sp3 Instruction Set

GFXIP 9.0 Family

December 19, 2016

Trade secret of Advanced Micro Devices, Inc.
Unpublished work, Copyright 2010-2015 Advanced Micro Devices, Inc.

All rights reserved. This notice is intended as a precaution against inadvertent publication and does not imply publication or any waiver of confidentiality. The year included in the foregoing notice is the year of creation of the work.

AMD Confidential; maintained by Justin David Smith (justins).

Legal Notices

Copyright 2010-2015 Advanced Micro Devices, Inc. All rights reserved.

AMD, the AMD Arrow logo and AMD technology, product and feature names are trademarks and/or registered trademarks of Advanced Micro Devices, Inc. All other company and/or product names may be trademarks and/or registered trademarks of their respective owners.

Except as otherwise expressly set forth in a signed, written agreement between you and AMD, you may review the specification only 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 to provide feedback about the information disclosed in the specification to AMD. All rights in and to the specification are retained by AMD, and this notice 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 notice or any notices from the specification, or (iii) give any part of the specification, or assign or otherwise provide your rights under this notice, to anyone else unless otherwise expressly set forth in a signed, written agreement between you and AMD.

Except as otherwise expressly agreed in a signed, written agreement between you and AMD, the specification may contain preliminary information, errors, or inaccuracies, or may not include certain necessary information and may be changed at any time. Additionally, AMD reserves the right to discontinue or make changes to the specification and its products at any time without notice and agrees that the product may not be exactly as shown. The specification is provided entirely "AS IS". Furthermore, except as expressly agreed upon in a signed, written agreement between you and AMD, 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.

You acknowledge that 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.

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 1334/2000 of 22 June 2000. 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/. 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.

This notice 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 notice 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 notice 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. In the event of conflict between the provisions of this notice and those of the Master Development Agreement effective as of September 1, 2010 executed between you and AMD, the provisions of such Master Development Agreement shall take precedence to the extent of such conflict.

Contents

1 Introduction

This is the sp3 instruction set for GFXIP 9. This document is automatically generated and should reflect the latest shader architectural definition checked into the tree. Please refer to the sp3 Language Reference Manual for general information about sp3, and the Shader Programming Guide for a more detailed explanation of these instructions.

NOTE: This document should be read together with the Shader Programming Guide, which explains how the instructions function and describes changes between each family of shader cores. Deltas in the instruction set between families are only described in the Shader Programming Guide. The Shader Programming Guide is the final authority on the behaviour of shader instructions.

The main sections cover each instruction encoding type. The appendices of this document describe the operand types, data formats, and special architectural constants available to all sp3 shaders.

1.1 Conventions

An instruction specification looks like this:

opcode_name *operand_name_0*, *operand_name_1*[4]
D0: type_0, ↔ S0: type_1, RSRC

The opcode name is always listed first, in bold. Opcodes for a given encoding are sorted in alphabetical order. The operands are then listed with names that describe their type and/or use. In this example we have two operands. The second operand, *operand_name_1*, requires four dwords (consecutive GPRs) and this is indicated by the suffix [4]. If no size is given, the operand is assumed to require at most one dword of data.

The second line describes each operand in more detail. It indicates whether a given operand is a destination (destinations are named `D0`, `D1`, ...) or a source (sources are named `S0`, `S1`, ...); these shorthand names are also used in instruction descriptions. If an operand is both an input and an output for the instruction then the symbol ↔ will appear below the operand as well; all such operands use `D*` shorthand names.

The type of the operand is given immediately after the shorthand name; in this example they are type_0 and type_1. The type determines which registers, numbers, etc. can be used in this operand position. The full definition of operand types is given in Appendix A; the hyperlinks for each operand type jump directly to the corresponding definitions in this appendix.

If the data in the operand should have a specific format it is also indicated on the second line, usually as an all-caps string. In this example the second operand, *operand_name_1*, should have its data formatted as a RSRC, which is a buffer resource constant. Common formats include:

- B16 — 16-bit untyped value.
- B32 — 32-bit untyped value.
- B64 — 64-bit untyped value, read or written to a consecutive run of GPRs. The first GPR always contains the least significant dword of the data.
- I16, U16 — 16-bit signed or unsigned integer value.
- I32, U32 — 32-bit signed or unsigned integer value.

