                  |
| 19     | V_AND_B32      | D.u = S0.u & S1.u.                                                                                                                                                                                         |

Input and output modifiers not supported.
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| 20     | V_OR_B32          | D.u = S0.u | S1.u.  
Input and output modifiers not supported.                                                                                                   |
| 21     | V_XOR_B32         | D.u = S0.u ^ S1.u.  
Input and output modifiers not supported.                                                                                                   |
| 22     | V_MAC_F32         | D.f = S0.f * S1.f + D.f.                                                                                                                   |
| 23     | V_MADMK_F32       | D.f = S0.f * K + S1.f.  // K is a 32-bit literal constant.                                                                                   |
|        |                   | This opcode cannot use the VOP3 encoding and cannot use input/output modifiers.                                                              |
| 24     | V_MADAK_F32       | D.f = S0.f * S1.f + K.  // K is a 32-bit literal constant.                                                                                   |
|        |                   | This opcode cannot use the VOP3 encoding and cannot use input/output modifiers.                                                              |
| 25     | V_ADD_C0_U32      | D.u = S0.u + S1.u;  
VCC[threadId] = (S0.u + S1.u >= 0x100000000ULL ? 1 : 0).  
// VCC is an UNSIGNED overflow/carry-out for V_ADDC_CO_U32.                                                                                     |
|        |                   | In VOP3 the VCC destination may be an arbitrary SGPR-pair.                                                                                     |
| 26     | V_SUB_C0_U32      | D.u = S0.u - S1.u;  
VCC[threadId] = (S1.u > S0.u ? 1 : 0).  
// VCC is an UNSIGNED overflow/carry-out for V_SUBB_CO_U32.                                                                                     |
|        |                   | In VOP3 the VCC destination may be an arbitrary SGPR-pair.                                                                                     |
| 27     | V_SUBREV_C0_U32   | D.u = S1.u - S0.u;  
VCC[threadId] = (S0.u > S1.u ? 1 : 0).  
// VCC is an UNSIGNED overflow/carry-out for V_SUBB_CO_U32.                                                                                     |
|        |                   | In VOP3 the VCC destination may be an arbitrary SGPR-pair.                                                                                     |
| 28     | V_ADDC_C0_U32     | D.u = S0.u + S1.u + VCC[threadId];  
VCC[threadId] = (S0.u + S1.u + VCC[threadId] >= 0x100000000ULL ? 1 : 0).  
// VCC is an UNSIGNED overflow.                                                                                                              |
|        |                   | In VOP3 the VCC destination may be an arbitrary SGPR-pair, and the VCC source comes from the SGPR-pair at S2.u.                                      |
| 29     | V_SUBB_C0_U32     | D.u = S0.u - S1.u - VCC[threadId];  
VCC[threadId] = (S1.u + VCC[threadId] > S0.u ? 1 : 0).  
// VCC is an UNSIGNED overflow.                                                                                                              |
<p>|        |                   | In VOP3 the VCC destination may be an arbitrary SGPR-pair, and the VCC source comes from the SGPR-pair at S2.u.                                      |</p>
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>V_SUBBREV_CO_U32</td>
<td>D.(u = S1.u - S0.u - VCC[\text{threadId}]);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VCC[\text{threadId}] = (S1.u + VCC[\text{threadId}] &gt; S0.u ? 1 : 0).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// VCC is an UNSIGNED overflow.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>In VOP3 the VCC destination may be an arbitrary SGPR-pair, and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the VCC source comes from the SGPR-pair at S2.u.</td>
</tr>
<tr>
<td>31</td>
<td>V_ADD_F16</td>
<td>D.f16 = S0.f16 + S1.f16.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Supports denormals, round mode, exception flags, saturation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.5ULP precision, denormals are supported.</td>
</tr>
<tr>
<td>32</td>
<td>V_SUB_F16</td>
<td>D.f16 = S0.f16 - S1.f16.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Supports denormals, round mode, exception flags, saturation.</td>
</tr>
<tr>
<td>33</td>
<td>V_SUBREV_F16</td>
<td>D.f16 = S1.f16 - S0.f16.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Supports denormals, round mode, exception flags, saturation.</td>
</tr>
<tr>
<td>34</td>
<td>V_MUL_F16</td>
<td>D.f16 = S0.f16 \times S1.f16.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Supports denormals, round mode, exception flags, saturation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.5ULP precision, denormals are supported.</td>
</tr>
<tr>
<td>35</td>
<td>V_MAC_F16</td>
<td>D.f16 = S0.f16 \times S1.f16 + D.f16.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Supports round mode, exception flags, saturation.</td>
</tr>
<tr>
<td>36</td>
<td>V_MADMK_F16</td>
<td>D.f16 = S0.f16 \times K.f16 + S1.f16.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// K is a 16-bit literal constant stored in the following literal DWORD.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This opcode cannot use the VOP3 encoding and cannot use input/output modifiers. Supports round mode, exception flags, saturation.</td>
</tr>
<tr>
<td>37</td>
<td>V_MADAK_F16</td>
<td>D.f16 = S0.f16 \times S1.f16 + K.f16.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// K is a 16-bit literal constant stored in the following literal DWORD.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This opcode cannot use the VOP3 encoding and cannot use input/output modifiers. Supports round mode, exception flags, saturation.</td>
</tr>
<tr>
<td>38</td>
<td>V_ADD_U16</td>
<td>D.u16 = S0.u16 + S1.u16.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Supports saturation (unsigned 16-bit integer domain).</td>
</tr>
<tr>
<td>39</td>
<td>V_SUB_U16</td>
<td>D.u16 = S0.u16 - S1.u16.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Supports saturation (unsigned 16-bit integer domain).</td>
</tr>
<tr>
<td>40</td>
<td>V_SUBREV_U16</td>
<td>D.u16 = S1.u16 - S0.u16.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Supports saturation (unsigned 16-bit integer domain).</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>41</td>
<td>V_MUL_LO_U16</td>
<td>(D.u16 = S0.u16 \times S1.u16). Supports saturation (unsigned 16-bit integer domain).</td>
</tr>
<tr>
<td>42</td>
<td>V_LSHLREV_B16</td>
<td>(D.u[15:0] = S1.u[15:0] &lt;&lt; S0.u[3:0]).</td>
</tr>
<tr>
<td>43</td>
<td>V_LSHRREV_B16</td>
<td>(D.u[15:0] = S1.u[15:0] &gt;&gt; S0.u[3:0]).</td>
</tr>
<tr>
<td>44</td>
<td>V_ASHRREV_I16</td>
<td>(D.i[15:0] = \text{signext}(S1.i[15:0]) &gt;&gt; S0.i[3:0]).</td>
</tr>
</tbody>
</table>
| 45     | V_MAX_F16             | if (IEEE_MODE && S0.f16 == sNaN)  
D.f16 = Quiet(S0.f16);  
else if (IEEE_MODE && S1.f16 == sNaN)  
D.f16 = Quiet(S1.f16);  
else if (S0.f16 == NaN)  
D.f16 = S1.f16;  
else if (S1.f16 == NaN)  
D.f16 = S0.f16;  
else if (S0.f16 == +0.0 && S1.f16 == -0.0)  
D.f16 = S0.f16;  
else if (S0.f16 == -0.0 && S1.f16 == +0.0)  
D.f16 = S1.f16;  
else if (IEEE_MODE)  
D.f16 = (S0.f16 >= S1.f16 ? S0.f16 : S1.f16);  
elses  
D.f16 = (S0.f16 > S1.f16 ? S0.f16 : S1.f16);  
endif.  
IEEE compliant. Supports denormals, round mode, exception flags, saturation. |
| 46     | V_MIN_F16             | if (IEEE_MODE && S0.f16 == sNaN)  
D.f16 = Quiet(S0.f16);  
else if (IEEE_MODE && S1.f16 == sNaN)  
D.f16 = Quiet(S1.f16);  
else if (S0.f16 == NaN)  
D.f16 = S1.f16;  
else if (S1.f16 == NaN)  
D.f16 = S0.f16;  
else if (S0.f16 == +0.0 && S1.f16 == -0.0)  
D.f16 = S0.f16;  
else if (S0.f16 == -0.0 && S1.f16 == +0.0)  
D.f16 = S1.f16;  
else  
// Note: there's no IEEE special case here like there is for V_MAX_F16.  
D.f16 = (S0.f16 < S1.f16 ? S0.f16 : S1.f16);  
endif.  
IEEE compliant. Supports denormals, round mode, exception flags, saturation. |
<p>| 47     | V_MAX_U16             | (D.u16 = (S0.u16 &gt;= S1.u16 ? S0.u16 : S1.u16)).                                        |</p>
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td>V_MAX_I16</td>
<td>D.i16 = (S0.i16 &gt;= S1.i16 ? S0.i16 : S1.i16).</td>
</tr>
<tr>
<td>49</td>
<td>V_MIN_U16</td>
<td>D.u16 = (S0.u16 &lt; S1.u16 ? S0.u16 : S1.u16).</td>
</tr>
<tr>
<td>50</td>
<td>V_MIN_I16</td>
<td>D.i16 = (S0.i16 &lt; S1.i16 ? S0.i16 : S1.i16).</td>
</tr>
<tr>
<td>51</td>
<td>V_LDEXP_F16</td>
<td>D.f16 = S0.f16 * (2 ** S1.i16). Note that the S1 has a format of f16 since floating point literal constants are interpreted as 16 bit value for this opcode</td>
</tr>
<tr>
<td>52</td>
<td>V_ADD_U32</td>
<td>D.u = S0.u + S1.u.</td>
</tr>
<tr>
<td>53</td>
<td>V_SUB_U32</td>
<td>D.u = S0.u - S1.u.</td>
</tr>
<tr>
<td>54</td>
<td>V_SUBREV_U32</td>
<td>D.u = S1.u - S0.u.</td>
</tr>
<tr>
<td>59</td>
<td>V_FM pérdida_F32</td>
<td>D.f32 = S0.f32 * S1.f32 + D.f32.</td>
</tr>
<tr>
<td>61</td>
<td>V_XNOR_B32</td>
<td>D.b32 = S0.b32 XNOR S1.b32.</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>V_NOP</td>
<td>Do nothing.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>---------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>1</td>
<td>V_MOV_B32</td>
<td>$D.u = S0.u.$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Input and output modifiers not supported; this is an untyped operation.</td>
</tr>
<tr>
<td>2</td>
<td>V_READFIRSTLANE_B</td>
<td>Copy one VGPR value to one SGPR. $D = SGPR$ destination, $S0 = source$ data (VGPR# or M0 for lds direct access), Lane# = FindFirst1fromLSB(exec) (Lane# = 0 if exec is zero). Ignores exec mask for the access.</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Input and output modifiers not supported; this is an untyped operation.</td>
</tr>
<tr>
<td>3</td>
<td>V_CVT_I32_F64</td>
<td>$D.i = (\text{int})S0.d.$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.5ULP accuracy, out-of-range floating point values (including infinity) saturate. NaN is converted to 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Generation of the INEXACT exception is controlled by the CLAMP bit. INEXACT exceptions are enabled for this conversion iff CLAMP == 1.</td>
</tr>
<tr>
<td>4</td>
<td>V_CVT_F64_I32</td>
<td>$D.d = (\text{double})S0.i.$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0ULP accuracy.</td>
</tr>
<tr>
<td>5</td>
<td>V_CVT_F32_I32</td>
<td>$D.f = (\text{float})S0.i.$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.5ULP accuracy.</td>
</tr>
<tr>
<td>6</td>
<td>V_CVT_F32_U32</td>
<td>$D.f = (\text{float})S0.u.$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.5ULP accuracy.</td>
</tr>
<tr>
<td>7</td>
<td>V_CVT_U32_F32</td>
<td>$D.u = (\text{unsigned})S0.f.$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1ULP accuracy, out-of-range floating point values (including infinity) saturate. NaN is converted to 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Generation of the INEXACT exception is controlled by the CLAMP bit. INEXACT exceptions are enabled for this conversion iff CLAMP == 1.</td>
</tr>
<tr>
<td>8</td>
<td>V_CVT_I32_F32</td>
<td>$D.i = (\text{int})S0.f.$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1ULP accuracy, out-of-range floating point values (including infinity) saturate. NaN is converted to 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Generation of the INEXACT exception is controlled by the CLAMP bit. INEXACT exceptions are enabled for this conversion iff CLAMP == 1.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>10</td>
<td>V_CVT_F16_F32</td>
<td>D.f16 = flt32_to_flt16(S0.f). 0.5ULP accuracy, supports input modifiers and creates FP16 denormals when appropriate.</td>
</tr>
<tr>
<td>11</td>
<td>V_CVT_F32_F16</td>
<td>D.f = flt16_to_flt32(S0.f16). 0.5ULP accuracy, FP16 denormal inputs are accepted.</td>
</tr>
<tr>
<td>12</td>
<td>V_CVT_RPI_I32_F32</td>
<td>D.i = (int)floor(S0.f + 0.5). 0.5ULP accuracy, denormals are supported.</td>
</tr>
<tr>
<td>13</td>
<td>V_CVT_FLR_I32_F32</td>
<td>D.i = (int)floor(S0.f). 1ULP accuracy, denormals are supported.</td>
</tr>
<tr>
<td>14</td>
<td>V_CVT_OFF_F32_I4</td>
<td>4-bit signed int to 32-bit float. Used for interpolation in shader. S0 Result</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1000  -0.5f</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1001  -0.4375f</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1010  -0.375f</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1011  -0.3125f</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1100  -0.25f</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1101  -0.1875f</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1110  -0.125f</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1111  -0.0625f</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0000  0.0f</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0001  0.0625f</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0010  0.125f</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0011  0.1875f</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0100  0.25f</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0101  0.3125f</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0110  0.375f</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0111  0.4375f</td>
</tr>
<tr>
<td>15</td>
<td>V_CVT_F32_F64</td>
<td>D.f = (float)S0.d. 0.5ULP accuracy, denormals are supported.</td>
</tr>
<tr>
<td>16</td>
<td>V_CVT_F64_F32</td>
<td>D.d = (double)S0.f. 0ULP accuracy, denormals are supported.</td>
</tr>
<tr>
<td>17</td>
<td>V_CVT_F32_UBYTE0</td>
<td>D.f = (float)(S0.u[7:0]).</td>
</tr>
<tr>
<td>18</td>
<td>V_CVT_F32_UBYTE1</td>
<td>D.f = (float)(S0.u[15:8]).</td>
</tr>
<tr>
<td>19</td>
<td>V_CVT_F32_UBYTE2</td>
<td>D.f = (float)(S0.u[23:16]).</td>
</tr>
<tr>
<td>20</td>
<td>V_CVT_F32_UBYTE3</td>
<td>D.f = (float)(S0.u[31:24]).</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| 21     | V_CVT_U32_F64         | D.u = (unsigned)S0.d.  
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. |
| 22     | V_CVT_F64_U32         | D.d = (double)S0.u.  
0ULP accuracy.                                                                                                                                  |
| 23     | V_TRUNC_F64           | D.d = trunc(S0.d).  
Return integer part of S0.d, round-to-zero semantics.                                                                                         |
| 24     | V_CEIL_F64            | D.d = trunc(S0.d);  
if(S0.d > 0.0 && S0.d != D.d) then  
D.d += 1.0;  
endif.  
Round up to next whole integer.                                                                                                               |
| 25     | V_RNDNE_F64           | D.d = floor(S0.d + 0.5);  
if(floor(S0.d) is even && fract(S0.d) == 0.5) then  
D.d -= 1.0;  
endif.  
Round-to-nearest-even semantics.                                                                                                               |
| 26     | V_FLOOR_F64           | D.d = trunc(S0.d);  
if(S0.d < 0.0 && S0.d != D.d) then  
D.d += -1.0;  
endif.  
Round down to previous whole integer.                                                                                                          |
| 27     | V_FRACT_F32           | D.f = S0.f + -floor(S0.f).  
Return fractional portion of a number. 0.5ULP accuracy, denormals are accepted.                                                                |
| 28     | V_TRUNC_F32           | D.f = trunc(S0.f).  
Return integer part of S0.f, round-to-zero semantics.                                                                                         |
| 29     | V_CEIL_F32            | D.f = trunc(S0.f);  
if(S0.f > 0.0 && S0.f != D.f) then  
D.f += 1.0;  
endif.  
Round up to next whole integer.                                                                                                               |
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>V_RNDNE_F32</td>
<td>( D.f = \text{floor}(S0.f + 0.5); ) if(floor(S0.f) is even &amp;&amp; fract(S0.f) == 0.5) then ( D.f -= 1.0; ) endif.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Round-to-nearest-even semantics.</td>
</tr>
<tr>
<td>31</td>
<td>V_FLOOR_F32</td>
<td>( D.f = \text{trunc}(S0.f); ) if(S0.f &lt; 0.0 &amp;&amp; S0.f != D.f) then ( D.f += -1.0; ) endif.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Round down to previous whole integer.</td>
</tr>
<tr>
<td>32</td>
<td>V_EXP_F32</td>
<td>( D.f = \text{pow}(2.0, S0.f). )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Base 2 exponentiation. 1ULP accuracy, denormals are flushed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Examples:</strong> ( \text{V_EXP_F32}(0xff800000) \Rightarrow 0x00000000 ) // exp(-INF) = 0  ( \text{V_EXP_F32}(0x80000000) \Rightarrow 0x3f800000 ) // exp(-0.0) = 1  ( \text{V_EXP_F32}(0x7f800000) \Rightarrow 0x7f800000 ) // exp(+INF) = +INF</td>
</tr>
<tr>
<td>33</td>
<td>V_LOG_F32</td>
<td>( D.f = \log_2(S0.f). )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Base 2 logarithm. 1ULP accuracy, denormals are flushed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Examples:</strong> ( \text{V_LOG_F32}(0xff800000) \Rightarrow 0xffc00000 ) // log(-INF) = NAN  ( \text{V_LOG_F32}(0xfb000000) \Rightarrow 0xffc00000 ) // log(-1.0) = NAN  ( \text{V_LOG_F32}(0x80000000) \Rightarrow 0xff800000 ) // log(-0.0) = -INF  ( \text{V_LOG_F32}(0x80000000) \Rightarrow 0xff800000 ) // log(+0.0) = -INF  ( \text{V_LOG_F32}(0x3f800000) \Rightarrow 0x00000000 ) // log(+1.0) = 0  ( \text{V_LOG_F32}(0x7f800000) \Rightarrow 0x7f800000 ) // log(+INF) = +INF</td>
</tr>
<tr>
<td>34</td>
<td>V_RCP_F32</td>
<td>( D.f = 1.0 / S0.f. )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reciprocal with IEEE rules and 1ULP accuracy. Accuracy converges to &lt; 0.5ULP when using the Newton-Raphson method and 2 FMA operations. Denormals are flushed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Examples:</strong> ( \text{V_RCP_F32}(0xff800000) \Rightarrow 0x80000000 ) // rcp(-INF) = -0  ( \text{V_RCP_F32}(0xc0000000) \Rightarrow 0xbf000000 ) // rcp(-2.0) = -0.5  ( \text{V_RCP_F32}(0x80000000) \Rightarrow 0xff800000 ) // rcp(-0.0) = -INF  ( \text{V_RCP_F32}(0x80000000) \Rightarrow 0x7f800000 ) // rcp(+0.0) = +INF  ( \text{V_RCP_F32}(0x7f800000) \Rightarrow 0x00000000 ) // rcp(+INF) = +0</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>35</td>
<td>V_RCP_FLAG_F32</td>
<td>D.f = 1.0 / S0.f. Reciprocal intended for integer division, 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 following sequences: Unsigned: CVT_F32_U32 RCP_FLAG_F32 MUL_F32 (2<strong>32 - 1) CVT_U32_F32 Signed: CVT_F32_I32 RCP_FLAG_F32 MUL_F32 (2</strong>31 - 1) CVT_I32_F32</td>
</tr>
<tr>
<td>36</td>
<td>V_RSQ_F32</td>
<td>D.f = 1.0 / sqrt(S0.f). Reciprocal square root with IEEE rules. 1ULP accuracy, denormals are flushed. Examples: V_RSQ_F32(0xff800000) =&gt; 0xffc00000 // rsq(-INF) = NAN V_RSQ_F32(0x80000000) =&gt; 0xff800000 // rsq(-0.0) = -INF V_RSQ_F32(0x00000000) =&gt; 0x7f800000 // rsq(+0.0) = +INF V_RSQ_F32(0x40800000) =&gt; 0x3f000000 // rsq(+4.0) = +0.5 V_RSQ_F32(0x7f800000) =&gt; 0x00000000 // rsq(+INF) = +0</td>
</tr>
<tr>
<td>37</td>
<td>V_RCP_F64</td>
<td>D.d = 1.0 / S0.d. Reciprocal with IEEE rules and perhaps not the accuracy you were hoping for -- (2**29)ULP accuracy. On the upside, denormals are supported.</td>
</tr>
<tr>
<td>38</td>
<td>V_RSQ_F64</td>
<td>D.f16 = 1.0 / sqrt(S0.f16). Reciprocal square root with IEEE rules and perhaps not the accuracy you were hoping for -- (2**29)ULP accuracy. On the upside, denormals are supported.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>----------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>39</td>
<td>V_SQRT_F32</td>
<td>( D.f = \sqrt{S0.f} ).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Square root. 1ULP accuracy, denormals are flushed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Examples:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[ V_{\text{SQRT_F32}}(0xff000000) \Rightarrow 0xffc00000 \quad \text{// sqrt(-INF) = NAN} ]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[ V_{\text{SQRT_F32}}(0x80000000) \Rightarrow 0x80000000 \quad \text{// sqrt(-0.0) = -0} ]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[ V_{\text{SQRT_F32}}(0x00000000) \Rightarrow 0x00000000 \quad \text{// sqrt(0.0) = +0} ]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[ V_{\text{SQRT_F32}}(0x40800000) \Rightarrow 0x40000000 \quad \text{// sqrt(+4.0) = +2.0} ]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[ V_{\text{SQRT_F32}}(0x7f800000) \Rightarrow 0x7f800000 \quad \text{// sqrt(+INF) = +INF} ]</td>
</tr>
<tr>
<td>40</td>
<td>V_SQRT_F64</td>
<td>( D.d = \sqrt{S0.d} ).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Square root with perhaps not the accuracy you were hoping for -- ((2^{29}))ULP accuracy. On the upside, denormals are supported.</td>
</tr>
<tr>
<td>41</td>
<td>V_SIN_F32</td>
<td>( D.f = \sin(S0.f \times 2 \times \pi) ).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Trigonometric sine. Denormals are supported.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Examples:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[ V_{\text{SIN_F32}}(0xff000000) \Rightarrow 0xffc00000 \quad \text{// sin(-INF) = NAN} ]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[ V_{\text{SIN_F32}}(0xff7fffff) \Rightarrow 0x00000000 \quad \text{// -MaxFloat, finite} ]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[ V_{\text{SIN_F32}}(0x80000000) \Rightarrow 0x80000000 \quad \text{// sin(-0.0) = -0} ]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[ V_{\text{SIN_F32}}(0x3e800000) \Rightarrow 0x3f800000 \quad \text{// sin(0.25) = 1} ]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[ V_{\text{SIN_F32}}(0x7f800000) \Rightarrow 0xffc00000 \quad \text{// sin(+INF) = NAN} ]</td>
</tr>
<tr>
<td>42</td>
<td>V_COS_F32</td>
<td>( D.f = \cos(S0.f \times 2 \times \pi) ).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Trigonometric cosine. Denormals are supported.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Examples:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[ V_{\text{COS_F32}}(0xff000000) \Rightarrow 0xffc00000 \quad \text{// cos(-INF) = NAN} ]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[ V_{\text{COS_F32}}(0xff7fffff) \Rightarrow 0x3f800000 \quad \text{// -MaxFloat, finite} ]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[ V_{\text{COS_F32}}(0x80000000) \Rightarrow 0x3f800000 \quad \text{// cos(-0.0) = 1} ]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[ V_{\text{COS_F32}}(0x3e800000) \Rightarrow 0x00000000 \quad \text{// cos(0.25) = 0} ]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[ V_{\text{COS_F32}}(0x7f800000) \Rightarrow 0xffc00000 \quad \text{// cos(+INF) = NAN} ]</td>
</tr>
<tr>
<td>43</td>
<td>V_NOT_B32</td>
<td>( D.u = \neg S0.u ).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bitwise negation. Input and output modifiers not supported.</td>
</tr>
<tr>
<td>44</td>
<td>V_BFREV_B32</td>
<td>( D.u[31:0] = S0.u[0:31] ).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bitfield reverse. Input and output modifiers not supported.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>45</td>
<td>V_FFBH_U32</td>
<td>D.i = -1; // Set if no ones are found for i in 0 ... 31 do</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// Note: search is from the MSB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if S0.u[31 - i] == 1 then</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D.i = i;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>break for;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>endif;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>endfor.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Counts how many zeros before the first one starting from the MSB.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Returns -1 if there are no ones.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Examples:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>V_FFBH_U32(0x00000000) =&gt; 0xffffffff</td>
</tr>
<tr>
<td></td>
<td></td>
<td>V_FFBH_U32(0x800000ff) =&gt; 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>V_FFBH_U32(0x100000ff) =&gt; 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>V_FFBH_U32(0x0000ffff) =&gt; 16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>V_FFBH_U32(0x00000001) =&gt; 31</td>
</tr>
<tr>
<td>46</td>
<td>V_FFBL_B32</td>
<td>D.i = -1; // Set if no ones are found for i in 0 ... 31 do // Search from LSB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if S0.u[i] == 1 then</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D.i = i;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>break for;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>endif;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>endfor.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Returns the bit position of the first one from the LSB, or -1 if there are no ones.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Examples:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>V_FFBL_B32(0x00000000) =&gt; 0xffffffff</td>
</tr>
<tr>
<td></td>
<td></td>
<td>V_FFBL_B32(0xff0000001) =&gt; 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>V_FFBL_B32(0xff000008) =&gt; 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>V_FFBL_B32(0xffff0000) =&gt; 16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>V_FFBL_B32(0x800000000) =&gt; 31</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| 47     | V_FFBH_I32            | D.i = -1; // Set if all bits are the same for i in 1 ... 31  
// Note: search is from the MSB  
if S0.i[31 - i] != S0.i[31] then  
    D.i = i;  
    break for;  
endif;  
endfor.  
Counts how many bits in a row (from MSB to LSB) are the same as the sign bit. Returns -1 if all bits are the same.  
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 |
| 48     | V_FREXP_EXP_I32_F64   | if(S0.d == +-INF || S0.d == NAN) then  
    D.i = 0;  
else  
    D.i = TwosComplement(Exponent(S0.d) - 1023 + 1);  
endif.  
Returns exponent of single precision float input, such that S0.d = significand * (2 ** exponent). See also V_FREXP_MANT_F64, which returns the significand. See the C library function frexp() for more information. |
| 49     | V_FREXP_MANT_F64      | if(S0.d == +-INF || S0.d == NAN) then  
    D.d = S0.d;  
else  
    D.d = Mantissa(S0.d);  
endif.  
Result range is in (-1.0,-0.5][0.5,1.0) in typical cases. Returns binary significand of double precision float input, such that S0.d = significand * (2 ** exponent). See also V_FREXP_EXP_I32_F64, which returns integer exponent. See the C library function frexp() for more information. |
| 50     | V_FRACT_F64           | D.d = S0.d + -floor(S0.d).  
Return fractional portion of a number. 0.5ULP accuracy, denormals are accepted. |
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>51</td>
<td>V_FREXP_EXP_I32_F32</td>
<td>if(S0.f == +INF</td>
</tr>
<tr>
<td>52</td>
<td>V_FREXP_MANT_F32</td>
<td>if(S0.f == +INF</td>
</tr>
<tr>
<td>53</td>
<td>V_CLREXCP</td>
<td>Clear wave's exception state in SIMD (SP).</td>
</tr>
</tbody>
</table>
"Vega" 7nm Instruction Set Architecture

Opcode

Name

55

V_SCREEN_PARTITIO
N_4SE_B32

Description
D.u = TABLE[S0.u[7:0]].
TABLE:
0x1, 0x3, 0x7, 0xf, 0x5, 0xf, 0xf, 0xf, 0x7, 0xf, 0xf, 0xf,
0xf, 0xf, 0xf, 0xf,
0xf, 0x2, 0x6, 0xe, 0xf, 0xa, 0xf, 0xf, 0xf, 0xb, 0xf, 0xf,
0xf, 0xf, 0xf, 0xf,
0xd, 0xf, 0x4, 0xc, 0xf, 0xf, 0x5, 0xf, 0xf, 0xf, 0xd, 0xf,
0xf, 0xf, 0xf, 0xf,
0x9, 0xb, 0xf, 0x8, 0xf, 0xf, 0xf, 0xa, 0xf, 0xf, 0xf, 0xe,
0xf, 0xf, 0xf, 0xf,
0xf, 0xf, 0xf, 0xf, 0x4, 0xc, 0xd, 0xf, 0x6, 0xf, 0xf, 0xf,
0xe, 0xf, 0xf, 0xf,
0xf, 0xf, 0xf, 0xf, 0xf, 0x8, 0x9, 0xb, 0xf, 0x9, 0x9, 0xf,
0xf, 0xd, 0xf, 0xf,
0xf, 0xf, 0xf, 0xf, 0x7, 0xf, 0x1, 0x3, 0xf, 0xf, 0x9, 0xf,
0xf, 0xf, 0xb, 0xf,
0xf, 0xf, 0xf, 0xf, 0x6, 0xe, 0xf, 0x2, 0x6, 0xf, 0xf, 0x6,
0xf, 0xf, 0xf, 0x7,
0xb, 0xf, 0xf, 0xf, 0xf, 0xf, 0xf, 0xf, 0x2, 0x3, 0xb, 0xf,
0xa, 0xf, 0xf, 0xf,
0xf, 0x7, 0xf, 0xf, 0xf, 0xf, 0xf, 0xf, 0xf, 0x1, 0x9, 0xd,
0xf, 0x5, 0xf, 0xf,
0xf, 0xf, 0xe, 0xf, 0xf, 0xf, 0xf, 0xf, 0xe, 0xf, 0x8, 0xc,
0xf, 0xf, 0xa, 0xf,
0xf, 0xf, 0xf, 0xd, 0xf, 0xf, 0xf, 0xf, 0x6, 0x7, 0xf, 0x4,
0xf, 0xf, 0xf, 0x5,
0x9, 0xf, 0xf, 0xf, 0xd, 0xf, 0xf, 0xf, 0xf, 0xf, 0xf, 0xf,
0x8, 0xc, 0xe, 0xf,
0xf, 0x6, 0x6, 0xf, 0xf, 0xe, 0xf, 0xf, 0xf, 0xf, 0xf, 0xf,
0xf, 0x4, 0x6, 0x7,
0xf, 0xf, 0x6, 0xf, 0xf, 0xf, 0x7, 0xf, 0xf, 0xf, 0xf, 0xf,
0xb, 0xf, 0x2, 0x3,
0x9, 0xf, 0xf, 0x9, 0xf, 0xf, 0xf, 0xb, 0xf, 0xf, 0xf, 0xf,
0x9, 0xd, 0xf, 0x1
4SE version of LUT instruction for screen partitioning/filtering.
This opcode is intended to accelerate screen partitioning in the
4SE case only. 2SE and 1SE cases use normal ALU instructions.
This opcode returns a 4-bit bitmask indicating which SE backends
are covered by a rectangle from (x_min, y_min) to (x_max, y_max).
With 32-pixel tiles the SE for (x, y) is given by

{ x[5] ^

. Using this formula we can determine which

SEs are covered by a larger rectangle.
The primitive shader must perform the following operation before
the opcode is called.
1. Compute the bounding box of the primitive (x_min, y_min)
(upper left) and (x_max, y_max) (lower right), in pixels.
2. Check for any extents that do not need to use the opcode ---

12.8. VOP1 Instructions

137 of 290


<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>57</td>
<td>V_CVT_F16_U16</td>
<td>( D.f16 = \text{uint16	extunderscore to	extunderscore flt16}(S.u16). ) 0.5ULP accuracy, supports denormals, rounding, exception flags and saturation.</td>
</tr>
<tr>
<td>58</td>
<td>V_CVT_F16_I16</td>
<td>( D.f16 = \text{int16	extunderscore to	extunderscore flt16}(S.i16). ) 0.5ULP accuracy, supports denormals, rounding, exception flags and saturation.</td>
</tr>
<tr>
<td>59</td>
<td>V_CVT_U16_F16</td>
<td>( D.u16 = \text{flt16	extunderscore to	extunderscore uint16}(S.f16). ) 1ULP accuracy, supports rounding, exception flags and saturation. 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.</td>
</tr>
<tr>
<td>60</td>
<td>V_CVT_I16_F16</td>
<td>( D.i16 = \text{flt16	extunderscore to	extunderscore int16}(S.f16). ) 1ULP accuracy, supports rounding, exception flags and saturation. 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.</td>
</tr>
</tbody>
</table>
| 61     | V_RCP_F16       | \( D.f16 = 1.0 / S0.f16. \) Reciprocal with IEEE rules and 0.51ULP accuracy. Examples: \[
V\_RCP\_F16(0xfc00) \Rightarrow 0x8000 \quad \text{rcp}(-\text{INF}) = -0 \\
V\_RCP\_F16(0xc000) \Rightarrow 0xb800 \quad \text{rcp}(-2.0) = -0.5 \\
V\_RCP\_F16(0x8000) \Rightarrow 0xcf00 \quad \text{rcp}(-0.0) = -\text{INF} \\
V\_RCP\_F16(0x0000) \Rightarrow 0x7c00 \quad \text{rcp}(+0.0) = +\text{INF} \\
V\_RCP\_F16(0x7c00) \Rightarrow 0x0000 \quad \text{rcp}(+\text{INF}) = +0
\] |
| 62     | V_SQRT_F16      | \( D.f16 = \text{sqrt}(S0.f16). \) Square root. 0.51ULP accuracy, denormals are supported. Examples: \[
V\_SQRT\_F16(0xfc00) \Rightarrow 0xfe00 \quad \text{sqrt}(-\text{INF}) = \text{NAN} \\
V\_SQRT\_F16(0x8000) \Rightarrow 0x8000 \quad \text{sqrt}(-0.0) = -0 \\
V\_SQRT\_F16(0x0000) \Rightarrow 0x0000 \quad \text{sqrt}(+0.0) = +0 \\
V\_SQRT\_F16(0x4400) \Rightarrow 0x4000 \quad \text{sqrt}(+4.0) = +2.0 \\
V\_SQRT\_F16(0x7c00) \Rightarrow 0x7c00 \quad \text{sqrt}(+\text{INF}) = +\text{INF}
\] |
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>V_RSQ_F16</td>
<td>$D.f16 = 1.0 / \sqrt{S0.f16}$.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reciprocal square root with IEEE rules. 0.51ULP accuracy,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>denormals are supported.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Examples:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- $V_{RSQ_F16}(0xfc00) \Rightarrow 0xfe00$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// rsq(-INF) = NAN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- $V_{RSQ_F16}(0x8000) \Rightarrow 0xfc00$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// rsq(-0.0) = -INF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- $V_{RSQ_F16}(0x8000) \Rightarrow 0x7c00$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// rsq(+0.0) = +INF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- $V_{RSQ_F16}(0x4400) \Rightarrow 0x3800$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// rsq(+4.0) = +0.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- $V_{RSQ_F16}(0x7c00) \Rightarrow 0x0000$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// rsq(+INF) = +0</td>
</tr>
<tr>
<td>64</td>
<td>V_LOG_F16</td>
<td>$D.f16 = \log_2(S0.f)$.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Base 2 logarithm. 0.51ULP accuracy, denormals are supported.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Examples:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- $V_{LOG_F16}(0xfc00) \Rightarrow 0xfe00$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// log(-INF) = NAN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- $V_{LOG_F16}(0xbc00) \Rightarrow 0xfe00$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// log(-1.0) = NAN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- $V_{LOG_F16}(0x8000) \Rightarrow 0xfc00$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// log(-0.0) = -INF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- $V_{LOG_F16}(0x8000) \Rightarrow 0x7c00$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// log(+0.0) = -INF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- $V_{LOG_F16}(0x3c00) \Rightarrow 0x0000$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// log(+1.0) = 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- $V_{LOG_F16}(0x7c00) \Rightarrow 0x7c00$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// log(+INF) = +INF</td>
</tr>
<tr>
<td>65</td>
<td>V_EXP_F16</td>
<td>$D.f16 = \text{pow}(2.0, S0.f16)$.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Base 2 exponentiation. 0.51ULP accuracy, denormals are supported.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Examples:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- $V_{EXP_F16}(0xfc00) \Rightarrow 0x0000$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// exp(-INF) = 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- $V_{EXP_F16}(0x8000) \Rightarrow 0x3c00$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// exp(-0.0) = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- $V_{EXP_F16}(0x7c00) \Rightarrow 0x7c00$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// exp(+INF) = +INF</td>
</tr>
<tr>
<td>66</td>
<td>V_FREXP_MANT_F16</td>
<td>if(S0.f16 == +INF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$D.f16 = S0.f16$;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$D.f16 = \text{Mantissa}(S0.f16)$;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>endif.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Result range is in $(-1.0,-0.5)[0.5,1.0)$ in typical cases. Returns</td>
</tr>
<tr>
<td></td>
<td></td>
<td>binary significand of half precision float input, such that S0.f16 =</td>
</tr>
<tr>
<td></td>
<td></td>
<td>significand * (2 ** exponent). See also V_FREXP_EXP_I16_F16, which</td>
</tr>
<tr>
<td></td>
<td></td>
<td>returns integer exponent. See the C library function frexp() for more</td>
</tr>
<tr>
<td></td>
<td></td>
<td>information.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| 67     | V_FREXP_EXP_I16_F16   | if(S0.f16 == +INF || S0.f16 == NAN) then  
|         |                       |   D.i = 0;  
|         |                       | else  
|         |                       |   D.i = TwosComplement(Exponent(S0.f16) - 15 + 1);  
|         |                       | endif.  
|         |                       | Returns exponent of half precision float input, such that S0.f16 = significand * (2 ** exponent). See also V_FREXP_MANT_F16, which returns the significand. See the C library function frexp() for more information. |
| 68     | V_FLOOR_F16           | D.f16 = trunc(S0.f16);  
|         |                       | if(S0.f16 < 0.0f && S0.f16 != D.f16) then  
|         |                       |   D.f16 -= 1.0;  
|         |                       | endif.  
|         |                       | Round down to previous whole integer. |
| 69     | V_CEIL_F16            | D.f16 = trunc(S0.f16);  
|         |                       | if(S0.f16 > 0.0f && S0.f16 != D.f16) then  
|         |                       |   D.f16 += 1.0;  
|         |                       | endif.  
|         |                       | Round up to next whole integer. |
| 70     | V_TRUNC_F16           | D.f16 = trunc(S0.f16).  
|         |                       | Return integer part of S0.f16, round-to-zero semantics. |
| 71     | V_RNDNE_F16           | D.f16 = floor(S0.f16 + 0.5);  
|         |                       | if(floor(S0.f16) is even && fract(S0.f16) == 0.5) then  
|         |                       |   D.f16 -= 1.0;  
|         |                       | endif.  
|         |                       | Round-to-nearest-even semantics. |
| 72     | V_FRACT_F16           | D.f16 = S0.f16 - floor(S0.f16).  
|         |                       | Return fractional portion of a number. 0.5ULP accuracy, denormals are accepted. |
| 73     | V_SIN_F16             | D.f16 = sin(S0.f16 * 2 * PI).  
|         |                       | Trigonometric sine. Denormals are supported.  
|         |                       | 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  

"Vega" 7nm Instruction Set Architecture

12.8. VOP1 Instructions
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>74</td>
<td>V_COS_F16</td>
<td>( D.f16 = \cos(S0.f16 \times 2 \times \pi) ). Trigonometric cosine. Denormals are supported.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Examples:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( V_{\text{COS}}_F16(0xfc00) \Rightarrow 0xfe00 \quad \text{//cos(-INF) = NAN} )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( V_{\text{COS}}_F16(0xfbff) \Rightarrow 0x3c00 \quad \text{//Most negative finite FP16}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( V_{\text{COS}}_F16(0x8000) \Rightarrow 0x3c00 \quad \text{//cos(0.0) = 1}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( V_{\text{COS}}_F16(0x3400) \Rightarrow 0x0000 \quad \text{//cos(0.25) = 0}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( V_{\text{COS}}_F16(0x7bff) \Rightarrow 0x3c00 \quad \text{//Most positive finite FP16}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( V_{\text{COS}}_F16(0x7c00) \Rightarrow 0xfe00 \quad \text{//cos(+INF) = NAN}</td>
</tr>
<tr>
<td>75</td>
<td>V_EXP_LEGACY_F32</td>
<td>( D.f = \text{pow}(2.0, S0.f) ). Power with legacy semantics.</td>
</tr>
<tr>
<td>76</td>
<td>V_LOG_LEGACY_F32</td>
<td>( D.f = \log2(S0.f) ). Base 2 logarithm with legacy semantics.</td>
</tr>
<tr>
<td>77</td>
<td>V_CVT_NORM_I16_F16</td>
<td>( D.i16 = \text{flt16}_\text{to}_\text{snorm16}(S.f16) ). 0.5ULP accuracy, supports rounding, exception flags and saturation, denormals are supported.</td>
</tr>
<tr>
<td>78</td>
<td>V_CVT_NORM_U16_F16</td>
<td>( D.u16 = \text{flt16}_\text{to}_\text{unorm16}(S.f16) ). 0.5ULP accuracy, supports rounding, exception flags and saturation, denormals are supported.</td>
</tr>
<tr>
<td>79</td>
<td>V_SAT_PK_U8_I16</td>
<td>( D.u32 = {16'b0, \text{sat8}(S.u[31:16]), \text{sat8}(S.u[15:0])} ).</td>
</tr>
<tr>
<td>81</td>
<td>V_SWAP_B32</td>
<td>( \text{tmp} = D.u; D.u = S0.u; S0.u = \text{tmp}. Swap operands. Input and output modifiers not supported; this is an untyped operation.</td>
</tr>
</tbody>
</table>

### 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:

<table>
<thead>
<tr>
<th>SRC0</th>
<th>VSRC1</th>
<th>OP</th>
<th>SRC2₀</th>
<th>SDST₁</th>
<th>VDST₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td></td>
<td>31</td>
<td>32</td>
<td>31</td>
<td>30</td>
</tr>
<tr>
<td>NEG</td>
<td>OMOD</td>
<td>OP₁₀</td>
<td>cmp₂</td>
<td>DST₄</td>
<td>DST₃</td>
</tr>
</tbody>
</table>

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 47. Instructions with Sixteen Compare Operations

<table>
<thead>
<tr>
<th>Compare Operation</th>
<th>Opcode Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>0</td>
<td>D.u = 0</td>
</tr>
<tr>
<td>LT</td>
<td>1</td>
<td>D.u = (S0 &lt; S1)</td>
</tr>
<tr>
<td>EQ</td>
<td>2</td>
<td>D.u = (S0 == S1)</td>
</tr>
<tr>
<td>LE</td>
<td>3</td>
<td>D.u = (S0 &lt;= S1)</td>
</tr>
<tr>
<td>GT</td>
<td>4</td>
<td>D.u = (S0 &gt; S1)</td>
</tr>
<tr>
<td>LG</td>
<td>5</td>
<td>D.u = (S0 &lt;&gt; S1)</td>
</tr>
<tr>
<td>GE</td>
<td>6</td>
<td>D.u = (S0 &gt;= S1)</td>
</tr>
<tr>
<td>O</td>
<td>7</td>
<td>D.u = !(isNaN(S0) &amp;&amp; !isNaN(S1))</td>
</tr>
<tr>
<td>Compare Operation</td>
<td>Opcode Offset</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>U</td>
<td>8</td>
<td>D.u = (!isNaN(S0)</td>
</tr>
<tr>
<td>NGE</td>
<td>9</td>
<td>D.u = !(S0 &gt;= S1)</td>
</tr>
<tr>
<td>NLG</td>
<td>10</td>
<td>D.u = !(S0 &lt;&gt; S1)</td>
</tr>
<tr>
<td>NGT</td>
<td>11</td>
<td>D.u = !(S0 &gt; S1)</td>
</tr>
<tr>
<td>NLE</td>
<td>12</td>
<td>D.u = !(S0 &lt;= S1)</td>
</tr>
<tr>
<td>NEQ</td>
<td>13</td>
<td>D.u = !(S0 == S1)</td>
</tr>
<tr>
<td>NLT</td>
<td>14</td>
<td>D.u = !(S0 &lt; S1)</td>
</tr>
<tr>
<td>TRU</td>
<td>15</td>
<td>D.u = 1</td>
</tr>
</tbody>
</table>

Table 48. Instructions with Sixteen Compare Operations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Hex Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>V_CMP_{COMPF}_F16</td>
<td>16-bit float compare.</td>
<td>0x20 to 0x2F</td>
</tr>
<tr>
<td>V_CMPX_{COMPF}_F16</td>
<td>16-bit float compare. Also writes EXEC.</td>
<td>0x30 to 0x3F</td>
</tr>
<tr>
<td>V_CMP_{COMPF}_F32</td>
<td>32-bit float compare.</td>
<td>0x40 to 0x4F</td>
</tr>
<tr>
<td>V_CMPX_{COMPF}_F32</td>
<td>32-bit float compare. Also writes EXEC.</td>
<td>0x50 to 0x5F</td>
</tr>
<tr>
<td>V_CMPS_{COMPF}_F64</td>
<td>64-bit float compare.</td>
<td>0x60 to 0x6F</td>
</tr>
<tr>
<td>V_CMPSX_{COMPF}_F64</td>
<td>64-bit float compare. Also writes EXEC.</td>
<td>0x70 to 0x7F</td>
</tr>
</tbody>
</table>

Table 49. Instructions with Sixteen Compare Operations

<table>
<thead>
<tr>
<th>Compare Operation</th>
<th>Opcode Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>0</td>
<td>D.u = 0</td>
</tr>
<tr>
<td>LT</td>
<td>1</td>
<td>D.u = (S0 &lt; S1)</td>
</tr>
<tr>
<td>EQ</td>
<td>2</td>
<td>D.u = (S0 == S1)</td>
</tr>
<tr>
<td>LE</td>
<td>3</td>
<td>D.u = (S0 &lt;= S1)</td>
</tr>
<tr>
<td>GT</td>
<td>4</td>
<td>D.u = (S0 &gt; S1)</td>
</tr>
<tr>
<td>LG</td>
<td>5</td>
<td>D.u = (S0 &lt;&gt; S1)</td>
</tr>
<tr>
<td>GE</td>
<td>6</td>
<td>D.u = (S0 &gt;= S1)</td>
</tr>
<tr>
<td>TRU</td>
<td>7</td>
<td>D.u = 1</td>
</tr>
</tbody>
</table>

Table 50. Instructions with Eight Compare Operations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Hex Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>V_CMP_{COMPI}_I16</td>
<td>16-bit signed integer compare.</td>
<td>0xA0 - 0xA7</td>
</tr>
<tr>
<td>V_CMP_{COMPI}_U16</td>
<td>16-bit signed integer compare. Also writes EXEC.</td>
<td>0xA8 - 0xAF</td>
</tr>
<tr>
<td>V_CMPX_{COMPI}_I16</td>
<td>16-bit unsigned integer compare.</td>
<td>0xB0 - 0xB7</td>
</tr>
</tbody>
</table>
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. Also writes EXEC. 0xC8 - 0xCF
V_CMPX_{COMPI}_I32 32-bit unsigned integer compare. 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. Also writes EXEC. 0xE8 - 0xEF
V_CMPX_{COMPI}_I64 64-bit unsigned integer compare. 0xF0 - 0xF7
V_CMPX_{COMPI}_U64 64-bit unsigned integer compare. Also writes EXEC. 0xF8 - 0xFF

Table 51. VOPC Compare Opcodes

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>V_CMP_CLASS_F32</td>
<td>VCC = IEEE numeric class function specified in S1.u, performed on S0.f</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The function reports true if the floating point value is <em>any</em> of the numeric types selected in S1.u according to the following list:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[0] -- value is a signaling NaN.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[1] -- value is a quiet NaN.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[2] -- value is negative infinity.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[3] -- value is a negative normal value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[4] -- value is a negative denormal value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[5] -- value is negative zero.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[6] -- value is positive zero.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[7] -- value is a positive denormal value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[8] -- value is a positive normal value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[9] -- value is positive infinity.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>---------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>17</td>
<td>V_CMPX_CLASS_F32</td>
<td>EXEC = VCC = IEEE numeric class function specified in S1.u, performed on S0.f</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The function reports true if the floating point value is <em>any</em> of the numeric types selected in S1.u according to the following list:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[0] -- value is a signaling NaN.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[1] -- value is a quiet NaN.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[2] -- value is negative infinity.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[3] -- value is a negative normal value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[4] -- value is a negative denormal value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[5] -- value is negative zero.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[6] -- value is positive zero.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[7] -- value is a positive denormal value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[8] -- value is a positive normal value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[9] -- value is positive infinity.</td>
</tr>
<tr>
<td>18</td>
<td>V_CMP_CLASS_F64</td>
<td>VCC = IEEE numeric class function specified in S1.u, performed on S0.d</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The function reports true if the floating point value is <em>any</em> of the numeric types selected in S1.u according to the following list:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[0] -- value is a signaling NaN.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[1] -- value is a quiet NaN.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[2] -- value is negative infinity.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[3] -- value is a negative normal value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[4] -- value is a negative denormal value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[5] -- value is negative zero.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[6] -- value is positive zero.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[7] -- value is a positive denormal value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[8] -- value is a positive normal value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[9] -- value is positive infinity.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| 19     | V_CMPX_CLASS_F64 | EXEC = VCC = IEEE numeric class function specified in S1.u, performed on S0.d  
The function reports true if the floating point value is *any* of the numeric types selected in S1.u 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. |
| 20     | V_CMP_CLASS_F16  | VCC = IEEE numeric class function specified in S1.u, performed on S0.f16.  
Note that the S1 has a format of f16 since floating point literal constants are interpreted as 16 bit value for this opcode  
The function reports true if the floating point value is *any* of the numeric types selected in S1.u 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. |
## 12.9. VOPC Instructions

### Opcode 21: V_CMPX_CLASS_F16

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>V_CMPX_CLASS_F16</td>
<td>EXEC = VCC = IEEE numeric class function specified in S1.u, performed on S0.f16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Note that the S1 has a format of f16 since floating point literal constants are interpreted as 16 bit value for this opcode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The function reports true if the floating point value is <em>any</em> of the numeric types selected in S1.u according to the following list:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[0] -- value is a signaling NaN.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[1] -- value is a quiet NaN.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[2] -- value is negative infinity.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[3] -- value is a negative normal value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[4] -- value is a negative denormal value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[5] -- value is negative zero.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[6] -- value is positive zero.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[7] -- value is a positive denormal value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[8] -- value is a positive normal value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1.u[9] -- value is positive infinity.</td>
</tr>
</tbody>
</table>

### Opcode 32: V_CMP_F_F16

D.u64[threadId] = 0.