- I64, U64 — 64-bit signed or unsigned integer value, read or written to a consecutive run of GPRs. The first GPR always contains the least significant dword of the data.
- F16 — 16-bit floating point value, S10E5 format.
- F32 — 32-bit IEEE single-precision float.
- F64 — 64-bit IEEE double-precision float, read or written to a consecutive run of GPRs. The first GPR always contains the least significant dword of the data.
- BUF — 64-bit virtual memory address.
- RSRC — 128-bit buffer resource constant.
- IMG — 256-bit image resource constant.
- SAMP — 128-bit sampler constant.

In instruction descriptions the operands are referred to as `D*` and `S*`. Often a suffix is used to indicate how the bits in the register will be interpreted for the operation. Suffixes used include:

- `S0.i16` — treat value as a signed 16-bit integer
- `S0.i` — treat value as a signed 32-bit integer
- `S0.i64` — treat value as a signed 64-bit integer
- `S0.u16` — treat value as an unsigned 16-bit integer
- `S0.u` — treat value as an unsigned 32-bit integer
- `S0.u64` — treat value as an unsigned 64-bit integer
- `S0.f16` — treat value as half-precision floating point value (S10E5)
- `S0.f` — treat value as a single-precision floating point value (S23E8)
- `S0.d` — treat value as a double-precision floating point value (S52E11)

You may compare the suffixes to a C union data structure:

```
union {
    int i;
    unsigned u;
    float f;
    double d;
    // ...
};
```

Slices are used to indicate that only a subset of register bits will be used by the instruction, for example `D[n]` indicates bit *n* (counting from the LSB) of `D` and `S[n:m]` indicates bits *n* through *m* (counting from the LSB).

If subencodings and/or opcode flags are used in the instruction set definition, they will appear in the description as SEN_NAME_OF_FLAG (subencoding) and OPF_NAME_OF_FLAG (opcode flag). These flags are not needed in sp3 programs directly but are frequently used to help classify opcodes in the C-Simulator, RTL code, tests and tools. Where available in the instruction set definition, they are described in Appendix F. The hyperlinks for these flags jump directly to their definitions in this appendix.

1.2 References

- sp3 Language Reference
  http://svdc-svc.amd.com/~gfxipdv/shdoc_gfx9/pub/doc/sp3/sp3_language_ref.pdf
- sp3 Instruction Set
  http://svdc-svc.amd.com/~gfxipdv/shdoc_gfx9/pub/doc/sp3/sp3_instructions.pdf
- sp3 Examples
  http://svdc-svc.amd.com/~gfxipdv/shdoc_gfx9/pub/doc/sp3/sp3_examples.pdf
- GFX9 Shader Programming Guide
  //gfxip/gfx9/doc/design/blocks/sq/arch/Gfx9_Shader_Programming.docx
- GFX9 SIMD Pair Micro-Architecture Specification
  //gfxip/gfx9/doc/design/blocks/sp/GFX9_SP_Specification.docx

2 Encoding SOP1

Scalar ALU operations with one destination and one source.

s_abs_i32 *sdst*, *ssrc*
D0: sdst, I32 S0: ssrc, 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
```

Flags: OPF_WRSCC

s_and_saveexec_b64 *sdst*[2], *ssrc*[2]
D0: sreg, U64 S0: ssrc, U64

```
D.u64 = EXEC;
EXEC = S0.u64 & EXEC;
SCC = (EXEC ≠ 0).
```

Flags: OPF_RDEX, OPF_WREX, OPF_WRSCC

s_andn1_saveexec_b64 *sdst*[2], *ssrc*[2]
D0: sreg, U64 S0: ssrc, U64

```
D.u64 = EXEC;
EXEC = ~S0.u64 & EXEC;
SCC = (EXEC ≠ 0).
```

Flags: OPF_RDEX, OPF_WREX, OPF_WRSCC

s_andn1_wrexec_b64 *sdst*[2], *ssrc*[2]
D0: sreg, U64 S0: ssrc, U64

```
EXEC = ~S0.u64 & EXEC;
D.u64 = EXEC;
SCC = (EXEC ≠ 0).
```

Flags: OPF_RDEX, OPF_WREX, OPF_WRSCC

s_andn2_saveexec_b64 *sdst*[2], *ssrc*[2]
D0: sreg, U64 S0: ssrc, U64

```
D.u64 = EXEC;
EXEC = S0.u64 & ~EXEC;
SCC = (EXEC ≠ 0).
```

Flags: OPF_RDEX, OPF_WREX, OPF_WRSCC

s_andn2_wrexec_b64 *sdst*[2], *ssrc*[2]
D0: sreg, U64 S0: ssrc, U64

```
EXEC = S0.u64 & ~EXEC;
D.u64 = EXEC;
SCC = (EXEC ≠ 0).
```

Flags: OPF_RDEX, OPF_WREX, OPF_WRSCC

s_bcnt0_i32_b32 *sdst*, *ssrc*
D0: sdst, I32 S0: ssrc, U32

s_bcnt0_i32_b64 *sdst*, *ssrc*[2]
D0: sdst, I32 S0: ssrc, U64

```
D = 0;
for i in 0 . . . opcode_size_in_bits - 1 do
        D += (S0[i] == 0 ? 1 : 0)
endfor;
SCC = (D ≠ 0).
```

Examples:

S_BCNT0_I32_B32(0x00000000) ⟹ 32
S_BCNT0_I32_B32(0xcccccccc) ⟹ 16
S_BCNT0_I32_B32(0xffffffff) ⟹ 0

Flags: OPF_WRSCC

s_bcnt1_i32_b32 *sdst*, *ssrc*
D0: sdst, I32 S0: ssrc, U32

s_bcnt1_i32_b64 *sdst*, *ssrc*[2]
D0: sdst, I32 S0: ssrc, U64

```
D = 0;
for i in 0 . . . opcode_size_in_bits - 1 do
        D += (S0[i] == 1 ? 1 : 0)
endfor;
SCC = (D ≠ 0).
```

Examples:

S_BCNT1_I32_B32(0x00000000) ⟹ 0
S_BCNT1_I32_B32(0xcccccccc) ⟹ 16
S_BCNT1_I32_B32(0xffffffff) ⟹ 32

Flags: OPF_WRSCC

s_bitreplicate_b64_b32 *sdst*[2], *ssrc*
D0: sdst, U64 S0: ssrc, U32

```
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.

s_bitset0_b32 *sdst*, *ssrc*
D0: sdst, ↔, B32 S0: ssrc, U32

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

Flags: OPF_DACCUM

s_bitset0_b64 *sdst*[2], *ssrc*
D0: sdst, ↔, B64 S0: ssrc, U32

```
D.u64[S0.u[5:0]] = 0.
```

Flags: OPF_DACCUM

s_bitset1_b32 *sdst*, *ssrc*
D0: sdst, ↔, B32 S0: ssrc, U32

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

Flags: OPF_DACCUM

s_bitset1_b64 *sdst*[2], *ssrc*
D0: sdst, ↔, B64 S0: ssrc, U32

```
D.u64[S0.u[5:0]] = 1.
```

Flags: OPF_DACCUM

s_brev_b32 *sdst*, *ssrc*
D0: sdst, U32 S0: ssrc, U32

```
D.u[31:0] = S0.u[0:31].
```

Reverse bits.

s_brev_b64 *sdst*[2], *ssrc*[2]
D0: sdst, U64 S0: ssrc, U64

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

Reverse bits.

s_cbranch_join *ssrc*
S0: sreg, B32

```
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.

Flags: SEN_NODST , OPF_WREX, OPF_WRPC

s_cmov_b32 *sdst*, *ssrc*
D0: sdst, ↔, B32 S0: ssrc, B32

```
if(SCC) then
        D.u = S0.u;
endif.
```

Conditional move.

Flags: OPF_DACCUM, OPF_RDSCC

s_cmov_b64 *sdst*[2], *ssrc*[2]
D0: sdst, ↔, B64 S0: ssrc, B64

```
if(SCC) then
        D.u64 = S0.u64;
endif.
```

Conditional move.

Flags: OPF_DACCUM, OPF_RDSCC

s_ff0_i32_b32 *sdst*, *ssrc*
D0: sdst, I32 S0: ssrc, U32
s_ff0_i32_b64 *sdst*, *ssrc*[2]
D0: sdst, I32 S0: ssrc, U64

```
D.i = -1; // Set if no zeros are found
for i in 0 . . . opcode_size_in_bits - 1 do // Search from LSB
        if S0[i] == 0 then
                D.i = i;
                break for;
        endif;
endfor.
```

Returns the bit position of the first zero from the LSB, or -1 if there are no zeros.

Examples:

S_FF0_I32_B32(0xaaaaaaaa) ⟹ 0
S_FF0_I32_B32(0x55555555) ⟹ 1
S_FF0_I32_B32(0x00000000) ⟹ 0
S_FF0_I32_B32(0xffffffff) ⟹ 0xffffffff
S_FF0_I32_B32(0xfffeffff) ⟹ 16

s_ff1_i32_b32 *sdst*, *ssrc*
D0: sdst, I32 S0: ssrc, U32
s_ff1_i32_b64 *sdst*, *ssrc*[2]
D0: sdst, I32 S0: ssrc, U64

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

s_flbit_i32 *sdst*, *ssrc*
D0: sdst, I32 S0: ssrc, 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

s_flbit_i32_b32 *sdst*, *ssrc*
D0: sdst, I32 S0: ssrc, U32
s_flbit_i32_b64 *sdst*, *ssrc*[2]
D0: sdst, I32 S0: ssrc, U64