### Opcode 33: V_CMP_LT_F16

D.u64[threadId] = (S0 < S1).

### Opcode 34: V_CMP_EQ_F16

D.u64[threadId] = (S0 == S1).

### Opcode 35: V_CMP_LE_F16

D.u64[threadId] = (S0 <= S1).

### Opcode 36: V_CMP_GT_F16

D.u64[threadId] = (S0 > S1).

### Opcode 37: V_CMP_LG_F16

D.u64[threadId] = (S0 <> S1).

### Opcode 38: V_CMP_GE_F16

D.u64[threadId] = (S0 >= S1).

### Opcode 39: V_CMP_O_F16

D.u64[threadId] = (!isNan(S0) && !isNan(S1)).

### Opcode 40: V_CMP_U_F16

D.u64[threadId] = (isNan(S0) || isNan(S1)).

### Opcode 41: V_CMP_NGE_F16

D.u64[threadId] = !(S0 >= S1) // With NAN inputs this is not the same operation as <=.

### Opcode 42: V_CMP_NLG_F16

D.u64[threadId] = !(S0 <> S1) // With NAN inputs this is not the same operation as ==.

### Opcode 43: V_CMP_NGT_F16

D.u64[threadId] = !(S0 > S1) // With NAN inputs this is not the same operation as <=.

### Opcode 44: V_CMP_NLE_F16

D.u64[threadId] = !(S0 <= S1) // With NAN inputs this is not the same operation as >.

### Opcode 45: V_CMP_NEQ_F16

D.u64[threadId] = !(S0 == S1) // With NAN inputs this is not the same operation as !=.

### Opcode 46: V_CMP_NLT_F16

D.u64[threadId] = !(S0 < S1) // With NAN inputs this is not the same operation as >=.

### Opcode 47: V_CMP_TRU_F16

D.u64[threadId] = 1.
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td>V_CMPX_F_F16</td>
<td>EXEC[threadId] = D.u64[threadId] = 0.</td>
</tr>
<tr>
<td>49</td>
<td>V_CMPX_LT_F16</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &lt; S1).</td>
</tr>
<tr>
<td>50</td>
<td>V_CMPX_EQ_F16</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 == S1).</td>
</tr>
<tr>
<td>51</td>
<td>V_CMPX_LE_F16</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &lt;= S1).</td>
</tr>
<tr>
<td>52</td>
<td>V_CMPX_GT_F16</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &gt; S1).</td>
</tr>
<tr>
<td>53</td>
<td>V_CMPX_GE_F16</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &gt;= S1).</td>
</tr>
<tr>
<td>54</td>
<td>V_CMPX_O_F16</td>
<td>EXEC[threadId] = D.u64[threadId] = (!isNan(S0) &amp;&amp; !isNan(S1)).</td>
</tr>
<tr>
<td>55</td>
<td>V_CMPX_U_F16</td>
<td>EXEC[threadId] = D.u64[threadId] = (isNan(S0)</td>
</tr>
<tr>
<td>56</td>
<td>V_CMPX_NGE_F16</td>
<td>EXEC[threadId] = D.u64[threadId] = !(S0 &gt;= S1) // With NAN inputs this is not the same operation as &lt;.</td>
</tr>
<tr>
<td>57</td>
<td>V_CMPX_NLG_F16</td>
<td>EXEC[threadId] = D.u64[threadId] = !(S0 &lt;&gt; S1) // With NAN inputs this is not the same operation as ==.</td>
</tr>
<tr>
<td>58</td>
<td>V_CMPX_NGT_F16</td>
<td>EXEC[threadId] = D.u64[threadId] = !(S0 &gt; S1) // With NAN inputs this is not the same operation as &lt;=.</td>
</tr>
<tr>
<td>59</td>
<td>V_CMPX_NLE_F16</td>
<td>EXEC[threadId] = D.u64[threadId] = !(S0 &lt;= S1) // With NAN inputs this is not the same operation as &gt;.</td>
</tr>
<tr>
<td>60</td>
<td>V_CMPX_NEQ_F16</td>
<td>EXEC[threadId] = D.u64[threadId] = !(S0 == S1) // With NAN inputs this is not the same operation as !=.</td>
</tr>
<tr>
<td>61</td>
<td>V_CMPX_NLT_F16</td>
<td>EXEC[threadId] = D.u64[threadId] = !(S0 &lt; S1) // With NAN inputs this is not the same operation as &gt;=.</td>
</tr>
<tr>
<td>62</td>
<td>V_CMPX_TRU_F16</td>
<td>EXEC[threadId] = D.u64[threadId] = 1.</td>
</tr>
<tr>
<td>63</td>
<td>V_CMP_F_F32</td>
<td>D.u64[threadId] = 0.</td>
</tr>
<tr>
<td>64</td>
<td>V_CMP_LT_F32</td>
<td>D.u64[threadId] = (S0 &lt; S1).</td>
</tr>
<tr>
<td>65</td>
<td>V_CMP_EQ_F32</td>
<td>D.u64[threadId] = (S0 == S1).</td>
</tr>
<tr>
<td>66</td>
<td>V_CMP_LE_F32</td>
<td>D.u64[threadId] = (S0 &lt;= S1).</td>
</tr>
<tr>
<td>67</td>
<td>V_CMP_GT_F32</td>
<td>D.u64[threadId] = (S0 &gt; S1).</td>
</tr>
<tr>
<td>68</td>
<td>V_CMP_GE_F32</td>
<td>D.u64[threadId] = (S0 &gt;= S1).</td>
</tr>
<tr>
<td>69</td>
<td>V_CMP_O_F32</td>
<td>D.u64[threadId] = (!isNan(S0) &amp;&amp; !isNan(S1)).</td>
</tr>
<tr>
<td>70</td>
<td>V_CMP_U_F32</td>
<td>D.u64[threadId] = (isNan(S0)</td>
</tr>
<tr>
<td>71</td>
<td>V_CMP_NGE_F32</td>
<td>D.u64[threadId] = !(S0 &gt;= S1) // With NAN inputs this is not the same operation as &lt;.</td>
</tr>
<tr>
<td>72</td>
<td>V_CMP_NLG_F32</td>
<td>D.u64[threadId] = !(S0 &lt;&gt; S1) // With NAN inputs this is not the same operation as ==.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>------------------</td>
<td>---------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>75</td>
<td>V_CMP_NGT_F32</td>
<td>D.u64[threadId] = !(S0 &gt; S1) // With NAN inputs this is not the same operation as &lt;=.</td>
</tr>
<tr>
<td>76</td>
<td>V_CMP_NLE_F32</td>
<td>D.u64[threadId] = !(S0 &lt;= S1) // With NAN inputs this is not the same operation as &gt;.</td>
</tr>
<tr>
<td>77</td>
<td>V_CMP_NEQ_F32</td>
<td>D.u64[threadId] = !(S0 == S1) // With NAN inputs this is not the same operation as !=.</td>
</tr>
<tr>
<td>78</td>
<td>V_CMP_NLT_F32</td>
<td>D.u64[threadId] = !(S0 &lt; S1) // With NAN inputs this is not the same operation as &gt;=.</td>
</tr>
<tr>
<td>79</td>
<td>V_CMP_TRU_F32</td>
<td>D.u64[threadId] = 1.</td>
</tr>
<tr>
<td>80</td>
<td>V_CMPX_F_F32</td>
<td>EXEC[threadId] = D.u64[threadId] = 0.</td>
</tr>
<tr>
<td>81</td>
<td>V_CMPX_LT_F32</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &lt; S1).</td>
</tr>
<tr>
<td>82</td>
<td>V_CMPX_EQ_F32</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 == S1).</td>
</tr>
<tr>
<td>83</td>
<td>V_CMPX_LE_F32</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &lt;= S1).</td>
</tr>
<tr>
<td>84</td>
<td>V_CMPX_GT_F32</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &gt; S1).</td>
</tr>
<tr>
<td>85</td>
<td>V_CMPX_LG_F32</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &gt; S1).</td>
</tr>
<tr>
<td>86</td>
<td>V_CMPX_GE_F32</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &gt;= S1).</td>
</tr>
<tr>
<td>87</td>
<td>V_CMPX_O_F32</td>
<td>EXEC[threadId] = D.u64[threadId] = (!isNan(S0) &amp;&amp; !isNan(S1)).</td>
</tr>
<tr>
<td>88</td>
<td>V_CMPX_U_F32</td>
<td>EXEC[threadId] = D.u64[threadId] = (isNan(S0)</td>
</tr>
<tr>
<td>89</td>
<td>V_CMPX_NGE_F32</td>
<td>EXEC[threadId] = D.u64[threadId] = !(S0 &gt;= S1) // With NAN inputs this is not the same operation as &lt;.</td>
</tr>
<tr>
<td>90</td>
<td>V_CMPX_NLG_F32</td>
<td>EXEC[threadId] = D.u64[threadId] = !(S0 &lt;= S1) // With NAN inputs this is not the same operation as ==.</td>
</tr>
<tr>
<td>91</td>
<td>V_CMPX_NGT_F32</td>
<td>EXEC[threadId] = D.u64[threadId] = !(S0 &gt; S1) // With NAN inputs this is not the same operation as &lt;=.</td>
</tr>
<tr>
<td>92</td>
<td>V_CMPX_NLE_F32</td>
<td>EXEC[threadId] = D.u64[threadId] = !(S0 &lt;= S1) // With NAN inputs this is not the same operation as &gt;.</td>
</tr>
<tr>
<td>93</td>
<td>V_CMPX_NEQ_F32</td>
<td>EXEC[threadId] = D.u64[threadId] = !(S0 == S1) // With NAN inputs this is not the same operation as !=.</td>
</tr>
<tr>
<td>94</td>
<td>V_CMPX_NLT_F32</td>
<td>EXEC[threadId] = D.u64[threadId] = !(S0 &lt; S1) // With NAN inputs this is not the same operation as &gt;=.</td>
</tr>
<tr>
<td>95</td>
<td>V_CMPX_TRU_F32</td>
<td>EXEC[threadId] = D.u64[threadId] = 1.</td>
</tr>
<tr>
<td>96</td>
<td>V_CMP_F_F64</td>
<td>D.u64[threadId] = 0.</td>
</tr>
<tr>
<td>97</td>
<td>V_CMP_LT_F64</td>
<td>D.u64[threadId] = (S0 &lt; S1).</td>
</tr>
<tr>
<td>98</td>
<td>V_CMP_EQ_F64</td>
<td>D.u64[threadId] = (S0 == S1).</td>
</tr>
<tr>
<td>99</td>
<td>V_CMP_LE_F64</td>
<td>D.u64[threadId] = (S0 &lt;= S1).</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>100</td>
<td>V_CMP_GT_F64</td>
<td>D.u64[threadId] = (S0 &gt; S1).</td>
</tr>
<tr>
<td>101</td>
<td>V_CMP_LG_F64</td>
<td>D.u64[threadId] = (S0 &lt;&gt; S1).</td>
</tr>
<tr>
<td>102</td>
<td>V_CMP_GE_F64</td>
<td>D.u64[threadId] = (S0 &gt;= S1).</td>
</tr>
<tr>
<td>103</td>
<td>V_CMP_O_F64</td>
<td>D.u64[threadId] = (!isNan(S0) &amp;&amp; !isNan(S1)).</td>
</tr>
<tr>
<td>104</td>
<td>V_CMP_U_F64</td>
<td>D.u64[threadId] = (isNan(S0)</td>
</tr>
<tr>
<td>105</td>
<td>V_CMP_NGE_F64</td>
<td>D.u64[threadId] = !(S0 &gt;= S1) // With NAN inputs this is not the same operation as &lt;.</td>
</tr>
<tr>
<td>106</td>
<td>V_CMP_NLG_F64</td>
<td>D.u64[threadId] = !(S0 &lt;&gt; S1) // With NAN inputs this is not the same operation as ==.</td>
</tr>
<tr>
<td>107</td>
<td>V_CMP_NGT_F64</td>
<td>D.u64[threadId] = !(S0 &gt; S1) // With NAN inputs this is not the same operation as &lt;=.</td>
</tr>
<tr>
<td>108</td>
<td>V_CMP_NLE_F64</td>
<td>D.u64[threadId] = !(S0 &lt;= S1) // With NAN inputs this is not the same operation as &gt;.</td>
</tr>
<tr>
<td>109</td>
<td>V_CMP_NEQ_F64</td>
<td>D.u64[threadId] = !(S0 == S1) // With NAN inputs this is not the same operation as !=.</td>
</tr>
<tr>
<td>110</td>
<td>V_CMP_NLT_F64</td>
<td>D.u64[threadId] = !(S0 &lt; S1) // With NAN inputs this is not the same operation as &gt;=.</td>
</tr>
<tr>
<td>111</td>
<td>V_CMP_TRU_F64</td>
<td>D.u64[threadId] = 1.</td>
</tr>
<tr>
<td>112</td>
<td>V_CMPX_F_F64</td>
<td>EXEC[threadId] = D.u64[threadId] = 0.</td>
</tr>
<tr>
<td>113</td>
<td>V_CMPX_LT_F64</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &lt; S1).</td>
</tr>
<tr>
<td>114</td>
<td>V_CMPX_EQ_F64</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 == S1).</td>
</tr>
<tr>
<td>115</td>
<td>V_CMPX_LE_F64</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &lt;= S1).</td>
</tr>
<tr>
<td>116</td>
<td>V_CMPX_GT_F64</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &gt; S1).</td>
</tr>
<tr>
<td>117</td>
<td>V_CMPX_LG_F64</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &lt;&gt; S1).</td>
</tr>
<tr>
<td>118</td>
<td>V_CMPX_GE_F64</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &gt;= S1).</td>
</tr>
<tr>
<td>119</td>
<td>V_CMPX_O_F64</td>
<td>EXEC[threadId] = D.u64[threadId] = (!isNan(S0) &amp;&amp; !isNan(S1)).</td>
</tr>
<tr>
<td>120</td>
<td>V_CMPX_U_F64</td>
<td>EXEC[threadId] = D.u64[threadId] = (isNan(S0)</td>
</tr>
<tr>
<td>121</td>
<td>V_CMPX_NGE_F64</td>
<td>EXEC[threadId] = D.u64[threadId] = !(S0 &gt;= S1) // With NAN inputs this is not the same operation as &lt;.</td>
</tr>
<tr>
<td>122</td>
<td>V_CMPX_NLG_F64</td>
<td>EXEC[threadId] = D.u64[threadId] = !(S0 &lt;&gt; S1) // With NAN inputs this is not the same operation as ==.</td>
</tr>
<tr>
<td>123</td>
<td>V_CMPX_NGT_F64</td>
<td>EXEC[threadId] = D.u64[threadId] = !(S0 &gt; S1) // With NAN inputs this is not the same operation as &lt;=.</td>
</tr>
<tr>
<td>124</td>
<td>V_CMPX_NLE_F64</td>
<td>EXEC[threadId] = D.u64[threadId] = !(S0 &lt;= S1) // With NAN inputs this is not the same operation as &gt;.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>125</td>
<td>V_CMPX_NEQ_F64</td>
<td>EXEC[threadId] = D.u64[threadId] = !(S0 == S1) // With NAN inputs this is not the same operation as !=.</td>
</tr>
<tr>
<td>126</td>
<td>V_CMPX_NLT_F64</td>
<td>EXEC[threadId] = D.u64[threadId] = !(S0 &lt; S1) // With NAN inputs this is not the same operation as &gt;=.</td>
</tr>
<tr>
<td>127</td>
<td>V_CMPX_TRU_F64</td>
<td>EXEC[threadId] = D.u64[threadId] = 1.</td>
</tr>
<tr>
<td>160</td>
<td>V_CMP_F_I16</td>
<td>D.u64[threadId] = 0.</td>
</tr>
<tr>
<td>161</td>
<td>V_CMP_LT_I16</td>
<td>D.u64[threadId] = (S0 &lt; S1).</td>
</tr>
<tr>
<td>162</td>
<td>V_CMP_EQ_I16</td>
<td>D.u64[threadId] = (S0 == S1).</td>
</tr>
<tr>
<td>163</td>
<td>V_CMP_LE_I16</td>
<td>D.u64[threadId] = (S0 &lt;= S1).</td>
</tr>
<tr>
<td>164</td>
<td>V_CMP_GT_I16</td>
<td>D.u64[threadId] = (S0 &gt; S1).</td>
</tr>
<tr>
<td>165</td>
<td>V_CMP_NE_I16</td>
<td>D.u64[threadId] = (S0 &lt;&gt; S1).</td>
</tr>
<tr>
<td>166</td>
<td>V_CMP_GT_U16</td>
<td>D.u64[threadId] = (S0 &gt;= S1).</td>
</tr>
<tr>
<td>167</td>
<td>V_CMP_T_I16</td>
<td>D.u64[threadId] = 1.</td>
</tr>
<tr>
<td>168</td>
<td>V_CMP_F_U16</td>
<td>D.u64[threadId] = 0.</td>
</tr>
<tr>
<td>169</td>
<td>V_CMP_LT_U16</td>
<td>D.u64[threadId] = (S0 &lt; S1).</td>
</tr>
<tr>
<td>170</td>
<td>V_CMP_EQ_U16</td>
<td>D.u64[threadId] = (S0 == S1).</td>
</tr>
<tr>
<td>171</td>
<td>V_CMP_LE_U16</td>
<td>D.u64[threadId] = (S0 &lt;= S1).</td>
</tr>
<tr>
<td>172</td>
<td>V_CMP_GT_U16</td>
<td>D.u64[threadId] = (S0 &gt; S1).</td>
</tr>
<tr>
<td>173</td>
<td>V_CMP_T_U16</td>
<td>D.u64[threadId] = 1.</td>
</tr>
<tr>
<td>174</td>
<td>V_CMPX_F_I16</td>
<td>EXEC[threadId] = D.u64[threadId] = 0.</td>
</tr>
<tr>
<td>177</td>
<td>V_CMPX_LT_I16</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &lt; S1).</td>
</tr>
<tr>
<td>178</td>
<td>V_CMPX_EQ_I16</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 == S1).</td>
</tr>
<tr>
<td>179</td>
<td>V_CMPX_LE_I16</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &lt;= S1).</td>
</tr>
<tr>
<td>180</td>
<td>V_CMPX_GT_I16</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &gt; S1).</td>
</tr>
<tr>
<td>181</td>
<td>V_CMPX_NLT_I16</td>
<td>EXEC[threadId] = D.u64[threadId] = !(S0 &lt; S1).</td>
</tr>
<tr>
<td>182</td>
<td>V_CMPX_EQ_I16</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 == S1).</td>
</tr>
<tr>
<td>183</td>
<td>V_CMPX_LE_I16</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &lt;= S1).</td>
</tr>
<tr>
<td>184</td>
<td>V_CMPX_GT_I16</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &gt; S1).</td>
</tr>
<tr>
<td>185</td>
<td>V_CMPX_LT_I16</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &lt; S1).</td>
</tr>
<tr>
<td>186</td>
<td>V_CMPX_EQ_U16</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 == S1).</td>
</tr>
<tr>
<td>187</td>
<td>V_CMPX_LE_U16</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &lt;= S1).</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>188</td>
<td>V_CMPX_GT_U16</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &gt; S1).</td>
</tr>
<tr>
<td>189</td>
<td>V_CMPX_NE_U16</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &lt;&gt; S1).</td>
</tr>
<tr>
<td>190</td>
<td>V_CMPX_GE_U16</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &gt;= S1).</td>
</tr>
<tr>
<td>191</td>
<td>V_CMPX_T_U16</td>
<td>EXEC[threadId] = D.u64[threadId] = 1.</td>
</tr>
<tr>
<td>192</td>
<td>V_CMP_F_I32</td>
<td>D.u64[threadId] = 0.</td>
</tr>
<tr>
<td>193</td>
<td>V_CMP_LT_I32</td>
<td>D.u64[threadId] = (S0 &lt; S1).</td>
</tr>
<tr>
<td>194</td>
<td>V_CMP_EQ_I32</td>
<td>D.u64[threadId] = (S0 == S1).</td>
</tr>
<tr>
<td>195</td>
<td>V_CMP_LE_I32</td>
<td>D.u64[threadId] = (S0 &lt;= S1).</td>
</tr>
<tr>
<td>196</td>
<td>V_CMP_GT_I32</td>
<td>D.u64[threadId] = (S0 &gt; S1).</td>
</tr>
<tr>
<td>197</td>
<td>V_CMP_NE_I32</td>
<td>D.u64[threadId] = (S0 &lt;&gt; S1).</td>
</tr>
<tr>
<td>198</td>
<td>V_CMP_GE_I32</td>
<td>D.u64[threadId] = (S0 &gt;= S1).</td>
</tr>
<tr>
<td>199</td>
<td>V_CMP_T_I32</td>
<td>D.u64[threadId] = 1.</td>
</tr>
<tr>
<td>200</td>
<td>V_CMP_F_U32</td>
<td>D.u64[threadId] = 0.</td>
</tr>
<tr>
<td>201</td>
<td>V_CMP_LT_U32</td>
<td>D.u64[threadId] = (S0 &lt; S1).</td>
</tr>
<tr>
<td>202</td>
<td>V_CMP_EQ_U32</td>
<td>D.u64[threadId] = (S0 == S1).</td>
</tr>
<tr>
<td>203</td>
<td>V_CMP_LE_U32</td>
<td>D.u64[threadId] = (S0 &lt;= S1).</td>
</tr>
<tr>
<td>204</td>
<td>V_CMP_GT_U32</td>
<td>D.u64[threadId] = (S0 &gt; S1).</td>
</tr>
<tr>
<td>205</td>
<td>V_CMP_NE_U32</td>
<td>D.u64[threadId] = (S0 &lt;&gt; S1).</td>
</tr>
<tr>
<td>206</td>
<td>V_CMP_GE_U32</td>
<td>D.u64[threadId] = (S0 &gt;= S1).</td>
</tr>
<tr>
<td>207</td>
<td>V_CMP_T_U32</td>
<td>D.u64[threadId] = 1.</td>
</tr>
<tr>
<td>208</td>
<td>V_CMPX_F_I32</td>
<td>EXEC[threadId] = D.u64[threadId] = 0.</td>
</tr>
<tr>
<td>209</td>
<td>V_CMPX_LT_I32</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &lt; S1).</td>
</tr>
<tr>
<td>210</td>
<td>V_CMPX_EQ_I32</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 == S1).</td>
</tr>
<tr>
<td>211</td>
<td>V_CMPX_LE_I32</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &lt;= S1).</td>
</tr>
<tr>
<td>212</td>
<td>V_CMPX_GT_I32</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &gt; S1).</td>
</tr>
<tr>
<td>213</td>
<td>V_CMPX_NE_I32</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &lt;&gt; S1).</td>
</tr>
<tr>
<td>214</td>
<td>V_CMPX_GE_I32</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &gt;= S1).</td>
</tr>
<tr>
<td>215</td>
<td>V_CMPX_T_I32</td>
<td>EXEC[threadId] = D.u64[threadId] = 1.</td>
</tr>
<tr>
<td>216</td>
<td>V_CMPX_F_U32</td>
<td>EXEC[threadId] = D.u64[threadId] = 0.</td>
</tr>
<tr>
<td>217</td>
<td>V_CMPX_LT_U32</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &lt; S1).</td>
</tr>
<tr>
<td>218</td>
<td>V_CMPX_EQ_U32</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 == S1).</td>
</tr>
<tr>
<td>219</td>
<td>V_CMPX_LE_U32</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &lt;= S1).</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>220</td>
<td>V_CMPX_GT_U32</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &gt; S1).</td>
</tr>
<tr>
<td>221</td>
<td>V_CMPX_NE_U32</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &lt;&gt; S1).</td>
</tr>
<tr>
<td>222</td>
<td>V_CMPX_GE_U32</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &gt;= S1).</td>
</tr>
<tr>
<td>223</td>
<td>V_CMPX_T_U32</td>
<td>EXEC[threadId] = D.u64[threadId] = 1.</td>
</tr>
<tr>
<td>224</td>
<td>V_CMP_F_I64</td>
<td>D.u64[threadId] = 0.</td>
</tr>
<tr>
<td>225</td>
<td>V_CMP_LT_I64</td>
<td>D.u64[threadId] = (S0 &lt; S1).</td>
</tr>
<tr>
<td>226</td>
<td>V_CMP_EQ_I64</td>
<td>D.u64[threadId] = (S0 == S1).</td>
</tr>
<tr>
<td>227</td>
<td>V_CMP_LE_I64</td>
<td>D.u64[threadId] = (S0 &lt;= S1).</td>
</tr>
<tr>
<td>228</td>
<td>V_CMP_GT_I64</td>
<td>D.u64[threadId] = (S0 &gt; S1).</td>
</tr>
<tr>
<td>229</td>
<td>V_CMP_NE_I64</td>
<td>D.u64[threadId] = (S0 &lt;&gt; S1).</td>
</tr>
<tr>
<td>230</td>
<td>V_CMP_GE_I64</td>
<td>D.u64[threadId] = (S0 &gt;= S1).</td>
</tr>
<tr>
<td>231</td>
<td>V_CMP_T_I64</td>
<td>D.u64[threadId] = 1.</td>
</tr>
<tr>
<td>232</td>
<td>V_CMP_F_U64</td>
<td>D.u64[threadId] = 0.</td>
</tr>
<tr>
<td>233</td>
<td>V_CMP_LT_U64</td>
<td>D.u64[threadId] = (S0 &lt; S1).</td>
</tr>
<tr>
<td>234</td>
<td>V_CMP_EQ_U64</td>
<td>D.u64[threadId] = (S0 == S1).</td>
</tr>
<tr>
<td>235</td>
<td>V_CMP_LE_U64</td>
<td>D.u64[threadId] = (S0 &lt;= S1).</td>
</tr>
<tr>
<td>236</td>
<td>V_CMP_GT_U64</td>
<td>D.u64[threadId] = (S0 &gt; S1).</td>
</tr>
<tr>
<td>237</td>
<td>V_CMP_NE_U64</td>
<td>D.u64[threadId] = (S0 &lt;&gt; S1).</td>
</tr>
<tr>
<td>238</td>
<td>V_CMP_GE_U64</td>
<td>D.u64[threadId] = (S0 &gt;= S1).</td>
</tr>
<tr>
<td>239</td>
<td>V_CMP_T_U64</td>
<td>D.u64[threadId] = 1.</td>
</tr>
<tr>
<td>240</td>
<td>V_CMPX_F_I64</td>
<td>EXEC[threadId] = D.u64[threadId] = 0.</td>
</tr>
<tr>
<td>241</td>
<td>V_CMPX_LT_I64</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &lt; S1).</td>
</tr>
<tr>
<td>242</td>
<td>V_CMPX_EQ_I64</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 == S1).</td>
</tr>
<tr>
<td>243</td>
<td>V_CMPX_LE_I64</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &lt;= S1).</td>
</tr>
<tr>
<td>244</td>
<td>V_CMPX_GT_I64</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &gt; S1).</td>
</tr>
<tr>
<td>245</td>
<td>V_CMPX_NE_I64</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &lt;&gt; S1).</td>
</tr>
<tr>
<td>246</td>
<td>V_CMPX_GE_I64</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &gt;= S1).</td>
</tr>
<tr>
<td>247</td>
<td>V_CMPX_T_I64</td>
<td>EXEC[threadId] = D.u64[threadId] = 1.</td>
</tr>
<tr>
<td>248</td>
<td>V_CMPX_F_U64</td>
<td>EXEC[threadId] = D.u64[threadId] = 0.</td>
</tr>
<tr>
<td>249</td>
<td>V_CMPX_LT_U64</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &lt; S1).</td>
</tr>
<tr>
<td>250</td>
<td>V_CMPX_EQ_U64</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 == S1).</td>
</tr>
<tr>
<td>251</td>
<td>V_CMPX_LE_U64</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &lt;= S1).</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>252</td>
<td>V_CMPX_GT_U64</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &gt; S1).</td>
</tr>
<tr>
<td>253</td>
<td>V_CMPX_NE_U64</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &lt;&gt; S1).</td>
</tr>
<tr>
<td>254</td>
<td>V_CMPX_GE_U64</td>
<td>EXEC[threadId] = D.u64[threadId] = (S0 &gt;= S1).</td>
</tr>
<tr>
<td>255</td>
<td>V_CMPX_T_U64</td>
<td>EXEC[threadId] = D.u64[threadId] = 1.</td>
</tr>
</tbody>
</table>

### 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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>V_PK_MAD_I16</td>
<td>(D.i[31:16] = S0.i[31:16] * S1.i[31:16] + S2.i[31:16] ). (D.i[15:0] = S0.i[15:0] * S1.i[15:0] + S2.i[15:0] ).</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>2</td>
<td>V_PK_ADD_I16</td>
<td>D.i[31:16] = S0.i[31:16] + S1.i[31:16] . D.i[15:0] = S0.i[15:0] + S1.i[15:0]</td>
</tr>
<tr>
<td>3</td>
<td>V_PK_SUB_I16</td>
<td>D.i[31:16] = S0.i[31:16] - S1.i[31:16] . D.i[15:0] = S0.i[15:0] - S1.i[15:0]</td>
</tr>
<tr>
<td>6</td>
<td>V_PK_ASHRREV_I16</td>
<td>D.i[31:16] = S1.i[31:16] &gt;&gt; S0.i[19:16] . D.i[15:0] = S1.i[15:0] &gt;&gt; S0.i[3:0]</td>
</tr>
<tr>
<td>7</td>
<td>V_PK_MAX_I16</td>
<td>D.i[31:16] = (S0.i[31:16] &gt;= S1.i[31:16]) ? S0.i[31:16] : S1.i[31:16] . D.i[15:0] = (S0.i[15:0] &gt;= S1.i[15:0]) ? S0.i[15:0] : S1.i[15:0]</td>
</tr>
<tr>
<td>8</td>
<td>V_PK_MIN_I16</td>
<td>D.i[31:16] = (S0.i[31:16] &lt; S1.i[31:16]) ? S0.i[31:16] : S1.i[31:16] . D.i[15:0] = (S0.i[15:0] &lt; S1.i[15:0]) ? S0.i[15:0] : S1.i[15:0]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fused half-precision multiply add.</td>
</tr>
<tr>
<td>17</td>
<td>V_PK_MIN_F16</td>
<td>D.f[31:16] = min(S0.f[31:16], S1.f[31:16]) . D.f[15:0] = min(S0.f[15:0], S1.u[15:0])</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>18</td>
<td>V_PK_MAX_F16</td>
<td>$D.f[31:16] = \max(S0.f[31:16], S1.f[31:16]) \cdot D.f[15:0] = \max(S0.f[15:0], S1.f[15:0])$.</td>
</tr>
<tr>
<td>32</td>
<td>V_MAD_MIX_F32</td>
<td>$D.f[31:0] = S0.f \cdot S1.f + S2.f$. Size and location of $S0$, $S1$ and $S2$ controlled by OPSEL: $0=src[31:0]$, $1=src[15:0]$, $2=src[15:0]$, $3=src[31:16]$. Also, for MAD_MIX, the NEG_HI field acts instead as an absolute-value modifier.</td>
</tr>
<tr>
<td>33</td>
<td>V_MAD_MIXLO_F16</td>
<td>$D.f[15:0] = S0.f \cdot S1.f + S2.f$. Size and location of $S0$, $S1$ and $S2$ controlled by OPSEL: $0=src[31:0]$, $1=src[15:0]$, $2=src[15:0]$, $3=src[31:16]$. Also, for MAD_MIX, the NEG_HI field acts instead as an absolute-value modifier.</td>
</tr>
<tr>
<td>34</td>
<td>V_MAD_MIXHI_F16</td>
<td>$D.f[31:16] = S0.f \cdot S1.f + S2.f$. Size and location of $S0$, $S1$ and $S2$ controlled by OPSEL: $0=src[31:0]$, $1=src[15:0]$, $2=src[15:0]$, $3=src[31:16]$. Also, for MAD_MIX, the NEG_HI field acts instead as an absolute-value modifier.</td>
</tr>
<tr>
<td>35</td>
<td>V_DOT2_F32_F16</td>
<td>$D.f32 = S0.f16[0] \cdot S1.f16[0] + S0.f16[1] \cdot S1.f16[1] + S2.f32$</td>
</tr>
<tr>
<td>38</td>
<td>V_DOT2_I32_I16</td>
<td>$D.i32 = S0.i16[0] \cdot S1.i16[0] + S0.i16[1] \cdot S1.i16[1] + S2.i32$</td>
</tr>
<tr>
<td>39</td>
<td>V_DOT2_U32_U16</td>
<td>$D.u32 = S0.u16[0] \cdot S1.u16[0] + S0.u16[1] \cdot S1.u16[1] + S2.u32$</td>
</tr>
<tr>
<td>40</td>
<td>V_DOT4_I32_I8</td>
<td>$D.i32 = S0.i8[0] \cdot S1.i8[0] + S0.i8[1] \cdot S1.i8[1] + S0.i8[2] \cdot S1.i8[2] + S0.i8[3] \cdot S1.i8[3] + S2.i32$</td>
</tr>
<tr>
<td>41</td>
<td>V_DOT4_U32_U8</td>
<td>$D.u32 = S0.u8[0] \cdot S1.u8[0] + S0.u8[1] \cdot S1.u8[1] + S0.u8[2] \cdot S1.u8[2] + S0.u8[3] \cdot S1.u8[3] + S2.u32$</td>
</tr>
</tbody>
</table>

### 12.11. VINTERP Instructions

**VINTERP**

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>VDEST (acc)</th>
<th>OPx</th>
<th>ATTR4</th>
<th>ATTR5</th>
<th>CHANx</th>
<th>VSRCx (LJ)</th>
</tr>
</thead>
</table>

12.11. VINTERP Instructions 156 of 290
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>V_INTERP_P1_F32</td>
<td>D.f = P10 * S.f + P0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Parameter interpolation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CAUTION: when in HALF_LDS mode, D must not be the same GPR as S; if D == S then data corruption will occur.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NOTE: In textual representations the I/J VGPR is the first source and the attribute is the second source; however in the VOP3 encoding the attribute is stored in the src0 field and the VGPR is stored in the src1 field.</td>
</tr>
<tr>
<td>1</td>
<td>V_INTERP_P2_F32</td>
<td>D.f = P20 * S.f + D.f.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Parameter interpolation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NOTE: In textual representations the I/J VGPR is the first source and the attribute is the second source; however in the VOP3 encoding the attribute is stored in the src0 field and the VGPR is stored in the src1 field.</td>
</tr>
<tr>
<td>2</td>
<td>V_INTERP_MOV_F32</td>
<td>D.f = {P10,P20,P0}[S.u].</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Parameter load. Used for custom interpolation in the shader.</td>
</tr>
</tbody>
</table>

### 12.11.1. VINTERP using VOP3 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 + 0x270.

![VOP3A Encoding Diagram](image)

### 12.12. VOP3A & VOP3B Instructions

VOP3 instructions use one of two encodings:

![VOP3A Encoding Diagram](image)

![VOP3B Encoding Diagram](image)
**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.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>448</td>
<td><code>V_MAD_LEGACY_F32</code></td>
<td>D.f = S0.f * S1.f + S2.f. // DX9 rules, 0.0 * x = 0.0</td>
</tr>
<tr>
<td>449</td>
<td><code>V_MAD_F32</code></td>
<td>D.f = S0.f * S1.f + S2.f. 1ULP accuracy, denormals are flushed.</td>
</tr>
<tr>
<td>450</td>
<td><code>V_MAD_I32_U24</code></td>
<td>D.\text{i} = S0.\text{i}[23:0] * S1.\text{i}[23:0] + S2.\text{i}.</td>
</tr>
<tr>
<td>451</td>
<td><code>V_MAD_U32_U24</code></td>
<td>D.\text{u} = S0.\text{u}[23:0] * S1.\text{u}[23:0] + S2.\text{u}.</td>
</tr>
<tr>
<td>452</td>
<td><code>V_CUBEID_F32</code></td>
<td>D.f = cubemap face ID ((0.0, 1.0, \ldots, 5.0)). XYZ coordinate is given in ((S0.f, S1.f, S2.f)). Cubemap Face ID determination. Result is a floating point face ID. S0.f = x S1.f = y S2.f = z If (Abs(S2.f) &gt;= Abs(S0.f) &amp;&amp; Abs(S2.f) &gt;= Abs(S1.f)) If (S2.f &lt; 0) D.f = 5.0 Else D.f = 4.0 Else if (Abs(S1.f) &gt;= Abs(S0.f)) If (S1.f &lt; 0) D.f = 3.0 Else D.f = 2.0 Else If (S0.f &lt; 0) D.f = 1.0 Else D.f = 0.0</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>--------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| 453    | V_CUBESC_F32 | D.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.f) >= Abs(S0.f) && Abs(S2.f) >= Abs(S1.f))  
  If (S2.f < 0) D.f = -S0.f  
  Else D.f = S0.f  
Else if (Abs(S1.f) >= Abs(S0.f))  
  D.f = S0.f  
Else  
  If (S0.f < 0) D.f = S2.f  
  Else D.f = -S2.f |
| 454    | V_CUBETC_F32 | D.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.f) >= Abs(S0.f) && Abs(S2.f) >= Abs(S1.f))  
  D.f = -S1.f  
Else if (Abs(S1.f) >= Abs(S0.f))  
  If (S1.f < 0) D.f = -S2.f  
  Else D.f = S2.f  
Else  
  D.f = -S1.f |
| 455    | V_CUBEMA_F32 | D.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.f) >= Abs(S0.f) && Abs(S2.f) >= Abs(S1.f))  
  D.f = 2.0*S2.f  
Else if (Abs(S1.f) >= Abs(S0.f))  
  D.f = 2.0 * S1.f  
Else  
  D.f = 2.0 * S0.f |
| 456    | V_BFE_U32    | D.u = (S0.u >> S1.u[4:0]) & ((1 << S2.u[4:0]) - 1).  
Bitfield extract with S0 = data, S1 = field_offset, S2 = field_width. |
| 457    | V_BFE_I32    | D.i = (S0.i >> S1.i[4:0]) & ((1 << S2.i[4:0]) - 1).  
Bitfield extract with S0 = data, S1 = field_offset, S2 = field_width. |
| 458    | V_BFI_B32    | D.u = (S0.u & S1.u) | (~S0.u & S2.u).  
Bitfield insert. |
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>459</td>
<td>V_FMA_F32</td>
<td>( D.f = S0.f \times S1.f + S2.f ).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fused single precision multiply add. 0.5ULP accuracy, denormals are supported.</td>
</tr>
<tr>
<td>460</td>
<td>V_FMA_F64</td>
<td>( D.d = S0.d \times S1.d + S2.d ).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fused double precision multiply add. 0.5ULP precision, denormals are supported.</td>
</tr>
<tr>
<td>461</td>
<td>V_LERP_U8</td>
<td>( D.u = ((S0.u[31:24] + S1.u[31:24] + S2.u[24]) \gg 1) \ll 24 )  ( D.u += ((S0.u[23:16] + S1.u[23:16] + S2.u[16]) \gg 1) \ll 16 )  ( D.u += ((S0.u[15:8] + S1.u[15:8] + S2.u[8]) \gg 1) \ll 8 )  ( D.u += ((S0.u[7:0] + S1.u[7:0] + S2.u[0]) \gg 1) ).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unsigned 8-bit pixel average on packed unsigned bytes (linear interpolation). S2 acts as a round mode; if set, 0.5 rounds up, otherwise 0.5 truncates.</td>
</tr>
<tr>
<td>462</td>
<td>V_ALIGNBIT_B32</td>
<td>( D.u = ((S0,S1) \gg S2.u[4:0]) &amp; 0xffffffff. )</td>
</tr>
<tr>
<td>463</td>
<td>V_ALIGNBYTE_B32</td>
<td>( D.u = ((S0,S1) \gg (8*S2.u[4:0])) &amp; 0xffffffff. )</td>
</tr>
<tr>
<td>464</td>
<td>V_MIN3_F32</td>
<td>( D.f = V_{MIN3}(V_{MIN3}(S0.f, S1.f), S2.f). )</td>
</tr>
<tr>
<td>465</td>
<td>V_MIN3_I32</td>
<td>( D.i = V_{MIN3}(V_{MIN3}(S0.i, S1.i), S2.i). )</td>
</tr>
<tr>
<td>466</td>
<td>V_MIN3_U32</td>
<td>( D.u = V_{MIN3}(V_{MIN3}(S0.u, S1.u), S2.u). )</td>
</tr>
<tr>
<td>467</td>
<td>V_MAX3_F32</td>
<td>( D.f = V_{MAX3}(V_{MAX3}(S0.f, S1.f), S2.f). )</td>
</tr>
<tr>
<td>468</td>
<td>V_MAX3_I32</td>
<td>( D.i = V_{MAX3}(V_{MAX3}(S0.i, S1.i), S2.i). )</td>
</tr>
<tr>
<td>469</td>
<td>V_MAX3_U32</td>
<td>( D.u = V_{MAX3}(V_{MAX3}(S0.u, S1.u), S2.u). )</td>
</tr>
<tr>
<td>470</td>
<td>V_MED3_F32</td>
<td>( D.f = V_{MED3}(S0.f, S1.f, S2.f). )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if ( \text{isNan}(S0.f) \lor \text{isNan}(S1.f) \lor \text{isNan}(S2.f) ) then ( D.f = V_{MIN3}(S0.f, S1.f, S2.f); )  else if ( \text{V_MAX3}(S0.f, S1.f, S2.f) = S0.f ) then ( D.f = V_{MAX3}(S1.f, S2.f); )  else if ( \text{V_MAX3}(S0.f, S1.f, S2.f) = S1.f ) then ( D.f = V_{MAX3}(S0.f, S2.f); )  else ( D.f = V_{MAX3}(S0.f, S1.f); ) end.</td>
</tr>
<tr>
<td>471</td>
<td>V_MED3_I32</td>
<td>( D.i = V_{MED3}(S0.i, S1.i, S2.i). )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if ( \text{V_MAX3}(S0.i, S1.i, S2.i) = S0.i ) then ( D.i = V_{MAX3}(S1.i, S2.i); )  else if ( \text{V_MAX3}(S0.i, S1.i, S2.i) = S1.i ) then ( D.i = V_{MAX3}(S0.i, S2.i); )  else ( D.i = V_{MAX3}(S0.i, S1.i); ) end.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>472</td>
<td>V_MED3_U32</td>
<td>if (V_MAX3_U32(S0.u, S1.u, S2.u) == S0.u)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>else if (V_MAX3_U32(S0.u, S1.u, S2.u) == S1.u)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td></td>
<td>endif.</td>
</tr>
<tr>
<td>473</td>
<td>V_SAD_U8</td>
<td>D.u = abs(S0.i[31:24] - S1.i[31:24]);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D.u += abs(S0.i[15:8] - S1.i[15:8]);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sum of absolute differences with accumulation, overflow into upper bits is allowed.</td>
</tr>
<tr>
<td>474</td>
<td>V_SAD_HI_U8</td>
<td>D.u = (SAD_U8(S0, S1, 0) &lt;&lt; 16) + S2.u.</td>
</tr>
<tr>
<td>475</td>
<td>V_SAD_U16</td>
<td>D.u = abs(S0.i[31:16] - S1.i[31:16]) + abs(S0.i[15:0] - S1.i[15:0]) + S2.u.</td>
</tr>
<tr>
<td>476</td>
<td>V_SAD_U32</td>
<td>D.u = abs(S0.i - S1.i) + S2.u.</td>
</tr>
<tr>
<td>477</td>
<td>V_CVT_PK_U8_F32</td>
<td>D.u = (S2.u &amp; ~((0xff &lt;&lt; (0 * S1.u[1:0]))));</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>---------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>478</td>
<td>V_DIV_FIXUP_F32</td>
<td></td>
</tr>
</tbody>
</table>

    sign_out = sign(S1.f)^sign(S2.f):
    if (S2.f == NAN)
        D.f = Quiet(S2.f);
    else if (S1.f == NAN)
        D.f = Quiet(S1.f);
    else if (S1.f == S2.f == 0)
        // 0/0
        D.f = 0xffc0_0000;
    else if (abs(S1.f) == abs(S2.f) == +-INF)
        // inf/inf
        D.f = 0xffc0_0000;
    else if (S1.f == S2.f == 0)
        // x/0, or inf/y
        D.f = sign_out ? -INF : +INF;
    else if (abs(S1.f) == +-INF || S2.f == 0)
        // x/inf, 0/y
        D.f = sign_out ? -0 : 0;
    else if (((exponent(S2.f) - exponent(S1.f)) < -150)
        D.f = sign_out ? -underflow : underflow;
    else if (exponent(S1.f) == 255)
        D.f = sign_out ? -overflow : overflow;
    else
        D.f = sign_out ? -abs(S0.f) : abs(S0.f);
    endif.