```
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

s_flbit_i32_i64 *sdst*, *ssrc*[2]
D0: sdst, I32 S0: ssrc, 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(0xffff3333) ⟹ 16
S_FLBIT_I32(0x7fffffff) ⟹ 1
S_FLBIT_I32(0x80000000) ⟹ 1
S_FLBIT_I32(0xffffffff) ⟹ 0xffffffff

s_getpc_b64 *sdst*[2]
D0: sdst, B64

```
D.u64 = PC + 4.
```

Destination receives the byte address of the next instruction. Note that this instruction is always 4 bytes.

Flags: SEN_NOSRC , OPF_RDPC

s_mov_b32 *sdst*, *ssrc*
D0: sdst, B32 S0: ssrc, B32

```
D.u = S0.u.
```

Flags: OPF_MOV

s_mov_b64 *sdst*[2], *ssrc*[2]
D0: sdst, B64 S0: ssrc, B64

```
D.u64 = S0.u64.
```

Flags: OPF_MOV

s_mov_fed_b32 *sdst*, *ssrc*
D0: sdst, B32 S0: ssrc, B32

```
D.u = Corrupted(S0.u).
```

Flags: ASIC_FED_INSTRUCTIONS, OPF_MOV

s_movreld_b32 *sdst*, *ssrc*
D0: sreg, B32 S0: ssrc, 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

Flags: OPF_MOVRELD, OPF_RDM0

s_movreld_b64 *sdst*[2], *ssrc*[2]
D0: sreg, B64 S0: ssrc, 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.

Flags: OPF_MOVRELD, OPF_RDM0

s_movrels_b32 *sdst*, *ssrc*
D0: sdst, B32 S0: sreg, 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

Flags: OPF_MOVRELS, OPF_RDM0

s_movrels_b64 *sdst*[2], *ssrc*[2]
D0: sdst, B64 S0: sreg, 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.

Flags: OPF_MOVRELS, OPF_RDM0

s_nand_saveexec_b64 *sdst*[2], *ssrc*[2]
D0: sreg, U64 S0: ssrc, U64

```
D.u64 = EXEC;
EXEC = ∼(S0.u64 & EXEC);
SCC = (EXEC ≠ 0).
```

Flags: OPF_RDEX, OPF_WREX, OPF_WRSCC

s_nor_saveexec_b64 *sdst*[2], *ssrc*[2]
D0: sreg, U64 S0: ssrc, U64

```
D.u64 = EXEC;
EXEC = ~(S0.u64 | EXEC);
SCC = (EXEC ≠ 0).
```

Flags: OPF_RDEX, OPF_WREX, OPF_WRSCC

s_not_b32 *sdst*, *ssrc*
D0: sdst, U32 S0: ssrc, U32

s_not_b64 *sdst*[2], *ssrc*[2]
D0: sdst, U64 S0: ssrc, U64

```
D = ~S0;
SCC = (D ≠ 0).
```

Bitwise negation.

Flags: OPF_WRSCC

s_or_saveexec_b64 *sdst*[2], *ssrc*[2]
D0: sreg, U64 S0: ssrc, U64

```
D.u64 = EXEC;
EXEC = S0.u64 | EXEC;
SCC = (EXEC ≠ 0).
```

Flags: OPF_RDEX, OPF_WREX, OPF_WRSCC

s_orn1_saveexec_b64 *sdst*[2], *ssrc*[2]
D0: sreg, U64 S0: ssrc, U64

```
D.u64 = EXEC;
EXEC = ~S0.u64 | EXEC;
SCC = (EXEC ≠ 0).
```

Flags: OPF_RDEX, OPF_WREX, OPF_WRSCC

s_orn2_saveexec_b64 *sdst*[2], *ssrc*[2]
D0: sreg, U64 S0: ssrc, U64

```
D.u64 = EXEC;
EXEC = S0.u64 | ~EXEC;
SCC = (EXEC ≠ 0).
```

Flags: OPF_RDEX, OPF_WREX, OPF_WRSCC

s_quadmask_b32 *sdst*, *ssrc*
D0: sdst, U32 S0: ssrc, U32
s_quadmask_b64 *sdst*[2], *ssrc*[2]
D0: sdst, U64 S0: ssrc, U64

```
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.
Flags: OPF_WRSCC

s_rfe_b64 *ssrc*[2]
S0: sreg, B64

```
PRIV = 0;
PC = S0.u64.
```

Return from exception handler and continue. This instruction may only be used within a trap handler.
Flags: SEN_NODST , OPF_BREAK_ISTREAM, OPF_WRPC

s_set_gpr_idx_idx *ssrc*
S0: ssrc, B32

```
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.
Flags: SEN_NODST , OPF_RDM0, OPF_WRM0

s_setpc_b64 *ssrc*[2]
S0: sreg, B64

```
PC = S0.u64.
```

S0.u64 is a byte address of the instruction to jump to.
Flags: SEN_NODST , OPF_BREAK_ISTREAM, OPF_WRPC

s_sext_i32_i16 *sdst*, *ssrc*
D0: sdst, I32 S0: ssrc, I16

```
D.i = signext(S0.i[15:0]).
```

Sign extension.

s_sext_i32_i8 *sdst*, *ssrc*
D0: sdst, I32 S0: ssrc, I16

```
D.i = signext(S0.i[7:0]).
```

Sign extension.

s_swappc_b64 *sdst*[2], *ssrc*[2]
D0: sdst, B64 S0: sreg, 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.

Flags: OPF_BREAK_ISTREAM, OPF_RDPC, OPF_WRPC

s_wqm_b32 *sdst*, *ssrc*
D0: sdst, U32 S0: ssrc, U32

s_wqm_b64 *sdst*[2], *ssrc*[2]
D0: sdst, U64 S0: ssrc, U64

```
for i in 0 . . .  opcode_size_in_bits - 1 do
       D[i] = (S0[(i & ~3):(i | 3)] ≠ 0);
endfor;
SCC = (D ≠ 0).
```

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.

Flags: OPF_WRSCC

s_xnor_saveexec_b64 *sdst*[2], *ssrc*[2]
D0: sreg, U64 S0: ssrc, U64

```
D.u64 = EXEC;
EXEC = ~(S0.u64 ^ EXEC);
SCC = (EXEC ≠ 0).
```

Flags: OPF_RDEX, OPF_WREX, OPF_WRSCC

s_xor_saveexec_b64 *sdst*[2], *ssrc*[2]
D0: sreg, U64 S0: ssrc, U64

```
D.u64 = EXEC;
EXEC = S0.u64 ^ EXEC;
SCC = (EXEC ≠ 0).
```

Flags: OPF_RDEX, OPF_WREX, OPF_WRSCC

3 Encoding SOPC

Scalar ALU comparison operations with two sources.

s_bitcmp0_b32 *ssrc_0*, *ssrc_1*
S0: ssrc, U32 S1: ssrc, U32

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

Flags: OPF_WRSCC

s_bitcmp0_b64 *ssrc_0*[2], *ssrc_1*
S0: ssrc, U64 S1: ssrc, U32

```
SCC = (S0.u64[S1.u[5:0]] == 0).
```

Flags: OPF_WRSCC

s_bitcmp1_b32 *ssrc_0*, *ssrc_1*
S0: ssrc, U32 S1: ssrc, U32

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

Flags: OPF_WRSCC

s_bitcmp1_b64 *ssrc_0*[2], *ssrc_1*
S0: ssrc, U64 S1: ssrc, U32

```
SCC = (S0.u64[S1.u[5:0]] == 1).
```

Flags: OPF_WRSCC

s_cmp_eq_i32 *ssrc_0*, *ssrc_1*
S0: ssrc, I32 S1: ssrc, I32
s_cmp_eq_u32 *ssrc_0*, *ssrc_1*
S0: ssrc, U32 S1: ssrc, U32

```
SCC = (S0 == S1).
```

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

Flags: OPF_WRSCC

s_cmp_eq_u64 *ssrc_0*[2], *ssrc_1*[2]
S0: ssrc, U64 S1: ssrc, U64

```
SCC = (S0.i64 == S1.i64).
```

Flags: OPF_WRSCC

s_cmp_ge_i32 *ssrc_0*, *ssrc_1*
S0: ssrc, I32 S1: ssrc, I32

```
SCC = (S0.i ≥ S1.i).
```