Single precision division fixup. S0 = Quotient, S1 = Denominator, S2 = Numerator.

Given a numerator, denominator, and quotient from a divide, this opcode will detect and apply special case numerics, touching up the quotient if necessary. This opcode also generates invalid, denorm and divide by zero exceptions caused by the division.
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>479</td>
<td>V_DIV_FIXUP_F64</td>
<td></td>
</tr>
</tbody>
</table>

    sign_out = sign(S1.d)^sign(S2.d);
        if (S2.d == NAN)
            D.d = Quiet(S2.d);
        else if (S1.d == NAN)
            D.d = Quiet(S1.d);
        else if (S1.d == S2.d == 0)
            // 0/0
            D.d = 0xfff8_0000_0000_0000;
        else if (abs(S1.d) == abs(S2.d) == +INF)
            // inf/inf
            D.d = 0xfff8_0000_0000_0000;
        else if (S1.d == 0 || abs(S2.d) == +INF)
            // x/0, or inf/y
            D.d = sign_out ? -INF : +INF;
        else if (abs(S1.d) == +INF || S2.d == 0)
            // x/inf, 0/y
            D.d = sign_out ? -0 : 0;
        else if ((exponent(S2.d) - exponent(S1.d)) < -1075)
            D.d = sign_out ? -underflow : underflow;
        else if (exponent(S1.d) == 2047)
            D.d = sign_out ? -overflow : overflow;
        else
            D.d = sign_out ? -abs(S0.d) : abs(S0.d);
    endif.

Double precision division fixup. S0 = Quotient, S1 = Denominator, S2 = Numerator.

Given a numerator, denominator, and quotient from a divide, this opcode will detect and apply special case numerics, touching up the quotient if necessary. This opcode also generates invalid, denorm and divide by zero exceptions caused by the division.
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| 480    | V_DIV_SCALE_F32       | VCC = 0;
                      if (S2.f == 0 || S1.f == 0)
                        D.f = NAN
                      else if (exponent(S2.f) - exponent(S1.f) >= 96)
                          // N/D near MAX_FLOAT
                          VCC = 1;
                          if (S0.f == S1.f)
                              // Only scale the denominator
                              D.f = ldexp(S0.f, 64);
                          end if
                      else if (S1.f == DENORM)
                          D.f = ldexp(S0.f, 64);
                      else if (1 / S1.f == DENORM && S2.f / S1.f == DENORM)
                          VCC = 1;
                          if (S0.f == S1.f)
                              // Only scale the denominator
                              D.f = ldexp(S0.f, 64);
                          end if
                      else if (1 / S1.f == DENORM)
                          D.f = ldexp(S0.f, -64);
                      else if (S2.f / S1.f == DENORM)
                          VCC = 1;
                          if (S0.f == S2.f)
                              // Only scale the numerator
                              D.f = ldexp(S0.f, 64);
                          end if
                      else if (exponent(S2.f) <= 23)
                          // Numerator is tiny
                          D.f = ldexp(S0.f, 64);
                      end if.

Single precision division pre-scale. S0 = Input to scale (either denominator or numerator), S1 = Denominator, S2 = Numerator.

Given a numerator and denominator, this opcode will appropriately scale inputs for division to avoid subnormal terms during Newton-Raphson correction algorithm. S0 must be the same value as either S1 or S2.

This opcode producses a VCC flag for post-scaling of the quotient (using V_DIV_FMAS_F32).
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>481</td>
<td>V_DIV_SCALE_F64</td>
<td></td>
</tr>
</tbody>
</table>

| VCC = 0; |
| if (S2.d == 0 || S1.d == 0) |
| D.d = NAN |
| else if (exponent(S2.d) - exponent(S1.d) >= 768) |
| // N/D near MAX_FLOAT |
| VCC = 1; |
| if (S0.d == S1.d) |
| // Only scale the denominator |
| D.d = ldexp(S0.d, 128); |
| end if |
| else if (S1.d == DENORM) |
| D.d = ldexp(S0.d, 128); |
| else if (1 / S1.d == DENORM && S2.d / S1.d == DENORM) |
| VCC = 1; |
| if (S0.d == S1.d) |
| // Only scale the denominator |
| D.d = ldexp(S0.d, 128); |
| end if |
| else if (1 / S1.d == DENORM) |
| D.d = ldexp(S0.d, -128); |
| else if (S2.d / S1.d==DENORM) |
| VCC = 1; |
| if (S0.d == S2.d) |
| // Only scale the numerator |
| D.d = ldexp(S0.d, 128); |
| end if |
| else if (exponent(S2.d) <= 53) |
| // Numerator is tiny |
| D.d = ldexp(S0.d, 128); |
| end if. |

Double precision division pre-scale. S0 = Input to scale (either denominator or numerator), S1 = Denominator, S2 = Numerator.

Given a numerator and denominator, this opcode will appropriately scale inputs for division to avoid subnormal terms during Newton-Raphson correction algorithm. S0 must be the same value as either S1 or S2.

This opcode produces a VCC flag for post-scaling of the quotient (using V_DIV_FMAS_F64).
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| 482    | V_DIV_FMAS_F32           | if (VCC[threadId])
D.f = 2**32 * (S0.f * S1.f + S2.f);
else
D.f = S0.f * S1.f + S2.f;
end if.
Single precision FMA with fused scale.
This opcode performs a standard Fused Multiply-Add operation and will conditionally scale the resulting exponent if VCC is set.
Input denormals are not flushed, but output flushing is allowed. |
| 483    | V_DIV_FMAS_F64           | if (VCC[threadId])
D.d = 2**64 * (S0.d * S1.d + S2.d);
else
D.d = S0.d * S1.d + S2.d;
end if.
Double precision FMA with fused scale.
This opcode performs a standard Fused Multiply-Add operation and will conditionally scale the resulting exponent if VCC is set.
Input denormals are not flushed, but output flushing is allowed. |
| 484    | V_MSAD_U8                | D.u = Masked Byte SAD with accum_lo(S0.u, S1.u, S2.u).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| 485    | V_QSAD_PK_U16_U8         | D.u = Quad-Byte SAD with 16-bit packed accum_lo/hi(S0.u[63:0], S1.u[31:0], S2.u[63:0])                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 486    | V_MQSAD_PK_U16_U8        | D.u = Masked Quad-Byte SAD with 16-bit packed accum_lo/hi(S0.u[63:0], S1.u[31:0], S2.u[63:0])                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 487    | V_MQSAD_U32_U8           | D.u128 = Masked Quad-Byte SAD with 32-bit accum_lo/hi(S0.u[63:0], S1.u[31:0], S2.u[127:0])                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| 488    | V_MAD_U64_U32            | {vcc_out,D.u64} = S0.u32 * S1.u32 + S2.u64.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| 489    | V_MAD_I64_I32            | {vcc_out,D.i64} = S0.i32 * S1.i32 + S2.i64.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| 490    | V_MAD_LEGACY_F16         | D.f16 = S0.f16 * S1.f16 + S2.f16.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|        |                          | Supports round mode, exception flags, saturation. If op_sel[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 op_sel[3] is 1 Result is written to 16 MSBs of destination VGPR and lo 16 bits are preserved. |
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>491</td>
<td>V_MAD_LEGACY_U16</td>
<td>D.u16 = S0.u16 * S1.u16 + S2.u16.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Supports saturation (unsigned 16-bit integer domain).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If op_sel[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).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If op_sel[3] is 1 Result is written to 16 MSBs of destination VGPR and lo 16 bits are preserved.</td>
</tr>
<tr>
<td>492</td>
<td>V_MAD_LEGACY_I16</td>
<td>D.i16 = S0.i16 * S1.i16 + S2.i16.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Supports saturation (signed 16-bit integer domain).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If op_sel[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).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If op_sel[3] is 1 Result is written to 16 MSBs of destination VGPR and lo 16 bits are preserved.</td>
</tr>
<tr>
<td>493</td>
<td>V_PERM_B32</td>
<td>D.u[31:24] = byte_permute({S0.u, S1.u}, S2.u[31:24]);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D.u[23:16] = byte_permute({S0.u, S1.u}, S2.u[23:16]);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D.u[15:8] = byte_permute({S0.u, S1.u}, S2.u[15:8]);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D.u[7:0] = byte_permute({S0.u, S1.u}, S2.u[7:0]);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>byte permute(byte in[8], byte sel) {</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if(sel&gt;=13) then return 0xff;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>elsif(sel==12) then return 0x00;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>elsif(sel==11) then return in[7][7] * 0xff;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>elsif(sel==10) then return in[5][7] * 0xff;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>elsif(sel==9) then return in[3][7] * 0xff;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>elsif(sel==8) then return in[1][7] * 0xff;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>else return in[sel];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Byte permute.</td>
</tr>
<tr>
<td>494</td>
<td>V_FMA_LEGACY_F16</td>
<td>D.f16 = S0.f16 * S1.f16 + S2.f16.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fused half precision multiply add.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| 495    | V_DIV_FIXUP_LEGACY_F16| \[
|        | CY_F16                | sign_out = sign(S1.f16)*sign(S2.f16);                                    |
|        |                       | if (S2.f16 == NAN)                                                         |
|        |                       | D.f16 = Quiet(S2.f16);                                                    |
|        |                       | else if (S1.f16 == NAN)                                                   |
|        |                       | D.f16 = Quiet(S1.f16);                                                    |
|        |                       | else if (S1.f16 == S2.f16 == 0)                                           |
|        |                       | // 0/0                                                                     |
|        |                       | D.f16 = 0xfe00;                                                           |
|        |                       | else if (abs(S1.f16) == abs(S2.f16) == +INF)                              |
|        |                       | // inf/inf                                                                |
|        |                       | D.f16 = 0xfe00;                                                           |
|        |                       | else if (S1.f16 == 0 || abs(S2.f16) == +INF)                              |
|        |                       | // x/0, or inf/y                                                          |
|        |                       | D.f16 = sign_out ? -INF : +INF;                                            |
|        |                       | else if (abs(S1.f16) == +INF || S2.f16 == 0)                              |
|        |                       | // x/inf, 0/y                                                             |
|        |                       | D.f16 = sign_out ? -0 : 0;                                                 |
|        |                       | else                                                                     |
|        |                       | D.f16 = sign_out ? -abs(S0.f16) : abs(S0.f16);                            |
|        |                       | end if.                                                                   |
|        |                       | Half precision division fixup. S0 = Quotient, S1 = Denominator,            |
|        |                       | S2 = Numerator.                                                           |
|        |                       | Given a numerator, denominator, and quotient from a divide, this          |
|        |                       | opcode will detect and apply special case numerics, touching up           |
|        |                       | the quotient if necessary. This opcode also generates invalid,           |
|        |                       | denorm and divide by zero exceptions caused by the division.              |
| 496    | V_CVT_PKACCUM_U8_F32  | \[
<p>|        |                       | byte = S1.u[1:0];                                                        |
|        |                       | bit = byte * 8;                                                          |
|        |                       | D.u[bit+7:bit] = flt32_to_uint8(S0.f).                                   |
|        |                       | Pack converted value of S0.f into byte S1 of the destination.             |
|        |                       | Note: this opcode uses src_c to pass destination in as a source.         |
| 497    | V_MAD_U32_U16         | D.u32 = S0.u16 * S1.u16 + S2.u32.                                        |
| 498    | V_MAD_I32_I16         | D.i32 = S0.i16 * S1.i16 + S2.i32.                                        |
| 499    | V_XAD_U32             | D.u32 = (S0.u32 ^ S1.u32) + S2.u32.                                      |
|        |                       | No carryin/carryout and no saturation. This opcode exists to              |
|        |                       | accelerate the SHA256 hash algorithm.                                     |
| 500    | V_MIN3_F16            | D.f16 = V_MIN_F16(V_MIN_F16(S0.f16, S1.f16), S2.f16).                     |
| 501    | V_MIN3_I16            | D.i16 = V_MIN_I16(V_MIN_I16(S0.i16, S1.i16), S2.i16).                     |
| 502    | V_MIN3_U16            | D.u16 = V_MIN_U16(V_MIN_U16(S0.u16, S1.u16), S2.u16).                     |
| 503    | V_MAX3_F16            | D.f16 = V_MAX_F16(V_MAX_F16(S0.f16, S1.f16), S2.f16).                     |
| 504    | V_MAX3_I16            | D.i16 = V_MAX_I16(V_MAX_I16(S0.i16, S1.i16), S2.i16).                     |</p>
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>505</td>
<td>V_MAX3_U16</td>
<td>D.u16 = V_MAX_U16(V_MAX_U16(S0.u16, S1.u16), S2.u16).</td>
</tr>
<tr>
<td>506</td>
<td>V_MED3_F16</td>
<td>if (isNan(S0.f16)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D.f16 = V_MIN3_F16(S0.f16, S1.f16, S2.f16);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>else if (V_MAX3_F16(S0.f16, S1.f16, S2.f16) == S0.f16)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D.f16 = V_MAX_F16(S1.f16, S2.f16);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>else if (V_MAX3_F16(S0.f16, S1.f16, S2.f16) == S1.f16)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D.f16 = V_MAX_F16(S0.f16, S2.f16);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>else D.f16 = V_MAX_F16(S0.f16, S1.f16);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>endif.</td>
</tr>
<tr>
<td>507</td>
<td>V_MED3_I16</td>
<td>if (V_MAX3_I16(S0.i16, S1.i16, S2.i16) == S0.i16)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D.i16 = V_MAX_I16(S1.i16, S2.i16);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>else if (V_MAX3_I16(S0.i16, S1.i16, S2.i16) == S1.i16)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D.i16 = V_MAX_I16(S0.i16, S2.i16);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>else D.i16 = V_MAX_I16(S0.i16, S1.i16);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>endif.</td>
</tr>
<tr>
<td>508</td>
<td>V_MED3_U16</td>
<td>if (V_MAX3_U16(S0.u16, S1.u16, S2.u16) == S0.u16)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D.u16 = V_MAX_U16(S1.u16, S2.u16);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>else if (V_MAX3_U16(S0.u16, S1.u16, S2.u16) == S1.u16)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D.u16 = V_MAX_U16(S0.u16, S2.u16);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>else D.u16 = V_MAX_U16(S0.u16, S1.u16);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>endif.</td>
</tr>
<tr>
<td>509</td>
<td>V_LSHL_ADD_U32</td>
<td>D.u = (S0.u &lt;&lt; S1.u[4:0]) + S2.u.</td>
</tr>
<tr>
<td>510</td>
<td>V_ADD_LSHL_U32</td>
<td>D.u = (S0.u + S1.u) &lt;&lt; S2.u[4:0].</td>
</tr>
<tr>
<td>511</td>
<td>V_ADD3_U32</td>
<td>D.u = S0.u + S1.u + S2.u.</td>
</tr>
<tr>
<td>512</td>
<td>V_LSHL_OR_B32</td>
<td>D.u = (S0.u &lt;&lt; S1.u[4:0])</td>
</tr>
<tr>
<td>513</td>
<td>V_AND_OR_B32</td>
<td>D.u = (S0.u &amp; S1.u)</td>
</tr>
<tr>
<td>514</td>
<td>V_OR3_B32</td>
<td>D.u = S0.u</td>
</tr>
<tr>
<td>515</td>
<td>V_MAD_F16</td>
<td>D.f16 = S0.f16 * S1.f16 + S2.f16.</td>
</tr>
</tbody>
</table>

Supports round mode, exception flags, saturation. 1ULP accuracy, denormals are flushed.

If op_sel[3] is 0 Result is written to 16 LSBs of destination VGPR and hi 16 bits are preserved.
If op_sel[3] is 1 Result is written to 16 MSBs of destination VGPR and lo 16 bits are preserved.
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| 516    | V_MAD_U16  | $D.u16 = S0.u16 \times S1.u16 + S2.u16$.  
Supports saturation (unsigned 16-bit integer domain).  
If op_sel[3] is 0 Result is written to 16 LSBs of destination VGPR and hi 16 bits are preserved.  
If op_sel[3] is 1 Result is written to 16 MSBs of destination VGPR and lo 16 bits are preserved. |
| 517    | V_MAD_I16  | $D.i16 = S0.i16 \times S1.i16 + S2.i16$.  
Supports saturation (signed 16-bit integer domain).  
If op_sel[3] is 0 Result is written to 16 LSBs of destination VGPR and hi 16 bits are preserved.  
If op_sel[3] is 1 Result is written to 16 MSBs of destination VGPR and lo 16 bits are preserved. |
| 518    | V_FMA_F16  | $D.f16 = S0.f16 \times S1.f16 + S2.f16$.  
Fused half precision multiply add. 0.5ULP accuracy, denormals are supported.  
If op_sel[3] is 0 Result is written to 16 LSBs of destination VGPR and hi 16 bits are preserved.  
If op_sel[3] is 1 Result is written to 16 MSBs of destination VGPR and lo 16 bits are preserved. |
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>519</td>
<td>V_DIV_FIXUP_F16</td>
<td>sign_out = sign(S1.f16)*sign(S2.f16); if (S2.f16 == NAN) D.f16 = Quiet(S2.f16); else if (S1.f16 == NAN) D.f16 = Quiet(S1.f16); else if (S1.f16 == S2.f16 == 0) // 0/0 D.f16 = 0xfe00; else if (abs(S1.f16) == abs(S2.f16) == +-INF) // inf/inf D.f16 = 0xfe00; else if (S1.f16 ==0</td>
</tr>
</tbody>
</table>

Half precision division fixup. S0 = Quotient, S1 = Denominator, S2 = Numerator.

Given a numerator, denominator, and quotient from a divide, this opcode will detect and apply special case numerics, touching up the quotient if necessary. This opcode also generates invalid, denorm and divide by zero exceptions caused by the division.

If op_sel[3] is 0 Result is written to 16 LSBs of destination VGPR and hi 16 bits are preserved.
If op_sel[3] is 1 Result is written to 16 MSBs of destination VGPR and lo 16 bits are preserved.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>628</td>
<td>V_INTERP_P1LL_F16</td>
<td>D.f32 = P10.f16 * S0.f32 + P0.f16.</td>
</tr>
</tbody>
</table>

'LL' stands for 'two LDS arguments'. attr_word selects the high or low half 16 bits of each LDS dword accessed. This opcode is available for 32-bank LDS only.

NOTE: In textual representations the I/J VGPR is the first source and the attribute is the second source; however in the VOP3 encoding the attribute is stored in the src0 field and the VGPR is stored in the src1 field.
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>629</td>
<td>V_INTERP_P1LV_F16</td>
<td>[D.f32 = \text{P10.f16} * \text{S0.f32} + (\text{S2.u32 &gt;&gt; (attr_word * 16))}.f16.] 'LV' stands for 'One LDS and one VGPR argument'. S2 holds two parameters, attr_word selects the high or low word of the VGPR for this calculation, as well as the high or low half of the LDS data. Meant for use with 16-bank LDS. NOTE: In textual representations the I/J VGPR is the first source and the attribute is the second source; however in the VOP3 encoding the attribute is stored in the src0 field and the VGPR is stored in the src1 field.</td>
</tr>
<tr>
<td>630</td>
<td>V_INTERP_P2_LEGA CY_F16</td>
<td>[D.f16 = \text{P20.f16} * \text{S0.f32} + \text{S2.f32}..] Final computation. attr_word selects LDS high or low 16bits. Used for both 16- and 32-bank LDS. Result is written to the 16 LSBs of the destination VGPR. NOTE: In textual representations the I/J VGPR is the first source and the attribute is the second source; however in the VOP3 encoding the attribute is stored in the src0 field and the VGPR is stored in the src1 field.</td>
</tr>
<tr>
<td>631</td>
<td>V_INTERP_P2_F16</td>
<td>[D.f16 = \text{P20.f16} * \text{S0.f32} + \text{S2.f32}..] Final computation. attr_word selects LDS high or low 16bits. Used for both 16- and 32-bank LDS. NOTE: In textual representations the I/J VGPR is the first source and the attribute is the second source; however in the VOP3 encoding the attribute is stored in the src0 field and the VGPR is stored in the src1 field. If op_sel[3] is 0 Result is written to 16 LSBs of destination VGPR and hi 16 bits are preserved. If op_sel[3] is 1 Result is written to 16 MSBs of destination VGPR and lo 16 bits are preserved.</td>
</tr>
<tr>
<td>640</td>
<td>V_ADD_F64</td>
<td>[D.d = \text{S0.d} + \text{S1.d}..] 0.5ULP precision, denormals are supported.</td>
</tr>
<tr>
<td>641</td>
<td>V_MUL_F64</td>
<td>[D.d = \text{S0.d} * \text{S1.d}..] 0.5ULP precision, denormals are supported.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| 642    | V_MIN_F64       | if (IEEE_MODE && S0.d == sNaN)  
D.d = Quiet(S0.d);  
else if (IEEE_MODE && S1.d == sNaN)  
D.d = Quiet(S1.d);  
else if (S0.d == NaN)  
D.d = S1.d;  
else if (S1.d == NaN)  
D.d = S0.d;  
else if (S0.d == +0.0 && S1.d == -0.0)  
D.d = S1.d;  
else if (S0.d == -0.0 && S1.d == +0.0)  
D.d = S0.d;  
else  
// Note: there's no IEEE special case here like there is for V_MAX_F64.  
D.d = (S0.d < S1.d ? S0.d : S1.d);  
endif.                                                   |
| 643    | V_MAX_F64       | if (IEEE_MODE && S0.d == sNaN)  
D.d = Quiet(S0.d);  
else if (IEEE_MODE && S1.d == sNaN)  
D.d = Quiet(S1.d);  
else if (S0.d == NaN)  
D.d = S1.d;  
else if (S1.d == NaN)  
D.d = S0.d;  
else if (S0.d == +0.0 && S1.d == -0.0)  
D.d = S1.d;  
else if (S0.d == -0.0 && S1.d == +0.0)  
D.d = S0.d;  
else if (IEEE_MODE)  
D.d = (S0.d >= S1.d ? S0.d : S1.d);  
else  
D.d = (S0.d > S1.d ? S0.d : S1.d);  
endif.                                                   |
| 644    | V_LDEXP_F64     | D.d = S0.d * (2 ** S1.i).                                                                                                                  |
| 645    | V_MUL_LO_U32    | D.u = S0.u * S1.u.                                                                                                                          |
| 646    | V_MUL_HI_U32    | D.u = (S0.u * S1.u) >> 32.                                                                                                                  |
| 647    | V_MUL_HI_I32    | D.i = (S0.i * S1.i) >> 32.                                                                                                                  |
| 648    | V_LDEXP_F32     | D.f = S0.f * (2 ** S1.i).                                                                                                                  |
| 649    | V_READLANE_B32  | Copy one VGPR value to one SGPR. D = SGPR-dest, S0 = Source Data (VGPR# or M0(lds-direct)), S1 = Lane Select (SGPR or M0). Ignores exec mask.  
Input and output modifiers not supported; this is an untyped operation. |
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>650</td>
<td>V_WRITELANE_B32</td>
<td>Write value into one VGPR in one lane. D = VGPR-dest, S0 = Source Data (sgpr, m0, exec or constants), S1 = Lane Select (SGPR or M0). Ignores exec mask. Input and output modifiers not supported; this is an untyped operation.</td>
</tr>
<tr>
<td>651</td>
<td>V_BCNT_U32_B32</td>
<td>D.u = 0; for i in 0 ... 31 do D.u += (S0.u[i] == 1 ? 1 : 0); endfor. Bit count.</td>
</tr>
<tr>
<td>652</td>
<td>V_MBCNT_LO_U32_B32</td>
<td>ThreadMask = (1LL &lt;&lt; ThreadPosition) - 1; MaskedValue = (S0.u &amp; ThreadMask[31:0]); D.u = S1.u; for i in 0 ... 31 do D.u += (MaskedValue[i] == 1 ? 1 : 0); endfor. Masked bit count, ThreadPosition is the position of this thread in the wavefront (in 0..63). See also V_MBCNT_HI_U32_B32.</td>
</tr>
<tr>
<td>653</td>
<td>V_MBCNT_HI_U32_B32</td>
<td>ThreadMask = (1LL &lt;&lt; ThreadPosition) - 1; MaskedValue = (S0.u &amp; ThreadMask[63:32]); D.u = S1.u; for i in 0 ... 31 do D.u += (MaskedValue[i] == 1 ? 1 : 0); endfor. Masked bit count, ThreadPosition is the position of this thread in the wavefront (in 0..63). See also V_MBCNT_LO_U32_B32.</td>
</tr>
<tr>
<td>655</td>
<td>V_LSHLREV_B64</td>
<td>D.u64 = S1.u64 &lt;&lt; S0.u[5:0].</td>
</tr>
<tr>
<td>656</td>
<td>V_LSHRREV_B64</td>
<td>D.u64 = S1.u64 &gt;&gt; S0.u[5:0].</td>
</tr>
<tr>
<td>657</td>
<td>V_ASHRREV_I64</td>
<td>D.u64 = signext(S1.u64) &gt;&gt; S0.u[5:0].</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| 658    | V_TRIG_PREOP_F64              | shift = S1.u * 53; if exponent(S0.d) > 1077 then shift += exponent(S0.d) - 1077; endif
result = (double) ((2/PI[1200:0] << shift) & 0x1ffffff_fffffff);
scale = (-53 - shift);
if exponent(S0.d) >= 1968 then scale += 128; endif
D.d = ldexp(result, scale).
Look Up 2/PI (S0.d) with segment select S1.u[4:0]. This operation returns an aligned, double precision segment of 2/PI needed to do range reduction on S0.d (double-precision value). Multiple segments can be specified through S1.u[4:0]. Rounding uses round-to-zero. Large inputs (exp > 1968) are scaled to avoid loss of precision through denormalization. |
| 659    | V_BFM_B32                     | D.u = ((1<<S0.u[4:0])-1) << S1.u[4:0]. Bitfield modify. S0 is the bitfield width and S1 is the bitfield offset.                                                                                                                                                                                                                             |
| 660    | V_CVT_PKNORM_I16_F32          | D = {(snorm)S1.f, (snorm)S0.f}.                                                                                                                                                                                                                                                                                                           |
| 661    | V_CVT_PKNORM_U16_F32          | D = {(unorm)S1.f, (unorm)S0.f}.                                                                                                                                                                                                                                                                                                           |
| 662    | V_CVT_PKRTZ_F16_F32           | D = {flt32_to_flt16(S1.f),flt32_to_flt16(S0.f)}. // Round-toward-zero regardless of current round mode setting in hardware.                                                                                                                                                                                                                       |
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.                                                                                                               |
<p>| 663    | V_CVT_PK_U16_U32              | D = {uint32_to_uint16(S1.u), uint32_to_uint16(S0.u)}.                                                                                                                                                                                                                                                                                      |
| 664    | V_CVT_PK_I16_I32              | D = {int32_to_int16(S1.i), int32_to_int16(S0.i)}.                                                                                                                                                                                                                                                                                           |
| 665    | V_CVT_PKNORM_I16_F16          | D = {(snorm)S1.f16, (snorm)S0.f16}.                                                                                                                                                                                                                                                                                                        |
| 666    | V_CVT_PKNORM_U16_F16          | D = {(unorm)S1.f16, (unorm)S0.f16}.                                                                                                                                                                                                                                                                                                        |
| 668    | V_ADD_I32                     | D.i = S0.i + S1.i. Supports saturation (signed 32-bit integer domain).                                                                                                                                                                                                                                                                     |
| 669    | V_SUB_I32                     | D.i = S0.i - S1.i. Supports saturation (signed 32-bit integer domain).                                                                                                                                                                                                                                                                     |</p>
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>670</td>
<td>V_ADD_I16</td>
<td>D.i16 = S0.i16 + S1.i16. Supports saturation (signed 16-bit integer domain).</td>
</tr>
<tr>
<td>671</td>
<td>V_SUB_I16</td>
<td>D.i16 = S0.i16 - S1.i16. Supports saturation (signed 16-bit integer domain).</td>
</tr>
<tr>
<td>672</td>
<td>V_PACK_B32_F16</td>
<td>D[31:16].f16 = S1.f16; D[15:0].f16 = S0.f16.</td>
</tr>
</tbody>
</table>

### 12.13. LDS & GDS Instructions

This suite of instructions operates on data stored within the data share memory. The instructions transfer data between VGPRs and data share memory.

The bitfield map for the LDS/GDS is:

```
<table>
<thead>
<tr>
<th>LDS, GDS</th>
<th></th>
<th>OPs</th>
<th>GDS</th>
<th>OFFSET1b</th>
<th>OFFSET0b</th>
</tr>
</thead>
<tbody>
<tr>
<td>VDSTb</td>
<td>DATA1b</td>
<td>DATA0b</td>
<td>ADDR0b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

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 GDS, cleared if LDS.
- OP = DS instructions.
- ADDR = Source LDS address VGPR 0 - 255.
- DATA0 = Source data0 VGPR 0 - 255.
- DATA1 = Source data1 VGPR 0 - 255.
- VDST = Destination VGPR 0- 255.

All instructions with RTN in the name return the value that was in memory before the operation was performed.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| 0      | DS_ADD_U32        | // 32bit
        |                   | tmp = MEM[ADDR];
        |                   | MEM[ADDR] += DATA;
        |                   | RETURN_DATA = tmp.                                                         |
| 1      | DS_SUB_U32        | // 32bit
        |                   | tmp = MEM[ADDR];
        |                   | MEM[ADDR] -= DATA;
<pre><code>    |                   | RETURN_DATA = tmp.                                                         |
</code></pre>
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>DS_RSUB_U32</td>
<td>// 32bit&lt;br&gt;tmp = MEM[ADDR];&lt;br&gt;MEM[ADDR] = DATA - MEM[ADDR];&lt;br&gt;RETURN_DATA = tmp.&lt;br&gt;Subtraction with reversed operands.</td>
</tr>
<tr>
<td>3</td>
<td>DS_INC_U32</td>
<td>// 32bit&lt;br&gt;tmp = MEM[ADDR];&lt;br&gt;MEM[ADDR] = (tmp &gt;= DATA) ? 0 : tmp + 1; // unsigned compare&lt;br&gt;RETURN_DATA = tmp.</td>
</tr>
<tr>
<td>4</td>
<td>DS_DEC_U32</td>
<td>// 32bit&lt;br&gt;tmp = MEM[ADDR];&lt;br&gt;MEM[ADDR] = (tmp == 0</td>
</tr>
<tr>
<td>5</td>
<td>DS_MIN_I32</td>
<td>// 32bit&lt;br&gt;tmp = MEM[ADDR];&lt;br&gt;MEM[ADDR] = (DATA &lt; tmp) ? DATA : tmp; // signed compare&lt;br&gt;RETURN_DATA = tmp.</td>
</tr>
<tr>
<td>6</td>
<td>DS_MAX_I32</td>
<td>// 32bit&lt;br&gt;tmp = MEM[ADDR];&lt;br&gt;MEM[ADDR] = (DATA &gt; tmp) ? DATA : tmp; // signed compare&lt;br&gt;RETURN_DATA = tmp.</td>
</tr>
<tr>
<td>7</td>
<td>DS_MIN_U32</td>
<td>// 32bit&lt;br&gt;tmp = MEM[ADDR];&lt;br&gt;MEM[ADDR] = (DATA &lt; tmp) ? DATA : tmp; // unsigned compare&lt;br&gt;RETURN_DATA = tmp.</td>
</tr>
<tr>
<td>8</td>
<td>DS_MAX_U32</td>
<td>// 32bit&lt;br&gt;tmp = MEM[ADDR];&lt;br&gt;MEM[ADDR] = (DATA &gt; tmp) ? DATA : tmp; // unsigned compare&lt;br&gt;RETURN_DATA = tmp.</td>
</tr>
<tr>
<td>9</td>
<td>DS_AND_B32</td>
<td>// 32bit&lt;br&gt;tmp = MEM[ADDR];&lt;br&gt;MEM[ADDR] &amp;= DATA;&lt;br&gt;RETURN_DATA = tmp.</td>
</tr>
<tr>
<td>10</td>
<td>DS_OR_B32</td>
<td>// 32bit&lt;br&gt;tmp = MEM[ADDR];&lt;br&gt;MEM[ADDR]</td>
</tr>
<tr>
<td>11</td>
<td>DS_XOR_B32</td>
<td>// 32bit&lt;br&gt;tmp = MEM[ADDR];&lt;br&gt;MEM[ADDR] ^= DATA;&lt;br&gt;RETURN_DATA = tmp.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| 12     | DS_MSKOR_B32                | // 32bit
|        | tmp = MEM[ADDR];            |             |
|        | MEM[ADDR] = (MEM[ADDR] & ~DATA) | DATA2;     |
|        | RETURN_DATA = tmp.          |             |
|        | Masked dword OR, D0 contains the mask and D1 contains the new value. |
| 13     | DS_WRITE_B32                | // 32bit
|        | MEM[ADDR] = DATA.           |             |
|        | Write dword.                |             |
| 14     | DS_WRITE2_B32               | // 32bit
|        | MEM[ADDR_BASE + OFFSET0 * 4] = DATA; |
|        | Write 2 dwords.             |             |
| 15     | DS_WRITE2ST64_B32           | // 32bit
|        | MEM[ADDR_BASE + OFFSET0 * 4 * 64] = DATA; |
|        | MEM[ADDR_BASE + OFFSET1 * 4 * 64] = DATA2. |
|        | Write 2 dwords.             |             |
| 16     | DS_CMPST_B32                | // 32bit
|        | tmp = MEM[ADDR];            |             |
|        | src = DATA2;                |             |
|        | cmp = DATA;                 |             |
|        | MEM[ADDR] = (tmp == cmp) ? src : tmp; |
|        | RETURN_DATA[0] = tmp.       |             |
|        | Compare and store. Caution, the order of src and cmp are the opposite* of the BUFFER_ATOMIC_CMPSWAP opcode. |
| 17     | DS_CMPST_F32                | // 32bit
|        | tmp = MEM[ADDR];            |             |
|        | src = DATA2;                |             |
|        | cmp = DATA;                 |             |
|        | MEM[ADDR] = (tmp == cmp) ? src : tmp; |
|        | RETURN_DATA[0] = tmp.       |             |
|        | Floating point compare and store that handles NaN/INF/denormal values. |
| 18     | DS_MIN_F32                  | // 32bit
<p>|        | tmp = MEM[ADDR];            |             |
|        | src = DATA;                 |             |
|        | cmp = DATA2;                |             |
|        | Floating point minimum that handles NaN/INF/denormal values. |</p>
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| 19     | DS_MAX_F32         | // 32bit<br>
tmp = MEM[ADDR];<br>src = DATA;<br>cmp = DATA2;<br>MEM[ADDR] = (tmp > cmp) ? src : tmp.<br>Floating point maximum that handles NaN/INF/denormal values. |
| 20     | DS_NOP             | Do nothing.                                                                 |
| 21     | DS_ADD_F32         | // 32bit<br>
tmp = MEM[ADDR];<br>MEM[ADDR] += DATA;<br>RETURN_DATA = tmp.<br>Floating point add that handles NaN/INF/denormal values. |
<p>| 29     | DS_WRITE_ADDTID_B32| // 32bit&lt;br&gt;MEM[ADDR_BASE + OFFSET + M0.OFFSET + TID*4] = DATA.&lt;br&gt;Write dword.               |
| 30     | DS_WRITE_B8        | MEM[ADDR] = DATA[7:0].&lt;br&gt;Byte write.                                       |
| 31     | DS_WRITE_B16       | MEM[ADDR] = DATA[15:0].&lt;br&gt;Short write.                                    |
| 32     | DS_ADD_RTN_U32     | // 32bit&lt;br&gt;tmp = MEM[ADDR];&lt;br&gt;MEM[ADDR] += DATA;&lt;br&gt;RETURN_DATA = tmp.     |
| 33     | DS_SUB_RTN_U32     | // 32bit&lt;br&gt;tmp = MEM[ADDR];&lt;br&gt;MEM[ADDR] -= DATA;&lt;br&gt;RETURN_DATA = tmp.     |
| 34     | DS_RSUB_RTN_U32    | // 32bit&lt;br&gt;tmp = MEM[ADDR];&lt;br&gt;MEM[ADDR] = DATA - MEM[ADDR];&lt;br&gt;RETURN_DATA = tmp.&lt;br&gt;Subtraction with reversed operands. |
| 35     | DS_INC_RTN_U32     | // 32bit&lt;br&gt;tmp = MEM[ADDR];&lt;br&gt;MEM[ADDR] = (tmp &gt;= DATA) ? 0 : tmp + 1; // unsigned compare&lt;br&gt;RETURN_DATA = tmp.     |</p>
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| 36     | DS_DEC_RTN_U32     | // 32bit
<pre><code>    |                    | tmp = MEM[ADDR]; |
    |                    | MEM[ADDR] = (tmp == 0 || tmp &gt; DATA) ? DATA : tmp - 1; // unsigned compare |
    |                    | RETURN_DATA = tmp. |
</code></pre>
<p>| 37     | DS_MIN_RTN_I32     | // 32bit |
|                    | tmp = MEM[ADDR]; |
|                    | MEM[ADDR] = (DATA &lt; tmp) ? DATA : tmp; // signed compare |
|                    | RETURN_DATA = tmp. |
| 38     | DS_MAX_RTN_I32     | // 32bit |
|                    | tmp = MEM[ADDR]; |
|                    | MEM[ADDR] = (DATA &gt; tmp) ? DATA : tmp; // signed compare |
|                    | RETURN_DATA = tmp. |
| 39     | DS_MIN_RTN_U32     | // 32bit |
|                    | tmp = MEM[ADDR]; |
|                    | MEM[ADDR] = (DATA &lt; tmp) ? DATA : tmp; // unsigned compare |
|                    | RETURN_DATA = tmp. |
| 40     | DS_MAX_RTN_U32     | // 32bit |
|                    | tmp = MEM[ADDR]; |
|                    | MEM[ADDR] = (DATA &gt; tmp) ? DATA : tmp; // unsigned compare |
|                    | RETURN_DATA = tmp. |
| 41     | DS_AND_RTN_B32     | // 32bit |
|                    | tmp = MEM[ADDR]; |
|                    | MEM[ADDR] &amp;= DATA; |
|                    | RETURN_DATA = tmp. |
| 42     | DS_OR_RTN_B32      | // 32bit |
|                    | tmp = MEM[ADDR]; |
|                    | MEM[ADDR] |= DATA; |
|                    | RETURN_DATA = tmp. |
| 43     | DS_XOR_RTN_B32     | // 32bit |
|                    | tmp = MEM[ADDR]; |
|                    | MEM[ADDR] ^= DATA; |
|                    | RETURN_DATA = tmp. |
| 44     | DS_MSKOR_RTN_B32   | // 32bit |
|                    | tmp = MEM[ADDR]; |
|                    | MEM[ADDR] = (MEM[ADDR] &amp; ~DATA) | DATA2; |
|                    | RETURN_DATA = tmp. |
|                    | Masked dword OR, D0 contains the mask and D1 contains the new value. |
| 45     | DS_WRXCHG_RTN_B32  | tmp = MEM[ADDR]; |
|                    | MEM[ADDR] = DATA; |
|                    | RETURN_DATA = tmp. |
|                    | Write-exchange operation. |</p>
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>46</td>
<td>DS_WRXCHG2_RTN_B32</td>
<td>Write-exchange 2 separate dwords.</td>
</tr>
<tr>
<td>47</td>
<td>DS_WRXCHG2ST64_RTN_B32</td>
<td>Write-exchange 2 separate dwords with a stride of 64 dwords.</td>
</tr>
</tbody>
</table>
| 48     | DS_CMPST_RTN_B32 | // 32bit  
|        |      | tmp = MEM[ADDR];  
|        |      | src = DATA2;  
|        |      | cmp = DATA;  
|        |      | MEM[ADDR] = (tmp == cmp) ? src : tmp;  
|        |      | RETURN_DATA[0] = tmp.  
|        |      | Compare and store. Caution, the order of src and cmp are the *opposite* of the BUFFER_ATOMIC_CMPSWAP opcode. |
| 49     | DS_CMPST_RTN_F32 | // 32bit  
|        |      | tmp = MEM[ADDR];  
|        |      | src = DATA2;  
|        |      | cmp = DATA;  
|        |      | MEM[ADDR] = (tmp == cmp) ? src : tmp;  
|        |      | RETURN_DATA[0] = tmp.  
|        |      | Floating point compare and store that handles NaN/INF/denormal values. |
| 50     | DS_MIN_RTN_F32 | // 32bit  
|        |      | tmp = MEM[ADDR];  
|        |      | src = DATA;  
|        |      | cmp = DATA2;  
|        |      | Floating point minimum that handles NaN/INF/denormal values. |
| 51     | DS_MAX_RTN_F32 | // 32bit  
|        |      | tmp = MEM[ADDR];  
|        |      | src = DATA;  
|        |      | cmp = DATA2;  
|        |      | Floating point maximum that handles NaN/INF/denormal values. |
| 52     | DS_WRAP_RTN_B32 | tmp = MEM[ADDR];  
|        |      | MEM[ADDR] = (tmp >= DATA) ? tmp - DATA : tmp + DATA2;  
|        |      | RETURN_DATA = tmp.  
| 53     | DS_ADD_RTN_F32 | // 32bit  
|        |      | tmp = MEM[ADDR];  
|        |      | MEM[ADDR] += DATA;  
|        |      | RETURN_DATA = tmp.  
<p>|        |      | Floating point add that handles NaN/INF/denormal values. |</p>
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>DS_READ_B32</td>
<td>RETURN_DATA = MEM[ADDR].</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dword read.</td>
</tr>
<tr>
<td>55</td>
<td>DS_READ2_B32</td>
<td>RETURN_DATA[0] = MEM[ADDR_BASE + OFFSET0 * 4]; RETURN_DATA[1] = MEM[ADDR_BASE + OFFSET1 * 4].</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read 2 dwords.</td>
</tr>
<tr>
<td>56</td>
<td>DS_READ2ST64_B32</td>
<td>RETURN_DATA[0] = MEM[ADDR_BASE + OFFSET0 * 4 * 64]; RETURN_DATA[1] = MEM[ADDR_BASE + OFFSET1 * 4 * 64].</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read 2 dwords.</td>
</tr>
<tr>
<td>57</td>
<td>DS_READ_I8</td>
<td>RETURN_DATA = signext(MEM[ADDR][7:0]).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Signed byte read.</td>
</tr>
<tr>
<td>58</td>
<td>DS_READ_U8</td>
<td>RETURN_DATA = {24'h0, MEM[ADDR][7:0]}.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unsigned byte read.</td>
</tr>
<tr>
<td>59</td>
<td>DS_READ_I16</td>
<td>RETURN_DATA = signext(MEM[ADDR][15:0]).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Signed short read.</td>
</tr>
<tr>
<td>60</td>
<td>DS_READ_U16</td>
<td>RETURN_DATA = {16'h0, MEM[ADDR][15:0]}.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unsigned short read.</td>
</tr>
<tr>
<td>61</td>
<td>DS_SWIZZLE_B32</td>
<td>Dword swizzle, no data is written to LDS memory. See next section for details.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>62</td>
<td>DS_PERMUTE_B32</td>
<td>// VGPR[index][thread_id] is the VGPR RAM&lt;br&gt;VDST, ADDR and DATA0 are from the microcode DS encoding&lt;br&gt;tmp[0..63] = 0&lt;br&gt;for i in 0..63 do&lt;br&gt;  // If a source thread is disabled, it will not propagate data.&lt;br&gt;  next if !EXEC[i]&lt;br&gt;  // ADDR needs to be divided by 4.&lt;br&gt;  // High-order bits are ignored.&lt;br&gt;  dst_lane = floor((VGPR[ADDR][i] + OFFSET) / 4) mod 64&lt;br&gt;  tmp[dst_lane] = VGPR[DATA0][i]&lt;br&gt;endfor&lt;br&gt;// Copy data into destination VGPRs. If multiple sources&lt;br// select the same destination thread, the highest-numbered&lt;br// source thread wins.&lt;br&gt;for i in 0..63 do&lt;br&gt;  next if !EXEC[i]&lt;br&gt;  VGPR[VDST][i] = tmp[i]&lt;br&gt;endfor</td>
</tr>
</tbody>
</table>

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 hardware to implement an arbitrary swizzle across threads in a wavefront.

Note the address passed in is the thread ID multiplied by 4. This is due to a limitation in the DS hardware design.

If multiple sources map to the same destination lane, standard LDS arbitration rules determine which write wins.

See also DS_BPERMUTE_B32.

Examples (simplified 4-thread wavefronts):