Flags: OPF_WRSCC

s_cmp_ge_u32 *ssrc_0*, *ssrc_1*
S0: ssrc, U32 S1: ssrc, U32

```
SCC = (S0.u ≥ S1.u).
```

Flags: OPF_WRSCC

s_cmp_gt_i32 *ssrc_0*, *ssrc_1*
S0: ssrc, I32 S1: ssrc, I32

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

Flags: OPF_WRSCC

s_cmp_gt_u32 *ssrc_0*, *ssrc_1*
S0: ssrc, U32 S1: ssrc, U32

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

Flags: OPF_WRSCC

s_cmp_le_i32 *ssrc_0*, *ssrc_1*
S0: ssrc, I32 S1: ssrc, I32

```
SCC = (S0.i ≤ S1.i).
```

Flags: OPF_WRSCC

s_cmp_le_u32 *ssrc_0*, *ssrc_1*
S0: ssrc, U32 S1: ssrc, U32

```
SCC = (S0.u ≤ S1.u).
```

Flags: OPF_WRSCC

s_cmp_lg_i32 *ssrc_0*, *ssrc_1*
S0: ssrc, I32 S1: ssrc, I32
s_cmp_lg_u32 *ssrc_0*, *ssrc_1*
S0: ssrc, U32 S1: ssrc, U32

```
SCC = (S0 ≠ S1).
```

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

Flags: OPF_WRSCC

s_cmp_lg_u64 *ssrc_0*[2], *ssrc_1*[2]
S0: ssrc, U64 S1: ssrc, U64

```
SCC = (S0.i64 ≠ S1.i64).
```

Flags: OPF_WRSCC

s_cmp_lt_i32 *ssrc_0*, *ssrc_1*
S0: ssrc, I32 S1: ssrc, I32

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

Flags: OPF_WRSCC

s_cmp_lt_u32 *ssrc_0*, *ssrc_1*
S0: ssrc, U32 S1: ssrc, U32

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

Flags: OPF_WRSCC

s_set_gpr_idx_on *ssrc_0*, *simm4*
S0: ssrc, B32 S1: simm4, B32

```
MODE.gpr_idx_en = 1;
M0[7:0] = S0.u[7:0];
M0[15:12] = SIMM4; // this is the direct content of S1 field
// Remaining bits of M0 are unmodified.
```

Enable GPR indexing mode. Vector operations after this will perform relative GPR addressing based on the contents of M0. The structure SQ_M0_GPR_IDX_WORD may be used to decode M0. The raw contents of the S1 field are read and used to set the enable bits. S1[0] = VSRC0_REL, S1[1] = VSRC1_REL, S1[2] = VSRC2_REL and S1[3] = VDST_REL.

S_SET_GPR_IDX_ON, S_SET_GPR_IDX_OFF, S_SET_GPR_IDX_MODE and S_SET_GPR_IDX_IDX are related instructions.

Flags: OPF_RDM0, OPF_WRM0

s_setvskip *ssrc_0*, *ssrc_1*
S0: ssrc, U32 S1: ssrc, U32

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

Enables and disables VSKIP mode. When VSKIP is enabled, no VOP*/M*BUF/MIMG/DS/FLAT/EXP instuctions are issued. Note that VSKIPped memory instructions do not manipulate the waitcnt counters; as a result, if you have outstanding memory requests you may want to issue S_WAITCNT 0 prior to enabling VSKIP, otherwise you'll need to be careful not to count VSKIPped instructions in your waitcnt calculations.

Examples:

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

4 Encoding SOPP

Scalar ALU operations for control flow.

s_barrier

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.

Flags: SEN_NOSRC

---

s_branch *label*

S0: label, B16

```
PC = PC + signext(SIMM16 * 4) + 4. // short jump.
```

For a long jump, use S_SETPC_B64.

Flags: OPF_BREAK_ISTREAM

---

s_cbranch_cdbgsys *label*

S0: label, B16

```
if(conditional_debug_system ≠ 0) then
       PC = PC + signext(SIMM16 * 4) + 4;
endif.
```

---

s_cbranch_cdbgsys_and_user *label*

S0: label, B16

```
if(conditional_debug_system && conditional_debug_user) then
       PC = PC + signext(SIMM16 * 4) + 4;
endif.
```

---

s_cbranch_cdbgsys_or_user *label*

S0: label, B16

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

---

s_cbranch_cdbguser *label*

S0: label, B16

```
if(conditional_debug_user ≠ 0) then
       PC = PC + signext(SIMM16 * 4) + 4;
endif.
```