```plaintext
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 }
```
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>DS_BPERMUTE_B32</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>// VGPR[index][thread_id] is the VGPR RAM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// VDST, ADDR and DATA0 are from the microcode DS encoding</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp[0..63] = 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for i in 0..63 do</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// ADDR needs to be divided by 4.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// High-order bits are ignored.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>src_lane = floor((VGPR[ADDR][i] + OFFSET) / 4) mod 64</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// EXEC is applied to the source VGPR reads.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>next if !EXEC[src_lane]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp[i] = VGPR[DATA0][src_lane]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>endfor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// Copy data into destination VGPRs. Some source</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// data may be broadcast to multiple lanes.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for i in 0..63 do</td>
</tr>
<tr>
<td></td>
<td></td>
<td>next if !EXEC[i]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VGPR[VDST][i] = tmp[i]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>endfor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Backward permute. This does not access LDS memory and may be</td>
</tr>
<tr>
<td></td>
<td></td>
<td>called even if no LDS memory is allocated to the wave. It uses</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LDS hardware to implement an arbitrary swizzle across threads</td>
</tr>
<tr>
<td></td>
<td></td>
<td>in a wavefront.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Note the address passed in is the thread ID multiplied by 4.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This is due to a limitation in the DS hardware design.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Note that EXEC mask is applied to both VGPR read and write. If</td>
</tr>
<tr>
<td></td>
<td></td>
<td>src_lane selects a disabled thread, zero will be returned.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See also DS_PERMUTE_B32.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Examples (simplified 4-thread wavefronts):</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VGPR[SRC0] = { A, B, C, D }</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VGPR[ADDR] = { 0, 0, 12, 4 }</td>
</tr>
<tr>
<td></td>
<td></td>
<td>EXEC = 0xF, OFFSET = 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VGPR[VDST] := { A, A, D, B }</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VGPR[SRC0] = { A, B, C, D }</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VGPR[ADDR] = { 0, 0, 12, 4 }</td>
</tr>
<tr>
<td></td>
<td></td>
<td>EXEC = 0xA, OFFSET = 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VGPR[VDST] := { -, 0, -, B }</td>
</tr>
<tr>
<td>64</td>
<td>DS_ADD_U64</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>// 64bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] += DATA[0:1];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA[0:1] = tmp.</td>
</tr>
<tr>
<td>65</td>
<td>DS_SUB_U64</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>// 64bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] -= DATA[0:1];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA[0:1] = tmp.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>------------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| 66     | DS_RSUB_U64 | // 64bit  
|        |            | tmp = MEM[ADDR];  
|        |            | MEM[ADDR] = DATA - MEM[ADDR];  
|        |            | RETURN_DATA = tmp.  
|        |            | Subtraction with reversed operands.  |
| 67     | DS_INC_U64  | // 64bit  
|        |            | tmp = MEM[ADDR];  
|        |            | MEM[ADDR] = (tmp >= DATA[0:1]) ? 0 : tmp + 1; // unsigned  
|        |            | compare  
|        |            | RETURN_DATA[0:1] = tmp.  |
| 68     | DS_DEC_U64  | // 64bit  
|        |            | tmp = MEM[ADDR];  
|        |            | MEM[ADDR] = (tmp == 0 || tmp > DATA[0:1]) ? DATA[0:1] : tmp - 1; // unsigned  
|        |            | compare  
|        |            | RETURN_DATA[0:1] = tmp.  |
| 69     | DS_MIN_I64  | // 64bit  
|        |            | tmp = MEM[ADDR];  
|        |            | MEM[ADDR] -= (DATA[0:1] < tmp) ? DATA[0:1] : tmp; // signed  
|        |            | compare  
|        |            | RETURN_DATA[0:1] = tmp.  |
| 70     | DS_MAX_I64  | // 64bit  
|        |            | tmp = MEM[ADDR];  
|        |            | MEM[ADDR] -= (DATA[0:1] > tmp) ? DATA[0:1] : tmp; // signed  
|        |            | compare  
|        |            | RETURN_DATA[0:1] = tmp.  |
| 71     | DS_MIN_U64  | // 64bit  
|        |            | tmp = MEM[ADDR];  
|        |            | MEM[ADDR] -= (DATA[0:1] < tmp) ? DATA[0:1] : tmp; // unsigned  
|        |            | compare  
|        |            | RETURN_DATA[0:1] = tmp.  |
| 72     | DS_MAX_U64  | // 64bit  
|        |            | tmp = MEM[ADDR];  
|        |            | MEM[ADDR] -= (DATA[0:1] > tmp) ? DATA[0:1] : tmp; // unsigned  
|        |            | compare  
|        |            | RETURN_DATA[0:1] = tmp.  |
| 73     | DS_AND_B64  | // 64bit  
|        |            | tmp = MEM[ADDR];  
|        |            | MEM[ADDR] &= DATA[0:1];  
|        |            | RETURN_DATA[0:1] = tmp.  |
| 74     | DS_OR_B64   | // 64bit  
|        |            | tmp = MEM[ADDR];  
|        |            | MEM[ADDR] |= DATA[0:1];  
<p>|        |            | RETURN_DATA[0:1] = tmp.  |</p>
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| 75     | DS_XOR_B64           | // 64bit  
|        |                      | tmp = MEM[ADDR];  
|        |                      | MEM[ADDR] ^= DATA[0:1];  
|        |                      | RETURN_DATA[0:1] = tmp.                                                                                                                                                                                                                                                   |
| 76     | DS_MSKOR_B64         | // 64bit  
|        |                      | tmp = MEM[ADDR];  
|        |                      | MEM[ADDR] = (MEM[ADDR] & ~DATA) | DATA2;  
|        |                      | RETURN_DATA = tmp.  
|        |                      | Masked dword OR, D0 contains the mask and D1 contains the new value.                                                                                                                                                                                                         |
| 77     | DS_WRITE_B64         | // 64bit  
|        |                      | MEM[ADDR] = DATA.                                                                                                                                                                                                                                                          |
|        |                      | Write qword.                                                                                                                                                                                                                                                                  |
| 78     | DS_WRITE2_B64        | // 64bit  
|        |                      | MEM[ADDR_BASE + OFFSET0 * 8] = DATA;  
|        |                      | Write 2 qwords.                                                                                                                                                                                                                                                                                                                        |
| 79     | DS_WRITE2ST64_B64    | // 64bit  
|        |                      | MEM[ADDR_BASE + OFFSET0 * 8 * 64] = DATA;  
|        |                      | MEM[ADDR_BASE + OFFSET1 * 8 * 64] = DATA2.                                                                                                                                                                                                                                  |
|        |                      | Write 2 qwords.                                                                                                                                                                                                                                                                                                                          |
| 80     | DS_CMPST_B64         | // 64bit  
|        |                      | tmp = MEM[ADDR];  
|        |                      | src = DATA2;  
|        |                      | cmp = DATA;  
|        |                      | MEM[ADDR] = (tmp == cmp) ? src : tmp;  
|        |                      | RETURN_DATA[0] = tmp.  
|        |                      | Compare and store. Caution, the order of src and cmp are the *opposite* of the BUFFER_ATOMIC_CMPSWAP_X2 opcode.                                                                                                                                                         |
| 81     | DS_CMPST_F64         | // 64bit  
|        |                      | tmp = MEM[ADDR];  
|        |                      | src = DATA2;  
|        |                      | cmp = DATA;  
|        |                      | MEM[ADDR] = (tmp == cmp) ? src : tmp;  
|        |                      | RETURN_DATA[0] = tmp.  
<p>|        |                      | Floating point compare and store that handles NaN/INF/denormal values.                                                                                                                                                                                                     |</p>
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| 82     | DS_MIN_F64         | // 64bit<br>
tmp = MEM[ADDR];<br>src = DATA;<br>cmp = DATA2;<br>MEM[ADDR] = (cmp < tmp) ? src : tmp.<br>Floating point minimum that handles NaN/INF/denormal values. |
| 83     | DS_MAX_F64         | // 64bit<br>
tmp = MEM[ADDR];<br>src = DATA;<br>cmp = DATA2;<br>MEM[ADDR] = (tmp > cmp) ? src : tmp.<br>Floating point maximum that handles NaN/INF/denormal values. |
<p>| 84     | DS_WRITE_B8_D16_HI | MEM[ADDR] = DATA[23:16].&lt;br&gt;Byte write in to high word.                     |
| 85     | DS_WRITE_B16_D16_HI| MEM[ADDR] = DATA[31:16].&lt;br&gt;Short write in to high word.                    |
| 86     | DS_READ_U8_D16     | RETURN_DATA[15:0] = {8'h0,MEM[ADDR][7:0]}.&lt;br&gt;Unsigned byte read with masked return to lower word. |
| 87     | DS_READ_U8_D16_HI  | RETURN_DATA[31:16] = {8'h0,MEM[ADDR][7:0]}.&lt;br&gt;Unsigned byte read with masked return to upper word. |
| 88     | DS_READ_I8_D16     | RETURN_DATA[15:0] = signext(MEM[ADDR][7:0]).&lt;br&gt;Signed byte read with masked return to lower word. |
| 89     | DS_READ_I8_D16_HI  | RETURN_DATA[31:16] = signext(MEM[ADDR][7:0]).&lt;br&gt;Signed byte read with masked return to upper word. |
| 90     | DS_READ_U16_D16    | RETURN_DATA[15:0] = MEM[ADDR][15:0].&lt;br&gt;Unsigned short read with masked return to upper word. |
| 91     | DS_READ_U16_D16_HI | RETURN_DATA[31:0] = MEM[ADDR][15:0].&lt;br&gt;Unsigned short read with masked return to upper word. |
| 96     | DS_ADD_RTN_U64     | // 64bit&lt;br&gt;tmp = MEM[ADDR];&lt;br&gt;MEM[ADDR] += DATA[0:1];&lt;br&gt;RETURN_DATA[0:1] = tmp. |</p>
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>97</td>
<td>DS_SUB_RTN_U64</td>
<td>// 64bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] -= DATA[0:1];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA[0:1] = tmp.</td>
</tr>
<tr>
<td>98</td>
<td>DS_RSUB_RTN_U64</td>
<td>// 64bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] = DATA - MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA = tmp.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Subtraction with reversed operands.</td>
</tr>
<tr>
<td>99</td>
<td>DS_INC_RTN_U64</td>
<td>// 64bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] = (tmp &gt;= DATA[0:1]) ? 0 : tmp + 1; // unsigned compare</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA[0:1] = tmp.</td>
</tr>
<tr>
<td>100</td>
<td>DS_DEC_RTN_U64</td>
<td>// 64bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] = (tmp == 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>compare</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA[0:1] = tmp.</td>
</tr>
<tr>
<td>101</td>
<td>DS_MIN_RTN_I64</td>
<td>// 64bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] -= (DATA[0:1] &lt; tmp) ? DATA[0:1] : tmp; // signed compare</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA[0:1] = tmp.</td>
</tr>
<tr>
<td>102</td>
<td>DS_MAX_RTN_I64</td>
<td>// 64bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] -= (DATA[0:1] &gt; tmp) ? DATA[0:1] : tmp; // signed compare</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA[0:1] = tmp.</td>
</tr>
<tr>
<td>103</td>
<td>DS_MIN_RTN_U64</td>
<td>// 64bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] -= (DATA[0:1] &lt; tmp) ? DATA[0:1] : tmp; // unsigned compare</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA[0:1] = tmp.</td>
</tr>
<tr>
<td>104</td>
<td>DS_MAX_RTN_U64</td>
<td>// 64bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] -= (DATA[0:1] &gt; tmp) ? DATA[0:1] : tmp; // unsigned compare</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA[0:1] = tmp.</td>
</tr>
<tr>
<td>105</td>
<td>DS_AND_RTN_B64</td>
<td>// 64bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] &amp;= DATA[0:1];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA[0:1] = tmp.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------</td>
<td>-------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| 106    | DS_OR_RTN_B64         | // 64bit  
|        |                       | tmp = MEM[ADDR];  
|        |                       | MEM[ADDR] |= DATA[0:1];  
|        |                       | RETURN_DATA[0:1] = tmp.                                                                     |
| 107    | DS_XOR_RTN_B64        | // 64bit  
|        |                       | tmp = MEM[ADDR];  
|        |                       | MEM[ADDR] ^= DATA[0:1];  
|        |                       | RETURN_DATA[0:1] = tmp.                                                                     |
| 108    | DS_MSKOR_RTN_B64      | // 64bit  
|        |                       | tmp = MEM[ADDR];  
|        |                       | MEM[ADDR] = (MEM[ADDR] & ~DATA) | DATA2;  
|        |                       | RETURN_DATA = tmp.  
|        |                       | Masked dword OR, D0 contains the mask and D1 contains the new value.                          |
| 109    | DS_WRXCHG_RTN_B64     | tmp = MEM[ADDR];  
|        |                       | MEM[ADDR] = DATA;  
|        |                       | RETURN_DATA = tmp.  
|        |                       | Write-exchange operation.                                                                    |
| 110    | DS_WRXCHG2_RTN_B64    | Write-exchange 2 separate qwords.                                                               |
| 111    | DS_WRXCHG2ST64_RTN_B64| Write-exchange 2 qwords with a stride of 64 qwords.                                             |
| 112    | DS_CMPST_RTN_B64      | // 64bit  
|        |                       | tmp = MEM[ADDR];  
|        |                       | src = DATA2;  
|        |                       | cmp = DATA;  
|        |                       | MEM[ADDR] = (tmp == cmp) ? src : tmp;  
|        |                       | RETURN_DATA[0] = tmp.  
|        |                       | Compare and store. Caution, the order of src and cmp are the *opposite* of the BUFFER_ATOMIC_CMP_SWAP_X2 opcode. |
| 113    | DS_CMPST_RTN_F64      | // 64bit  
|        |                       | tmp = MEM[ADDR];  
|        |                       | src = DATA2;  
|        |                       | cmp = DATA;  
|        |                       | MEM[ADDR] = (tmp == cmp) ? src : tmp;  
|        |                       | RETURN_DATA[0] = tmp.  
<p>|        |                       | Floating point compare and store that handles NaN/INF/denormal values.                       |</p>
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| 114    | DS_MIN_RTN_F64 | // 64bit
|        |      | tmp = MEM[ADDR]; |
|        |      | src = DATA; |
|        |      | cmp = DATA2; |
|        |      | Floating point minimum that handles NaN/INF/denormal values. |
| 115    | DS_MAX_RTN_F64 | // 64bit
|        |      | tmp = MEM[ADDR]; |
|        |      | src = DATA; |
|        |      | cmp = DATA2; |
|        |      | Floating point maximum that handles NaN/INF/denormal values. |
| 118    | DS_READ_B64 | RETURN_DATA = MEM[ADDR]. |
|        |      | Read 1 qword. |
| 119    | DS_READ2_B64 | RETURN_DATA[0] = MEM[ADDR_BASE + OFFSET0 * 8]; |
|        |      | RETURN_DATA[1] = MEM[ADDR_BASE + OFFSET1 * 8]. |
|        |      | Read 2 qwords. |
| 120    | DS_READ2ST64_B64 | RETURN_DATA[0] = MEM[ADDR_BASE + OFFSET0 * 8 * 64]; |
|        |      | RETURN_DATA[1] = MEM[ADDR_BASE + OFFSET1 * 8 * 64]. |
|        |      | Read 2 qwords. |
| 126    | DS_CONDXCHG32_RTN_B64 | Conditional write exchange. |
| 128    | DS_ADD_SRC2_U32 | //32bit
|        |      | A = ADDR_BASE; |
| 129    | DS_SUB_SRC2_U32 | //32bit
|        |      | A = ADDR_BASE; |
| 130    | DS_RSUB_SRC2_U32 | //32bit
<p>|        |      | A = ADDR_BASE; |</p>
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>152</td>
<td>DS_GWS_SEMA_RELEA SE_ALL</td>
<td>GDS Only: The GWS resource (rid) indicated will process this opcode by updating the counter and labeling the specified resource as a semaphore.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// Determine the GWS resource to work on rid[5:0] = SH_SX_EXPCMD.gds_base[5:0] + offset0[5:0];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// Incr the state counter of the resource state.counter[rid] = state.wave_in_queue; state.type = SEMAPHORE; return rd_done; //release calling wave</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This action will release ALL queued waves; it will have no effect if no waves are present.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>153</td>
<td>DS_GWS_INIT</td>
<td>GDS Only: Initialize a barrier or semaphore resource.</td>
</tr>
</tbody>
</table>

```c
// Determine the GWS resource to work on
rid[5:0] = SH_SX_EXPCMD.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
```

| 154    | DS_GWS_SEMA_V | GDS Only: The GWS resource indicated will process this opcode by updating the counter and labeling the resource as a semaphore. |

```c
// Determine the GWS resource to work on
rid[5:0] = SH_SX_EXPCMD.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 will release one wave if any are queued in this resource.
```

| 155    | DS_GWS_SEMA_BR | GDS Only: The GWS resource indicated will process this opcode by updating the counter by the bulk release delivered count and labeling the resource as a semaphore. |

```c
// Determine the GWS resource to work on
rid[5:0] = SH_SX_EXPCMD.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 will release count number of waves, immediately if queued, or as they arrive from the noted resource.
```
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>156</td>
<td>DS_GWS_SEMA_P</td>
<td>GDS Only: The GWS resource indicated will process this opcode by queueing it until counter enables a release and then decrementing the counter of the resource as a semaphore.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>//Determine the GWS resource to work on</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rid[5:0] = SH_SX_EXPCMD.gds_base[5:0] + offset0[5:0]; state.type = SEMAPHORE;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ENQUEUE until(state[rid].counter &gt; 0) state[rid].counter -= 1; return rd_done;</td>
</tr>
<tr>
<td>157</td>
<td>DS_GWS_BARRIER</td>
<td>GDS Only: The GWS resource indicated will process this opcode by queueing it until barrier is satisfied. The number of waves needed is passed in as DATA of first valid thread.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>//Determine the GWS resource to work on</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rid[5:0] = SH_SX_EXPCMD.gds_base[5:0] + OFFSET0[5:0]; index = find first valid (vector mask); value = DATA[thread: index];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// Input Decision Machine</td>
</tr>
<tr>
<td></td>
<td></td>
<td>state.type[rid] = BARRIER; if(state[rid].counter &lt;= 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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Since the waves deliver the count for the next barrier, this function can have a different size barrier for each occurrence.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// Release Machine</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if(state.type == BARRIER) then if(state.flag != thread.flag) then return rd_done; endif;</td>
</tr>
<tr>
<td>182</td>
<td>DS_READ_ADDTID_B32</td>
<td>RETURN_DATA = MEM[ADDR_BASE + OFFSET + M0.OFFSET + TID*4]. Dword read.</td>
</tr>
<tr>
<td>189</td>
<td>DS_CONSUME</td>
<td>LDS &amp; 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.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>190</td>
<td>DS_APPEND</td>
<td>LDS &amp; GDS. Add ((\text{count_bits}(\text{exec_mask}))) to the value stored in DS memory at ((\text{M0.base} + \text{instr_offset})). Return the pre-operation value to VGPRs.</td>
</tr>
<tr>
<td>191</td>
<td>DS_ORDERED_COUNT</td>
<td>GDS-only. Add ((\text{count_bits}(\text{exec_mask}))) to one of 4 dedicated ordered-count counters (aka 'packers'). Additional bits of instr.offset field are overloaded to hold packer-id, 'last'.</td>
</tr>
<tr>
<td>197</td>
<td>DS_MIN_SRC2_I64</td>
<td>//64bit&lt;br&gt;A = ADDR_BASE;&lt;br&gt;B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0});&lt;br&gt;MEM[A] = \text{min}(MEM[A], MEM[B]).</td>
</tr>
<tr>
<td>198</td>
<td>DS_MAX_SRC2_I64</td>
<td>//64bit&lt;br&gt;A = ADDR_BASE;&lt;br&gt;B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0});&lt;br&gt;MEM[A] = \text{max}(MEM[A], MEM[B]).</td>
</tr>
</tbody>
</table>

"Vega" 7nm Instruction Set Architecture

12.13. LDS & GDS Instructions
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>199</td>
<td>DS_MIN_SRC2_U64</td>
<td>//64bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A = ADDR_BASE;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[A] = min(MEM[A], MEM[B]).</td>
</tr>
<tr>
<td>200</td>
<td>DS_MAX_SRC2_U64</td>
<td>//64bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A = ADDR_BASE;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[A] = max(MEM[A], MEM[B]).</td>
</tr>
<tr>
<td>201</td>
<td>DS_AND_SRC2_B64</td>
<td>//64bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A = ADDR_BASE;</td>
</tr>
<tr>
<td>202</td>
<td>DS_OR_SRC2_B64</td>
<td>//64bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A = ADDR_BASE;</td>
</tr>
<tr>
<td>203</td>
<td>DS_XOR_SRC2_B64</td>
<td>//64bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A = ADDR_BASE;</td>
</tr>
<tr>
<td>205</td>
<td>DS_WRITE_SRC2_B64</td>
<td>//64bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A = ADDR_BASE;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[A] = MEM[B].</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Write qword.</td>
</tr>
<tr>
<td>210</td>
<td>DS_MIN_SRC2_F64</td>
<td>//64bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A = ADDR_BASE;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Float, handles NaN/INF/denorm.</td>
</tr>
<tr>
<td>211</td>
<td>DS_MAX_SRC2_F64</td>
<td>//64bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A = ADDR_BASE;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Float, handles NaN/INF/denorm.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>----------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>222</td>
<td>DS_WRITE_B96</td>
<td>{MEM[ADDR + 8], MEM[ADDR + 4], MEM[ADDR]} = DATA[95:0]. Tri-dword write.</td>
</tr>
<tr>
<td>223</td>
<td>DS_WRITE_B128</td>
<td>{MEM[ADDR + 12], MEM[ADDR + 8], MEM[ADDR + 4], MEM[ADDR]} = DATA[127:0]. Quad-dword write.</td>
</tr>
<tr>
<td>254</td>
<td>DS_READ_B96</td>
<td>Tri-dword read.</td>
</tr>
<tr>
<td>255</td>
<td>DS_READ_B128</td>
<td>Quad-dword read.</td>
</tr>
</tbody>
</table>

12.13.1. DS_SWIZZLE_B32 Details

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 special 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,1a,6,e,4,c,8,10,11,19,15,1d,13,1b,17,1f,12,1a,16,1e,14,1c,18,10,20}
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 will swap 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, rotate left:
{2,3,4,5,6,7,8,9,a,b,c,d,e,f,10,1c,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,20,1}
0x01, mask=1, rotate left:
{2,1,4,7,6,5,8,a,9,c,f,e,d,10,11,12,13,14,15,16,17,18,19,1a,1b,1c,1d,1e,1f,20,3}
0x01, mask=1, rotate right:
{1e,1,4,3,2,5,8,7,6,9,c,b,a,d,10,f,e,11,14,13,12,15,18,17,16,19,1c,1b,1a,1d,20,1f}

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 will grab thread1, thread4 will grab 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;
    }
} else if (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;
    }
} else if (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
    }
}

### 12.13.2. LDS Instruction Limitations

Some of the DS instructions are available only to GDS, not LDS. These are:

- `DS_GWS_SEMA_RELEASE_ALL`
- `DS_GWS_INIT`
- `DS_GWS_SEMA_V`
- `DS_GWS_SEMA_BR`


12.14. MUBUF Instructions

The bitfield map of the MUBUF format is:

```
  31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
MUBUF   1 1 1 0 0 0  r f OP_s SLC  f GLC  OFFSET_12 0
       |     |   |   |   |   |   |   |   |    |   |   |   |   |   |   |   |   |
63     |     |   |   |   |   |   |   |   |    |   |   |   |   |   |   |   |   |
SOFFSET_8 (sgpr) TFE 2 SRSRC6 (T# sgpr) VDATA6 (vgpr: src or dst) VADDR6 (vgpr)
```

where:

- **OFFSET** = Unsigned immediate byte offset.
- **OFFEN** = Send offset either as VADDR or as zero.
- **IDXEN** = Send index either as VADDR or as zero.
- **GLC** = Global coherency.
- **ADDR64** = Buffer address of 64 bits.
- **LDS** = Data read from/written to LDS or VGPR.
- **OP** = Opcode instructions.
- **VADDR** = VGPR address source.
- **VDATA** = Destination vector GPR.
- **SRSRC** = Scalar GPR that specifies resource constant.
- **SLC** = System level coherent.
- **TFE** = Texture fail enable.
- **SOFFSET** = Byte offset added to the memory address of an SGPR.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>BUFFER_LOAD_FORMAT_X</td>
<td>Untyped buffer load 1 dword with format conversion.</td>
</tr>
<tr>
<td>1</td>
<td>BUFFER_LOAD_FORMAT_XY</td>
<td>Untyped buffer load 2 dwords with format conversion.</td>
</tr>
<tr>
<td>2</td>
<td>BUFFER_LOAD_FORMAT_XYZ</td>
<td>Untyped buffer load 3 dwords with format conversion.</td>
</tr>
<tr>
<td>3</td>
<td>BUFFER_STORE_FORMAT_X</td>
<td>Untyped buffer store 1 dword with format conversion.</td>
</tr>
<tr>
<td>4</td>
<td>BUFFER_STORE_FORMAT_XY</td>
<td>Untyped buffer store 2 dwords with format conversion.</td>
</tr>
<tr>
<td>5</td>
<td>BUFFER_STORE_FORMAT_XYZ</td>
<td>Untyped buffer store 3 dwords with format conversion.</td>
</tr>
<tr>
<td>6</td>
<td>BUFFER_STORE_FORMAT_XYZW</td>
<td>Untyped buffer store 4 dwords with format conversion.</td>
</tr>
<tr>
<td>8</td>
<td>BUFFER_LOAD_FORMAT_D16_X</td>
<td>Untyped buffer load 1 dword with format conversion.</td>
</tr>
<tr>
<td>9</td>
<td>BUFFER_LOAD_FORMAT_D16_XY</td>
<td>Untyped buffer load 1 dword with format conversion.</td>
</tr>
<tr>
<td>10</td>
<td>BUFFER_LOAD_FORMAT_D16_XYZ</td>
<td>Untyped buffer load 2 dwords with format conversion.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------------------------------------</td>
<td>----------------------------------------------------------------</td>
</tr>
<tr>
<td>11</td>
<td>BUFFER_LOAD_FORMAT_D16_XY ZW</td>
<td>Untyped buffer load 2 dwords with format conversion.</td>
</tr>
<tr>
<td>12</td>
<td>BUFFER_STORE_FORMAT_D16_X</td>
<td>Untyped buffer store 1 dword with format conversion.</td>
</tr>
<tr>
<td>13</td>
<td>BUFFER_STORE_FORMAT_D16_XY</td>
<td>Untyped buffer store 1 dword with format conversion.</td>
</tr>
<tr>
<td>14</td>
<td>BUFFER_STORE_FORMAT_D16_XYZ</td>
<td>Untyped buffer store 2 dwords with format conversion.</td>
</tr>
<tr>
<td>15</td>
<td>BUFFER_STORE_FORMAT_D16_XYZ</td>
<td>Untyped buffer store 2 dwords with format conversion.</td>
</tr>
<tr>
<td>16</td>
<td>BUFFER_LOAD_UBYTE</td>
<td>Untyped buffer load unsigned byte (zero extend to VGPR destination).</td>
</tr>
<tr>
<td>17</td>
<td>BUFFER_LOAD_SBYTE</td>
<td>Untyped buffer load signed byte (sign extend to VGPR destination).</td>
</tr>
<tr>
<td>18</td>
<td>BUFFER_LOAD_USHORT</td>
<td>Untyped buffer load unsigned short (zero extend to VGPR destination).</td>
</tr>
<tr>
<td>19</td>
<td>BUFFER_LOAD_SSHORT</td>
<td>Untyped buffer load signed short (sign extend to VGPR destination).</td>
</tr>
<tr>
<td>20</td>
<td>BUFFER_LOAD_DWORD</td>
<td>Untyped buffer load dword.</td>
</tr>
<tr>
<td>21</td>
<td>BUFFER_LOAD_DWORDX2</td>
<td>Untyped buffer load 2 dwords.</td>
</tr>
<tr>
<td>22</td>
<td>BUFFER_LOAD_DWORDX3</td>
<td>Untyped buffer load 3 dwords.</td>
</tr>
<tr>
<td>23</td>
<td>BUFFER_LOAD_DWORDX4</td>
<td>Untyped buffer load 4 dwords.</td>
</tr>
<tr>
<td>24</td>
<td>BUFFER_STORE_BYTE</td>
<td>Untyped buffer store byte. Stores S0[7:0].</td>
</tr>
<tr>
<td>25</td>
<td>BUFFER_STORE_BYTE_D16_HI</td>
<td>Untyped buffer store byte. Stores S0[23:16].</td>
</tr>
<tr>
<td>26</td>
<td>BUFFER_STORE_SHORT</td>
<td>Untyped buffer store short. Stores S0[15:0].</td>
</tr>
<tr>
<td>27</td>
<td>BUFFER_STORE_SHORT_D16_HI</td>
<td>Untyped buffer store short. Stores S0[31:16].</td>
</tr>
<tr>
<td>28</td>
<td>BUFFER_STORE_DWORD</td>
<td>Untyped buffer store dword.</td>
</tr>
<tr>
<td>29</td>
<td>BUFFER_STORE_DWORDX2</td>
<td>Untyped buffer store 2 dwords.</td>
</tr>
<tr>
<td>30</td>
<td>BUFFER_STORE_DWORDX3</td>
<td>Untyped buffer store 3 dwords.</td>
</tr>
<tr>
<td>31</td>
<td>BUFFER_STORE_DWORDX4</td>
<td>Untyped buffer store 4 dwords.</td>
</tr>
<tr>
<td>32</td>
<td>BUFFER_LOAD_UBYTE_D16</td>
<td>D0[15:0] = {8'h0, MEM[ADDR]}.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Untyped buffer load unsigned byte.</td>
</tr>
<tr>
<td>33</td>
<td>BUFFER_LOAD_UBYTE_D16_HI</td>
<td>D0[31:16] = {8'h0, MEM[ADDR]}.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Untyped buffer load unsigned byte.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>34</td>
<td>BUFFER_LOAD_SBYTE_D16</td>
<td>(D0[15:0] = {8'h0, MEM[ADDR]}). Untyped buffer load signed byte.</td>
</tr>
<tr>
<td>35</td>
<td>BUFFER_LOAD_SBYTE_D16_HI</td>
<td>(D0[31:16] = {8'h0, MEM[ADDR]}). Untyped buffer load signed byte.</td>
</tr>
<tr>
<td>36</td>
<td>BUFFER_LOAD_SHORT_D16</td>
<td>(D0[15:0] = MEM[ADDR]). Untyped buffer load short.</td>
</tr>
<tr>
<td>37</td>
<td>BUFFER_LOAD_SHORT_D16_HI</td>
<td>(D0[31:16] = MEM[ADDR]). Untyped buffer load short.</td>
</tr>
<tr>
<td>38</td>
<td>BUFFER_LOAD_FORMAT_D16_HI_X</td>
<td>(D0[31:16] = MEM[ADDR]). Untyped buffer load 1 dword with format conversion.</td>
</tr>
<tr>
<td>39</td>
<td>BUFFER_STORE_FORMAT_D16_HI_X</td>
<td>Untyped buffer store 1 dword with format conversion.</td>
</tr>
<tr>
<td>61</td>
<td>BUFFER_STORE_LDS_DWORD</td>
<td>Store one DWORD from LDS memory to system memory without utilizing VGPRs.</td>
</tr>
<tr>
<td>62</td>
<td>BUFFER_WBINVL1</td>
<td>Write back and invalidate the shader L1. Returns ACK to shader.</td>
</tr>
<tr>
<td>63</td>
<td>BUFFER_WBINVL1_VOL</td>
<td>Write back and invalidate the shader L1 only for lines that are marked volatile. Returns ACK to shader.</td>
</tr>
<tr>
<td>64</td>
<td>BUFFER_ATOMIC_SWAP</td>
<td>// 32bit (tmp = MEM[ADDR]); (MEM[ADDR] = DATA); (RETURN_DATA = tmp).</td>
</tr>
<tr>
<td>65</td>
<td>BUFFER_ATOMIC_CMPSWAP</td>
<td>// 32bit (tmp = MEM[ADDR]); (src = DATA[0]); (cmp = DATA[1]); (MEM[ADDR] = (tmp == cmp) ? src : tmp); (RETURN_DATA[0] = tmp).</td>
</tr>
<tr>
<td>66</td>
<td>BUFFER_ATOMIC_ADD</td>
<td>// 32bit (tmp = MEM[ADDR]); (MEM[ADDR] += DATA); (RETURN_DATA = tmp).</td>
</tr>
<tr>
<td>67</td>
<td>BUFFER_ATOMIC_SUB</td>
<td>// 32bit (tmp = MEM[ADDR]); (MEM[ADDR] -= DATA); (RETURN_DATA = tmp).</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>68</td>
<td>BUFFER_ATOMIC_SMIN</td>
<td>// 32bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] = (DATA &lt; tmp) ? DATA : tmp; // signed compare</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA = tmp.</td>
</tr>
<tr>
<td>69</td>
<td>BUFFER_ATOMIC_UMIN</td>
<td>// 32bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] = (DATA &lt; tmp) ? DATA : tmp; // unsigned compare</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA = tmp.</td>
</tr>
<tr>
<td>70</td>
<td>BUFFER_ATOMIC_SMAX</td>
<td>// 32bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] = (DATA &gt; tmp) ? DATA : tmp; // signed compare</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA = tmp.</td>
</tr>
<tr>
<td>71</td>
<td>BUFFER_ATOMIC_UMAX</td>
<td>// 32bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] = (DATA &gt; tmp) ? DATA : tmp; // unsigned compare</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA = tmp.</td>
</tr>
<tr>
<td>72</td>
<td>BUFFER_ATOMIC_AND</td>
<td>// 32bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] &amp;= DATA;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA = tmp.</td>
</tr>
<tr>
<td>73</td>
<td>BUFFER_ATOMIC_OR</td>
<td>// 32bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA = tmp.</td>
</tr>
<tr>
<td>74</td>
<td>BUFFER_ATOMIC_XOR</td>
<td>// 32bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] ^= DATA;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA = tmp.</td>
</tr>
<tr>
<td>75</td>
<td>BUFFER_ATOMIC_INC</td>
<td>// 32bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] = (tmp &gt;= DATA) ? 0 : tmp + 1; // unsigned compare</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA = tmp.</td>
</tr>
<tr>
<td>76</td>
<td>BUFFER_ATOMIC_DEC</td>
<td>// 32bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] = (tmp == 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA = tmp.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>96</td>
<td>BUFFER_ATOMIC_SWAP_X2</td>
<td>// 64bit&lt;br&gt;tmp = MEM[ADDR]; &lt;br&gt;MEM[ADDR] = DATA[0:1]; &lt;br&gt;RETURN_DATA[0:1] = tmp.</td>
</tr>
<tr>
<td>97</td>
<td>BUFFER_ATOMIC_CMPSWAP_X2</td>
<td>// 64bit&lt;br&gt;tmp = MEM[ADDR]; &lt;br&gt;src = DATA[0:1]; &lt;br&gt;cmp = DATA[2:3]; &lt;br&gt;MEM[ADDR] = (tmp == cmp) ? src : tmp; &lt;br&gt;RETURN_DATA[0:1] = tmp.</td>
</tr>
<tr>
<td>98</td>
<td>BUFFER_ATOMIC_ADD_X2</td>
<td>// 64bit&lt;br&gt;tmp = MEM[ADDR]; &lt;br&gt;MEM[ADDR] += DATA[0:1]; &lt;br&gt;RETURN_DATA[0:1] = tmp.</td>
</tr>
<tr>
<td>99</td>
<td>BUFFER_ATOMIC_SUB_X2</td>
<td>// 64bit&lt;br&gt;tmp = MEM[ADDR]; &lt;br&gt;MEM[ADDR] -= DATA[0:1]; &lt;br&gt;RETURN_DATA[0:1] = tmp.</td>
</tr>
<tr>
<td>100</td>
<td>BUFFER_ATOMIC_SMIN_X2</td>
<td>// 64bit&lt;br&gt;tmp = MEM[ADDR]; &lt;br&gt;MEM[ADDR] -= (DATA[0:1] &lt; tmp) ? DATA[0:1] : tmp; // signed compare &lt;br&gt;RETURN_DATA[0:1] = tmp.</td>
</tr>
<tr>
<td>104</td>
<td>BUFFER_ATOMIC_AND_X2</td>
<td>// 64bit&lt;br&gt;tmp = MEM[ADDR]; &lt;br&gt;MEM[ADDR] &amp;= DATA[0:1]; &lt;br&gt;RETURN_DATA[0:1] = tmp.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>105</td>
<td>BUFFER_ATOMIC_OR_X2</td>
<td>// 64bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA[0:1] = tmp.</td>
</tr>
<tr>
<td>106</td>
<td>BUFFER_ATOMIC_XOR_X2</td>
<td>// 64bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] ^= DATA[0:1];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA[0:1] = tmp.</td>
</tr>
<tr>
<td>107</td>
<td>BUFFER_ATOMIC_INC_X2</td>
<td>// 64bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] = (tmp &gt;= DATA[0:1]) ? 0 : tmp + 1; // unsigned compare</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA[0:1] = tmp.</td>
</tr>
<tr>
<td>108</td>
<td>BUFFER_ATOMIC_DEC_X2</td>
<td>// 64bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] = (tmp == 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA[0:1] = tmp.</td>
</tr>
</tbody>
</table>

### 12.15. MTBUF Instructions

The bitfield map of the MTBUF format is:

```
+----------+----------+----------+----------+----------+----------+
|   31      |   24     |   17     |   10     |   3       |   0       |
| OFFSET12  | SRSRC3   | DFMT4    | OP4      | NFMT3     | MTBUF     |
+----------+----------+----------+----------+----------+----------+
| 63        | 56       | 49       | 42       | 35       | 28       |
| SOFFSET8 (sgpr) | TFE SLC | SRSRC1 (T# sgpr) | VDATA6 (vgpr: src or dst) | VADDR6 (vgpr) |
```

where:

- OFFSET = Unsigned immediate byte offset.
- OFFEN = Send offset either as VADDR or as zero.
- IDXEN = Send index either as VADDR or as zero.
- GLC = Global coherency.
- ADDR64 = Buffer address of 64 bits.
- OP = Opcode instructions.
- DFMT = Data format for typed buffer.
- NFMT = Number format for typed buffer.
- VADDR = VGPR address source.
- VDATA = Vector GPR for read/write result.
- SRSRC = Scalar GPR that specifies resource constant.
- SOFFSET = Unsigned byte offset from an SGPR.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>TBUFFER_LOAD_FORMAT_X</td>
<td>Typed buffer load 1 dword with format conversion.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------</td>
<td>--------------------------------------------------------------</td>
</tr>
<tr>
<td>1</td>
<td>TBUFFER_LOAD_FORMAT_XY</td>
<td>Typed buffer load 2 dwords with format conversion.</td>
</tr>
<tr>
<td>2</td>
<td>TBUFFER_LOAD_FORMAT_XYZ</td>
<td>Typed buffer load 3 dwords with format conversion.</td>
</tr>
<tr>
<td>3</td>
<td>TBUFFER_LOAD_FORMAT_XYZW</td>
<td>Typed buffer load 4 dwords with format conversion.</td>
</tr>
<tr>
<td>4</td>
<td>TBUFFER_STORE_FORMAT_X</td>
<td>Typed buffer store 1 dword with format conversion.</td>
</tr>
<tr>
<td>5</td>
<td>TBUFFER_STORE_FORMAT_XY</td>
<td>Typed buffer store 2 dwords with format conversion.</td>
</tr>
<tr>
<td>6</td>
<td>TBUFFER_STORE_FORMAT_XYZ</td>
<td>Typed buffer store 3 dwords with format conversion.</td>
</tr>
<tr>
<td>7</td>
<td>TBUFFER_STORE_FORMAT_XYZW</td>
<td>Typed buffer store 4 dwords with format conversion.</td>
</tr>
<tr>
<td>8</td>
<td>TBUFFER_LOAD_FORMAT_D16_X</td>
<td>Typed buffer load 1 dword with format conversion.</td>
</tr>
<tr>
<td>9</td>
<td>TBUFFER_LOAD_FORMAT_D16_XY</td>
<td>Typed buffer load 1 dword with format conversion.</td>
</tr>
<tr>
<td>10</td>
<td>TBUFFER_LOAD_FORMAT_D16_XYZ</td>
<td>Typed buffer load 2 dwords with format conversion.</td>
</tr>
<tr>
<td>11</td>
<td>TBUFFER_LOAD_FORMAT_D16_XYZW</td>
<td>Typed buffer load 2 dwords with format conversion.</td>
</tr>
<tr>
<td>12</td>
<td>TBUFFER_STORE_FORMAT_D16_X</td>
<td>Typed buffer store 1 dword with format conversion.</td>
</tr>
<tr>
<td>13</td>
<td>TBUFFER_STORE_FORMAT_D16_XY</td>
<td>Typed buffer store 1 dword with format conversion.</td>
</tr>
<tr>
<td>14</td>
<td>TBUFFER_STORE_FORMAT_D16_XYZ</td>
<td>Typed buffer store 2 dwords with format conversion.</td>
</tr>
<tr>
<td>15</td>
<td>TBUFFER_STORE_FORMAT_D16_XYZW</td>
<td>Typed buffer store 2 dwords with format conversion.</td>
</tr>
</tbody>
</table>

### 12.16. MIMG Instructions

The bitfield map of the MIMG format is:

![MIMG Bitfield Map](image-url)
where:

DMASK = Enable mask for image read/write data components.
UNRM = Force address to be unnormalized.
GLC = Global coherency.
DA = Declare an array.
A16 = Texture address component size.
TFE = Texture fail enable.
LWE = LOD warning enable.
OP = Opcode instructions.
SLC = System level coherent.
VADDR = VGPR address source.
VDATA = Vector GPR for read/write result.
SRSRC = Scalar GPR that specifies resource constant.
SSAMP = Scalar GPR that specifies sampler constant.
D16 = Data in VGPRs is 16 bits, not 32 bits.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>IMAGE_LOAD</td>
<td>Image memory load with format conversion specified in T#. No sampler.</td>
</tr>
<tr>
<td>1</td>
<td>IMAGE_LOAD_MIP</td>
<td>Image memory load with user-supplied mip level. No sampler.</td>
</tr>
<tr>
<td>2</td>
<td>IMAGE_LOAD_PCK</td>
<td>Image memory load with no format conversion. No sampler.</td>
</tr>
<tr>
<td>3</td>
<td>IMAGE_LOAD_PCK_SGN</td>
<td>Image memory load with no format conversion and sign extension. No sampler.</td>
</tr>
<tr>
<td>4</td>
<td>IMAGE_LOAD_MIP_PCK</td>
<td>Image memory load with user-supplied mip level, no format conversion. No sampler.</td>
</tr>
<tr>
<td>5</td>
<td>IMAGE_LOAD_MIP_PCK_SGN</td>
<td>Image memory load with user-supplied mip level, no format conversion and with sign extension. No sampler.</td>
</tr>
<tr>
<td>8</td>
<td>IMAGE_STORE</td>
<td>Image memory store with format conversion specified in T#. No sampler.</td>
</tr>
<tr>
<td>9</td>
<td>IMAGE_STORE_MIP</td>
<td>Image memory store with format conversion specified in T# to user specified mip level. No sampler.</td>
</tr>
<tr>
<td>10</td>
<td>IMAGE_STORE_PCK</td>
<td>Image memory store of packed data without format conversion. No sampler.</td>
</tr>
<tr>
<td>11</td>
<td>IMAGE_STORE_MIP_PCK</td>
<td>Image memory store of packed data without format conversion to user-supplied mip level. No sampler.</td>
</tr>
<tr>
<td>14</td>
<td>IMAGE_GET_RESINFO</td>
<td>return resource info for a given mip level specified in the address vgpr. No sampler. Returns 4 integer values into VGPRs 3-0: {num_mip_levels, depth, height, width}.</td>
</tr>
</tbody>
</table>
| 16     | IMAGE_ATOMIC_SWAP | // 32bit
|        |                    | tmp = MEM[ADDR];
|        |                    | MEM[ADDR] = DATA;
<p>|        |                    | RETURN_DATA = tmp. |</p>
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| 17     | IMAGE_ATOMIC_CMPSWP   | `// 32bit`  
         |          | `tmp = MEM[ADDR];`  
         |          | `src = DATA[0];`  
         |          | `cmp = DATA[1];`  
         |          | `MEM[ADDR] = (tmp == cmp) ? src : tmp;`  
         |          | `RETURN_DATA[0] = tmp.`  
| 18     | IMAGE_ATOMIC_ADD      | `// 32bit`  
         |          | `tmp = MEM[ADDR];`  
         |          | `MEM[ADDR] += DATA;`  
         |          | `RETURN_DATA = tmp.`  
| 19     | IMAGE_ATOMIC_SUB      | `// 32bit`  
         |          | `tmp = MEM[ADDR];`  
         |          | `MEM[ADDR] -= DATA;`  
         |          | `RETURN_DATA = tmp.`  
| 20     | IMAGE_ATOMIC_SMIN     | `// 32bit`  
         |          | `tmp = MEM[ADDR];`  
         |          | `MEM[ADDR] = (DATA < tmp) ? DATA : tmp;`  
         |          | `// signed compare`  
         |          | `RETURN_DATA = tmp.`  
| 21     | IMAGE_ATOMIC_UMIN     | `// 32bit`  
         |          | `tmp = MEM[ADDR];`  
         |          | `MEM[ADDR] = (DATA < tmp) ? DATA : tmp;`  
         |          | `// unsigned compare`  
         |          | `RETURN_DATA = tmp.`  
| 22     | IMAGE_ATOMIC_SMAX     | `// 32bit`  
         |          | `tmp = MEM[ADDR];`  
         |          | `MEM[ADDR] = (DATA > tmp) ? DATA : tmp;`  
         |          | `// signed compare`  
         |          | `RETURN_DATA = tmp.`  
| 23     | IMAGE_ATOMIC_UMAX     | `// 32bit`  
         |          | `tmp = MEM[ADDR];`  
         |          | `MEM[ADDR] = (DATA > tmp) ? DATA : tmp;`  
         |          | `// unsigned compare`  
         |          | `RETURN_DATA = tmp.`  
| 24     | IMAGE_ATOMIC_AND      | `// 32bit`  
         |          | `tmp = MEM[ADDR];`  
         |          | `MEM[ADDR] &= DATA;`  
         |          | `RETURN_DATA = tmp.`  
| 25     | IMAGE_ATOMIC_OR       | `// 32bit`  
         |          | `tmp = MEM[ADDR];`  
         |          | `MEM[ADDR] |= DATA;`  
         |          | `RETURN_DATA = tmp.`  
| 26     | IMAGE_ATOMIC_XOR      | `// 32bit`  
         |          | `tmp = MEM[ADDR];`  
         |          | `MEM[ADDR] ^= DATA;`  
         |          | `RETURN_DATA = tmp.`  

"Vega" 7nm Instruction Set Architecture

12.16. MIMG Instructions
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| 27     | IMAGE_ATOMIC_INC   | // 32bit  
|        | mem = MEM[ADDR];   |  
|        | MEM[ADDR] = (tmp >= DATA) ? 0 : tmp + 1; // unsigned compare  
|        | return DATA = tmp. |                                                                            |
| 28     | IMAGE_ATOMIC_DEC   | // 32bit  
|        | mem = MEM[ADDR];   |  
|        | MEM[ADDR] = (tmp == 0 || tmp > DATA) ? DATA : tmp - 1; // unsigned compare  
<p>|        | return DATA = tmp. |                                                                            |
| 32     | IMAGE_SAMPLE       | sample texture map.                                                         |
| 33     | IMAGE_SAMPLE_CL    | sample texture map, with LOD clamp specified in shader.                     |
| 34     | IMAGE_SAMPLE_D     | sample texture map, with user derivatives                                    |
| 35     | IMAGE_SAMPLE_D_CL  | sample texture map, with LOD clamp specified in shader, with user derivatives. |
| 36     | IMAGE_SAMPLE_L     | sample texture map, with user LOD.                                          |
| 37     | IMAGE_SAMPLE_B     | sample texture map, with lod bias.                                          |
| 38     | IMAGE_SAMPLE_B_CL  | sample texture map, with LOD clamp specified in shader, with lod bias.      |
| 39     | IMAGE_SAMPLE_LZ    | sample texture map, from level 0.                                           |
| 40     | IMAGE_SAMPLE_C     | sample texture map, with PCF.                                               |
| 41     | IMAGE_SAMPLE_C_CL  | SAMPLE_C, with LOD clamp specified in shader.                               |
| 42     | IMAGE_SAMPLE_C_D   | SAMPLE_C, with user derivatives.                                            |
| 43     | IMAGE_SAMPLE_C_D_CL| SAMPLE_C, with LOD clamp specified in shader, with user derivatives.       |
| 44     | IMAGE_SAMPLE_C_L   | SAMPLE_C, with user LOD.                                                    |
| 45     | IMAGE_SAMPLE_C_B   | SAMPLE_C, with lod bias.                                                    |
| 46     | IMAGE_SAMPLE_C_B_CL| SAMPLE_C, with LOD clamp specified in shader, with lod bias.               |
| 47     | IMAGE_SAMPLE_C_LZ  | SAMPLE_C, from level 0.                                                     |
| 48     | IMAGE_SAMPLE_O     | sample texture map, with user offsets.                                      |
| 49     | IMAGE_SAMPLE_CL_O  | SAMPLE_0 with LOD clamp specified in shader.                                |
| 50     | IMAGE_SAMPLE_D_O   | SAMPLE_0, with user derivatives.                                            |
| 51     | IMAGE_SAMPLE_D_CL_O| SAMPLE_0, with LOD clamp specified in shader, with user derivatives.       |
| 52     | IMAGE_SAMPLE_L_O   | SAMPLE_0, with user LOD.                                                    |
| 53     | IMAGE_SAMPLE_B_O   | SAMPLE_0, with lod bias.                                                    |</p>
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>IMAGE_SAMPLE_B_CL_O</td>
<td>SAMPLE_0, with LOD clamp specified in shader, with lod bias.</td>
</tr>
<tr>
<td>55</td>
<td>IMAGE_SAMPLE_LZ_O</td>
<td>SAMPLE_0, from level 0.</td>
</tr>
<tr>
<td>56</td>
<td>IMAGE_SAMPLE_C_O</td>
<td>SAMPLE_C with user specified offsets.</td>
</tr>
<tr>
<td>57</td>
<td>IMAGE_SAMPLE_C_CL_O</td>
<td>SAMPLE_C_0, with LOD clamp specified in shader.</td>
</tr>
<tr>
<td>58</td>
<td>IMAGE_SAMPLE_C_D_O</td>
<td>SAMPLE_C_0, with user derivatives.</td>
</tr>
<tr>
<td>59</td>
<td>IMAGE_SAMPLE_C_D_CL_O</td>
<td>SAMPLE_C_0, with LOD clamp specified in shader, with user derivatives.</td>
</tr>
<tr>
<td>60</td>
<td>IMAGE_SAMPLE_C_L_O</td>
<td>SAMPLE_C_0, with user LOD.</td>
</tr>
<tr>
<td>61</td>
<td>IMAGE_SAMPLE_C_B_O</td>
<td>SAMPLE_C_0, with lod bias.</td>
</tr>
<tr>
<td>62</td>
<td>IMAGE_SAMPLE_C_B_CL_O</td>
<td>SAMPLE_C_0, with LOD clamp specified in shader, with lod bias.</td>
</tr>
<tr>
<td>63</td>
<td>IMAGE_SAMPLE_C_LZ_O</td>
<td>SAMPLE_C_0, from level 0.</td>
</tr>
<tr>
<td>64</td>
<td>IMAGE_GATHER4</td>
<td>gather 4 single component elements (2x2).</td>
</tr>
<tr>
<td>65</td>
<td>IMAGE_GATHER4_CL</td>
<td>gather 4 single component elements (2x2) with user LOD clamp.</td>
</tr>
<tr>
<td>66</td>
<td>IMAGE_GATHER4H</td>
<td>Same as Gather4, but fetches one component per texel, from a 4x1 group of texels.</td>
</tr>
<tr>
<td>68</td>
<td>IMAGE_GATHER4_L</td>
<td>gather 4 single component elements (2x2) with user LOD.</td>
</tr>
<tr>
<td>69</td>
<td>IMAGE_GATHER4_B</td>
<td>gather 4 single component elements (2x2) with user bias.</td>
</tr>
<tr>
<td>70</td>
<td>IMAGE_GATHER4_B_CL</td>
<td>gather 4 single component elements (2x2) with user bias and clamp.</td>
</tr>
<tr>
<td>71</td>
<td>IMAGE_GATHER4_LZ</td>
<td>gather 4 single component elements (2x2) at level 0.</td>
</tr>
<tr>
<td>72</td>
<td>IMAGE_GATHER4_C</td>
<td>gather 4 single component elements (2x2) with PCF.</td>
</tr>
<tr>
<td>73</td>
<td>IMAGE_GATHER4_C_CL</td>
<td>gather 4 single component elements (2x2) with user LOD clamp and PCF.</td>
</tr>
<tr>
<td>74</td>
<td>IMAGE_GATHER4H_PCK</td>
<td>Same as GATHER4H, but fetched elements are treated as a single component and packed into GPR(s).</td>
</tr>
<tr>
<td>75</td>
<td>IMAGE_GATHER8H_PCK</td>
<td>Similar to GATHER4H_PCK, but packs eight elements from a 8x1 group of texels.</td>
</tr>
<tr>
<td>76</td>
<td>IMAGE_GATHER4_C_L</td>
<td>gather 4 single component elements (2x2) with user LOD and PCF.</td>
</tr>
<tr>
<td>77</td>
<td>IMAGE_GATHER4_C_B</td>
<td>gather 4 single component elements (2x2) with user bias and PCF.</td>
</tr>
<tr>
<td>78</td>
<td>IMAGE_GATHER4_C_B_CL</td>
<td>gather 4 single component elements (2x2) with user bias, clamp and PCF.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>79</td>
<td>IMAGE_GATHER4_C_LZ</td>
<td>gather 4 single component elements (2x2) at level 0, with PCF.</td>
</tr>
<tr>
<td>80</td>
<td>IMAGE_GATHER4_O</td>
<td>GATHER4, with user offsets.</td>
</tr>
<tr>
<td>81</td>
<td>IMAGE_GATHER4_CL_O</td>
<td>GATHER4_CL, with user offsets.</td>
</tr>
<tr>
<td>84</td>
<td>IMAGE_GATHER4_L_O</td>
<td>GATHER4_L, with user offsets.</td>
</tr>
<tr>
<td>85</td>
<td>IMAGE_GATHER4_B_O</td>
<td>GATHER4_B, with user offsets.</td>
</tr>
<tr>
<td>86</td>
<td>IMAGE_GATHER4_B_CL_O</td>
<td>GATHER4_B_CL, with user offsets.</td>
</tr>
<tr>
<td>87</td>
<td>IMAGE_GATHER4_LZ_O</td>
<td>GATHER4_LZ, with user offsets.</td>
</tr>
<tr>
<td>88</td>
<td>IMAGE_GATHER4_C_O</td>
<td>GATHER4_C, with user offsets.</td>
</tr>
<tr>
<td>89</td>
<td>IMAGE_GATHER4_C_CL_O</td>
<td>GATHER4_C_CL, with user offsets.</td>
</tr>
<tr>
<td>92</td>
<td>IMAGE_GATHER4_C_L_O</td>
<td>GATHER4_C_L, with user offsets.</td>
</tr>
<tr>
<td>93</td>
<td>IMAGE_GATHER4_B_O</td>
<td>GATHER4_B, with user offsets.</td>
</tr>
<tr>
<td>94</td>
<td>IMAGE_GATHER4_B_CL_O</td>
<td>GATHER4_B_CL, with user offsets.</td>
</tr>
<tr>
<td>95</td>
<td>IMAGE_GATHER4_C_LZ_O</td>
<td>GATHER4_C_LZ, with user offsets.</td>
</tr>
<tr>
<td>96</td>
<td>IMAGE_GET_LOD</td>
<td>Return calculated LOD. Vdata gets 2 32bit integer values: { \text{rawLOD}, \text{clampedLOD} }.</td>
</tr>
<tr>
<td>104</td>
<td>IMAGE_SAMPLE_CD</td>
<td>sample texture map, with user derivatives (LOD per quad)</td>
</tr>
<tr>
<td>105</td>
<td>IMAGE_SAMPLE_CD_CL</td>
<td>sample texture map, with LOD clamp specified in shader, with user</td>
</tr>
<tr>
<td></td>
<td></td>
<td>derivatives (LOD per quad).</td>
</tr>
<tr>
<td>106</td>
<td>IMAGE_SAMPLE_C_CD</td>
<td>SAMPLE_C, with user derivatives (LOD per quad).</td>
</tr>
<tr>
<td>107</td>
<td>IMAGE_SAMPLE_C_CD_CL</td>
<td>SAMPLE_C, with LOD clamp specified in shader, with user</td>
</tr>
<tr>
<td></td>
<td></td>
<td>derivatives (LOD per quad).</td>
</tr>
<tr>
<td>108</td>
<td>IMAGE_SAMPLE_CD_O</td>
<td>SAMPLE_0, with user derivatives (LOD per quad).</td>
</tr>
<tr>
<td>109</td>
<td>IMAGE_SAMPLE_CD_CL_O</td>
<td>SAMPLE_0, with LOD clamp specified in shader, with user</td>
</tr>
<tr>
<td></td>
<td></td>
<td>derivatives (LOD per quad).</td>
</tr>
<tr>
<td>110</td>
<td>IMAGESAMPLE_C_CD_O</td>
<td>SAMPLE_C_0, with user derivatives (LOD per quad).</td>
</tr>
<tr>
<td>111</td>
<td>IMAGE_SAMPLE_C_CD_CL_O</td>
<td>SAMPLE_C_0, with LOD clamp specified in shader, with user</td>
</tr>
<tr>
<td></td>
<td></td>
<td>derivatives (LOD per quad).</td>
</tr>
</tbody>
</table>

### 12.17. EXPORT Instructions

Transfer vertex position, vertex parameter, pixel color, or pixel depth information to the output buffer. Every pixel shader must do at least one export to a color, depth or NULL target with the VM bit set to 1. This communicates the pixel-valid mask to the color and depth buffers. Every pixel does only one of the above export types with the DONE bit set to 1. Vertex shaders must do one or more position exports, and at least one parameter export. The final position export
must have the DONE bit set to 1.

### 12.18. FLAT, Scratch and Global Instructions

The bitfield map of the FLAT format is:

![Bitfield Map of FLAT Format]

where:

- **GLC** = Global coherency.
- **SLC** = System level coherency.
- **OP** = Opcode instructions.
- **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.

### 12.18.1. 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.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>FLAT_LOAD_UBYTE</td>
<td>Untyped buffer load unsigned byte (zero extend to VGPR destination).</td>
</tr>
<tr>
<td>17</td>
<td>FLAT_LOAD_SBYTE</td>
<td>Untyped buffer load signed byte (sign extend to VGPR destination).</td>
</tr>
<tr>
<td>18</td>
<td>FLAT_LOAD_USHORT</td>
<td>Untyped buffer load unsigned short (zero extend to VGPR destination).</td>
</tr>
<tr>
<td>19</td>
<td>FLAT_LOAD_SSHORT</td>
<td>Untyped buffer load signed short (sign extend to VGPR destination).</td>
</tr>
<tr>
<td>20</td>
<td>FLAT_LOAD_DWORD</td>
<td>Untyped buffer load dword.</td>
</tr>
<tr>
<td>21</td>
<td>FLAT_LOAD_DWORDX2</td>
<td>Untyped buffer load 2 dwords.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>22</td>
<td>FLAT_LOAD_DWORDX3</td>
<td>Untyped buffer load 3 dwords.</td>
</tr>
<tr>
<td>23</td>
<td>FLAT_LOAD_DWORDX4</td>
<td>Untyped buffer load 4 dwords.</td>
</tr>
<tr>
<td>24</td>
<td>FLAT_STORE_BYTE</td>
<td>Untyped buffer store byte. Stores $S0[7:0]$.</td>
</tr>
<tr>
<td>26</td>
<td>FLAT_STORE_SHORT</td>
<td>Untyped buffer store short. Stores $S0[15:0]$.</td>
</tr>
<tr>
<td>28</td>
<td>FLAT_STORE_DWORD</td>
<td>Untyped buffer store dword.</td>
</tr>
<tr>
<td>29</td>
<td>FLAT_STORE_DWORDX2</td>
<td>Untyped buffer store 2 dwords.</td>
</tr>
<tr>
<td>30</td>
<td>FLAT_STORE_DWORDX3</td>
<td>Untyped buffer store 3 dwords.</td>
</tr>
<tr>
<td>31</td>
<td>FLAT_STORE_DWORDX4</td>
<td>Untyped buffer store 4 dwords.</td>
</tr>
<tr>
<td>32</td>
<td>FLAT_LOAD_UBYTE_D16</td>
<td>$D0[15:0] = {8'h0, MEM[ADDR]}$.</td>
</tr>
<tr>
<td>33</td>
<td>FLAT_LOAD_UBYTE_D16_HI</td>
<td>$D0[31:16] = {8'h0, MEM[ADDR]}$.</td>
</tr>
<tr>
<td>34</td>
<td>FLAT_LOAD_SBYTE_D16</td>
<td>$D0[15:0] = {8'h0, MEM[ADDR]}$.</td>
</tr>
<tr>
<td>35</td>
<td>FLAT_LOAD_SBYTE_D16_HI</td>
<td>$D0[31:16] = {8'h0, MEM[ADDR]}$.</td>
</tr>
<tr>
<td>36</td>
<td>FLAT_LOAD_SHORT_D16</td>
<td>$D0[15:0] = MEM[ADDR]$.</td>
</tr>
<tr>
<td>37</td>
<td>FLAT_LOAD_SHORT_D16_HI</td>
<td>$D0[31:16] = MEM[ADDR]$.</td>
</tr>
<tr>
<td>64</td>
<td>FLAT_ATOMIC_SWAP</td>
<td>// 32bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] = DATA;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA = tmp.</td>
</tr>
<tr>
<td>65</td>
<td>FLAT_ATOMIC_CMP_SWAP</td>
<td>// 32bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>src = DATA[0];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cmp = DATA[1];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] = (tmp == cmp) ? src : tmp;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA[0] = tmp.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| 66     | FLAT_ATOMIC_ADD   | // 32bit  
|        |                   | tmp = MEM[ADDR];  
|        |                   | MEM[ADDR] += DATA;  
|        |                   | RETURN_DATA = tmp.                                                   |
| 67     | FLAT_ATOMIC_SUB   | // 32bit  
|        |                   | tmp = MEM[ADDR];  
|        |                   | MEM[ADDR] -= DATA;  
|        |                   | RETURN_DATA = tmp.                                                   |
| 68     | FLAT_ATOMIC_SMIN  | // 32bit  
|        |                   | tmp = MEM[ADDR];  
|        |                   | MEM[ADDR] = (DATA < tmp) ? DATA : tmp; // signed compare  
|        |                   | RETURN_DATA = tmp.                                                   |
| 69     | FLAT_ATOMIC_UMIN  | // 32bit  
|        |                   | tmp = MEM[ADDR];  
|        |                   | MEM[ADDR] = (DATA < tmp) ? DATA : tmp; // unsigned compare  
|        |                   | RETURN_DATA = tmp.                                                   |
| 70     | FLAT_ATOMIC_SMAX  | // 32bit  
|        |                   | tmp = MEM[ADDR];  
|        |                   | MEM[ADDR] = (DATA > tmp) ? DATA : tmp; // signed compare  
|        |                   | RETURN_DATA = tmp.                                                   |
| 71     | FLAT_ATOMIC_UMAX  | // 32bit  
|        |                   | tmp = MEM[ADDR];  
|        |                   | MEM[ADDR] = (DATA > tmp) ? DATA : tmp; // unsigned compare  
|        |                   | RETURN_DATA = tmp.                                                   |
| 72     | FLAT_ATOMIC_AND   | // 32bit  
|        |                   | tmp = MEM[ADDR];  
|        |                   | MEM[ADDR] &= DATA;  
|        |                   | RETURN_DATA = tmp.                                                   |
| 73     | FLAT_ATOMIC_OR    | // 32bit  
|        |                   | tmp = MEM[ADDR];  
|        |                   | MEM[ADDR] |= DATA;  
|        |                   | RETURN_DATA = tmp.                                                   |
| 74     | FLAT_ATOMIC_XOR   | // 32bit  
|        |                   | tmp = MEM[ADDR];  
|        |                   | MEM[ADDR] ^= DATA;  
|        |                   | RETURN_DATA = tmp.                                                   |
| 75     | FLAT_ATOMIC_INC   | // 32bit  
|        |                   | tmp = MEM[ADDR];  
|        |                   | MEM[ADDR] = (tmp >= DATA) ? 0 : tmp + 1; // unsigned compare  
<p>|        |                   | RETURN_DATA = tmp.                                                   |</p>
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| 76     | FLAT_ATOMIC_DEC       | // 32bit
|        |                       | tmp = MEM[ADDR];
|        |                       | MEM[ADDR] = (tmp == 0 || tmp > DATA) ? DATA : tmp - 1; // unsigned compare
|        |                       | RETURN_DATA = tmp. |
| 96     | FLAT_ATOMIC_SWAP_X2   | // 64bit
|        |                       | tmp = MEM[ADDR];
|        |                       | MEM[ADDR] = DATA[0:1];
|        |                       | RETURN_DATA[0:1] = tmp. |
| 97     | FLAT_ATOMIC_CMPSwap_X2| // 64bit
|        |                       | tmp = MEM[ADDR];
|        |                       | src = DATA[0:1];
|        |                       | cmp = DATA[2:3];
|        |                       | MEM[ADDR] = (tmp == cmp) ? src : tmp;
|        |                       | RETURN_DATA[0:1] = tmp. |
| 98     | FLAT_ATOMIC_ADD_X2    | // 64bit
|        |                       | tmp = MEM[ADDR];
|        |                       | MEM[ADDR] += DATA[0:1];
|        |                       | RETURN_DATA[0:1] = tmp. |
| 99     | FLAT_ATOMIC_SUB_X2    | // 64bit
|        |                       | tmp = MEM[ADDR];
|        |                       | MEM[ADDR] -= DATA[0:1];
|        |                       | RETURN_DATA[0:1] = tmp. |
| 100    | FLAT_ATOMIC_SMIN_X2   | // 64bit
|        |                       | tmp = MEM[ADDR];
|        |                       | MEM[ADDR] -= (DATA[0:1] < tmp) ? DATA[0:1] : tmp; // signed compare
|        |                       | RETURN_DATA[0:1] = tmp. |
| 101    | FLAT_ATOMIC_UMIN_X2   | // 64bit
|        |                       | tmp = MEM[ADDR];
|        |                       | MEM[ADDR] -= (DATA[0:1] < tmp) ? DATA[0:1] : tmp; // unsigned compare
|        |                       | RETURN_DATA[0:1] = tmp. |
| 102    | FLAT_ATOMIC_SMAX_X2   | // 64bit
|        |                       | tmp = MEM[ADDR];
|        |                       | MEM[ADDR] -= (DATA[0:1] > tmp) ? DATA[0:1] : tmp; // signed compare
|        |                       | RETURN_DATA[0:1] = tmp. |
| 103    | FLAT_ATOMIC_UMAX_X2   | // 64bit
|        |                       | tmp = MEM[ADDR];
|        |                       | MEM[ADDR] -= (DATA[0:1] > tmp) ? DATA[0:1] : tmp; // unsigned compare
|        |                       | RETURN_DATA[0:1] = tmp. |
### 12.18.2. Scratch Instructions

Scratch instructions are like Flat, but assume all workitem addresses fall in scratch (private) space.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>SCRATCH_LOAD_UBYTE</td>
<td>Untyped buffer load unsigned byte (zero extend to VGPR destination).</td>
</tr>
<tr>
<td>17</td>
<td>SCRATCH_LOAD_SBYTE</td>
<td>Untyped buffer load signed byte (sign extend to VGPR destination).</td>
</tr>
<tr>
<td>18</td>
<td>SCRATCH_LOAD_USHORT</td>
<td>Untyped buffer load unsigned short (zero extend to VGPR destination).</td>
</tr>
<tr>
<td>19</td>
<td>SCRATCH_LOAD_SSHORT</td>
<td>Untyped buffer load signed short (sign extend to VGPR destination).</td>
</tr>
<tr>
<td>20</td>
<td>SCRATCH_LOAD_DWORD</td>
<td>Untyped buffer load dword.</td>
</tr>
<tr>
<td>21</td>
<td>SCRATCH_LOAD_DWORDX2</td>
<td>Untyped buffer load 2 dwords.</td>
</tr>
<tr>
<td>22</td>
<td>SCRATCH_LOAD_DWORDX3</td>
<td>Untyped buffer load 3 dwords.</td>
</tr>
<tr>
<td>23</td>
<td>SCRATCH_LOAD_DWORDX4</td>
<td>Untyped buffer load 4 dwords.</td>
</tr>
</tbody>
</table>
12.18.3. Global Instructions

Global instructions are like Flat, but assume all workitem addresses fall in global memory space.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>GLOBAL_LOAD_UBYTE</td>
<td>Untyped buffer load unsigned byte (zero extend to VGPR destination).</td>
</tr>
<tr>
<td>17</td>
<td>GLOBAL_LOAD_SBYTE</td>
<td>Untyped buffer load signed byte (sign extend to VGPR destination).</td>
</tr>
<tr>
<td>18</td>
<td>GLOBAL_LOAD_USHORT</td>
<td>Untyped buffer load unsigned short (zero extend to VGPR destination).</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>---------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>19</td>
<td>GLOBAL_LOAD_SSHORT</td>
<td>Untyped buffer load signed short (sign extend to VGPR destination).</td>
</tr>
<tr>
<td>20</td>
<td>GLOBAL_LOAD_DWORD</td>
<td>Untyped buffer load dword.</td>
</tr>
<tr>
<td>21</td>
<td>GLOBAL_LOAD_DWORDX2</td>
<td>Untyped buffer load 2 dwords.</td>
</tr>
<tr>
<td>22</td>
<td>GLOBAL_LOAD_DWORDX3</td>
<td>Untyped buffer load 3 dwords.</td>
</tr>
<tr>
<td>23</td>
<td>GLOBAL_LOAD_DWORDX4</td>
<td>Untyped buffer load 4 dwords.</td>
</tr>
<tr>
<td>24</td>
<td>GLOBAL_STORE_BYTE</td>
<td>Untyped buffer store byte. Stores $0[7:0]</td>
</tr>
<tr>
<td>25</td>
<td>GLOBAL_STORE_BYTE_D16_HI</td>
<td>Untyped buffer store byte. Stores $0[23:16].</td>
</tr>
<tr>
<td>26</td>
<td>GLOBAL_STORE_SHORT</td>
<td>Untyped buffer store short. Stores $0[15:0].</td>
</tr>
<tr>
<td>27</td>
<td>GLOBAL_STORE_SHORT_D16_HI</td>
<td>Untyped buffer store short. Stores $0[31:16].</td>
</tr>
<tr>
<td>28</td>
<td>GLOBAL_STORE_DWORD</td>
<td>Untyped buffer store dword.</td>
</tr>
<tr>
<td>29</td>
<td>GLOBAL_STORE_DWORDX2</td>
<td>Untyped buffer store 2 dwords.</td>
</tr>
<tr>
<td>30</td>
<td>GLOBAL_STORE_DWORDX3</td>
<td>Untyped buffer store 3 dwords.</td>
</tr>
<tr>
<td>31</td>
<td>GLOBAL_STORE_DWORDX4</td>
<td>Untyped buffer store 4 dwords.</td>
</tr>
<tr>
<td>32</td>
<td>GLOBAL_LOAD_UBYTE_D16</td>
<td>$0[15:0] = {8'h0, MEM[ADDR]}.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Untyped buffer load unsigned byte.</td>
</tr>
<tr>
<td>33</td>
<td>GLOBAL_LOAD_UBYTE_D16_HI</td>
<td>$0[31:16] = {8'h0, MEM[ADDR]}.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Untyped buffer load unsigned byte.</td>
</tr>
<tr>
<td>34</td>
<td>GLOBAL_LOAD_SBYTE_D16</td>
<td>$0[15:0] = {8'h0, MEM[ADDR]}.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Untyped buffer load signed byte.</td>
</tr>
<tr>
<td>35</td>
<td>GLOBAL_LOAD_SBYTE_D16_HI</td>
<td>$0[31:16] = {8'h0, MEM[ADDR]}.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Untyped buffer load signed byte.</td>
</tr>
<tr>
<td>36</td>
<td>GLOBAL_LOAD_SHORT_D16</td>
<td>$0[15:0] = MEM[ADDR].</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Untyped buffer load short.</td>
</tr>
<tr>
<td>37</td>
<td>GLOBAL_LOAD_SHORT_D16_HI</td>
<td>$0[31:16] = MEM[ADDR].</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Untyped buffer load short.</td>
</tr>
<tr>
<td>64</td>
<td>GLOBAL_ATOMIC_SWAP</td>
<td>// 32bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] = DATA;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA = tmp.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>65</td>
<td>GLOBAL_ATOMIC_CMPSWAP</td>
<td>// 32bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR]; src = DATA[0]; cmp = DATA[1];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] = (tmp == cmp) ? src : tmp;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA[0] = tmp.</td>
</tr>
<tr>
<td>66</td>
<td>GLOBAL_ATOMIC_ADD</td>
<td>// 32bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] += DATA;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA = tmp.</td>
</tr>
<tr>
<td>67</td>
<td>GLOBAL_ATOMIC_SUB</td>
<td>// 32bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] -= DATA;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA = tmp.</td>
</tr>
<tr>
<td>68</td>
<td>GLOBAL_ATOMIC_SMIN</td>
<td>// 32bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] = (DATA &lt; tmp) ? DATA : tmp; // signed compare</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA = tmp.</td>
</tr>
<tr>
<td>69</td>
<td>GLOBAL_ATOMIC_UMIN</td>
<td>// 32bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] = (DATA &lt; tmp) ? DATA : tmp; // unsigned compare</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA = tmp.</td>
</tr>
<tr>
<td>70</td>
<td>GLOBAL_ATOMIC_SMAX</td>
<td>// 32bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] = (DATA &gt; tmp) ? DATA : tmp; // signed compare</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA = tmp.</td>
</tr>
<tr>
<td>71</td>
<td>GLOBAL_ATOMIC_UMAX</td>
<td>// 32bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] = (DATA &gt; tmp) ? DATA : tmp; // unsigned compare</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA = tmp.</td>
</tr>
<tr>
<td>72</td>
<td>GLOBAL_ATOMIC_AND</td>
<td>// 32bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] &amp;= DATA;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA = tmp.</td>
</tr>
<tr>
<td>73</td>
<td>GLOBAL_ATOMIC_OR</td>
<td>// 32bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA = tmp.</td>
</tr>
<tr>
<td>74</td>
<td>GLOBAL_ATOMIC_XOR</td>
<td>// 32bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] ^= DATA;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA = tmp.</td>
</tr>
<tr>
<td>Opcode</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| 75     | GLOBAL_ATOMIC_INC    | // 32bit  
|       |                       | tmp = MEM[ADDR];  
|       |                       | MEM[ADDR] = (tmp >= DATA) ? 0 : tmp + 1; // unsigned compare  
|       |                       | RETURN_DATA = tmp.                                                                                                                         |
| 76     | GLOBAL_ATOMIC_DEC    | // 32bit  
|       |                       | tmp = MEM[ADDR];  
|       |                       | MEM[ADDR] = (tmp == 0 || tmp > DATA) ? DATA : tmp - 1; // unsigned compare  
|       |                       | RETURN_DATA = tmp.                                                                                                                         |
| 96     | GLOBAL_ATOMIC_SWAP_X2| // 64bit  
|       |                       | tmp = MEM[ADDR];  
|       |                       | MEM[ADDR] = DATA[0:1];  
|       |                       | RETURN_DATA[0:1] = tmp.                                                                                                                   |
| 97     | GLOBAL_ATOMIC_CMPSWAP_X2 | // 64bit  
|       |                       | tmp = MEM[ADDR];  
|       |                       | src = DATA[0:1];  
|       |                       | cmp = DATA[2:3];  
|       |                       | MEM[ADDR] = (tmp == cmp) ? src : tmp;  
|       |                       | RETURN_DATA[0:1] = tmp.                                                                                                                   |
| 98     | GLOBAL_ATOMIC_ADD_X2  | // 64bit  
|       |                       | tmp = MEM[ADDR];  
|       |                       | MEM[ADDR] += DATA[0:1];  
|       |                       | RETURN_DATA[0:1] = tmp.                                                                                                                   |
| 99     | GLOBAL_ATOMIC_SUB_X2  | // 64bit  
|       |                       | tmp = MEM[ADDR];  
|       |                       | MEM[ADDR] -= DATA[0:1];  
|       |                       | RETURN_DATA[0:1] = tmp.                                                                                                                   |
| 100    | GLOBAL_ATOMIC_SMIN_X2 | // 64bit  
|       |                       | tmp = MEM[ADDR];  
|       |                       | MEM[ADDR] -= (DATA[0:1] < tmp) ? DATA[0:1] : tmp; // signed compare  
|       |                       | RETURN_DATA[0:1] = tmp.                                                                                                                   |
| 101    | GLOBAL_ATOMIC_UMIN_X2 | // 64bit  
|       |                       | tmp = MEM[ADDR];  
|       |                       | MEM[ADDR] -= (DATA[0:1] < tmp) ? DATA[0:1] : tmp; // unsigned compare  
|       |                       | RETURN_DATA[0:1] = tmp.                                                                                                                   |
| 102    | GLOBAL_ATOMIC_SMAX_X2 | // 64bit  
|       |                       | tmp = MEM[ADDR];  
|       |                       | MEM[ADDR] -= (DATA[0:1] > tmp) ? DATA[0:1] : tmp; // signed compare  
<p>|       |                       | RETURN_DATA[0:1] = tmp.                                                                                                                   |</p>
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>103</td>
<td>GLOBAL_ATOMIC_UMAX_X2</td>
<td>// 64bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] -= (DATA[0:1] &gt; tmp) ? DATA[0:1] : tmp; //</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned compare</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA[0:1] = tmp.</td>
</tr>
<tr>
<td>104</td>
<td>GLOBAL_ATOMIC_AND_X2</td>
<td>// 64bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] &amp;= DATA[0:1];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA[0:1] = tmp.</td>
</tr>
<tr>
<td>105</td>
<td>GLOBAL_ATOMIC_OR_X2</td>
<td>// 64bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA[0:1] = tmp.</td>
</tr>
<tr>
<td>106</td>
<td>GLOBAL_ATOMIC_XOR_X2</td>
<td>// 64bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] ^= DATA[0:1];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA[0:1] = tmp.</td>
</tr>
<tr>
<td>107</td>
<td>GLOBAL_ATOMIC_INC_X2</td>
<td>// 64bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] = (tmp &gt;= DATA[0:1]) ? 0 : tmp + 1; //</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned compare</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA[0:1] = tmp.</td>
</tr>
<tr>
<td>108</td>
<td>GLOBAL_ATOMIC_DEC_X2</td>
<td>// 64bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp = MEM[ADDR];</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MEM[ADDR] = (tmp == 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tmp - 1; // unsigned compare</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RETURN_DATA[0:1] = tmp.</td>
</tr>
</tbody>
</table>

### 12.19. Instruction Limitations

#### 12.19.1. 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

12.19.2. 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 GCN 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>
<thead>
<tr>
<th>Microcode Formats</th>
<th>Reference</th>
<th>Width (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scalar ALU and Control Formats</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SOP2</td>
<td>SOP2</td>
<td>32</td>
</tr>
<tr>
<td>SOP1</td>
<td>SOP1</td>
<td></td>
</tr>
<tr>
<td>SOPK</td>
<td>SOPK</td>
<td></td>
</tr>
<tr>
<td>SOPP</td>
<td>SOPP</td>
<td></td>
</tr>
<tr>
<td>SOPC</td>
<td>SOPC</td>
<td></td>
</tr>
<tr>
<td><strong>Scalar Memory Format</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SMEM</td>
<td>SMEM</td>
<td>64</td>
</tr>
<tr>
<td><strong>Vector ALU Format</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VOP1</td>
<td>VOP1</td>
<td>32</td>
</tr>
<tr>
<td>VOP2</td>
<td>VOP2</td>
<td>32</td>
</tr>
<tr>
<td>VOPC</td>
<td>VOPC</td>
<td>32</td>
</tr>
<tr>
<td>VOP3A</td>
<td>VOP3A</td>
<td>64</td>
</tr>
<tr>
<td>VOP3B</td>
<td>VOP3B</td>
<td>64</td>
</tr>
<tr>
<td>VOP3P</td>
<td>VOP3P</td>
<td>64</td>
</tr>
<tr>
<td>DPP</td>
<td>DPP</td>
<td>32</td>
</tr>
<tr>
<td>SDWA</td>
<td>VOP2</td>
<td>32</td>
</tr>
<tr>
<td><strong>Vector Parameter Interpolation Format</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VINTRP</td>
<td>VINTRP</td>
<td>32</td>
</tr>
<tr>
<td><strong>LDS/GDS Format</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DS</td>
<td>DS</td>
<td>64</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>SOP2</th>
<th>31</th>
<th>0</th>
<th>OP7</th>
<th>SDST7</th>
<th>SSRC1s</th>
<th>SSRC0s</th>
</tr>
</thead>
</table>

**Format**  
SOP2

**Description**  
This is a scalar instruction with two inputs and one output. Can be followed by a 32-bit literal constant.

*Table 53. SOP2 Fields*
<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bits</th>
<th>Format or Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSRC0</td>
<td>[7:0]</td>
<td>Source 0. First operand for the instruction.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SGPR0 to SGPR101: Scalar general-purpose registers.</td>
</tr>
<tr>
<td></td>
<td>0 - 101</td>
<td>FLAT_SCRATCH_LO.</td>
</tr>
<tr>
<td></td>
<td>102</td>
<td>FLAT_SCRATCH_HI.</td>
</tr>
<tr>
<td></td>
<td>103</td>
<td>XNACK_MASK_LO.</td>
</tr>
<tr>
<td></td>
<td>104</td>
<td>XNACK_MASK_HI.</td>
</tr>
<tr>
<td></td>
<td>105</td>
<td>VCC_LO: vcc[31:0].</td>
</tr>
<tr>
<td></td>
<td>106</td>
<td>VCC_HI: vcc[63:32].</td>
</tr>
<tr>
<td></td>
<td>107</td>
<td>108-123 TTMP0 - TTMP15: Trap handler temporary register.</td>
</tr>
<tr>
<td></td>
<td>124</td>
<td>M0. Memory register 0.</td>
</tr>
<tr>
<td></td>
<td>125</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>129-192</td>
<td>Signed integer 1 to 64.</td>
</tr>
<tr>
<td></td>
<td>193-208</td>
<td>Signed integer -1 to -16.</td>
</tr>
<tr>
<td></td>
<td>209-234</td>
<td>Reserved.</td>
</tr>
<tr>
<td></td>
<td>235</td>
<td>SHARED_BASE (Memory Aperture definition).</td>
</tr>
<tr>
<td></td>
<td>236</td>
<td>SHARED_LIMIT (Memory Aperture definition).</td>
</tr>
<tr>
<td></td>
<td>237</td>
<td>PRIVATE_BASE (Memory Aperture definition).</td>
</tr>
<tr>
<td></td>
<td>238</td>
<td>PRIVATE_LIMIT (Memory Aperture definition).</td>
</tr>
<tr>
<td></td>
<td>239</td>
<td>POPS_EXITING_WAVE_ID.</td>
</tr>
<tr>
<td></td>
<td>240</td>
<td>0.5.</td>
</tr>
<tr>
<td></td>
<td>241</td>
<td>-0.5.</td>
</tr>
<tr>
<td></td>
<td>242</td>
<td>1.0.</td>
</tr>
<tr>
<td></td>
<td>243</td>
<td>-1.0.</td>
</tr>
<tr>
<td></td>
<td>244</td>
<td>2.0.</td>
</tr>
<tr>
<td></td>
<td>245</td>
<td>-2.0.</td>
</tr>
<tr>
<td></td>
<td>246</td>
<td>4.0.</td>
</tr>
<tr>
<td></td>
<td>247</td>
<td>-4.0.</td>
</tr>
<tr>
<td></td>
<td>248</td>
<td>1/(2*PI).</td>
</tr>
<tr>
<td></td>
<td>249 - 250</td>
<td>Reserved.</td>
</tr>
<tr>
<td></td>
<td>251</td>
<td>VCCZ.</td>
</tr>
<tr>
<td></td>
<td>252</td>
<td>EXECZ.</td>
</tr>
<tr>
<td></td>
<td>253</td>
<td>SCC.</td>
</tr>
<tr>
<td></td>
<td>254</td>
<td>Reserved.</td>
</tr>
<tr>
<td></td>
<td>255</td>
<td>Literal constant.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Same codes as SSRC0, above.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Same codes as SSRC0, above except only codes 0-127 are valid.</td>
</tr>
<tr>
<td>ENCODING</td>
<td>[31:30]</td>
<td>Must be: 10</td>
</tr>
</tbody>
</table>

**Table 54. SOP2 Opcodes**

<table>
<thead>
<tr>
<th>Opcode #</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S_ADD_U32</td>
</tr>
<tr>
<td>Opcode #</td>
<td>Name</td>
</tr>
<tr>
<td>----------</td>
<td>--------------</td>
</tr>
<tr>
<td>1</td>
<td>S_SUB_U32</td>
</tr>
<tr>
<td>2</td>
<td>S_ADD_I32</td>
</tr>
<tr>
<td>3</td>
<td>S_SUB_I32</td>
</tr>
<tr>
<td>4</td>
<td>S_ADDC_U32</td>
</tr>
<tr>
<td>5</td>
<td>S_SUBB_U32</td>
</tr>
<tr>
<td>6</td>
<td>S_MIN_I32</td>
</tr>
<tr>
<td>7</td>
<td>S_MIN_U32</td>
</tr>
<tr>
<td>8</td>
<td>S_MAX_I32</td>
</tr>
<tr>
<td>9</td>
<td>S_MAX_U32</td>
</tr>
<tr>
<td>10</td>
<td>S_CSELECT_B32</td>
</tr>
<tr>
<td>11</td>
<td>S_CSELECT_B64</td>
</tr>
<tr>
<td>12</td>
<td>S_AND_B32</td>
</tr>
<tr>
<td>13</td>
<td>S_AND_B64</td>
</tr>
<tr>
<td>14</td>
<td>S_OR_B32</td>
</tr>
<tr>
<td>15</td>
<td>S_OR_B64</td>
</tr>
<tr>
<td>16</td>
<td>S_XOR_B32</td>
</tr>
<tr>
<td>17</td>
<td>S_XOR_B64</td>
</tr>
<tr>
<td>18</td>
<td>S_ANDN2_B32</td>
</tr>
<tr>
<td>19</td>
<td>S_ANDN2_B64</td>
</tr>
<tr>
<td>20</td>
<td>S_ORN2_B32</td>
</tr>
<tr>
<td>21</td>
<td>S_ORN2_B64</td>
</tr>
<tr>
<td>22</td>
<td>S_NAND_B32</td>
</tr>
<tr>
<td>23</td>
<td>S_NAND_B64</td>
</tr>
<tr>
<td>24</td>
<td>S_NOR_B32</td>
</tr>
<tr>
<td>25</td>
<td>S_NOR_B64</td>
</tr>
<tr>
<td>26</td>
<td>S_XNOR_B32</td>
</tr>
<tr>
<td>27</td>
<td>S_XNOR_B64</td>
</tr>
<tr>
<td>28</td>
<td>S_LSHL_B32</td>
</tr>
<tr>
<td>29</td>
<td>S_LSHL_B64</td>
</tr>
<tr>
<td>30</td>
<td>S_LSHR_B32</td>
</tr>
<tr>
<td>31</td>
<td>S_LSHR_B64</td>
</tr>
<tr>
<td>32</td>
<td>S_ASHR_I32</td>
</tr>
<tr>
<td>33</td>
<td>S_ASHR_I64</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>Field Name</th>
<th>Bits</th>
<th>Format or Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMM16</td>
<td>[15:0]</td>
<td>Signed immediate 16-bit value.</td>
</tr>
</tbody>
</table>
### Table 56. SOPK Opcodes

<table>
<thead>
<tr>
<th>Opcode #</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S_MOVK_I32</td>
</tr>
<tr>
<td>1</td>
<td>S_CMOVK_I32</td>
</tr>
<tr>
<td>2</td>
<td>S_CMPK_EQ_I32</td>
</tr>
<tr>
<td>3</td>
<td>S_CMPK_LT_I32</td>
</tr>
<tr>
<td>4</td>
<td>S_CMPK_GT_I32</td>
</tr>
<tr>
<td>5</td>
<td>S_CMPK_GE_I32</td>
</tr>
<tr>
<td>6</td>
<td>S_CMPK_LT_I32</td>
</tr>
<tr>
<td>7</td>
<td>S_CMPK_LE_I32</td>
</tr>
<tr>
<td>8</td>
<td>S_CMPK_EQ_U32</td>
</tr>
<tr>
<td>9</td>
<td>S_CMPK_GT_U32</td>
</tr>
<tr>
<td>10</td>
<td>S_CMPK_GE_U32</td>
</tr>
<tr>
<td>11</td>
<td>S_CMPK_LT_U32</td>
</tr>
<tr>
<td>12</td>
<td>S_CMPK_LE_U32</td>
</tr>
<tr>
<td>13</td>
<td>S_ADDK_I32</td>
</tr>
<tr>
<td>15</td>
<td>S_MULK_I32</td>
</tr>
<tr>
<td>17</td>
<td>S_GETREG_B32</td>
</tr>
<tr>
<td>18</td>
<td>S_SETREG_B32</td>
</tr>
<tr>
<td>20</td>
<td>S_SETREG_IMM32_B32</td>
</tr>
</tbody>
</table>
13.1.3. SOP1

This is a scalar instruction with two inputs and one output. Can be followed by a 32-bit literal constant.

Table 57. SOP1 Fields
<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bits</th>
<th>Format or Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDST</td>
<td>[22:16]</td>
<td>Scalar destination. Same codes as SSRC0, above except only codes 0-127 are valid.</td>
</tr>
<tr>
<td>ENCODING</td>
<td>[31:23]</td>
<td>Must be: 10_1111101</td>
</tr>
</tbody>
</table>

**Table 58. SOP1 Opcodes**

<table>
<thead>
<tr>
<th>Opcode #</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S_MOV_B32</td>
</tr>
<tr>
<td>1</td>
<td>S_MOV_B64</td>
</tr>
<tr>
<td>2</td>
<td>S_CMOV_B32</td>
</tr>
<tr>
<td>Opcode #</td>
<td>Name</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>3</td>
<td>S_CMOV_B64</td>
</tr>
<tr>
<td>4</td>
<td>S_NOT_B32</td>
</tr>
<tr>
<td>5</td>
<td>S_NOT_B64</td>
</tr>
<tr>
<td>6</td>
<td>S_WQM_B32</td>
</tr>
<tr>
<td>7</td>
<td>S_WQM_B64</td>
</tr>
<tr>
<td>8</td>
<td>S_BREV_B32</td>
</tr>
<tr>
<td>9</td>
<td>S_BREV_B64</td>
</tr>
<tr>
<td>10</td>
<td>S_BCNT0_I32_B32</td>
</tr>
<tr>
<td>11</td>
<td>S_BCNT0_I32_B64</td>
</tr>
<tr>
<td>12</td>
<td>S_BCNT1_I32_B32</td>
</tr>
<tr>
<td>13</td>
<td>S_BCNT1_I32_B64</td>
</tr>
<tr>
<td>14</td>
<td>S_FF0_I32_B32</td>
</tr>
<tr>
<td>15</td>
<td>S_FF0_I32_B64</td>
</tr>
<tr>
<td>16</td>
<td>S_FF1_I32_B32</td>
</tr>
<tr>
<td>17</td>
<td>S_FF1_I32_B64</td>
</tr>
<tr>
<td>18</td>
<td>S_FLBIT_I32_B32</td>
</tr>
<tr>
<td>19</td>
<td>S_FLBIT_I32_B64</td>
</tr>
<tr>
<td>20</td>
<td>S_FLBIT_I32</td>
</tr>
<tr>
<td>21</td>
<td>S_FLBIT_I32_I64</td>
</tr>
<tr>
<td>22</td>
<td>S_SEXT_I32_I8</td>
</tr>
<tr>
<td>23</td>
<td>S_SEXT_I32_I16</td>
</tr>
<tr>
<td>24</td>
<td>S_BITSET0_B32</td>
</tr>
<tr>
<td>25</td>
<td>S_BITSET0_B64</td>
</tr>
<tr>
<td>26</td>
<td>S_BITSET1_B32</td>
</tr>
<tr>
<td>27</td>
<td>S_BITSET1_B64</td>
</tr>
<tr>
<td>28</td>
<td>S_GETPC_B64</td>
</tr>
<tr>
<td>29</td>
<td>S_SETPC_B64</td>
</tr>
<tr>
<td>30</td>
<td>S_SWAPPC_B64</td>
</tr>
<tr>
<td>31</td>
<td>S_RFE_B64</td>
</tr>
<tr>
<td>32</td>
<td>S_AND_SAVEEXEC_B64</td>
</tr>
<tr>
<td>33</td>
<td>S_OR_SAVEEXEC_B64</td>
</tr>
<tr>
<td>34</td>
<td>S_XOR_SAVEEXEC_B64</td>
</tr>
<tr>
<td>35</td>
<td>S_ANDN2_SAVEEXEC_B64</td>
</tr>
<tr>
<td>Opcode #</td>
<td>Name</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>36</td>
<td>S_ORN2_SAVEEXEC_B64</td>
</tr>
<tr>
<td>37</td>
<td>S_NAND_SAVEEXEC_B64</td>
</tr>
<tr>
<td>38</td>
<td>S_NOR_SAVEEXEC_B64</td>
</tr>
<tr>
<td>39</td>
<td>S_XNOR_SAVEEXEC_B64</td>
</tr>
<tr>
<td>40</td>
<td>S_QUADMASK_B32</td>
</tr>
<tr>
<td>41</td>
<td>S_QUADMASK_B64</td>
</tr>
<tr>
<td>42</td>
<td>S_MOVRELS_B32</td>
</tr>
<tr>
<td>43</td>
<td>S_MOVRELS_B64</td>
</tr>
<tr>
<td>44</td>
<td>S_MOVRELD_B32</td>
</tr>
<tr>
<td>45</td>
<td>S_MOVRELD_B64</td>
</tr>
<tr>
<td>46</td>
<td>S_CBRANCH_JOIN</td>
</tr>
<tr>
<td>48</td>
<td>S_ABS_I32</td>
</tr>
<tr>
<td>50</td>
<td>S_SET_GPR_IDX_IDX</td>
</tr>
<tr>
<td>51</td>
<td>S_ANDN1_SAVEEXEC_B64</td>
</tr>
<tr>
<td>52</td>
<td>S_ORN1_SAVEEXEC_B64</td>
</tr>
<tr>
<td>53</td>
<td>S_ANDN1_WREXEC_B64</td>
</tr>
<tr>
<td>54</td>
<td>S_ANDN2_WREXEC_B64</td>
</tr>
<tr>
<td>55</td>
<td>S_BITREPLICATE_B64_B32</td>
</tr>
</tbody>
</table>

### 13.1.4. SOPC

**Format**

SOPC

**Description**

This is a scalar instruction with two inputs which are compared and produce SCC as a result. Can be followed by a 32-bit literal constant.

*Table 59. SOPC Fields*
<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bits</th>
<th>Format or Description</th>
</tr>
</thead>
</table>
| 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    | POPS_EXITING_WAVE_ID .  
|            | 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.  
|            |        | Same codes as SSRC0, above.  
| ENCODING   | [31:23] | Must be: 10_1111110  

Table 60. SOPC Opcodes

<table>
<thead>
<tr>
<th>Opcode #</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S_CMP_EQ_I32</td>
</tr>
<tr>
<td>1</td>
<td>S_CMP_LG_I32</td>
</tr>
<tr>
<td>2</td>
<td>S_CMP_GT_I32</td>
</tr>
</tbody>
</table>

13.1. Scalar ALU and Control Formats
<table>
<thead>
<tr>
<th>Opcode #</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>S_CMP_GE_I32</td>
</tr>
<tr>
<td>4</td>
<td>S_CMP_LT_I32</td>
</tr>
<tr>
<td>5</td>
<td>S_CMP_LE_I32</td>
</tr>
<tr>
<td>6</td>
<td>S_CMP_EQ_U32</td>
</tr>
<tr>
<td>7</td>
<td>S_CMP_LG_U32</td>
</tr>
<tr>
<td>8</td>
<td>S_CMP_GT_U32</td>
</tr>
<tr>
<td>9</td>
<td>S_CMP_GE_U32</td>
</tr>
<tr>
<td>10</td>
<td>S_CMP_LT_U32</td>
</tr>
<tr>
<td>11</td>
<td>S_CMP_LE_U32</td>
</tr>
<tr>
<td>12</td>
<td>S_BITCMP0_B32</td>
</tr>
<tr>
<td>13</td>
<td>S_BITCMP1_B32</td>
</tr>
<tr>
<td>14</td>
<td>S_BITCMP0_B64</td>
</tr>
<tr>
<td>15</td>
<td>S_BITCMP1_B64</td>
</tr>
<tr>
<td>16</td>
<td>S_SETVSKIP</td>
</tr>
<tr>
<td>17</td>
<td>S_SET_GPR_IDX_ON</td>
</tr>
<tr>
<td>18</td>
<td>S_CMP_EQ_U64</td>
</tr>
<tr>
<td>19</td>
<td>S_CMP_LG_U64</td>
</tr>
</tbody>
</table>

### 13.1.5. SOPP

**Format**  
SOPP

**Description**  
This is a scalar instruction with one 16-bit signed immediate (SIMM16) input.

**Table 61. SOPP Fields**

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bits</th>
<th>Format or Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMM16</td>
<td>[15:0]</td>
<td>Signed immediate 16-bit value.</td>
</tr>
<tr>
<td>ENCODING</td>
<td>[31:23]</td>
<td>Must be: 10_111111</td>
</tr>
</tbody>
</table>

**Table 62. SOPP Opcodes**
<table>
<thead>
<tr>
<th>Opcode #</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S_NOP</td>
</tr>
<tr>
<td>1</td>
<td>S_ENDPGM</td>
</tr>
<tr>
<td>2</td>
<td>S_BRANCH</td>
</tr>
<tr>
<td>3</td>
<td>S_WAKEUP</td>
</tr>
<tr>
<td>4</td>
<td>S_CBRANCH_SCC0</td>
</tr>
<tr>
<td>5</td>
<td>S_CBRANCH_SCC1</td>
</tr>
<tr>
<td>6</td>
<td>S_CBRANCH_VCCZ</td>
</tr>
<tr>
<td>7</td>
<td>S_CBRANCH_VCCNZ</td>
</tr>
<tr>
<td>8</td>
<td>S_CBRANCH_EXECZ</td>
</tr>
<tr>
<td>9</td>
<td>S_CBRANCH_EXECH</td>
</tr>
<tr>
<td>10</td>
<td>S_BARRIER</td>
</tr>
<tr>
<td>11</td>
<td>S_SETKILL</td>
</tr>
<tr>
<td>12</td>
<td>S_WAITCNT</td>
</tr>
<tr>
<td>13</td>
<td>S_SETHALT</td>
</tr>
<tr>
<td>14</td>
<td>S_SLEEP</td>
</tr>
<tr>
<td>15</td>
<td>S_SETPRIO</td>
</tr>
<tr>
<td>16</td>
<td>S_SENDMSG</td>
</tr>
<tr>
<td>17</td>
<td>S_SENDMSGHALT</td>
</tr>
<tr>
<td>18</td>
<td>S_TRAP</td>
</tr>
<tr>
<td>19</td>
<td>S_ICACHE_INV</td>
</tr>
<tr>
<td>20</td>
<td>S_INCPERFLEVEL</td>
</tr>
<tr>
<td>21</td>
<td>S_DECPERFLEVEL</td>
</tr>
<tr>
<td>22</td>
<td>S_TTRACEDATA</td>
</tr>
<tr>
<td>23</td>
<td>S_CBRANCH_CDBGSYS</td>
</tr>
<tr>
<td>24</td>
<td>S_CBRANCH_CDBGUSER</td>
</tr>
<tr>
<td>25</td>
<td>S_CBRANCH_CDBGSYS_OR_USER</td>
</tr>
<tr>
<td>26</td>
<td>S_CBRANCH_CDBGSYS_AND_USER</td>
</tr>
<tr>
<td>27</td>
<td>S_ENDPGM_SAVED</td>
</tr>
<tr>
<td>28</td>
<td>S_SET_GPR_IDX_OFF</td>
</tr>
<tr>
<td>29</td>
<td>S_SET_GPR_IDX_MODE</td>
</tr>
<tr>
<td>30</td>
<td>S_ENDPGM_ORDERED_PS_DONE</td>
</tr>
</tbody>
</table>
13.2. Scalar Memory Format

13.2.1. SMEM

Table 63. SMEM Fields

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bits</th>
<th>Format or Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBASE</td>
<td>[5:0]</td>
<td>SGPR-pair which provides base address or SGPR-quad which provides V#. (LSB of SGPR address is omitted).</td>
</tr>
<tr>
<td>SDATA</td>
<td>[12:6]</td>
<td>SGPR which provides write data or accepts return data.</td>
</tr>
<tr>
<td>NV</td>
<td>[15]</td>
<td>Non-volatile</td>
</tr>
<tr>
<td>GLC</td>
<td>[16]</td>
<td>Globally memory Coherent. Force bypass of L1 cache, or for atomics, cause pre-op value to be returned.</td>
</tr>
<tr>
<td>IMM</td>
<td>[17]</td>
<td>Immediate enable.</td>
</tr>
<tr>
<td>ENCODING</td>
<td>[31:26]</td>
<td>Must be: 110000</td>
</tr>
<tr>
<td>OFFSET</td>
<td>[52:32]</td>
<td>An immediate signed byte offset, or the address of an SGPR holding the unsigned byte offset. Signed offsets only work with S_LOAD/STORE.</td>
</tr>
<tr>
<td>SOFFSET</td>
<td>[63:57]</td>
<td>SGPR offset. Used only when SOFFSET_EN = 1 May only specify an SGPR or M0.</td>
</tr>
</tbody>
</table>

Table 64. SMEM Opcodes

<table>
<thead>
<tr>
<th>Opcode #</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S_LOAD_DWORD</td>
</tr>
<tr>
<td>1</td>
<td>S_LOAD_DWORDX2</td>
</tr>
<tr>
<td>2</td>
<td>S_LOAD_DWORDX4</td>
</tr>
<tr>
<td>3</td>
<td>S_LOAD_DWORDX8</td>
</tr>
<tr>
<td>4</td>
<td>S_LOAD_DWORDX16</td>
</tr>
<tr>
<td>5</td>
<td>S_SCRATCH_LOAD_DWORD</td>
</tr>
<tr>
<td>6</td>
<td>S_SCRATCH_LOAD_DWORDX2</td>
</tr>
<tr>
<td>Opcode #</td>
<td>Name</td>
</tr>
<tr>
<td>----------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>7</td>
<td>S_SCRATCH_LOAD_DWORDX4</td>
</tr>
<tr>
<td>8</td>
<td>S_BUFFER_LOAD_DWORD</td>
</tr>
<tr>
<td>9</td>
<td>S_BUFFER_LOAD_DWORDX2</td>
</tr>
<tr>
<td>10</td>
<td>S_BUFFER_LOAD_DWORDX4</td>
</tr>
<tr>
<td>11</td>
<td>S_BUFFER_LOAD_DWORDX8</td>
</tr>
<tr>
<td>12</td>
<td>S_BUFFER_LOAD_DWORDX16</td>
</tr>
<tr>
<td>16</td>
<td>S_STORE_DWORD</td>
</tr>
<tr>
<td>17</td>
<td>S_STORE_DWORDX2</td>
</tr>
<tr>
<td>18</td>
<td>S_STORE_DWORDX4</td>
</tr>
<tr>
<td>21</td>
<td>S_SCRATCH_STORE_DWORD</td>
</tr>
<tr>
<td>22</td>
<td>S_SCRATCH_STORE_DWORDX2</td>
</tr>
<tr>
<td>23</td>
<td>S_SCRATCH_STORE_DWORDX4</td>
</tr>
<tr>
<td>24</td>
<td>S_BUFFER_STORE_DWORD</td>
</tr>
<tr>
<td>25</td>
<td>S_BUFFER_STORE_DWORDX2</td>
</tr>
<tr>
<td>26</td>
<td>S_BUFFER_STORE_DWORDX4</td>
</tr>
<tr>
<td>32</td>
<td>S_DCACHE_INV</td>
</tr>
<tr>
<td>33</td>
<td>S_DCACHE_WB</td>
</tr>
<tr>
<td>34</td>
<td>S_DCACHE_INV_VOL</td>
</tr>
<tr>
<td>35</td>
<td>S_DCACHE_WB_VOL</td>
</tr>
<tr>
<td>36</td>
<td>S_MEMTIME</td>
</tr>
<tr>
<td>37</td>
<td>S_MEMREALTIME</td>
</tr>
<tr>
<td>38</td>
<td>S_ATC_PROBE</td>
</tr>
<tr>
<td>39</td>
<td>S_ATC_PROBE_BUFFER</td>
</tr>
<tr>
<td>40</td>
<td>S_DCACHE_DISCARD</td>
</tr>
<tr>
<td>41</td>
<td>S_DCACHE_DISCARD_X2</td>
</tr>
<tr>
<td>64</td>
<td>S_BUFFER_ATOMIC_SWAP</td>
</tr>
<tr>
<td>65</td>
<td>S_BUFFER_ATOMIC_CMPSWAP</td>
</tr>
<tr>
<td>66</td>
<td>S_BUFFER_ATOMIC_ADD</td>
</tr>
<tr>
<td>67</td>
<td>S_BUFFER_ATOMIC_SUB</td>
</tr>
<tr>
<td>68</td>
<td>S_BUFFER_ATOMIC_SMIN</td>
</tr>
<tr>
<td>69</td>
<td>S_BUFFER_ATOMIC_UMIN</td>
</tr>
<tr>
<td>70</td>
<td>S_BUFFER_ATOMIC_SMAX</td>
</tr>
<tr>
<td>71</td>
<td>S_BUFFER_ATOMIC_UMAX</td>
</tr>
<tr>
<td>Opcode #</td>
<td>Name</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>72</td>
<td>S_BUFFER_ATOMIC_AND</td>
</tr>
<tr>
<td>73</td>
<td>S_BUFFER_ATOMIC_OR</td>
</tr>
<tr>
<td>74</td>
<td>S_BUFFER_ATOMIC_XOR</td>
</tr>
<tr>
<td>75</td>
<td>S_BUFFER_ATOMIC_INC</td>
</tr>
<tr>
<td>76</td>
<td>S_BUFFER_ATOMIC_DEC</td>
</tr>
<tr>
<td>96</td>
<td>S_BUFFER_ATOMIC_SWAP_X2</td>
</tr>
<tr>
<td>97</td>
<td>S_BUFFER_ATOMIC_CMPSWAP_X2</td>
</tr>
<tr>
<td>98</td>
<td>S_BUFFER_ATOMIC_ADD_X2</td>
</tr>
<tr>
<td>99</td>
<td>S_BUFFER_ATOMIC_SUB_X2</td>
</tr>
<tr>
<td>100</td>
<td>S_BUFFER_ATOMIC_SMIN_X2</td>
</tr>
<tr>
<td>101</td>
<td>S_BUFFER_ATOMIC_UMIN_X2</td>
</tr>
<tr>
<td>102</td>
<td>S_BUFFER_ATOMIC_SMAX_X2</td>
</tr>
<tr>
<td>103</td>
<td>S_BUFFER_ATOMIC_UMAX_X2</td>
</tr>
<tr>
<td>104</td>
<td>S_BUFFER_ATOMIC_AND_X2</td>
</tr>
<tr>
<td>105</td>
<td>S_BUFFER_ATOMIC_OR_X2</td>
</tr>
<tr>
<td>106</td>
<td>S_BUFFER_ATOMIC_XOR_X2</td>
</tr>
<tr>
<td>107</td>
<td>S_BUFFER_ATOMIC_INC_X2</td>
</tr>
<tr>
<td>108</td>
<td>S_BUFFER_ATOMIC_DEC_X2</td>
</tr>
<tr>
<td>128</td>
<td>S_ATOMIC_SWAP</td>
</tr>
<tr>
<td>129</td>
<td>S_ATOMIC_CMPSWAP</td>
</tr>
<tr>
<td>130</td>
<td>S_ATOMIC_ADD</td>
</tr>
<tr>
<td>131</td>
<td>S_ATOMIC_SUB</td>
</tr>
<tr>
<td>132</td>
<td>S_ATOMIC_SMIN</td>
</tr>
<tr>
<td>133</td>
<td>S_ATOMIC_UMIN</td>
</tr>
<tr>
<td>134</td>
<td>S_ATOMIC_SMAX</td>
</tr>
<tr>
<td>135</td>
<td>S_ATOMIC_UMAX</td>
</tr>
<tr>
<td>136</td>
<td>S_ATOMIC_AND</td>
</tr>
<tr>
<td>137</td>
<td>S_ATOMIC_OR</td>
</tr>
<tr>
<td>138</td>
<td>S_ATOMIC_XOR</td>
</tr>
<tr>
<td>139</td>
<td>S_ATOMIC_INC</td>
</tr>
<tr>
<td>140</td>
<td>S_ATOMIC_DEC</td>
</tr>
<tr>
<td>160</td>
<td>S_ATOMIC_SWAP_X2</td>
</tr>
<tr>
<td>161</td>
<td>S_ATOMIC_CMPSWAP_X2</td>
</tr>
</tbody>
</table>
13.3. Vector ALU Formats

13.3.1. VOP2

| Format | VOP2 |
| Description | Vector ALU format with two operands |

Table 65. VOP2 Fields
<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bits</th>
<th>Format or Description</th>
</tr>
</thead>
</table>

VSRC1       [16:9] VGPR which provides the second operand.  
VDST        [24:17] Destination VGPR.  
ENCODING    [31] Must be: 0

Table 66. VOP2 Opcodes

<table>
<thead>
<tr>
<th>Opcode #</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>V_CNDMASK_B32</td>
</tr>
</tbody>
</table>

13.3. Vector ALU Formats
<table>
<thead>
<tr>
<th>Opcode #</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>V_ADD_F32</td>
</tr>
<tr>
<td>2</td>
<td>V_SUB_F32</td>
</tr>
<tr>
<td>3</td>
<td>V_SUBREV_F32</td>
</tr>
<tr>
<td>4</td>
<td>V_MUL_LEGACY_F32</td>
</tr>
<tr>
<td>5</td>
<td>V_MUL_F32</td>
</tr>
<tr>
<td>6</td>
<td>V_MUL_I32_I24</td>
</tr>
<tr>
<td>7</td>
<td>V_MUL_HI_I32_I24</td>
</tr>
<tr>
<td>8</td>
<td>V_MUL_U32_U24</td>
</tr>
<tr>
<td>9</td>
<td>V_MUL_HI_U32_U24</td>
</tr>
<tr>
<td>10</td>
<td>V_MIN_F32</td>
</tr>
<tr>
<td>11</td>
<td>V_MAX_F32</td>
</tr>
<tr>
<td>12</td>
<td>V_MIN_I32</td>
</tr>
<tr>
<td>13</td>
<td>V_MAX_I32</td>
</tr>
<tr>
<td>14</td>
<td>V_MIN_U32</td>
</tr>
<tr>
<td>15</td>
<td>V_MAX_U32</td>
</tr>
<tr>
<td>16</td>
<td>V_LSHRREV_B32</td>
</tr>
<tr>
<td>17</td>
<td>V_ASHRREV_I32</td>
</tr>
<tr>
<td>18</td>
<td>V_LSHLREV_B32</td>
</tr>
<tr>
<td>19</td>
<td>V_AND_B32</td>
</tr>
<tr>
<td>20</td>
<td>V_OR_B32</td>
</tr>
<tr>
<td>21</td>
<td>V_XOR_B32</td>
</tr>
<tr>
<td>22</td>
<td>V_MAC_F32</td>
</tr>
<tr>
<td>23</td>
<td>V_MADMK_F32</td>
</tr>
<tr>
<td>24</td>
<td>V_MADAK_F32</td>
</tr>
<tr>
<td>25</td>
<td>V_ADD_CO_U32</td>
</tr>
<tr>
<td>26</td>
<td>V_SUB_CO_U32</td>
</tr>
<tr>
<td>27</td>
<td>V_SUBREV_CO_U32</td>
</tr>
<tr>
<td>28</td>
<td>V_ADDC_CO_U32</td>
</tr>
<tr>
<td>29</td>
<td>V_SUBB_CO_U32</td>
</tr>
<tr>
<td>30</td>
<td>V_SUBBBREV_CO_U32</td>
</tr>
<tr>
<td>31</td>
<td>V_ADD_F16</td>
</tr>
<tr>
<td>32</td>
<td>V_SUB_F16</td>
</tr>
<tr>
<td>33</td>
<td>V_SUBREV_F16</td>
</tr>
<tr>
<td>Opcode #</td>
<td>Name</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------</td>
</tr>
<tr>
<td>34</td>
<td>V_MUL_F16</td>
</tr>
<tr>
<td>35</td>
<td>V_MAC_F16</td>
</tr>
<tr>
<td>36</td>
<td>V_MADMK_F16</td>
</tr>
<tr>
<td>37</td>
<td>V_MADAK_F16</td>
</tr>
<tr>
<td>38</td>
<td>V_ADD_U16</td>
</tr>
<tr>
<td>39</td>
<td>V_SUB_U16</td>
</tr>
<tr>
<td>40</td>
<td>V_SUBREV_U16</td>
</tr>
<tr>
<td>41</td>
<td>V_MUL_LO_U16</td>
</tr>
<tr>
<td>42</td>
<td>V_LSHLREV_B16</td>
</tr>
<tr>
<td>43</td>
<td>V_LSHRREV_B16</td>
</tr>
<tr>
<td>44</td>
<td>V_ASHRREV_I16</td>
</tr>
<tr>
<td>45</td>
<td>V_MAX_F16</td>
</tr>
<tr>
<td>46</td>
<td>V_MIN_F16</td>
</tr>
<tr>
<td>47</td>
<td>V_MAX_U16</td>
</tr>
<tr>
<td>48</td>
<td>V_MAX_I16</td>
</tr>
<tr>
<td>49</td>
<td>V_MIN_U16</td>
</tr>
<tr>
<td>50</td>
<td>V_MIN_I16</td>
</tr>
<tr>
<td>51</td>
<td>V_LDEXP_F16</td>
</tr>
<tr>
<td>52</td>
<td>V_ADD_U32</td>
</tr>
<tr>
<td>53</td>
<td>V_SUB_U32</td>
</tr>
<tr>
<td>54</td>
<td>V_SUBREV_U32</td>
</tr>
<tr>
<td>59</td>
<td>V_FMAC_F32</td>
</tr>
<tr>
<td>61</td>
<td>V_XNOR_B32</td>
</tr>
</tbody>
</table>

### 13.3.2. VOP1

**Format**

<table>
<thead>
<tr>
<th>VOP1</th>
<th>31</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>VDST</td>
<td>OP</td>
<td>SRC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Description**

Vector ALU format with one operand

*Table 67. VOP1 Fields*
<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bits</th>
<th>Format or Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRC0</td>
<td>[8:0]</td>
<td>Source 0. First operand for the instruction.</td>
</tr>
<tr>
<td></td>
<td>0 - 101</td>
<td>SGPR0 to SGPR101: Scalar general-purpose registers.</td>
</tr>
<tr>
<td></td>
<td>102</td>
<td>FLAT_SCRATCH_LO.</td>
</tr>
<tr>
<td></td>
<td>103</td>
<td>FLAT_SCRATCH_HI.</td>
</tr>
<tr>
<td></td>
<td>104</td>
<td>XNACK_MASK_LO.</td>
</tr>
<tr>
<td></td>
<td>105</td>
<td>XNACK_MASK_HI.</td>
</tr>
<tr>
<td></td>
<td>106</td>
<td>VCC_LO: vcc[31:0].</td>
</tr>
<tr>
<td></td>
<td>107</td>
<td>VCC_HI: vcc[63:32].</td>
</tr>
<tr>
<td></td>
<td>108-123</td>
<td>TTMP0 - TTMP15: Trap handler temporary register.</td>
</tr>
<tr>
<td></td>
<td>124</td>
<td>M0. Memory register 0.</td>
</tr>
<tr>
<td></td>
<td>125</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>126</td>
<td>EXEC_LO: exec[31:0].</td>
</tr>
<tr>
<td></td>
<td>127</td>
<td>EXEC_HI: exec[63:32].</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>0.</td>
</tr>
<tr>
<td></td>
<td>129-192</td>
<td>Signed integer 1 to 64.</td>
</tr>
<tr>
<td></td>
<td>193-208</td>
<td>Signed integer -1 to -16.</td>
</tr>
<tr>
<td></td>
<td>209-234</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>235</td>
<td>SHARED_BASE (Memory Aperture definition).</td>
</tr>
<tr>
<td></td>
<td>236</td>
<td>SHARED_LIMIT (Memory Aperture definition).</td>
</tr>
<tr>
<td></td>
<td>237</td>
<td>PRIVATE_BASE (Memory Aperture definition).</td>
</tr>
<tr>
<td></td>
<td>238</td>
<td>PRIVATE_LIMIT (Memory Aperture definition).</td>
</tr>
<tr>
<td></td>
<td>239</td>
<td>POPS_EXITING_WAVE_ID.</td>
</tr>
<tr>
<td></td>
<td>240</td>
<td>0.5.</td>
</tr>
<tr>
<td></td>
<td>241</td>
<td>-0.5.</td>
</tr>
<tr>
<td></td>
<td>242</td>
<td>1.0.</td>
</tr>
<tr>
<td></td>
<td>243</td>
<td>-1.0.</td>
</tr>
<tr>
<td></td>
<td>244</td>
<td>2.0.</td>
</tr>
<tr>
<td></td>
<td>245</td>
<td>-2.0.</td>
</tr>
<tr>
<td></td>
<td>246</td>
<td>4.0.</td>
</tr>
<tr>
<td></td>
<td>247</td>
<td>-4.0.</td>
</tr>
<tr>
<td></td>
<td>248</td>
<td>1/(2*PI).</td>
</tr>
<tr>
<td></td>
<td>249</td>
<td>SDWA</td>
</tr>
<tr>
<td></td>
<td>250</td>
<td>DPP</td>
</tr>
<tr>
<td></td>
<td>251</td>
<td>VCCZ.</td>
</tr>
<tr>
<td></td>
<td>252</td>
<td>EXECZ.</td>
</tr>
<tr>
<td></td>
<td>253</td>
<td>SCC.</td>
</tr>
<tr>
<td></td>
<td>254</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>255</td>
<td>Literal constant.</td>
</tr>
<tr>
<td></td>
<td>256 - 511</td>
<td>VGPR 0 - 255</td>
</tr>
<tr>
<td>VDST</td>
<td>[24:17]</td>
<td>Destination VGPR.</td>
</tr>
<tr>
<td>ENCODING</td>
<td>[31:25]</td>
<td>Must be: 0_111111</td>
</tr>
</tbody>
</table>

**Table 68. VOP1 Opcodes**

<table>
<thead>
<tr>
<th>Opcode #</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>V_NOP</td>
</tr>
<tr>
<td>1</td>
<td>V_MOV_B32</td>
</tr>
</tbody>
</table>

13.3. Vector ALU Formats
<table>
<thead>
<tr>
<th>Opcode #</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>V_READFIRSTLANE_B32</td>
</tr>
<tr>
<td>3</td>
<td>V_CVT_I32_F64</td>
</tr>
<tr>
<td>4</td>
<td>V_CVT_F64_I32</td>
</tr>
<tr>
<td>5</td>
<td>V_CVT_F32_I32</td>
</tr>
<tr>
<td>6</td>
<td>V_CVT_F32_U32</td>
</tr>
<tr>
<td>7</td>
<td>V_CVT_U32_F32</td>
</tr>
<tr>
<td>8</td>
<td>V_CVT_I32_F32</td>
</tr>
<tr>
<td>10</td>
<td>V_CVT_F16_F32</td>
</tr>
<tr>
<td>11</td>
<td>V_CVT_F32_F16</td>
</tr>
<tr>
<td>12</td>
<td>V_CVT_RPI_I32_F32</td>
</tr>
<tr>
<td>13</td>
<td>V_CVT_FLR_I32_F32</td>
</tr>
<tr>
<td>14</td>
<td>V_CVT_OFF_F32_I4</td>
</tr>
<tr>
<td>15</td>
<td>V_CVT_F32_F64</td>
</tr>
<tr>
<td>16</td>
<td>V_CVT_F64_F32</td>
</tr>
<tr>
<td>17</td>
<td>V_CVT_F32_UBYTE0</td>
</tr>
<tr>
<td>18</td>
<td>V_CVT_F32_UBYTE1</td>
</tr>
<tr>
<td>19</td>
<td>V_CVT_F32_UBYTE2</td>
</tr>
<tr>
<td>20</td>
<td>V_CVT_F32_UBYTE3</td>
</tr>
<tr>
<td>21</td>
<td>V_CVT_U32_F64</td>
</tr>
<tr>
<td>22</td>
<td>V_CVT_F64_U32</td>
</tr>
<tr>
<td>23</td>
<td>V_TRUNC_F64</td>
</tr>
<tr>
<td>24</td>
<td>V_CEIL_F64</td>
</tr>
<tr>
<td>25</td>
<td>V_RNDNE_F64</td>
</tr>
<tr>
<td>26</td>
<td>V_FLOOR_F64</td>
</tr>
<tr>
<td>27</td>
<td>V_FRACT_F32</td>
</tr>
<tr>
<td>28</td>
<td>V_TRUNC_F32</td>
</tr>
<tr>
<td>29</td>
<td>V_CEIL_F32</td>
</tr>
<tr>
<td>30</td>
<td>V_RNDNE_F32</td>
</tr>
<tr>
<td>31</td>
<td>V_FLOOR_F32</td>
</tr>
<tr>
<td>32</td>
<td>V_EXP_F32</td>
</tr>
<tr>
<td>33</td>
<td>V_LOG_F32</td>
</tr>
<tr>
<td>34</td>
<td>V_RCP_F32</td>
</tr>
<tr>
<td>35</td>
<td>V_RCP_IFLAG_F32</td>
</tr>
<tr>
<td>Opcode #</td>
<td>Name</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>36</td>
<td>V_RSQ_F32</td>
</tr>
<tr>
<td>37</td>
<td>V_RCP_F64</td>
</tr>
<tr>
<td>38</td>
<td>V_RSQ_F64</td>
</tr>
<tr>
<td>39</td>
<td>V_SQRT_F32</td>
</tr>
<tr>
<td>40</td>
<td>V_SQRT_F64</td>
</tr>
<tr>
<td>41</td>
<td>V_SIN_F32</td>
</tr>
<tr>
<td>42</td>
<td>V_COS_F32</td>
</tr>
<tr>
<td>43</td>
<td>V_NOT_B32</td>
</tr>
<tr>
<td>44</td>
<td>V_BFREV_B32</td>
</tr>
<tr>
<td>45</td>
<td>V_FFBH_U32</td>
</tr>
<tr>
<td>46</td>
<td>V_FFBL_B32</td>
</tr>
<tr>
<td>47</td>
<td>V_FFBH_I32</td>
</tr>
<tr>
<td>48</td>
<td>V_FREXP_EXP_I32_F64</td>
</tr>
<tr>
<td>49</td>
<td>V_FREXP_MANT_F64</td>
</tr>
<tr>
<td>50</td>
<td>V_FRACT_F64</td>
</tr>
<tr>
<td>51</td>
<td>V_FREXP_EXP_I32_F32</td>
</tr>
<tr>
<td>52</td>
<td>V_FREXP_MANT_F32</td>
</tr>
<tr>
<td>53</td>
<td>V_CLREXCP</td>
</tr>
<tr>
<td>55</td>
<td>V_SCREEN_PARTITION_4SE_B32</td>
</tr>
<tr>
<td>57</td>
<td>V_CVT_F16_U16</td>
</tr>
<tr>
<td>58</td>
<td>V_CVT_F16_I16</td>
</tr>
<tr>
<td>59</td>
<td>V_CVT_U16_F16</td>
</tr>
<tr>
<td>60</td>
<td>V_CVT_I16_F16</td>
</tr>
<tr>
<td>61</td>
<td>V_RCP_F16</td>
</tr>
<tr>
<td>62</td>
<td>V_SQRT_F16</td>
</tr>
<tr>
<td>63</td>
<td>V_RSQ_F16</td>
</tr>
<tr>
<td>64</td>
<td>V_LOG_F16</td>
</tr>
<tr>
<td>65</td>
<td>V_EXP_F16</td>
</tr>
<tr>
<td>66</td>
<td>V_FREXP_MANT_F16</td>
</tr>
<tr>
<td>67</td>
<td>V_FREXP_EXP_I16_F16</td>
</tr>
<tr>
<td>68</td>
<td>V_FLOOR_F16</td>
</tr>
<tr>
<td>69</td>
<td>V_CEIL_F16</td>
</tr>
<tr>
<td>70</td>
<td>V_TRUNC_F16</td>
</tr>
</tbody>
</table>
### Opcode Table

<table>
<thead>
<tr>
<th>Opcode #</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>71</td>
<td>V_RNDNE_F16</td>
</tr>
<tr>
<td>72</td>
<td>V_FRACT_F16</td>
</tr>
<tr>
<td>73</td>
<td>V_SIN_F16</td>
</tr>
<tr>
<td>74</td>
<td>V_COS_F16</td>
</tr>
<tr>
<td>75</td>
<td>V_EXP_LEGACY_F32</td>
</tr>
<tr>
<td>76</td>
<td>V_LOG_LEGACY_F32</td>
</tr>
<tr>
<td>77</td>
<td>V_CVT_NORM_I16_F16</td>
</tr>
<tr>
<td>78</td>
<td>V_CVT_NORM_U16_F16</td>
</tr>
<tr>
<td>79</td>
<td>V_SAT_PK_U8_I16</td>
</tr>
<tr>
<td>81</td>
<td>V_SWAP_B32</td>
</tr>
</tbody>
</table>