---

s_cbranch_execnz *label*

S0: label, B16

```
if(EXEC ≠ 0) then
       PC = PC + signext(SIMM16 * 4) + 4;
endif.
```

Flags: OPF_RDEX

---

s_cbranch_execz *label*

S0: label, B16

```
if(EXEC == 0) then
        PC = PC + signext(SIMM16 * 4) + 4;
endif.
```

Flags: OPF_RDEX

s_cbranch_scc0 *label*

S0: label, B16

```
if(SCC == 0) then
        PC = PC + signext(SIMM16 * 4) + 4;
endif.
```

Flags: OPF_RDSCC

s_cbranch_scc1 *label*

S0: label, B16

```
if(SCC == 1) then
        PC = PC + signext(SIMM16 * 4) + 4;
endif.
```

Flags: OPF_RDSCC

s_cbranch_vccnz *label*

S0: label, B16

```
if(VCC ≠ 0) then
        PC = PC + signext(SIMM16 * 4) + 4;
endif.
```

Flags: OPF_RDVCC

s_cbranch_vccz *label*

S0: label, B16

```
if(VCC == 0) then
        PC = PC + signext(SIMM16 * 4) + 4;
endif.
```

Flags: OPF_RDVCC

s_decperflevel *simm16*

S0: simm16, B16

Decrement performance counter specified in SIMM16[3:0] by 1.

s_endpgm

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.

Flags: SEN_NOSRC , OPF_BREAK_ISTREAM

s_endpgm_ordered_ps_done

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.

Flags: SEN_NOSRC , OPF_BREAK_ISTREAM

s_endpgm_saved

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.

Flags: SEN_NOSRC , OPF_BREAK_ISTREAM

s_icache_inv

Invalidate entire L1 instruction cache.

You must have 16 separate S_NOP instructions or a jump/branch instruction after this instruction to ensure the SQ instruction buffer is purged.

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.

Flags: SEN_NOSRC

s_incperflevel *simm16*

S0: simm16, B16

Increment performance counter specified in SIMM16[3:0] by 1.

s_nop *simm16*

S0: simm16, B16

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; see the shader programming guide for details. Compare with S_SLEEP.

s_sendmsg *simm16*

S0: simm16, B16

Send a message upstream to VGT or the interrupt handler. SIMM16[9:0] contains the message type and is documented in the shader programming guide. In sp3 you may use the builtin function sendmsg(msg, gsop, streamid) to compose values for this instruction.

Example:

```
// Assume wave ID is stored in s0
s_mov_b32 m0, s0
s_sendmsg sendmsg(VGT_EMITCUT)
```

Flags: OPF_RDM0

s_sendmsghalt *simm16*

S0: simm16, B16

Send a message and then HALT the wavefront; see S_SENDMSG for details.

Flags: OPF_RDM0

s_set_gpr_idx_mode *simm16*

S0: simm16, B16

```
M0[15:12] = SIMM16[3:0].
```

Modify the mode used for vector GPR indexing. The raw contents of the source field are read and used to set the enable bits. SIMM16[0] = VSRC0_REL, SIMM16[1] = VSRC1_REL, SIMM16[2] = VSRC2_REL and SIMM16[3] = VDST_REL.

S_SET_GPR_IDX_ON, S_SET_GPR_IDX_OFF, S_SET_GPR_IDX_MODE and S_SET_GPR_IDX_IDX are related instructions.

Flags: OPF_RDM0, OPF_WRM0

s_set_gpr_idx_off

```
MODE.gpr_idx_en = 0.
```

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.

S_SET_GPR_IDX_ON, S_SET_GPR_IDX_OFF, S_SET_GPR_IDX_MODE and S_SET_GPR_IDX_IDX are related instructions.

Flags: SEN_NOSRC

s_sethalt *simm16*

S0: simm16, B16

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.

s_setkill *simm16*

S0: simm16, B16

Set KILL bit to value of SIMM16[0]. Used primarily for debugging kill wave host command behavior.

s_setprio *simm16*

S0: simm16, B16

User settable wave priority is set to SIMM16[1:0]. 0 = lowest, 3 = highest. The overall wave priority is {SPIPrio[1:0] + UserPrio[1:0], WaveAge[3:0]}.

s_sleep *simm16*

S0: simm16, B16

Cause a wave to sleep for (64 * SIMM16[6:0] + 1..64) clocks. The exact amount of delay is approximate. Compare with S_NOP.