### 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 only a single comparison operation.

The final opcode number is determined by adding the base for the opcode family plus the offset from the compare op. Every compare instruction writes a result to VCC (for VOPC) or an SGPR (for VOP3). Additionally, every compare instruction has a variant that also writes to the EXEC mask. The destination of the compare result is VCC when encoded using the VOPC format, and can be an arbitrary SGPR when encoded in the VOP3 format.

### Comparison Operations

**Table 69. Comparison Operations**

<table>
<thead>
<tr>
<th>Compare Operation</th>
<th>Opcode Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sixteen Compare Operations (OP16)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>0</td>
<td>D.u = 0</td>
</tr>
<tr>
<td>Compare Operation</td>
<td>Opcode Offset</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>LT</td>
<td>1</td>
<td>$D.u = (S0 &lt; S1)$</td>
</tr>
<tr>
<td>EQ</td>
<td>2</td>
<td>$D.u = (S0 == S1)$</td>
</tr>
<tr>
<td>LE</td>
<td>3</td>
<td>$D.u = (S0 &lt;= S1)$</td>
</tr>
<tr>
<td>GT</td>
<td>4</td>
<td>$D.u = (S0 &gt; S1)$</td>
</tr>
<tr>
<td>LG</td>
<td>5</td>
<td>$D.u = (S0 &lt;&gt; S1)$</td>
</tr>
<tr>
<td>GE</td>
<td>6</td>
<td>$D.u = (S0 &gt;= S1)$</td>
</tr>
<tr>
<td>O</td>
<td>7</td>
<td>$D.u = (!isNaN(S0) &amp;&amp; !isNaN(S1))$</td>
</tr>
<tr>
<td>U</td>
<td>8</td>
<td>$D.u = (!isNaN(S0)</td>
</tr>
<tr>
<td>NGE</td>
<td>9</td>
<td>$D.u = !(S0 &gt;= S1)$</td>
</tr>
<tr>
<td>NLG</td>
<td>10</td>
<td>$D.u = !(S0 &lt;= S1)$</td>
</tr>
<tr>
<td>NGT</td>
<td>11</td>
<td>$D.u = !(S0 &lt; S1)$</td>
</tr>
<tr>
<td>NLE</td>
<td>12</td>
<td>$D.u = !(S0 &lt;= S1)$</td>
</tr>
<tr>
<td>NEQ</td>
<td>13</td>
<td>$D.u = !(S0 == S1)$</td>
</tr>
<tr>
<td>NLT</td>
<td>14</td>
<td>$D.u = !(S0 &lt; S1)$</td>
</tr>
<tr>
<td>TRU</td>
<td>15</td>
<td>$D.u = 1$</td>
</tr>
</tbody>
</table>

Eight Compare Operations (OP8)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Opcode Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>0</td>
<td>$D.u = 0$</td>
</tr>
<tr>
<td>LT</td>
<td>1</td>
<td>$D.u = (S0 &lt; S1)$</td>
</tr>
<tr>
<td>EQ</td>
<td>2</td>
<td>$D.u = (S0 == S1)$</td>
</tr>
<tr>
<td>LE</td>
<td>3</td>
<td>$D.u = (S0 &lt;= S1)$</td>
</tr>
<tr>
<td>GT</td>
<td>4</td>
<td>$D.u = (S0 &gt; S1)$</td>
</tr>
<tr>
<td>LG</td>
<td>5</td>
<td>$D.u = (S0 &lt;&gt; S1)$</td>
</tr>
<tr>
<td>GE</td>
<td>6</td>
<td>$D.u = (S0 &gt;= S1)$</td>
</tr>
<tr>
<td>TRU</td>
<td>7</td>
<td>$D.u = 1$</td>
</tr>
</tbody>
</table>

Table 70. VOPC Fields

13.3. Vector ALU Formats
### 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 | TMP0 - TMP15: 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 | POPS_EXITING_WAVE_ID.  
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.


ENCODING | [31:25] | Must be: 0_11110

### Table 71. VOPC Opcodes

<table>
<thead>
<tr>
<th>Opcode #</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>V_CMP_CLASS_F32</td>
</tr>
<tr>
<td>17</td>
<td>V_CMPX_CLASS_F32</td>
</tr>
</tbody>
</table>

13.3. Vector ALU Formats
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>V_CMP_CLASS_F64</td>
</tr>
<tr>
<td>19</td>
<td>V_CMPX_CLASS_F64</td>
</tr>
<tr>
<td>20</td>
<td>V_CMP_CLASS_F16</td>
</tr>
<tr>
<td>21</td>
<td>V_CMPX_CLASS_F16</td>
</tr>
<tr>
<td>22</td>
<td>V_CMP_F_F16</td>
</tr>
<tr>
<td>23</td>
<td>V_CMP_LT_F16</td>
</tr>
<tr>
<td>24</td>
<td>V_CMP_EQ_F16</td>
</tr>
<tr>
<td>25</td>
<td>V_CMP_LE_F16</td>
</tr>
<tr>
<td>26</td>
<td>V_CMP_GT_F16</td>
</tr>
<tr>
<td>27</td>
<td>V_CMP_LG_F16</td>
</tr>
<tr>
<td>28</td>
<td>V_CMP_GE_F16</td>
</tr>
<tr>
<td>29</td>
<td>V_CMP_O_F16</td>
</tr>
<tr>
<td>30</td>
<td>V_CMP_U_F16</td>
</tr>
<tr>
<td>31</td>
<td>V_CMP_NGE_F16</td>
</tr>
<tr>
<td>32</td>
<td>V_CMP_NLG_F16</td>
</tr>
<tr>
<td>33</td>
<td>V_CMP_NGT_F16</td>
</tr>
<tr>
<td>34</td>
<td>V_CMP_NLE_F16</td>
</tr>
<tr>
<td>35</td>
<td>V_CMP_NEQ_F16</td>
</tr>
<tr>
<td>36</td>
<td>V_CMP_NLT_F16</td>
</tr>
<tr>
<td>37</td>
<td>V_CMP_TRU_F16</td>
</tr>
<tr>
<td>38</td>
<td>V_CMPX_F_F16</td>
</tr>
<tr>
<td>39</td>
<td>V_CMPX_LT_F16</td>
</tr>
<tr>
<td>40</td>
<td>V_CMPX_EQ_F16</td>
</tr>
<tr>
<td>41</td>
<td>V_CMPX_LE_F16</td>
</tr>
<tr>
<td>42</td>
<td>V_CMPX_GT_F16</td>
</tr>
<tr>
<td>43</td>
<td>V_CMPX_LG_F16</td>
</tr>
<tr>
<td>44</td>
<td>V_CMPX_GE_F16</td>
</tr>
<tr>
<td>45</td>
<td>V_CMPX_O_F16</td>
</tr>
<tr>
<td>46</td>
<td>V_CMPX_U_F16</td>
</tr>
<tr>
<td>47</td>
<td>V_CMPX_NGE_F16</td>
</tr>
<tr>
<td>48</td>
<td>V_CMPX_NLG_F16</td>
</tr>
<tr>
<td>49</td>
<td>V_CMPX_NGT_F16</td>
</tr>
<tr>
<td>50</td>
<td>V_CMPX_NLE_F16</td>
</tr>
</tbody>
</table>

"Vega" 7nm Instruction Set Architecture
<table>
<thead>
<tr>
<th>Opcode #</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>61</td>
<td>V_CMPX_NEQ_F16</td>
</tr>
<tr>
<td>62</td>
<td>V_CMPX_NLT_F16</td>
</tr>
<tr>
<td>63</td>
<td>V_CMPX_TRU_F16</td>
</tr>
<tr>
<td>64</td>
<td>V_CMP_F_F32</td>
</tr>
<tr>
<td>65</td>
<td>V_CMP_LT_F32</td>
</tr>
<tr>
<td>66</td>
<td>V_CMP_EQ_F32</td>
</tr>
<tr>
<td>67</td>
<td>V_CMP_LE_F32</td>
</tr>
<tr>
<td>68</td>
<td>V_CMP_GT_F32</td>
</tr>
<tr>
<td>69</td>
<td>V_CMP_LG_F32</td>
</tr>
<tr>
<td>70</td>
<td>V_CMP_GE_F32</td>
</tr>
<tr>
<td>71</td>
<td>V_CMP_O_F32</td>
</tr>
<tr>
<td>72</td>
<td>V_CMP_U_F32</td>
</tr>
<tr>
<td>73</td>
<td>V_CMP_NGE_F32</td>
</tr>
<tr>
<td>74</td>
<td>V_CMP_NLG_F32</td>
</tr>
<tr>
<td>75</td>
<td>V_CMP_NGT_F32</td>
</tr>
<tr>
<td>76</td>
<td>V_CMP_NLE_F32</td>
</tr>
<tr>
<td>77</td>
<td>V_CMP_NEQ_F32</td>
</tr>
<tr>
<td>78</td>
<td>V_CMP_NLT_F32</td>
</tr>
<tr>
<td>79</td>
<td>V_CMP_TRU_F32</td>
</tr>
<tr>
<td>80</td>
<td>V_CMPX_F_F32</td>
</tr>
<tr>
<td>81</td>
<td>V_CMPX_LT_F32</td>
</tr>
<tr>
<td>82</td>
<td>V_CMPX_EQ_F32</td>
</tr>
<tr>
<td>83</td>
<td>V_CMPX_LE_F32</td>
</tr>
<tr>
<td>84</td>
<td>V_CMPX_GT_F32</td>
</tr>
<tr>
<td>85</td>
<td>V_CMPX_LG_F32</td>
</tr>
<tr>
<td>86</td>
<td>V_CMPX_GE_F32</td>
</tr>
<tr>
<td>87</td>
<td>V_CMPX_O_F32</td>
</tr>
<tr>
<td>88</td>
<td>V_CMPX_U_F32</td>
</tr>
<tr>
<td>89</td>
<td>V_CMPX_NGE_F32</td>
</tr>
<tr>
<td>90</td>
<td>V_CMPX_NLG_F32</td>
</tr>
<tr>
<td>91</td>
<td>V_CMPX_NGT_F32</td>
</tr>
<tr>
<td>92</td>
<td>V_CMPX_NLE_F32</td>
</tr>
<tr>
<td>93</td>
<td>V_CMPX_NEQ_F32</td>
</tr>
<tr>
<td>Opcode #</td>
<td>Name</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>94</td>
<td>V_CMPX_NLT_F32</td>
</tr>
<tr>
<td>95</td>
<td>V_CMPX_TRU_F32</td>
</tr>
<tr>
<td>96</td>
<td>V_CMP_F_F64</td>
</tr>
<tr>
<td>97</td>
<td>V_CMP_LT_F64</td>
</tr>
<tr>
<td>98</td>
<td>V_CMP_EQ_F64</td>
</tr>
<tr>
<td>99</td>
<td>V_CMP_LE_F64</td>
</tr>
<tr>
<td>100</td>
<td>V_CMP_GT_F64</td>
</tr>
<tr>
<td>101</td>
<td>V_CMP_LG_F64</td>
</tr>
<tr>
<td>102</td>
<td>V_CMP_GE_F64</td>
</tr>
<tr>
<td>103</td>
<td>V_CMP_O_F64</td>
</tr>
<tr>
<td>104</td>
<td>V_CMP_U_F64</td>
</tr>
<tr>
<td>105</td>
<td>V_CMP_NGE_F64</td>
</tr>
<tr>
<td>106</td>
<td>V_CMP_NLG_F64</td>
</tr>
<tr>
<td>107</td>
<td>V_CMP_NGT_F64</td>
</tr>
<tr>
<td>108</td>
<td>V_CMP_NLE_F64</td>
</tr>
<tr>
<td>109</td>
<td>V_CMP_NEQ_F64</td>
</tr>
<tr>
<td>110</td>
<td>V_CMP_NLT_F64</td>
</tr>
<tr>
<td>111</td>
<td>V_CMP_TRU_F64</td>
</tr>
<tr>
<td>112</td>
<td>V_CMPX_F_F64</td>
</tr>
<tr>
<td>113</td>
<td>V_CMPX_LT_F64</td>
</tr>
<tr>
<td>114</td>
<td>V_CMPX_EQ_F64</td>
</tr>
<tr>
<td>115</td>
<td>V_CMPX_LE_F64</td>
</tr>
<tr>
<td>116</td>
<td>V_CMPX_GT_F64</td>
</tr>
<tr>
<td>117</td>
<td>V_CMPX_LG_F64</td>
</tr>
<tr>
<td>118</td>
<td>V_CMPX_GE_F64</td>
</tr>
<tr>
<td>119</td>
<td>V_CMPX_O_F64</td>
</tr>
<tr>
<td>120</td>
<td>V_CMPX_U_F64</td>
</tr>
<tr>
<td>121</td>
<td>V_CMPX_NGE_F64</td>
</tr>
<tr>
<td>122</td>
<td>V_CMPX_NLG_F64</td>
</tr>
<tr>
<td>123</td>
<td>V_CMPX_NGT_F64</td>
</tr>
<tr>
<td>124</td>
<td>V_CMPX_NLE_F64</td>
</tr>
<tr>
<td>125</td>
<td>V_CMPX_NEQ_F64</td>
</tr>
<tr>
<td>126</td>
<td>V_CMPX_NLT_F64</td>
</tr>
<tr>
<td>Opcode #</td>
<td>Name</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>127</td>
<td>V_CMPX_TRU_F64</td>
</tr>
<tr>
<td>160</td>
<td>V_CMP_F_I16</td>
</tr>
<tr>
<td>161</td>
<td>V_CMP_LT_I16</td>
</tr>
<tr>
<td>162</td>
<td>V_CMP_EQ_I16</td>
</tr>
<tr>
<td>163</td>
<td>V_CMP_LE_I16</td>
</tr>
<tr>
<td>164</td>
<td>V_CMP_GT_I16</td>
</tr>
<tr>
<td>165</td>
<td>V_CMP_NE_I16</td>
</tr>
<tr>
<td>166</td>
<td>V_CMP_GE_I16</td>
</tr>
<tr>
<td>167</td>
<td>V_CMP_T_I16</td>
</tr>
<tr>
<td>168</td>
<td>V_CMP_F_U16</td>
</tr>
<tr>
<td>169</td>
<td>V_CMP_LT_U16</td>
</tr>
<tr>
<td>170</td>
<td>V_CMP_EQ_U16</td>
</tr>
<tr>
<td>171</td>
<td>V_CMP_LE_U16</td>
</tr>
<tr>
<td>172</td>
<td>V_CMP_GT_U16</td>
</tr>
<tr>
<td>173</td>
<td>V_CMP_NE_U16</td>
</tr>
<tr>
<td>174</td>
<td>V_CMP_GE_U16</td>
</tr>
<tr>
<td>175</td>
<td>V_CMP_T_U16</td>
</tr>
<tr>
<td>176</td>
<td>V_CMPX_F_I16</td>
</tr>
<tr>
<td>177</td>
<td>V_CMPX_LT_I16</td>
</tr>
<tr>
<td>178</td>
<td>V_CMPX_EQ_I16</td>
</tr>
<tr>
<td>179</td>
<td>V_CMPX_LE_I16</td>
</tr>
<tr>
<td>180</td>
<td>V_CMPX_GT_I16</td>
</tr>
<tr>
<td>181</td>
<td>V_CMPX_NE_I16</td>
</tr>
<tr>
<td>182</td>
<td>V_CMPX_GE_I16</td>
</tr>
<tr>
<td>183</td>
<td>V_CMPX_T_I16</td>
</tr>
<tr>
<td>184</td>
<td>V_CMPX_F_U16</td>
</tr>
<tr>
<td>185</td>
<td>V_CMPX_LT_U16</td>
</tr>
<tr>
<td>186</td>
<td>V_CMPX_EQ_U16</td>
</tr>
<tr>
<td>187</td>
<td>V_CMPX_LE_U16</td>
</tr>
<tr>
<td>188</td>
<td>V_CMPX_GT_U16</td>
</tr>
<tr>
<td>189</td>
<td>V_CMPX_NE_U16</td>
</tr>
<tr>
<td>190</td>
<td>V_CMPX_GE_U16</td>
</tr>
<tr>
<td>191</td>
<td>V_CMPX_T_U16</td>
</tr>
<tr>
<td>Opcode #</td>
<td>Name</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------</td>
</tr>
<tr>
<td>192</td>
<td>V_CMP_F_I32</td>
</tr>
<tr>
<td>193</td>
<td>V_CMP_LT_I32</td>
</tr>
<tr>
<td>194</td>
<td>V_CMP_EQ_I32</td>
</tr>
<tr>
<td>195</td>
<td>V_CMP_LE_I32</td>
</tr>
<tr>
<td>196</td>
<td>V_CMP_GT_I32</td>
</tr>
<tr>
<td>197</td>
<td>V_CMP_NE_I32</td>
</tr>
<tr>
<td>198</td>
<td>V_CMP_GE_I32</td>
</tr>
<tr>
<td>199</td>
<td>V_CMP_T_I32</td>
</tr>
<tr>
<td>200</td>
<td>V_CMP_F_U32</td>
</tr>
<tr>
<td>201</td>
<td>V_CMP_LT_U32</td>
</tr>
<tr>
<td>202</td>
<td>V_CMP_EQ_U32</td>
</tr>
<tr>
<td>203</td>
<td>V_CMP_LE_U32</td>
</tr>
<tr>
<td>204</td>
<td>V_CMP_GT_U32</td>
</tr>
<tr>
<td>205</td>
<td>V_CMP_NE_U32</td>
</tr>
<tr>
<td>206</td>
<td>V_CMP_GE_U32</td>
</tr>
<tr>
<td>207</td>
<td>V_CMP_T_U32</td>
</tr>
<tr>
<td>208</td>
<td>V_CMPX_F_I32</td>
</tr>
<tr>
<td>209</td>
<td>V_CMPX_LT_I32</td>
</tr>
<tr>
<td>210</td>
<td>V_CMPX_EQ_I32</td>
</tr>
<tr>
<td>211</td>
<td>V_CMPX_LE_I32</td>
</tr>
<tr>
<td>212</td>
<td>V_CMPX_GT_I32</td>
</tr>
<tr>
<td>213</td>
<td>V_CMPX_NE_I32</td>
</tr>
<tr>
<td>214</td>
<td>V_CMPX_GE_I32</td>
</tr>
<tr>
<td>215</td>
<td>V_CMPX_T_I32</td>
</tr>
<tr>
<td>216</td>
<td>V_CMPX_F_U32</td>
</tr>
<tr>
<td>217</td>
<td>V_CMPX_LT_U32</td>
</tr>
<tr>
<td>218</td>
<td>V_CMPX_EQ_U32</td>
</tr>
<tr>
<td>219</td>
<td>V_CMPX_LE_U32</td>
</tr>
<tr>
<td>220</td>
<td>V_CMPX_GT_U32</td>
</tr>
<tr>
<td>221</td>
<td>V_CMPX_NE_U32</td>
</tr>
<tr>
<td>222</td>
<td>V_CMPX_GE_U32</td>
</tr>
<tr>
<td>223</td>
<td>V_CMPX_T_U32</td>
</tr>
<tr>
<td>224</td>
<td>V_CMP_F_I64</td>
</tr>
<tr>
<td>Opcode #</td>
<td>Name</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------</td>
</tr>
<tr>
<td>225</td>
<td>V_CMP_LT_I64</td>
</tr>
<tr>
<td>226</td>
<td>V_CMP_EQ_I64</td>
</tr>
<tr>
<td>227</td>
<td>V_CMP_LE_I64</td>
</tr>
<tr>
<td>228</td>
<td>V_CMP_GT_I64</td>
</tr>
<tr>
<td>229</td>
<td>V_CMP_NE_I64</td>
</tr>
<tr>
<td>230</td>
<td>V_CMP_GE_I64</td>
</tr>
<tr>
<td>231</td>
<td>V_CMP_T_I64</td>
</tr>
<tr>
<td>232</td>
<td>V_CMP_F_U64</td>
</tr>
<tr>
<td>233</td>
<td>V_CMP_LT_U64</td>
</tr>
<tr>
<td>234</td>
<td>V_CMP_EQ_U64</td>
</tr>
<tr>
<td>235</td>
<td>V_CMP_LE_U64</td>
</tr>
<tr>
<td>236</td>
<td>V_CMP_GT_U64</td>
</tr>
<tr>
<td>237</td>
<td>V_CMP_NE_U64</td>
</tr>
<tr>
<td>238</td>
<td>V_CMP_GE_U64</td>
</tr>
<tr>
<td>239</td>
<td>V_CMP_T_U64</td>
</tr>
<tr>
<td>240</td>
<td>V_CMPX_F_I64</td>
</tr>
<tr>
<td>241</td>
<td>V_CMPX_LT_I64</td>
</tr>
<tr>
<td>242</td>
<td>V_CMPX_EQ_I64</td>
</tr>
<tr>
<td>243</td>
<td>V_CMPX_LE_I64</td>
</tr>
<tr>
<td>244</td>
<td>V_CMPX_GT_I64</td>
</tr>
<tr>
<td>245</td>
<td>V_CMPX_NE_I64</td>
</tr>
<tr>
<td>246</td>
<td>V_CMPX_GE_I64</td>
</tr>
<tr>
<td>247</td>
<td>V_CMPX_T_I64</td>
</tr>
<tr>
<td>248</td>
<td>V_CMPX_F_U64</td>
</tr>
<tr>
<td>249</td>
<td>V_CMPX_LT_U64</td>
</tr>
<tr>
<td>250</td>
<td>V_CMPX_EQ_U64</td>
</tr>
<tr>
<td>251</td>
<td>V_CMPX_LE_U64</td>
</tr>
<tr>
<td>252</td>
<td>V_CMPX_GT_U64</td>
</tr>
<tr>
<td>253</td>
<td>V_CMPX_NE_U64</td>
</tr>
<tr>
<td>254</td>
<td>V_CMPX_GE_U64</td>
</tr>
<tr>
<td>255</td>
<td>V_CMPX_T_U64</td>
</tr>
</tbody>
</table>
13.3. Vector ALU Formats

### 13.3.4. VOP3A

**Format**  
VOP3A

**Description**  
Vector ALU format with three operands

#### Table 72. VOP3A Fields

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bits</th>
<th>Format or Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VDST</td>
<td>[7:0]</td>
<td>Destination VGPR</td>
</tr>
<tr>
<td>CLMP</td>
<td>[15]</td>
<td>Clamp output</td>
</tr>
<tr>
<td>ENCODING</td>
<td>[31:26]</td>
<td>Must be: 110100</td>
</tr>
<tr>
<td>Field Name</td>
<td>Bits</td>
<td>Format or Description</td>
</tr>
<tr>
<td>------------</td>
<td>--------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>SRC0</td>
<td>[40:32]</td>
<td>Source 0. First operand for the instruction.</td>
</tr>
<tr>
<td></td>
<td>0 - 101</td>
<td>SGPR0 to SGPR101: Scalar general-purpose registers.</td>
</tr>
<tr>
<td></td>
<td>102</td>
<td>FLAT_SCRATCH_LO.</td>
</tr>
<tr>
<td></td>
<td>103</td>
<td>FLAT_SCRATCH_HI.</td>
</tr>
<tr>
<td></td>
<td>104</td>
<td>XNACK_MASK_LO.</td>
</tr>
<tr>
<td></td>
<td>105</td>
<td>XNACK_MASK_HI.</td>
</tr>
<tr>
<td></td>
<td>106</td>
<td>VCC_LO: vcc[31:0].</td>
</tr>
<tr>
<td></td>
<td>107</td>
<td>VCC_HI: vcc[63:32].</td>
</tr>
<tr>
<td></td>
<td>108-123</td>
<td>TTMP0 - TTMP15: Trap handler temporary register.</td>
</tr>
<tr>
<td></td>
<td>124</td>
<td>M0. Memory register 0.</td>
</tr>
<tr>
<td></td>
<td>125</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>126</td>
<td>EXEC_LO: exec[31:0].</td>
</tr>
<tr>
<td></td>
<td>127</td>
<td>EXEC_HI: exec[63:32].</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>0.</td>
</tr>
<tr>
<td></td>
<td>129-192</td>
<td>Signed integer 1 to 64.</td>
</tr>
<tr>
<td></td>
<td>193-208</td>
<td>Signed integer -1 to -16.</td>
</tr>
<tr>
<td></td>
<td>209-234</td>
<td>Reserved.</td>
</tr>
<tr>
<td></td>
<td>235</td>
<td>SHARED_BASE (Memory Aperture definition).</td>
</tr>
<tr>
<td></td>
<td>236</td>
<td>SHARED_LIMIT (Memory Aperture definition).</td>
</tr>
<tr>
<td></td>
<td>237</td>
<td>PRIVATE_BASE (Memory Aperture definition).</td>
</tr>
<tr>
<td></td>
<td>238</td>
<td>PRIVATE_LIMIT (Memory Aperture definition).</td>
</tr>
<tr>
<td></td>
<td>239</td>
<td>POPS_EXITING_WAVE_ID .</td>
</tr>
<tr>
<td></td>
<td>240</td>
<td>0.5.</td>
</tr>
<tr>
<td></td>
<td>241</td>
<td>-0.5.</td>
</tr>
<tr>
<td></td>
<td>242</td>
<td>1.0.</td>
</tr>
<tr>
<td></td>
<td>243</td>
<td>-1.0.</td>
</tr>
<tr>
<td></td>
<td>244</td>
<td>2.0.</td>
</tr>
<tr>
<td></td>
<td>245</td>
<td>-2.0.</td>
</tr>
<tr>
<td></td>
<td>246</td>
<td>4.0.</td>
</tr>
<tr>
<td></td>
<td>247</td>
<td>-4.0.</td>
</tr>
<tr>
<td></td>
<td>248</td>
<td>1/(2*pi).</td>
</tr>
<tr>
<td></td>
<td>249</td>
<td>SDWA</td>
</tr>
<tr>
<td></td>
<td>250</td>
<td>DPP</td>
</tr>
<tr>
<td></td>
<td>251</td>
<td>VCCZ.</td>
</tr>
<tr>
<td></td>
<td>252</td>
<td>EXECZ.</td>
</tr>
<tr>
<td></td>
<td>253</td>
<td>SCC.</td>
</tr>
<tr>
<td></td>
<td>254</td>
<td>Reserved.</td>
</tr>
<tr>
<td></td>
<td>255</td>
<td>Literal constant.</td>
</tr>
<tr>
<td></td>
<td>256 - 511</td>
<td>VGPR 0 - 255</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bits</th>
<th>Format or Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRC1</td>
<td>[49:41]</td>
<td>Second input operand. Same options as SRC0.</td>
</tr>
<tr>
<td>SRC2</td>
<td>[58:50]</td>
<td>Third input operand. Same options as SRC0.</td>
</tr>
<tr>
<td>OMOD</td>
<td>[60:59]</td>
<td>Output Modifier: 0=none, 1=*2, 2=*4, 3=div-2</td>
</tr>
</tbody>
</table>

Table 73. VOP3A Opcodes

<table>
<thead>
<tr>
<th>Opcode #</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>448</td>
<td>V_MAD_LEGACY_F32</td>
</tr>
<tr>
<td>Opcode #</td>
<td>Name</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------</td>
</tr>
<tr>
<td>449</td>
<td>V_MAD_F32</td>
</tr>
<tr>
<td>450</td>
<td>V_MAD_I32_I24</td>
</tr>
<tr>
<td>451</td>
<td>V_MAD_U32_U24</td>
</tr>
<tr>
<td>452</td>
<td>V_CUBEID_F32</td>
</tr>
<tr>
<td>453</td>
<td>V_CUBESC_F32</td>
</tr>
<tr>
<td>454</td>
<td>V_CUBETC_F32</td>
</tr>
<tr>
<td>455</td>
<td>V_CUBEMA_F32</td>
</tr>
<tr>
<td>456</td>
<td>V_BFE_U32</td>
</tr>
<tr>
<td>457</td>
<td>V_BFE_I32</td>
</tr>
<tr>
<td>458</td>
<td>V_BFI_B32</td>
</tr>
<tr>
<td>459</td>
<td>V_FMA_F32</td>
</tr>
<tr>
<td>460</td>
<td>V_FMA_F64</td>
</tr>
<tr>
<td>461</td>
<td>V_LERP_U8</td>
</tr>
<tr>
<td>462</td>
<td>V_ALIGNBIT_B32</td>
</tr>
<tr>
<td>463</td>
<td>V_ALIGNBYTE_B32</td>
</tr>
<tr>
<td>464</td>
<td>V_MIN3_F32</td>
</tr>
<tr>
<td>465</td>
<td>V_MIN3_I32</td>
</tr>
<tr>
<td>466</td>
<td>V_MIN3_U32</td>
</tr>
<tr>
<td>467</td>
<td>V_MAX3_F32</td>
</tr>
<tr>
<td>468</td>
<td>V_MAX3_I32</td>
</tr>
<tr>
<td>469</td>
<td>V_MAX3_U32</td>
</tr>
<tr>
<td>470</td>
<td>V_MED3_F32</td>
</tr>
<tr>
<td>471</td>
<td>V_MED3_I32</td>
</tr>
<tr>
<td>472</td>
<td>V_MED3_U32</td>
</tr>
<tr>
<td>473</td>
<td>V_SAD_U8</td>
</tr>
<tr>
<td>474</td>
<td>V_SAD_HI_U8</td>
</tr>
<tr>
<td>475</td>
<td>V_SAD_U16</td>
</tr>
<tr>
<td>476</td>
<td>V_SAD_U32</td>
</tr>
<tr>
<td>477</td>
<td>V_CVT_PK_U8_F32</td>
</tr>
<tr>
<td>478</td>
<td>V_DIV_FIXUP_F32</td>
</tr>
<tr>
<td>479</td>
<td>V_DIV_FIXUP_F64</td>
</tr>
<tr>
<td>482</td>
<td>V_DIV_FMAS_F32</td>
</tr>
<tr>
<td>483</td>
<td>V_DIV_FMAS_F64</td>
</tr>
<tr>
<td>Opcode #</td>
<td>Name</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------------------------------</td>
</tr>
<tr>
<td>484</td>
<td>V_MSAD_U8</td>
</tr>
<tr>
<td>485</td>
<td>V_QSAD_PK_U16_U8</td>
</tr>
<tr>
<td>486</td>
<td>V_MQSAD_PK_U16_U8</td>
</tr>
<tr>
<td>487</td>
<td>V_MQSAD_U32_U8</td>
</tr>
<tr>
<td>490</td>
<td>V_MAD_LEGACY_F16</td>
</tr>
<tr>
<td>491</td>
<td>V_MAD_LEGACY_U16</td>
</tr>
<tr>
<td>492</td>
<td>V_MAD_LEGACY_I16</td>
</tr>
<tr>
<td>493</td>
<td>V_PERM_B32</td>
</tr>
<tr>
<td>494</td>
<td>V_FMA_LEGACY_F16</td>
</tr>
<tr>
<td>495</td>
<td>V_DIV_FIXUP_LEGACY_F16</td>
</tr>
<tr>
<td>496</td>
<td>V_CVT_PKACCUM_U8_F32</td>
</tr>
<tr>
<td>497</td>
<td>V_MAD_U32_U16</td>
</tr>
<tr>
<td>498</td>
<td>V_MAD_I32_I16</td>
</tr>
<tr>
<td>499</td>
<td>V_XAD_U32</td>
</tr>
<tr>
<td>500</td>
<td>V_MIN3_F16</td>
</tr>
<tr>
<td>501</td>
<td>V_MIN3_I16</td>
</tr>
<tr>
<td>502</td>
<td>V_MIN3_U16</td>
</tr>
<tr>
<td>503</td>
<td>V_MAX3_F16</td>
</tr>
<tr>
<td>504</td>
<td>V_MAX3_I16</td>
</tr>
<tr>
<td>505</td>
<td>V_MAX3_U16</td>
</tr>
<tr>
<td>506</td>
<td>V_MED3_F16</td>
</tr>
<tr>
<td>507</td>
<td>V_MED3_I16</td>
</tr>
<tr>
<td>508</td>
<td>V_MED3_U16</td>
</tr>
<tr>
<td>509</td>
<td>V_LSHL_ADD_U32</td>
</tr>
<tr>
<td>510</td>
<td>V_ADD_LSHL_U32</td>
</tr>
<tr>
<td>511</td>
<td>V_ADD3_U32</td>
</tr>
<tr>
<td>512</td>
<td>V_LSHL_OR_B32</td>
</tr>
<tr>
<td>513</td>
<td>V_AND_OR_B32</td>
</tr>
<tr>
<td>514</td>
<td>V_OR3_B32</td>
</tr>
<tr>
<td>515</td>
<td>V_MAD_F16</td>
</tr>
<tr>
<td>516</td>
<td>V_MAD_U16</td>
</tr>
<tr>
<td>517</td>
<td>V_MAD_I16</td>
</tr>
<tr>
<td>518</td>
<td>V_FMA_F16</td>
</tr>
<tr>
<td>Opcode #</td>
<td>Name</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>519</td>
<td>V_DIV_FIXUP_F16</td>
</tr>
<tr>
<td>628</td>
<td>V_INTERP_P1LL_F16</td>
</tr>
<tr>
<td>629</td>
<td>V_INTERP_P1LV_F16</td>
</tr>
<tr>
<td>630</td>
<td>V_INTERP_P2_LEGACY_F16</td>
</tr>
<tr>
<td>631</td>
<td>V_INTERP_P2_F16</td>
</tr>
<tr>
<td>640</td>
<td>V_ADD_F64</td>
</tr>
<tr>
<td>641</td>
<td>V_MUL_F64</td>
</tr>
<tr>
<td>642</td>
<td>V_MIN_F64</td>
</tr>
<tr>
<td>643</td>
<td>V_MAX_F64</td>
</tr>
<tr>
<td>644</td>
<td>V_LDEXP_F64</td>
</tr>
<tr>
<td>645</td>
<td>V_MUL_LO_U32</td>
</tr>
<tr>
<td>646</td>
<td>V_MUL_HI_U32</td>
</tr>
<tr>
<td>647</td>
<td>V_MUL_HI_I32</td>
</tr>
<tr>
<td>648</td>
<td>V_LDEXP_F32</td>
</tr>
<tr>
<td>649</td>
<td>V_READLANE_B32</td>
</tr>
<tr>
<td>650</td>
<td>V_WRITELANE_B32</td>
</tr>
<tr>
<td>651</td>
<td>V_BCNT_U32_B32</td>
</tr>
<tr>
<td>652</td>
<td>V_MBCNT_LO_U32_B32</td>
</tr>
<tr>
<td>653</td>
<td>V_MBCNT_HI_U32_B32</td>
</tr>
<tr>
<td>655</td>
<td>V_LSHLREV_B64</td>
</tr>
<tr>
<td>656</td>
<td>V_LSHRREV_B64</td>
</tr>
<tr>
<td>657</td>
<td>V_ASHRREV_I64</td>
</tr>
<tr>
<td>658</td>
<td>V_TRIG_PREOP_F64</td>
</tr>
<tr>
<td>659</td>
<td>V_BFM_B32</td>
</tr>
<tr>
<td>660</td>
<td>V_CVT_PKNORM_I16_F32</td>
</tr>
<tr>
<td>661</td>
<td>V_CVT_PKNORM_U16_F32</td>
</tr>
<tr>
<td>662</td>
<td>V_CVT_PKRTZ_F16_F32</td>
</tr>
<tr>
<td>663</td>
<td>V_CVT_PK_U16_U32</td>
</tr>
<tr>
<td>664</td>
<td>V_CVT_PK_I16_I32</td>
</tr>
<tr>
<td>665</td>
<td>V_CVT_PKNORM_I16_F16</td>
</tr>
<tr>
<td>666</td>
<td>V_CVT_PKNORM_U16_F16</td>
</tr>
<tr>
<td>668</td>
<td>V_ADD_I32</td>
</tr>
<tr>
<td>669</td>
<td>V_SUB_I32</td>
</tr>
</tbody>
</table>
 Opcode # | Name                
----------|------------------
   670    | V_ADD_I16       
   671    | V_SUB_I16       
   672    | V_PACK_B32_F16  

### 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 74. VOP3B Fields**

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bits</th>
<th>Format or Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VDST</td>
<td>[7:0]</td>
<td>Destination VGPR</td>
</tr>
<tr>
<td>SDST</td>
<td>[14:8]</td>
<td>Scalar destination</td>
</tr>
<tr>
<td>CLMP</td>
<td>[15]</td>
<td>Clamp result</td>
</tr>
<tr>
<td>ENCODING</td>
<td>[31:26]</td>
<td>Must be: 110100</td>
</tr>
</tbody>
</table>
### 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 | POPS_EXITING_WAVE_ID.
| 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

<table>
<thead>
<tr>
<th>Opcode #</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>480</td>
<td>V_DIV_SCALE_F32</td>
</tr>
</tbody>
</table>

**Table 75. VOP3B Opcodes**
<table>
<thead>
<tr>
<th>Opcode #</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>481</td>
<td>V_DIV_SCALE_F64</td>
</tr>
<tr>
<td>488</td>
<td>V_MAD_U64_U32</td>
</tr>
<tr>
<td>489</td>
<td>V_MAD_I64_I32</td>
</tr>
</tbody>
</table>

### 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 76. VOP3P Fields**

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bits</th>
<th>Format or Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VDST</td>
<td>[7:0]</td>
<td>Destination VGPR</td>
</tr>
<tr>
<td>NEG_HI</td>
<td>[10:8]</td>
<td>Negate sources 0,1,2 of the high 16-bits.</td>
</tr>
<tr>
<td>OPSEL</td>
<td>[13:11]</td>
<td>Select low or high for low sources 0=[11], 1=[12], 2=[13].</td>
</tr>
<tr>
<td>OPSEL_HI2</td>
<td>[14]</td>
<td>Select low or high for high sources 0=[14], 1=[60], 2=[59].</td>
</tr>
<tr>
<td>ENCODING</td>
<td>[31:24]</td>
<td>Must be: 11010011</td>
</tr>
<tr>
<td>Field Name</td>
<td>Bits</td>
<td>Format or Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>SRC0</td>
<td>[40:32]</td>
<td>Source 0. First operand for the instruction.</td>
</tr>
<tr>
<td></td>
<td>0 - 101</td>
<td>SGPR0 to SGPR101: Scalar general-purpose registers.</td>
</tr>
<tr>
<td></td>
<td>102</td>
<td>FLAT_SCRATCH_LO.</td>
</tr>
<tr>
<td></td>
<td>103</td>
<td>FLAT_SCRATCH_HI.</td>
</tr>
<tr>
<td></td>
<td>104</td>
<td>XNACK_MASK_LO.</td>
</tr>
<tr>
<td></td>
<td>105</td>
<td>XNACK_MASK_HI.</td>
</tr>
<tr>
<td></td>
<td>106</td>
<td>VCC_LO: vcc[31:0].</td>
</tr>
<tr>
<td></td>
<td>107</td>
<td>VCC_HI: vcc[63:32].</td>
</tr>
<tr>
<td></td>
<td>108-123</td>
<td>TTMP0 - TTMP15: Trap handler temporary register.</td>
</tr>
<tr>
<td></td>
<td>124</td>
<td>M0. Memory register 0.</td>
</tr>
<tr>
<td></td>
<td>125</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>126</td>
<td>EXEC_LO: exec[31:0].</td>
</tr>
<tr>
<td></td>
<td>127</td>
<td>EXEC_HI: exec[63:32].</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>0.</td>
</tr>
<tr>
<td></td>
<td>129-192</td>
<td>Signed integer 1 to 64.</td>
</tr>
<tr>
<td></td>
<td>193-208</td>
<td>Signed integer -1 to -16.</td>
</tr>
<tr>
<td></td>
<td>209-234</td>
<td>Reserved.</td>
</tr>
<tr>
<td></td>
<td>235</td>
<td>SHARED_BASE (Memory Aperture definition).</td>
</tr>
<tr>
<td></td>
<td>236</td>
<td>SHARED_LIMIT (Memory Aperture definition).</td>
</tr>
<tr>
<td></td>
<td>237</td>
<td>PRIVATE_BASE (Memory Aperture definition).</td>
</tr>
<tr>
<td></td>
<td>238</td>
<td>PRIVATE_LIMIT (Memory Aperture definition).</td>
</tr>
<tr>
<td></td>
<td>239</td>
<td>POPS_EXITING_WAVE_ID.</td>
</tr>
<tr>
<td></td>
<td>240</td>
<td>0.5.</td>
</tr>
<tr>
<td></td>
<td>241</td>
<td>-0.5.</td>
</tr>
<tr>
<td></td>
<td>242</td>
<td>1.0.</td>
</tr>
<tr>
<td></td>
<td>243</td>
<td>-1.0.</td>
</tr>
<tr>
<td></td>
<td>244</td>
<td>2.0.</td>
</tr>
<tr>
<td></td>
<td>245</td>
<td>-2.0.</td>
</tr>
<tr>
<td></td>
<td>246</td>
<td>4.0.</td>
</tr>
<tr>
<td></td>
<td>247</td>
<td>-4.0.</td>
</tr>
<tr>
<td></td>
<td>248</td>
<td>1/(2*PI).</td>
</tr>
<tr>
<td></td>
<td>249</td>
<td>SDWA</td>
</tr>
<tr>
<td></td>
<td>250</td>
<td>DPP</td>
</tr>
<tr>
<td></td>
<td>251</td>
<td>VCCZ.</td>
</tr>
<tr>
<td></td>
<td>252</td>
<td>EXECZ.</td>
</tr>
<tr>
<td></td>
<td>253</td>
<td>SCC.</td>
</tr>
<tr>
<td></td>
<td>254</td>
<td>Reserved.</td>
</tr>
<tr>
<td></td>
<td>255</td>
<td>Literal constant.</td>
</tr>
<tr>
<td></td>
<td>256 - 511</td>
<td>VGPR 0 - 255</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bits</th>
<th>Format or Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRC1</td>
<td>[49:41]</td>
<td>Second input operand. Same options as SRC0.</td>
</tr>
<tr>
<td>SRC2</td>
<td>[58:50]</td>
<td>Third input operand. Same options as SRC0.</td>
</tr>
</tbody>
</table>

**Table 77. VOP3P Opcodes**

<table>
<thead>
<tr>
<th>Opcode #</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>V_PK_MAD_I16</td>
</tr>
</tbody>
</table>

13.3. Vector ALU Formats
### 13.3.7. SDWA

<table>
<thead>
<tr>
<th>Opcode #</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>V_PK_MUL_LO_U16</td>
</tr>
<tr>
<td>2</td>
<td>V_PK_ADD_I16</td>
</tr>
<tr>
<td>3</td>
<td>V_PK_SUB_I16</td>
</tr>
<tr>
<td>4</td>
<td>V_PK_LSHLREV_B16</td>
</tr>
<tr>
<td>5</td>
<td>V_PK_LSHRREV_B16</td>
</tr>
<tr>
<td>6</td>
<td>V_PK_ASHRREV_I16</td>
</tr>
<tr>
<td>7</td>
<td>V_PK_MAX_I16</td>
</tr>
<tr>
<td>8</td>
<td>V_PK_MIN_I16</td>
</tr>
<tr>
<td>9</td>
<td>V_PK_MAD_U16</td>
</tr>
<tr>
<td>10</td>
<td>V_PK_ADD_U16</td>
</tr>
<tr>
<td>11</td>
<td>V_PK_SUB_U16</td>
</tr>
<tr>
<td>12</td>
<td>V_PK_MAX_U16</td>
</tr>
<tr>
<td>13</td>
<td>V_PK_MIN_U16</td>
</tr>
<tr>
<td>14</td>
<td>V_PK_FMA_F16</td>
</tr>
<tr>
<td>15</td>
<td>V_PK_ADD_F16</td>
</tr>
<tr>
<td>16</td>
<td>V_PK_MUL_F16</td>
</tr>
<tr>
<td>17</td>
<td>V_PK_MIN_F16</td>
</tr>
<tr>
<td>18</td>
<td>V_PK_MAX_F16</td>
</tr>
<tr>
<td>32</td>
<td>V_MAD_MIX_F32</td>
</tr>
<tr>
<td>33</td>
<td>V_MAD_MIXLO_F16</td>
</tr>
<tr>
<td>34</td>
<td>V_MAD_MIXHI_F16</td>
</tr>
<tr>
<td>35</td>
<td>V_DOT2_F32_F16</td>
</tr>
<tr>
<td>38</td>
<td>V_DOT2_I32_I16</td>
</tr>
<tr>
<td>39</td>
<td>V_DOT2_U32_U16</td>
</tr>
<tr>
<td>40</td>
<td>V_DOT4_I32_I8</td>
</tr>
<tr>
<td>41</td>
<td>V_DOT4_U32_U8</td>
</tr>
<tr>
<td>42</td>
<td>V_DOT8_I32_I4</td>
</tr>
<tr>
<td>43</td>
<td>V_DOT8_U32_U4</td>
</tr>
</tbody>
</table>

---

**13.3. Vector ALU Formats**
**Format**  
SDWA

**Description**  
Sub-Dword Addressing. This is a second dwpord 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>
<thead>
<tr>
<th>Field Name</th>
<th>Bits</th>
<th>Format or Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRC0</td>
<td>[39:32]</td>
<td>Real SRC0 operand (VGPR).</td>
</tr>
</tbody>
</table>
| DST_SEL      | [42:40] | Select the data destination:  
0 = data[7:0]  
1 = data[15:8]  
2 = data[23:16]  
3 = data[31:24]  
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                                                                       |
| 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>
<thead>
<tr>
<th>Field Name</th>
<th>Bits</th>
<th>Format or Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRC0</td>
<td>[39:32]</td>
<td>Real SRC0 operand (VGPR).</td>
</tr>
<tr>
<td>SDST</td>
<td>[46:40]</td>
<td>Scalar GPR destination.</td>
</tr>
<tr>
<td>SD</td>
<td>[47]</td>
<td>Scalar destination type: 0 = VCC, 1 = normal SGPR.</td>
</tr>
<tr>
<td>SRC0_SEL</td>
<td>[50:48]</td>
<td>Source 0 select. Same options as DST_SEL.</td>
</tr>
<tr>
<td>SRC0_SEXT</td>
<td>[51]</td>
<td>Sign extend modifier for source 0.</td>
</tr>
<tr>
<td>SRC0_NEG</td>
<td>[52]</td>
<td>1 = negate source 0.</td>
</tr>
<tr>
<td>SRC0_ABS</td>
<td>[53]</td>
<td>1 = Absolute value of source 0.</td>
</tr>
<tr>
<td>S0</td>
<td>[55]</td>
<td>0 = source 0 is VGPR, 1 = is SGPR.</td>
</tr>
<tr>
<td>SRC1_SEL</td>
<td>[58:56]</td>
<td>Same options as SRC0_SEL.</td>
</tr>
<tr>
<td>SRC1_SEXT</td>
<td>[59]</td>
<td>Sign extend modifier for source 1.</td>
</tr>
<tr>
<td>SRC1_NEG</td>
<td>[60]</td>
<td>1 = negate source 1.</td>
</tr>
<tr>
<td>SRC1_ABS</td>
<td>[61]</td>
<td>1 = Absolute value of source 1.</td>
</tr>
<tr>
<td>S1</td>
<td>[63]</td>
<td>0 = source 1 is VGPR, 1 = is SGPR.</td>
</tr>
</tbody>
</table>

13.3.9. DPP

**Format**

DPP

13.3. Vector ALU Formats
**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 80. DPP Fields**

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bits</th>
<th>Format or Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRC0</td>
<td>[39:32]</td>
<td>Real SRC0 operand (VGPR).</td>
</tr>
<tr>
<td>DPP_CTRL</td>
<td>[48:40]</td>
<td>See next table: &quot;DPP_CTRL Enumeration&quot;</td>
</tr>
<tr>
<td>BC</td>
<td>[51]</td>
<td>Bounds Control: 0 = do not write when source is out of range, 1 = write.</td>
</tr>
<tr>
<td>SRC0_NEG</td>
<td>[52]</td>
<td>1 = negate source 0.</td>
</tr>
<tr>
<td>SRC0_ABS</td>
<td>[53]</td>
<td>1 = Absolute value of source 0.</td>
</tr>
<tr>
<td>SRC1_NEG</td>
<td>[54]</td>
<td>1 = negate source 1.</td>
</tr>
<tr>
<td>SRC1_ABS</td>
<td>[55]</td>
<td>1 = Absolute value of source 1.</td>
</tr>
</tbody>
</table>
| 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  
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 81. DPP_CTRL Enumeration**

<table>
<thead>
<tr>
<th>DPP_Cntl Enumeration</th>
<th>Hex Value</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPP_QUAD_PERM*</td>
<td>000-0FF</td>
<td>pix[n].srca = pix[(n&amp;0x3c)+ dpp_cntl[n%4<em>2+1 : n%4</em>2]].srca</td>
<td>Permute of four threads.</td>
</tr>
<tr>
<td>DPP_UNUSED</td>
<td>100</td>
<td>Undefined</td>
<td>Reserved.</td>
</tr>
<tr>
<td>DPP_ROW_SL*</td>
<td>101-10F</td>
<td>if (n&amp;0xf) &lt; (16-cntl[3:0]) pix[n].srca = pix[n + cntl[3:0]].srca else use bound_cntl</td>
<td>Row shift left by 1-15 threads.</td>
</tr>
<tr>
<td>DPP_ROW_SR*</td>
<td>111-11F</td>
<td>if ((n&amp;0xf) &gt;= cntl[3:0]) pix[n].srca = pix[n - cntl[3:0]].srca else use bound_cntl</td>
<td>Row shift right by 1-15 threads.</td>
</tr>
<tr>
<td>DPP_WF_SL1*</td>
<td>130</td>
<td>if (n&lt;63) pix[n].srca = pix[n+1].srca else use bound_cntl</td>
<td>Wavefront left shift by 1 thread.</td>
</tr>
</tbody>
</table>
### 13.4. Vector Parameter Interpolation Format

#### 13.4.1. VINTRP

**Format**

VINTRP

**Description**

Vector Parameter Interpolation. These opcodes perform parameter interpolation using vertex data in pixel shaders.

**Table 82. VINTRP Fields**

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bits</th>
<th>Format or Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSRC</td>
<td>[7:0]</td>
<td>SRC0 operand (VGPR).</td>
</tr>
<tr>
<td>ATTR_CHAN</td>
<td>[9:8]</td>
<td>Attribute channel: 0=X, 1=Y, 2=Z, 3=W</td>
</tr>
<tr>
<td>OP</td>
<td>[17:16]</td>
<td>Opcode:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0: v_interp_p1_f32 : VDST = P10 * VSRC + P0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: v_interp_p2_f32: VDST = P20 * VSRC + VDST</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2: v_interp_mov_f32: VDST = (P0, P10 or P20 selected by VSRC[1:0])</td>
</tr>
<tr>
<td>VDST</td>
<td>[25:18]</td>
<td>Destination VGPR</td>
</tr>
</tbody>
</table>
Table 83. DS Fields

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bits</th>
<th>Format or Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFFSET0</td>
<td>[7:0]</td>
<td>First address offset</td>
</tr>
<tr>
<td>OFFSET1</td>
<td>[15:8]</td>
<td>Second address offset. For some opcodes this is concatenated with OFFSET0.</td>
</tr>
<tr>
<td>GDS</td>
<td>[16]</td>
<td>1=GDS, 0=LDS operation.</td>
</tr>
<tr>
<td>ENCODING</td>
<td>[31:26]</td>
<td>Must be: 110110</td>
</tr>
<tr>
<td>ADDR</td>
<td>[39:32]</td>
<td>VGPR which supplies the address.</td>
</tr>
<tr>
<td>DATA0</td>
<td>[47:40]</td>
<td>First data VGPR.</td>
</tr>
<tr>
<td>DATA1</td>
<td>[55:48]</td>
<td>Second data VGPR.</td>
</tr>
<tr>
<td>VDST</td>
<td>[63:56]</td>
<td>Destination VGPR when results returned to VGPRs.</td>
</tr>
</tbody>
</table>

Table 84. DS Opcodes

<table>
<thead>
<tr>
<th>Opcode #</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DS_ADD_U32</td>
</tr>
<tr>
<td>1</td>
<td>DS_SUB_U32</td>
</tr>
<tr>
<td>2</td>
<td>DS_RSUB_U32</td>
</tr>
<tr>
<td>3</td>
<td>DS_INC_U32</td>
</tr>
<tr>
<td>4</td>
<td>DS_DEC_U32</td>
</tr>
</tbody>
</table>

VSRC must be different from VDST.

13.5. LDS and GDS format

13.5.1. DS

Format LDS and GDS

Description Local and Global Data Sharing instructions
<table>
<thead>
<tr>
<th>Opcode #</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>DS_MIN_I32</td>
</tr>
<tr>
<td>6</td>
<td>DS_MAX_I32</td>
</tr>
<tr>
<td>7</td>
<td>DS_MIN_U32</td>
</tr>
<tr>
<td>8</td>
<td>DS_MAX_U32</td>
</tr>
<tr>
<td>9</td>
<td>DS_AND_B32</td>
</tr>
<tr>
<td>10</td>
<td>DS_OR_B32</td>
</tr>
<tr>
<td>11</td>
<td>DS_XOR_B32</td>
</tr>
<tr>
<td>12</td>
<td>DS_MSKOR_B32</td>
</tr>
<tr>
<td>13</td>
<td>DS_WRITE_B32</td>
</tr>
<tr>
<td>14</td>
<td>DS_WRITE2_B32</td>
</tr>
<tr>
<td>15</td>
<td>DS_WRITE2ST64_B32</td>
</tr>
<tr>
<td>16</td>
<td>DS_CMPST_B32</td>
</tr>
<tr>
<td>17</td>
<td>DS_CMPST_F32</td>
</tr>
<tr>
<td>18</td>
<td>DS_MIN_F32</td>
</tr>
<tr>
<td>19</td>
<td>DS_MAX_F32</td>
</tr>
<tr>
<td>20</td>
<td>DS_NOP</td>
</tr>
<tr>
<td>21</td>
<td>DS_ADD_F32</td>
</tr>
<tr>
<td>29</td>
<td>DS_WRITE_ADDTID_B32</td>
</tr>
<tr>
<td>30</td>
<td>DS_WRITE_B8</td>
</tr>
<tr>
<td>31</td>
<td>DS_WRITE_B16</td>
</tr>
<tr>
<td>32</td>
<td>DS_ADD_RTN_U32</td>
</tr>
<tr>
<td>33</td>
<td>DS_SUB_RTN_U32</td>
</tr>
<tr>
<td>34</td>
<td>DS_RSUB_RTN_U32</td>
</tr>
<tr>
<td>35</td>
<td>DS_INC_RTN_U32</td>
</tr>
<tr>
<td>36</td>
<td>DS_DEC_RTN_U32</td>
</tr>
<tr>
<td>37</td>
<td>DS_MIN_RTN_I32</td>
</tr>
<tr>
<td>38</td>
<td>DS_MAX_RTN_I32</td>
</tr>
<tr>
<td>39</td>
<td>DS_MIN_RTN_U32</td>
</tr>
<tr>
<td>40</td>
<td>DS_MAX_RTN_U32</td>
</tr>
<tr>
<td>41</td>
<td>DS_AND_RTN_B32</td>
</tr>
<tr>
<td>42</td>
<td>DS_OR_RTN_B32</td>
</tr>
<tr>
<td>43</td>
<td>DS_XOR_RTN_B32</td>
</tr>
<tr>
<td>44</td>
<td>DS_MSKOR_RTN_B32</td>
</tr>
</tbody>
</table>

"Vega" 7nm Instruction Set Architecture
<table>
<thead>
<tr>
<th>Opcode #</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>DS_WRXCHG_RTN_B32</td>
</tr>
<tr>
<td>46</td>
<td>DS_WRXCHG2_RTN_B32</td>
</tr>
<tr>
<td>47</td>
<td>DS_WRXCHG2ST64_RTN_B32</td>
</tr>
<tr>
<td>48</td>
<td>DS_CMPST_RTN_B32</td>
</tr>
<tr>
<td>49</td>
<td>DS_CMPST_RTN_F32</td>
</tr>
<tr>
<td>50</td>
<td>DS_MIN_RTN_F32</td>
</tr>
<tr>
<td>51</td>
<td>DS_MAX_RTN_F32</td>
</tr>
<tr>
<td>52</td>
<td>DS_WRAP_RTN_B32</td>
</tr>
<tr>
<td>53</td>
<td>DS_ADD_RTN_F32</td>
</tr>
<tr>
<td>54</td>
<td>DS_READ_B32</td>
</tr>
<tr>
<td>55</td>
<td>DS_READ2_B32</td>
</tr>
<tr>
<td>56</td>
<td>DS_READ2ST64_B32</td>
</tr>
<tr>
<td>57</td>
<td>DS_READ_I8</td>
</tr>
<tr>
<td>58</td>
<td>DS_READ_U8</td>
</tr>
<tr>
<td>59</td>
<td>DS_READ_I16</td>
</tr>
<tr>
<td>60</td>
<td>DS_READ_U16</td>
</tr>
<tr>
<td>61</td>
<td>DS_SWIZZLE_B32</td>
</tr>
<tr>
<td>62</td>
<td>DS_PERMUTE_B32</td>
</tr>
<tr>
<td>63</td>
<td>DS_BPERMUTE_B32</td>
</tr>
<tr>
<td>64</td>
<td>DS_ADD_U64</td>
</tr>
<tr>
<td>65</td>
<td>DS_SUB_U64</td>
</tr>
<tr>
<td>66</td>
<td>DS_RSUB_U64</td>
</tr>
<tr>
<td>67</td>
<td>DS_INC_U64</td>
</tr>
<tr>
<td>68</td>
<td>DS_DEC_U64</td>
</tr>
<tr>
<td>69</td>
<td>DS_MIN_I64</td>
</tr>
<tr>
<td>70</td>
<td>DS_MAX_I64</td>
</tr>
<tr>
<td>71</td>
<td>DS_MIN_U64</td>
</tr>
<tr>
<td>72</td>
<td>DS_MAX_U64</td>
</tr>
<tr>
<td>73</td>
<td>DS_AND_B64</td>
</tr>
<tr>
<td>74</td>
<td>DS_OR_B64</td>
</tr>
<tr>
<td>75</td>
<td>DS_XOR_B64</td>
</tr>
<tr>
<td>76</td>
<td>DS_MSKOR_B64</td>
</tr>
<tr>
<td>77</td>
<td>DS_WRITE_B64</td>
</tr>
<tr>
<td>Opcode #</td>
<td>Name</td>
</tr>
<tr>
<td>---------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>78</td>
<td>DS_WRITE2_B64</td>
</tr>
<tr>
<td>79</td>
<td>DS_WRITE2ST64_B64</td>
</tr>
<tr>
<td>80</td>
<td>DS_CMPST_B64</td>
</tr>
<tr>
<td>81</td>
<td>DS_CMPST_F64</td>
</tr>
<tr>
<td>82</td>
<td>DS_MIN_F64</td>
</tr>
<tr>
<td>83</td>
<td>DS_MAX_F64</td>
</tr>
<tr>
<td>84</td>
<td>DS_WRITE_B8_D16_HI</td>
</tr>
<tr>
<td>85</td>
<td>DS_WRITE_B16_D16_HI</td>
</tr>
<tr>
<td>86</td>
<td>DS_READ_U8_D16</td>
</tr>
<tr>
<td>87</td>
<td>DS_READ_U8_D16_HI</td>
</tr>
<tr>
<td>88</td>
<td>DS_READ_I8_D16</td>
</tr>
<tr>
<td>89</td>
<td>DS_READ_I8_D16_HI</td>
</tr>
<tr>
<td>90</td>
<td>DS_READ_U16_D16</td>
</tr>
<tr>
<td>91</td>
<td>DS_READ_U16_D16_HI</td>
</tr>
<tr>
<td>96</td>
<td>DS_ADD_RTN_U64</td>
</tr>
<tr>
<td>97</td>
<td>DS_SUB_RTN_U64</td>
</tr>
<tr>
<td>98</td>
<td>DS_RSUB_RTN_U64</td>
</tr>
<tr>
<td>99</td>
<td>DS_INC_RTN_U64</td>
</tr>
<tr>
<td>100</td>
<td>DS_DEC_RTN_U64</td>
</tr>
<tr>
<td>101</td>
<td>DS_MIN_RTN_I64</td>
</tr>
<tr>
<td>102</td>
<td>DS_MAX_RTN_I64</td>
</tr>
<tr>
<td>103</td>
<td>DS_MIN_RTN_U64</td>
</tr>
<tr>
<td>104</td>
<td>DS_MAX_RTN_U64</td>
</tr>
<tr>
<td>105</td>
<td>DS_AND_RTN_B64</td>
</tr>
<tr>
<td>106</td>
<td>DS_OR_RTN_B64</td>
</tr>
<tr>
<td>107</td>
<td>DS_XOR_RTN_B64</td>
</tr>
<tr>
<td>108</td>
<td>DS_MSKOR_RTN_B64</td>
</tr>
<tr>
<td>109</td>
<td>DS_WRXCHG_RTN_B64</td>
</tr>
<tr>
<td>110</td>
<td>DS_WRXCHG2_RTN_B64</td>
</tr>
<tr>
<td>111</td>
<td>DS_WRXCHG2ST64_RTN_B64</td>
</tr>
<tr>
<td>112</td>
<td>DS_CMPST_RTN_B64</td>
</tr>
<tr>
<td>113</td>
<td>DS_CMPST_RTN_F64</td>
</tr>
<tr>
<td>114</td>
<td>DS_MIN_RTN_F64</td>
</tr>
<tr>
<td>Opcode #</td>
<td>Name</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------------------------</td>
</tr>
<tr>
<td>115</td>
<td>DS_MAX_RTN_F64</td>
</tr>
<tr>
<td>118</td>
<td>DS_READ_B64</td>
</tr>
<tr>
<td>119</td>
<td>DS_READ2_B64</td>
</tr>
<tr>
<td>120</td>
<td>DS_READ2ST64_B64</td>
</tr>
<tr>
<td>126</td>
<td>DS_CONDXCHG32_SUBT64_RTN_B64</td>
</tr>
<tr>
<td>128</td>
<td>DS_ADD_SRC2_U32</td>
</tr>
<tr>
<td>129</td>
<td>DS_SUB_SRC2_U32</td>
</tr>
<tr>
<td>130</td>
<td>DS_RSUB_SRC2_U32</td>
</tr>
<tr>
<td>131</td>
<td>DS_INC_SRC2_U32</td>
</tr>
<tr>
<td>132</td>
<td>DS_DEC_SRC2_U32</td>
</tr>
<tr>
<td>133</td>
<td>DS_MIN_SRC2_I32</td>
</tr>
<tr>
<td>134</td>
<td>DS_MAX_SRC2_I32</td>
</tr>
<tr>
<td>135</td>
<td>DS_MIN_SRC2_U32</td>
</tr>
<tr>
<td>136</td>
<td>DS_MAX_SRC2_U32</td>
</tr>
<tr>
<td>137</td>
<td>DS_AND_SRC2_B32</td>
</tr>
<tr>
<td>138</td>
<td>DS_OR_SRC2_B32</td>
</tr>
<tr>
<td>139</td>
<td>DS_XOR_SRC2_B32</td>
</tr>
<tr>
<td>141</td>
<td>DS_WRITE_SRC2_B32</td>
</tr>
<tr>
<td>146</td>
<td>DS_MIN_SRC2_F32</td>
</tr>
<tr>
<td>147</td>
<td>DS_MAX_SRC2_F32</td>
</tr>
<tr>
<td>149</td>
<td>DS_ADD_SRC2_F32</td>
</tr>
<tr>
<td>152</td>
<td>DS_GWS_SEMA_RELEASE_ALL</td>
</tr>
<tr>
<td>153</td>
<td>DS_GWS_INIT</td>
</tr>
<tr>
<td>154</td>
<td>DS_GWS_SEMA_V</td>
</tr>
<tr>
<td>155</td>
<td>DS_GWS_SEMA_BR</td>
</tr>
<tr>
<td>156</td>
<td>DS_GWS_SEMA_P</td>
</tr>
<tr>
<td>157</td>
<td>DS_GWS_BARRIER</td>
</tr>
<tr>
<td>182</td>
<td>DS_READ_ADDTID_B32</td>
</tr>
<tr>
<td>189</td>
<td>DS_CONSUME</td>
</tr>
<tr>
<td>190</td>
<td>DS_APPEND</td>
</tr>
<tr>
<td>191</td>
<td>DS_ORDERED_COUNT</td>
</tr>
<tr>
<td>192</td>
<td>DS_ADD_SRC2_U64</td>
</tr>
<tr>
<td>193</td>
<td>DS_SUB_SRC2_U64</td>
</tr>
<tr>
<td>Opcode #</td>
<td>Name</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>194</td>
<td>DS_RSUB_SRC2_U64</td>
</tr>
<tr>
<td>195</td>
<td>DS_INC_SRC2_U64</td>
</tr>
<tr>
<td>196</td>
<td>DS_DEC_SRC2_U64</td>
</tr>
<tr>
<td>197</td>
<td>DS_MIN_SRC2_I64</td>
</tr>
<tr>
<td>198</td>
<td>DS_MAX_SRC2_I64</td>
</tr>
<tr>
<td>199</td>
<td>DS_MIN_SRC2_U64</td>
</tr>
<tr>
<td>200</td>
<td>DS_MAX_SRC2_U64</td>
</tr>
<tr>
<td>201</td>
<td>DS_AND_SRC2_B64</td>
</tr>
<tr>
<td>202</td>
<td>DS_OR_SRC2_B64</td>
</tr>
<tr>
<td>203</td>
<td>DS_XOR_SRC2_B64</td>
</tr>
<tr>
<td>205</td>
<td>DS_WRITE_SRC2_B64</td>
</tr>
<tr>
<td>210</td>
<td>DS_MIN_SRC2_F64</td>
</tr>
<tr>
<td>211</td>
<td>DS_MAX_SRC2_F64</td>
</tr>
<tr>
<td>222</td>
<td>DS_WRITE_B96</td>
</tr>
<tr>
<td>223</td>
<td>DS_WRITE_B128</td>
</tr>
<tr>
<td>254</td>
<td>DS_READ_B96</td>
</tr>
<tr>
<td>255</td>
<td>DS_READ_B128</td>
</tr>
</tbody>
</table>

13.6. 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)

13.6.1. MTBUF
### Table 85. MTBUF Fields

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bits</th>
<th>Format or Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFFEN</td>
<td>[12]</td>
<td>1 = enable offset VGPR, 0 = use zero for address offset</td>
</tr>
<tr>
<td>IDXEN</td>
<td>[13]</td>
<td>1 = enable index VGPR, 0 = use zero for address index</td>
</tr>
<tr>
<td>GLC</td>
<td>[14]</td>
<td>0 = normal, 1 = globally coherent (bypass L0 cache) or for atomics, return pre-op value to VGPR.</td>
</tr>
<tr>
<td>DFMT</td>
<td>22:19</td>
<td>Data Format of data in memory buffer:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 invalid</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 8_8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 16_16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6 10_11_11</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 10_10_10_2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9 2_10_10_10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10 8_8_8_8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11 32_32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12 16_16_16_16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13 32_32_32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>14 32_32_32_32</td>
</tr>
<tr>
<td>NFMT</td>
<td>25:23</td>
<td>Numeric format of data in memory:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 unorm</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 snorm</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 uscaled</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 sscaled</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 uint</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 sint</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6 reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7 float</td>
</tr>
<tr>
<td>ENCODING</td>
<td>[31:26]</td>
<td>Must be: 111010</td>
</tr>
<tr>
<td>VADDR</td>
<td>[39:32]</td>
<td>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.</td>
</tr>
<tr>
<td>VDATA</td>
<td>[47:40]</td>
<td>Address of VGPR to supply first component of write data or receive first component of read-data.</td>
</tr>
<tr>
<td>SRSRC</td>
<td>[52:48]</td>
<td>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.</td>
</tr>
<tr>
<td>SLC</td>
<td>[54]</td>
<td>System level coherent: bypass L2 cache.</td>
</tr>
<tr>
<td>TFE</td>
<td>[55]</td>
<td>Partially resident texture, texture fail enable.</td>
</tr>
</tbody>
</table>
### Table 86. MTBUF Opcodes

<table>
<thead>
<tr>
<th>Opcode #</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>TBUFFER_LOAD_FORMAT_X</td>
</tr>
<tr>
<td>1</td>
<td>TBUFFER_LOAD_FORMAT_XY</td>
</tr>
<tr>
<td>2</td>
<td>TBUFFER_LOAD_FORMAT_XYZ</td>
</tr>
<tr>
<td>3</td>
<td>TBUFFER_LOAD_FORMAT_XYZW</td>
</tr>
<tr>
<td>4</td>
<td>TBUFFER_STORE_FORMAT_X</td>
</tr>
<tr>
<td>5</td>
<td>TBUFFER_STORE_FORMAT_XY</td>
</tr>
<tr>
<td>6</td>
<td>TBUFFER_STORE_FORMAT_XYZ</td>
</tr>
<tr>
<td>7</td>
<td>TBUFFER_STORE_FORMAT_XYZW</td>
</tr>
<tr>
<td>8</td>
<td>TBUFFER_LOAD_FORMAT_D16_X</td>
</tr>
<tr>
<td>9</td>
<td>TBUFFER_STORE_FORMAT_D16_XY</td>
</tr>
<tr>
<td>10</td>
<td>TBUFFER_LOAD_FORMAT_D16_XYZ</td>
</tr>
<tr>
<td>11</td>
<td>TBUFFER_LOAD_FORMAT_D16_XYZW</td>
</tr>
<tr>
<td>12</td>
<td>TBUFFER_STORE_FORMAT_D16_X</td>
</tr>
<tr>
<td>13</td>
<td>TBUFFER_STORE_FORMAT_D16_XY</td>
</tr>
<tr>
<td>14</td>
<td>TBUFFER_STORE_FORMAT_D16_XYZ</td>
</tr>
<tr>
<td>15</td>
<td>TBUFFER_STORE_FORMAT_D16_XYZW</td>
</tr>
</tbody>
</table>

### 13.6.2. MUBUF

**Format**

MUBUF

**Description**

Memory Untyped-Buffer Instructions

### Table 87. MUBUF Fields

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bits</th>
<th>Format or Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFFEN</td>
<td>[12]</td>
<td>1 = enable offset VGPR, 0 = use zero for address offset</td>
</tr>
<tr>
<td>Field Name</td>
<td>Bits</td>
<td>Format or Description</td>
</tr>
<tr>
<td>------------</td>
<td>------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>IDXEN</td>
<td>[13]</td>
<td>1 = enable index VGPR, 0 = use zero for address index</td>
</tr>
<tr>
<td>GLC</td>
<td>[14]</td>
<td>0 = normal, 1 = globally coherent (bypass L0 cache) or for atomics, return pre-op value to VGPR.</td>
</tr>
<tr>
<td>LDS</td>
<td>[16]</td>
<td>0 = normal, 1 = transfer data between LDS and memory instead of VGPRs and memory.</td>
</tr>
<tr>
<td>ENCODING</td>
<td>[31:26]</td>
<td>Must be: 111000</td>
</tr>
<tr>
<td>VADDR</td>
<td>[39:32]</td>
<td>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.</td>
</tr>
<tr>
<td>VDATA</td>
<td>[47:40]</td>
<td>Address of VGPR to supply first component of write data or receive first component of read-data.</td>
</tr>
<tr>
<td>SRSRC</td>
<td>[52:48]</td>
<td>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.</td>
</tr>
<tr>
<td>TFE</td>
<td>[55]</td>
<td>Partially resident texture, texture fail enable.</td>
</tr>
<tr>
<td>SOFFSET</td>
<td>[63:56]</td>
<td>Address offset, unsigned byte.</td>
</tr>
</tbody>
</table>

Table 88. MUBUF Opcodes

<table>
<thead>
<tr>
<th>Opcode #</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>BUFFER_LOAD_FORMAT_X</td>
</tr>
<tr>
<td>1</td>
<td>BUFFER_LOAD_FORMAT_XY</td>
</tr>
<tr>
<td>2</td>
<td>BUFFER_LOAD_FORMAT_XYZ</td>
</tr>
<tr>
<td>3</td>
<td>BUFFER_LOAD_FORMAT_XYZW</td>
</tr>
<tr>
<td>4</td>
<td>BUFFER_STORE_FORMAT_X</td>
</tr>
<tr>
<td>5</td>
<td>BUFFER_STORE_FORMAT_XY</td>
</tr>
<tr>
<td>6</td>
<td>BUFFER_STORE_FORMAT_XYZ</td>
</tr>
<tr>
<td>7</td>
<td>BUFFER_STORE_FORMAT_XYZW</td>
</tr>
<tr>
<td>8</td>
<td>BUFFER_LOAD_FORMAT_D16_X</td>
</tr>
<tr>
<td>9</td>
<td>BUFFER_LOAD_FORMAT_D16_XY</td>
</tr>
<tr>
<td>10</td>
<td>BUFFER_LOAD_FORMAT_D16_XYZ</td>
</tr>
<tr>
<td>11</td>
<td>BUFFER_LOAD_FORMAT_D16_XYZW</td>
</tr>
<tr>
<td>12</td>
<td>BUFFER_STORE_FORMAT_D16_X</td>
</tr>
<tr>
<td>13</td>
<td>BUFFER_STORE_FORMAT_D16_XY</td>
</tr>
<tr>
<td>14</td>
<td>BUFFER_STORE_FORMAT_D16_XYZ</td>
</tr>
<tr>
<td>Opcode #</td>
<td>Name</td>
</tr>
<tr>
<td>---------</td>
<td>-------------------------------------------</td>
</tr>
<tr>
<td>15</td>
<td>BUFFER_STORE_FORMAT_D16_XYZW</td>
</tr>
<tr>
<td>16</td>
<td>BUFFER_LOAD_UBYTE</td>
</tr>
<tr>
<td>17</td>
<td>BUFFER_LOAD_SBYTE</td>
</tr>
<tr>
<td>18</td>
<td>BUFFER_LOAD_U SHORT</td>
</tr>
<tr>
<td>19</td>
<td>BUFFER_LOAD_SSHORT</td>
</tr>
<tr>
<td>20</td>
<td>BUFFER_LOAD_DWORD</td>
</tr>
<tr>
<td>21</td>
<td>BUFFER_LOAD_DWORDX2</td>
</tr>
<tr>
<td>22</td>
<td>BUFFER_LOAD_DWORDX3</td>
</tr>
<tr>
<td>23</td>
<td>BUFFER_LOAD_DWORDX4</td>
</tr>
<tr>
<td>24</td>
<td>BUFFER_STORE_BYTE</td>
</tr>
<tr>
<td>25</td>
<td>BUFFER_STORE_BYTE_D16_HI</td>
</tr>
<tr>
<td>26</td>
<td>BUFFER_STORE_SHORT</td>
</tr>
<tr>
<td>27</td>
<td>BUFFER_STORE_SHORT_D16_HI</td>
</tr>
<tr>
<td>28</td>
<td>BUFFER_STORE_DWORD</td>
</tr>
<tr>
<td>29</td>
<td>BUFFER_STORE_DWORDX2</td>
</tr>
<tr>
<td>30</td>
<td>BUFFER_STORE_DWORDX3</td>
</tr>
<tr>
<td>31</td>
<td>BUFFER_STORE_DWORDX4</td>
</tr>
<tr>
<td>32</td>
<td>BUFFER_LOAD_UBYTE_D16</td>
</tr>
<tr>
<td>33</td>
<td>BUFFER_LOAD_UBYTE_D16_HI</td>
</tr>
<tr>
<td>34</td>
<td>BUFFER_LOAD_SBYTE_D16</td>
</tr>
<tr>
<td>35</td>
<td>BUFFER_LOAD_SBYTE_D16_HI</td>
</tr>
<tr>
<td>36</td>
<td>BUFFER_LOAD_SHORT_D16</td>
</tr>
<tr>
<td>37</td>
<td>BUFFER_LOAD_SHORT_D16_HI</td>
</tr>
<tr>
<td>38</td>
<td>BUFFER_LOAD_FORMAT_D16_HI_X</td>
</tr>
<tr>
<td>39</td>
<td>BUFFER_STORE_FORMAT_D16_HI_X</td>
</tr>
<tr>
<td>61</td>
<td>BUFFER_STORE_LDS_DWORD</td>
</tr>
<tr>
<td>62</td>
<td>BUFFER_WBINVL1</td>
</tr>
<tr>
<td>63</td>
<td>BUFFER_WBINVL1_VOL</td>
</tr>
<tr>
<td>64</td>
<td>BUFFER_ATOMIC_SWAP</td>
</tr>
<tr>
<td>65</td>
<td>BUFFER_ATOMIC_CMPSWAP</td>
</tr>
<tr>
<td>66</td>
<td>BUFFER_ATOMIC_ADD</td>
</tr>
<tr>
<td>67</td>
<td>BUFFER_ATOMIC_SUB</td>
</tr>
<tr>
<td>68</td>
<td>BUFFER_ATOMIC_SMIN</td>
</tr>
<tr>
<td>Opcode #</td>
<td>Name</td>
</tr>
<tr>
<td>---------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>69</td>
<td>BUFFER_ATOMIC_UMIN</td>
</tr>
<tr>
<td>70</td>
<td>BUFFER_ATOMIC_SMAX</td>
</tr>
<tr>
<td>71</td>
<td>BUFFER_ATOMIC_UMAX</td>
</tr>
<tr>
<td>72</td>
<td>BUFFER_ATOMIC_AND</td>
</tr>
<tr>
<td>73</td>
<td>BUFFER_ATOMIC_OR</td>
</tr>
<tr>
<td>74</td>
<td>BUFFER_ATOMIC_XOR</td>
</tr>
<tr>
<td>75</td>
<td>BUFFER_ATOMIC_INC</td>
</tr>
<tr>
<td>76</td>
<td>BUFFER_ATOMIC_DEC</td>
</tr>
<tr>
<td>96</td>
<td>BUFFER_ATOMIC_SWAP_X2</td>
</tr>
<tr>
<td>97</td>
<td>BUFFER_ATOMIC_CMPSWAP_X2</td>
</tr>
<tr>
<td>98</td>
<td>BUFFER_ATOMIC_ADD_X2</td>
</tr>
<tr>
<td>99</td>
<td>BUFFER_ATOMIC_SUB_X2</td>
</tr>
<tr>
<td>100</td>
<td>BUFFER_ATOMIC_SMIN_X2</td>
</tr>
<tr>
<td>101</td>
<td>BUFFER_ATOMIC_UMIN_X2</td>
</tr>
<tr>
<td>102</td>
<td>BUFFER_ATOMIC_SMAX_X2</td>
</tr>
<tr>
<td>103</td>
<td>BUFFER_ATOMIC_UMAX_X2</td>
</tr>
<tr>
<td>104</td>
<td>BUFFER_ATOMIC_AND_X2</td>
</tr>
<tr>
<td>105</td>
<td>BUFFER_ATOMIC_OR_X2</td>
</tr>
<tr>
<td>106</td>
<td>BUFFER_ATOMIC_XOR_X2</td>
</tr>
<tr>
<td>107</td>
<td>BUFFER_ATOMIC_INC_X2</td>
</tr>
<tr>
<td>108</td>
<td>BUFFER_ATOMIC_DEC_X2</td>
</tr>
</tbody>
</table>

13.7. Vector Memory Image Format

13.7.1. MIMG

<table>
<thead>
<tr>
<th>Format</th>
<th>MIMG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Memory Image Instructions</td>
</tr>
</tbody>
</table>
**Table 89. MIMG Fields**

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bits</th>
<th>Format or Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMASK</td>
<td>[11:8]</td>
<td>Data VGPR enable mask: 1 .. 4 consecutive VGPRs. Reads: defines which components are returned: 0=red, 1=green, 2=blue, 3=alpha. Writes: defines which components are written with data from VGPRs (missing components get 0). Enabled components come from consecutive VGPRs. E.G. dmask=1001 : Red is in VGPRn and alpha in VGPRn+1. For D16 writes, DMASK is only used as a word count: each bit represents 16 bits of data to be written starting at the LSB's of VDATA, then MSBs, then VDATA+1 etc. Bit position is ignored.</td>
</tr>
<tr>
<td>UNRM</td>
<td>[12]</td>
<td>Force address to be un-normalized. Must be set to 1 for Image stores &amp; atomics.</td>
</tr>
<tr>
<td>GLC</td>
<td>[13]</td>
<td>0 = normal, 1 = globally coherent (bypass L0 cache) or for atomics, return pre-op value to VGPR.</td>
</tr>
<tr>
<td>DA</td>
<td>[14]</td>
<td>Declare an Array. 1 Kernel has declared this resource to be an array of texture maps. 0 Kernel has declared this resource to be a single texture map.</td>
</tr>
<tr>
<td>A16</td>
<td>[15]</td>
<td>Address components are 16-bits (instead of the usual 32 bits). When set, all address components are 16 bits (packed into 2 per dword), except: Texel offsets (3 6bit UINT packed into 1 dword) PCF reference (for &quot;_C&quot; instructions) Address components are 16b uint for image ops without sampler; 16b float with sampler.</td>
</tr>
<tr>
<td>TFE</td>
<td>[16]</td>
<td>Partially resident texture, texture fail enable.</td>
</tr>
<tr>
<td>LWE</td>
<td>[17]</td>
<td>LOD Warning Enable. When set to 1, a texture fetch may return &quot;LOD.Clamp = 1&quot;.</td>
</tr>
<tr>
<td>OP</td>
<td>[0],[24:18]</td>
<td>Opcode. See table below. (combine bits zero and 18-24 to form opcode).</td>
</tr>
<tr>
<td>ENCODING</td>
<td>[31:26]</td>
<td>Must be: 111100</td>
</tr>
<tr>
<td>VADDR</td>
<td>[39:32]</td>
<td>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.</td>
</tr>
<tr>
<td>VDATA</td>
<td>[47:40]</td>
<td>Address of VGPR to supply first component of write data or receive first component of read-data.</td>
</tr>
<tr>
<td>SRSRC</td>
<td>[52:48]</td>
<td>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.</td>
</tr>
<tr>
<td>SSAMP</td>
<td>[57:53]</td>
<td>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.</td>
</tr>
<tr>
<td>D16</td>
<td>[63]</td>
<td>Address offset, unsigned byte.</td>
</tr>
</tbody>
</table>

**Table 90. MIMG Opcodes**

"Vega" 7nm Instruction Set Architecture

13.7. Vector Memory Image Format
<table>
<thead>
<tr>
<th>Opcode #</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>IMAGE_LOAD</td>
</tr>
<tr>
<td>1</td>
<td>IMAGE_LOAD_MIP</td>
</tr>
<tr>
<td>2</td>
<td>IMAGE_LOAD_PCK</td>
</tr>
<tr>
<td>3</td>
<td>IMAGE_LOAD_PCK_SGN</td>
</tr>
<tr>
<td>4</td>
<td>IMAGE_LOAD_MIP_PCK</td>
</tr>
<tr>
<td>5</td>
<td>IMAGE_LOAD_MIP_PCK_SGN</td>
</tr>
<tr>
<td>8</td>
<td>IMAGE_STORE</td>
</tr>
<tr>
<td>9</td>
<td>IMAGE_STORE_MIP</td>
</tr>
<tr>
<td>10</td>
<td>IMAGE_STORE_PCK</td>
</tr>
<tr>
<td>11</td>
<td>IMAGE_STORE_MIP_PCK</td>
</tr>
<tr>
<td>14</td>
<td>IMAGE_GET_RESINFO</td>
</tr>
<tr>
<td>16</td>
<td>IMAGE_ATOMIC_SWAP</td>
</tr>
<tr>
<td>17</td>
<td>IMAGE_ATOMIC_CMPSwap</td>
</tr>
<tr>
<td>18</td>
<td>IMAGE_ATOMIC_ADD</td>
</tr>
<tr>
<td>19</td>
<td>IMAGE_ATOMIC_SUB</td>
</tr>
<tr>
<td>20</td>
<td>IMAGE_ATOMIC_SMIN</td>
</tr>
<tr>
<td>21</td>
<td>IMAGE_ATOMIC_UMIN</td>
</tr>
<tr>
<td>22</td>
<td>IMAGE_ATOMIC_SMAX</td>
</tr>
<tr>
<td>23</td>
<td>IMAGE_ATOMIC_UMAX</td>
</tr>
<tr>
<td>24</td>
<td>IMAGE_ATOMIC_AND</td>
</tr>
<tr>
<td>25</td>
<td>IMAGE_ATOMIC_OR</td>
</tr>
<tr>
<td>26</td>
<td>IMAGE_ATOMIC_XOR</td>
</tr>
<tr>
<td>27</td>
<td>IMAGE_ATOMIC_INC</td>
</tr>
<tr>
<td>28</td>
<td>IMAGE_ATOMIC_DEC</td>
</tr>
<tr>
<td>32</td>
<td>IMAGE_SAMPLE</td>
</tr>
<tr>
<td>33</td>
<td>IMAGE_SAMPLE_CL</td>
</tr>
<tr>
<td>34</td>
<td>IMAGE_SAMPLE_D</td>
</tr>
<tr>
<td>35</td>
<td>IMAGE_SAMPLE_D_CL</td>
</tr>
<tr>
<td>36</td>
<td>IMAGE_SAMPLE_L</td>
</tr>
<tr>
<td>37</td>
<td>IMAGE_SAMPLE_B</td>
</tr>
<tr>
<td>38</td>
<td>IMAGE_SAMPLE_B_CL</td>
</tr>
<tr>
<td>39</td>
<td>IMAGE_SAMPLE_LZ</td>
</tr>
<tr>
<td>40</td>
<td>IMAGE_SAMPLE_C</td>
</tr>
<tr>
<td>Opcode #</td>
<td>Name</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>41</td>
<td>IMAGE_SAMPLE_C_CL</td>
</tr>
<tr>
<td>42</td>
<td>IMAGE_SAMPLE_C_D</td>
</tr>
<tr>
<td>43</td>
<td>IMAGE_SAMPLE_C_D_CL</td>
</tr>
<tr>
<td>44</td>
<td>IMAGE_SAMPLE_C_L</td>
</tr>
<tr>
<td>45</td>
<td>IMAGE_SAMPLE_C_B</td>
</tr>
<tr>
<td>46</td>
<td>IMAGE_SAMPLE_C_B_CL</td>
</tr>
<tr>
<td>47</td>
<td>IMAGE_SAMPLE_C_LZ</td>
</tr>
<tr>
<td>48</td>
<td>IMAGE_SAMPLE_O</td>
</tr>
<tr>
<td>49</td>
<td>IMAGE_SAMPLE_CL_O</td>
</tr>
<tr>
<td>50</td>
<td>IMAGE_SAMPLE_D_O</td>
</tr>
<tr>
<td>51</td>
<td>IMAGE_SAMPLE_D_CL_O</td>
</tr>
<tr>
<td>52</td>
<td>IMAGE_SAMPLE_L_O</td>
</tr>
<tr>
<td>53</td>
<td>IMAGE_SAMPLE_B_O</td>
</tr>
<tr>
<td>54</td>
<td>IMAGE_SAMPLE_B_CL_O</td>
</tr>
<tr>
<td>55</td>
<td>IMAGE_SAMPLE_LZ_O</td>
</tr>
<tr>
<td>56</td>
<td>IMAGE_SAMPLE_C_O</td>
</tr>
<tr>
<td>57</td>
<td>IMAGE_SAMPLE_C_CL_O</td>
</tr>
<tr>
<td>58</td>
<td>IMAGE_SAMPLE_C_D_O</td>
</tr>
<tr>
<td>59</td>
<td>IMAGE_SAMPLE_C_D_CL_O</td>
</tr>
<tr>
<td>60</td>
<td>IMAGE_SAMPLE_C_L_O</td>
</tr>
<tr>
<td>61</td>
<td>IMAGE_SAMPLE_C_B_O</td>
</tr>
<tr>
<td>62</td>
<td>IMAGE_SAMPLE_C_B_CL_O</td>
</tr>
<tr>
<td>63</td>
<td>IMAGE_SAMPLE_C_LZ_O</td>
</tr>
<tr>
<td>64</td>
<td>IMAGE_GATHER4</td>
</tr>
<tr>
<td>65</td>
<td>IMAGE_GATHER4_CL</td>
</tr>
<tr>
<td>66</td>
<td>IMAGE_GATHER4H</td>
</tr>
<tr>
<td>68</td>
<td>IMAGE_GATHER4_L</td>
</tr>
<tr>
<td>69</td>
<td>IMAGE_GATHER4_B</td>
</tr>
<tr>
<td>70</td>
<td>IMAGE_GATHER4_B_CL</td>
</tr>
<tr>
<td>71</td>
<td>IMAGE_GATHER4_LZ</td>
</tr>
<tr>
<td>72</td>
<td>IMAGE_GATHER4_C</td>
</tr>
<tr>
<td>73</td>
<td>IMAGE_GATHER4_C_CL</td>
</tr>
<tr>
<td>74</td>
<td>IMAGE_GATHER4H_PCK</td>
</tr>
</tbody>
</table>

"Vega" 7nm Instruction Set Architecture
### Opcode # | Name
---|---
75 | IMAGE_GATHER8H_PCK
76 | IMAGE_GATHER4_C_L
77 | IMAGE_GATHER4_C_B
78 | IMAGE_GATHER4_C_B_CL
79 | IMAGE_GATHER4_C_LZ
80 | IMAGE_GATHER4_O
81 | IMAGE_GATHER4_CL_O
84 | IMAGE_GATHER4_L_O
85 | IMAGE_GATHER4_B_O
86 | IMAGE_GATHER4_B_CL_O
87 | IMAGE_GATHER4_LZ_O
88 | IMAGE_GATHER4_C_O
89 | IMAGE_GATHER4_C_CL_O
92 | IMAGE_GATHER4_C_L_O
93 | IMAGE_GATHER4_C_B_O
94 | IMAGE_GATHER4_C_B_CL_O
95 | IMAGE_GATHER4_C_LZ_O
96 | IMAGE_GET_LOD
104 | IMAGE_SAMPLE_CD
105 | IMAGE_SAMPLE_CD_CL
106 | IMAGE_SAMPLE_C_CD
107 | IMAGE_SAMPLE_C_CD_CL
108 | IMAGE_SAMPLE_CD_O
109 | IMAGE_SAMPLE_CD_CL_O
110 | IMAGE_SAMPLE_C_CD_O
111 | IMAGE_SAMPLE_C_CD_CL_O

### 13.8. Flat Formats

Flat memory instruction 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.

### 13.8.1. FLAT

**Format**  
FLAT

**Description**  
FLAT Memory Access

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bits</th>
<th>Format or Description</th>
</tr>
</thead>
</table>
| OFFSET     | [12:0]     | Address offset  
Scratch, Global: 13-bit signed byte offset  
FLAT: 12-bit unsigned offset (MSB is ignored) |
| 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.                  |
| GLC        | [16]       | 0 = normal, 1 = globally coherent (bypass L0 cache) or for atomics, return pre-op value to VGPR. |
| 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. |
| VDST       | [63:56]    | Destination VGPR for data returned from memory to VGPRs.                                |
## Table 92. FLAT Opcodes

<table>
<thead>
<tr>
<th>Opcode #</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>FLAT_LOAD_UBYTE</td>
</tr>
<tr>
<td>17</td>
<td>FLAT_LOAD_SBYTE</td>
</tr>
<tr>
<td>18</td>
<td>FLAT_LOAD_USHORT</td>
</tr>
<tr>
<td>19</td>
<td>FLAT_LOAD_SSHORT</td>
</tr>
<tr>
<td>20</td>
<td>FLAT_LOAD_DWORD</td>
</tr>
<tr>
<td>21</td>
<td>FLAT_LOAD_DWORDX2</td>
</tr>
<tr>
<td>22</td>
<td>FLAT_LOAD_DWORDX3</td>
</tr>
<tr>
<td>23</td>
<td>FLAT_LOAD_DWORDX4</td>
</tr>
<tr>
<td>24</td>
<td>FLAT_STORE_BYTE</td>
</tr>
<tr>
<td>25</td>
<td>FLAT_STORE_BYTE_D16_HI</td>
</tr>
<tr>
<td>26</td>
<td>FLAT_STORE_SHORT</td>
</tr>
<tr>
<td>27</td>
<td>FLAT_STORE_SHORT_D16_HI</td>
</tr>
<tr>
<td>28</td>
<td>FLAT_STORE_DWORD</td>
</tr>
<tr>
<td>29</td>
<td>FLAT_STORE_DWORDX2</td>
</tr>
<tr>
<td>30</td>
<td>FLAT_STORE_DWORDX3</td>
</tr>
<tr>
<td>31</td>
<td>FLAT_STORE_DWORDX4</td>
</tr>
<tr>
<td>32</td>
<td>FLAT_LOAD_UBYTE_D16</td>
</tr>
<tr>
<td>33</td>
<td>FLAT_LOAD_UBYTE_D16_HI</td>
</tr>
<tr>
<td>34</td>
<td>FLAT_LOAD_SBYTE_D16</td>
</tr>
<tr>
<td>35</td>
<td>FLAT_LOAD_SBYTE_D16_HI</td>
</tr>
<tr>
<td>36</td>
<td>FLAT_LOAD_SHORT_D16</td>
</tr>
<tr>
<td>37</td>
<td>FLAT_LOAD_SHORT_D16_HI</td>
</tr>
<tr>
<td>64</td>
<td>FLAT_ATOMIC_SWAP</td>
</tr>
<tr>
<td>65</td>
<td>FLAT_ATOMIC_CMPSWAP</td>
</tr>
<tr>
<td>66</td>
<td>FLAT_ATOMIC_ADD</td>
</tr>
<tr>
<td>67</td>
<td>FLAT_ATOMIC_SUB</td>
</tr>
<tr>
<td>68</td>
<td>FLAT_ATOMIC_SMIN</td>
</tr>
<tr>
<td>69</td>
<td>FLAT_ATOMIC_UMIN</td>
</tr>
<tr>
<td>70</td>
<td>FLAT_ATOMIC_SMAX</td>
</tr>
<tr>
<td>71</td>
<td>FLAT_ATOMIC_UMAX</td>
</tr>
<tr>
<td>72</td>
<td>FLAT_ATOMIC_AND</td>
</tr>
<tr>
<td>73</td>
<td>FLAT_ATOMIC_OR</td>
</tr>
</tbody>
</table>
### Opcode # | Name
--- | ---
74 | FLAT_ATOMIC_XOR
75 | FLAT_ATOMIC_INC
76 | FLAT_ATOMIC_DEC
96 | FLAT_ATOMIC_SWAP_X2
97 | FLAT_ATOMIC_CMP_SWAP_X2
98 | FLAT_ATOMIC_ADD_X2
99 | FLAT_ATOMIC_SUB_X2
100 | FLAT_ATOMIC_SMIN_X2
101 | FLAT_ATOMIC_UMIN_X2
102 | FLAT_ATOMIC_SMAX_X2
103 | FLAT_ATOMIC_UMAX_X2
104 | FLAT_ATOMIC_AND_X2
105 | FLAT_ATOMIC_OR_X2
106 | FLAT_ATOMIC_XOR_X2
107 | FLAT_ATOMIC_INC_X2
108 | FLAT_ATOMIC_DEC_X2

### 13.8.2. GLOBAL

Table 93. GLOBAL Opcodes

<table>
<thead>
<tr>
<th>Opcode #</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>GLOBAL_LOAD_UBYTE</td>
</tr>
<tr>
<td>17</td>
<td>GLOBAL_LOAD_SBYTE</td>
</tr>
<tr>
<td>18</td>
<td>GLOBAL_LOAD_USHORT</td>
</tr>
<tr>
<td>19</td>
<td>GLOBAL_LOAD_SSHORT</td>
</tr>
<tr>
<td>20</td>
<td>GLOBAL_LOAD_DWORD</td>
</tr>
<tr>
<td>21</td>
<td>GLOBAL_LOAD_DWORDX2</td>
</tr>
<tr>
<td>22</td>
<td>GLOBAL_LOAD_DWORDX3</td>
</tr>
<tr>
<td>23</td>
<td>GLOBAL_LOAD_DWORDX4</td>
</tr>
<tr>
<td>24</td>
<td>GLOBAL_STORE_BYTE</td>
</tr>
<tr>
<td>25</td>
<td>GLOBAL_STORE_BYTE_D16_HI</td>
</tr>
<tr>
<td>26</td>
<td>GLOBAL_STORE_SHORT</td>
</tr>
<tr>
<td>27</td>
<td>GLOBAL_STORE_SHORT_D16_HI</td>
</tr>
<tr>
<td>Opcode #</td>
<td>Name</td>
</tr>
<tr>
<td>---------</td>
<td>-------------------------------------------</td>
</tr>
<tr>
<td>28</td>
<td>GLOBAL_STORE_DWORD</td>
</tr>
<tr>
<td>29</td>
<td>GLOBAL_STORE_DWORDX2</td>
</tr>
<tr>
<td>30</td>
<td>GLOBAL_STORE_DWORDX3</td>
</tr>
<tr>
<td>31</td>
<td>GLOBAL_STORE_DWORDX4</td>
</tr>
<tr>
<td>32</td>
<td>GLOBAL_LOAD_UBYTE_D16</td>
</tr>
<tr>
<td>33</td>
<td>GLOBAL_LOAD_UBYTE_D16_HI</td>
</tr>
<tr>
<td>34</td>
<td>GLOBAL_LOAD_SBYTE_D16</td>
</tr>
<tr>
<td>35</td>
<td>GLOBAL_LOAD_SBYTE_D16_HI</td>
</tr>
<tr>
<td>36</td>
<td>GLOBAL_LOAD_SHORT_D16</td>
</tr>
<tr>
<td>37</td>
<td>GLOBAL_LOAD_SHORT_D16_HI</td>
</tr>
<tr>
<td>64</td>
<td>GLOBAL_ATOMIC_SWAP</td>
</tr>
<tr>
<td>65</td>
<td>GLOBAL_ATOMIC_CMPSWAP</td>
</tr>
<tr>
<td>66</td>
<td>GLOBAL_ATOMIC_ADD</td>
</tr>
<tr>
<td>67</td>
<td>GLOBAL_ATOMIC_SUB</td>
</tr>
<tr>
<td>68</td>
<td>GLOBAL_ATOMIC_SMIN</td>
</tr>
<tr>
<td>69</td>
<td>GLOBAL_ATOMIC_UMIN</td>
</tr>
<tr>
<td>70</td>
<td>GLOBAL_ATOMIC_SMAX</td>
</tr>
<tr>
<td>71</td>
<td>GLOBAL_ATOMIC_UMAX</td>
</tr>
<tr>
<td>72</td>
<td>GLOBAL_ATOMIC_AND</td>
</tr>
<tr>
<td>73</td>
<td>GLOBAL_ATOMIC_OR</td>
</tr>
<tr>
<td>74</td>
<td>GLOBAL_ATOMIC_XOR</td>
</tr>
<tr>
<td>75</td>
<td>GLOBAL_ATOMIC_INC</td>
</tr>
<tr>
<td>76</td>
<td>GLOBAL_ATOMIC_DEC</td>
</tr>
<tr>
<td>96</td>
<td>GLOBAL_ATOMIC_SWAP_X2</td>
</tr>
<tr>
<td>97</td>
<td>GLOBAL_ATOMIC_CMPSWAP_X2</td>
</tr>
<tr>
<td>98</td>
<td>GLOBAL_ATOMIC_ADD_X2</td>
</tr>
<tr>
<td>99</td>
<td>GLOBAL_ATOMIC_SUB_X2</td>
</tr>
<tr>
<td>100</td>
<td>GLOBAL_ATOMIC_SMIN_X2</td>
</tr>
<tr>
<td>101</td>
<td>GLOBAL_ATOMIC_UMIN_X2</td>
</tr>
<tr>
<td>102</td>
<td>GLOBAL_ATOMIC_SMAX_X2</td>
</tr>
<tr>
<td>103</td>
<td>GLOBAL_ATOMIC_UMAX_X2</td>
</tr>
<tr>
<td>104</td>
<td>GLOBAL_ATOMIC_AND_X2</td>
</tr>
<tr>
<td>105</td>
<td>GLOBAL_ATOMIC_OR_X2</td>
</tr>
</tbody>
</table>
## 13.8.3. SCRATCH

Table 94. SCRATCH Opcodes

<table>
<thead>
<tr>
<th>Opcode #</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>SCRATCH_LOAD_UBYTE</td>
</tr>
<tr>
<td>17</td>
<td>SCRATCH_LOAD_SBYTE</td>
</tr>
<tr>
<td>18</td>
<td>SCRATCH_LOAD_USHORT</td>
</tr>
<tr>
<td>19</td>
<td>SCRATCH_LOAD_SSHORT</td>
</tr>
<tr>
<td>20</td>
<td>SCRATCH_LOAD_DWORD</td>
</tr>
<tr>
<td>21</td>
<td>SCRATCH_LOAD_DWORDX2</td>
</tr>
<tr>
<td>22</td>
<td>SCRATCH_LOAD_DWORDX3</td>
</tr>
<tr>
<td>23</td>
<td>SCRATCH_LOAD_DWORDX4</td>
</tr>
<tr>
<td>24</td>
<td>SCRATCH_STORE_BYTE</td>
</tr>
<tr>
<td>25</td>
<td>SCRATCH_STORE_BYTE_D16_HI</td>
</tr>
<tr>
<td>26</td>
<td>SCRATCH_STORE_SHORT</td>
</tr>
<tr>
<td>27</td>
<td>SCRATCH_STORE_SHORT_D16_HI</td>
</tr>
<tr>
<td>28</td>
<td>SCRATCH_STORE_DWORD</td>
</tr>
<tr>
<td>29</td>
<td>SCRATCH_STORE_DWORDX2</td>
</tr>
<tr>
<td>30</td>
<td>SCRATCH_STORE_DWORDX3</td>
</tr>
<tr>
<td>31</td>
<td>SCRATCH_STORE_DWORDX4</td>
</tr>
<tr>
<td>32</td>
<td>SCRATCH_LOAD_UBYTE_D16</td>
</tr>
<tr>
<td>33</td>
<td>SCRATCH_LOAD_UBYTE_D16_HI</td>
</tr>
<tr>
<td>34</td>
<td>SCRATCH_LOAD_SBYTE_D16</td>
</tr>
<tr>
<td>35</td>
<td>SCRATCH_LOAD_SBYTE_D16_HI</td>
</tr>
<tr>
<td>36</td>
<td>SCRATCH_LOAD_SHORT_D16</td>
</tr>
<tr>
<td>37</td>
<td>SCRATCH_LOAD_SHORT_D16_HI</td>
</tr>
</tbody>
</table>
13.9. Export Format

13.9.1. EXP

**Format**

EXP

**Description**

EXPORT instructions

The export format has only a single opcode, "EXPORT".

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Bits</th>
<th>Format or Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EN</td>
<td>[3:0]</td>
<td>COMPR==1: export half-dword enable. Valid values are: 0x0,3,c,f [0] enables VSRC0 : R,G from one VGPR (R in low bits, G high) [2] enables VSRC1 : B,A from one VGPR (B in low bits, A high) COMPR==0: [0-3] = enables for VSRC0..3. EN may be zero only for &quot;NULL Pixel Shader&quot; exports (used when exporting only valid mask to NULL target).</td>
</tr>
<tr>
<td>COMPR</td>
<td>[10]</td>
<td>Indicates that data is float-16/short/byte (compressed). Data is written to consecutive components (rgba or xyzw).</td>
</tr>
<tr>
<td>DONE</td>
<td>[11]</td>
<td>Indicates that this is the last export from the shader. Used only for Position and Pixel/color data.</td>
</tr>
<tr>
<td>VM</td>
<td>[12]</td>
<td>1 = the exec mask IS the valid mask for this export. Can be sent multiple times, must be sent at least once per pixel shader. This bit is only used for Pixel Shaders.</td>
</tr>
<tr>
<td>ENCODING</td>
<td>[31:26]</td>
<td>Must be: 110001</td>
</tr>
<tr>
<td>VSRC0</td>
<td>[39:32]</td>
<td>VGPR for source 0.</td>
</tr>
<tr>
<td>VSRC1</td>
<td>[47:40]</td>
<td>VGPR for source 1.</td>
</tr>
<tr>
<td>VSRC3</td>
<td>[63:56]</td>
<td>VGPR for source 3.</td>
</tr>
</tbody>
</table>