s_trap *simm16*

S0: simm16, B16

```
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
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.

s_ttracedata

Send M0 as user data to the thread trace stream.

Flags: SEN_NOSRC , OPF_RDM0

s_waitcnt *simm16*

S0: simm16, B16

Wait for the counts of outstanding lds, vector-memory and export/vmem-write-data to be at or below the specified levels.

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

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

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

SIMM16[15:14] = vmcount (vector memory operations) upper bits [5:4],

In sp3 you may use builtin functions vmcnt(n), expcnt(n) and lgkmcnt(n) to compose values for this instruction. They may be used in isolation or bitwise-AND'd together to compose the SIMM16 operand. If some functions are omitted, their fields will default to the maximum value.

Examples:

```
s_waitcnt 0 // Wait for all counts to reach zero.
s_waitcnt vmcnt(0) // Wait for VM counter to reach zero.
s_waitcnt vmcnt(0) & expcnt(0) // Wait for VM and export counters to reach zero.
```

s_wakeup

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 more 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 (whoever missed it just completes 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.

Flags: SEN_NOSRC

5 Encoding SOPK

Scalar ALU operations with one destination and one 16-bit constant source.

s_addk_i32 *sdst*, *simm16*
D0: sdst, ↔, B32 S0: simm16, B16

```
tmp = D.i; // save value so we can check sign bits for overflow later.
D.i = D.i + signext(SIMM16);
SCC = (tmp[31] == SIMM16[15] && tmp[31] ≠ D.i[31]). // signed overflow.
```

Flags: OPF_DACCUM, OPF_WRSCC

s_call_b64 *sdst*[2], *label*
D0: sdst, B64 S0: label

```
D.u64 = PC + 4;
PC = PC + signext(SIMM16 * 4) + 4.
```

Implements a short call, where the return address (the next instruction after the S_CALL_B64) is saved to D. Long calls should consider S_SWAPPC_B64 instead. Note that this instruction is always 4 bytes.

Flags: OPF_BREAK_ISTREAM, OPF_RDPC, OPF_WRPC

s_cbranch_i_fork *sdst*[2], *label_1*
S0: sdst, B64 S1: label

```
mask_pass = S0.u64 & EXEC;
mask_fail = ~S0.u64 & EXEC;
target_addr = PC + signext(SIMM16 * 4) + 4;
if(mask_pass == EXEC)
        PC = target_addr;
elsif(mask_fail == EXEC)
        PC += 4;
elsif(bitcount(mask_fail) < bitcount(mask_pass))
        EXEC = mask_fail;
        SGPR[CSP*4] = { target_addr, mask_pass };
        CSP += 1;
        PC += 4;
else
        EXEC = mask_pass;
        SGPR[CSP*4] = { PC + 4, mask_fail };
        CSP += 1;
        PC = target_addr;
endif.
```

Conditional branch using branch-stack. S0 = compare mask(vcc or any sgpr), and SIMM16 = signed DWORD branch offset relative to next instruction. See also S_CBRANCH_JOIN.

Flags: OPF_LABEL

s_cmovk_i32 *sdst*, *simm16*
D0: sdst, ↔, B32 S0: simm16, B16

```
if(SCC) then
        D.i = signext(SIMM16);
endif.
```

Conditional move with sign extension.

Flags: OPF_DACCUM, OPF_RDSCC

s_cmpk_eq_i32 *ssrc*, *simm16*
S0: sdst, B32 S1: simm16, B16

```
SCC = (S0.i == signext(SIMM16)).
```

Flags: OPF_WRSCC

s_cmpk_eq_u32 *ssrc*, *simm16*
S0: sdst, B32 S1: simm16, B16

```
SCC = (S0.u == SIMM16).
```

Flags: OPF_WRSCC

s_cmpk_ge_i32 *ssrc*, *simm16*
S0: sdst, B32 S1: simm16, B16

```
SCC = (S0.i ≥ signext(SIMM16)).
```

Flags: OPF_WRSCC

s_cmpk_ge_u32 *ssrc*, *simm16*
S0: sdst, B32 S1: simm16, B16

```
SCC = (S0.u ≥ SIMM16).
```

Flags: OPF_WRSCC

s_cmpk_gt_i32 *ssrc*, *simm16*
S0: sdst, B32 S1: simm16, B16

```
SCC = (S0.i > signext(SIMM16)).
```

Flags: OPF_WRSCC

s_cmpk_gt_u32 *ssrc*, *simm16*
S0: sdst, B32 S1: simm16, B16

```
SCC = (S0.u > SIMM16).
```

Flags: OPF_WRSCC

s_cmpk_le_i32 *ssrc*, *simm16*
S0: sdst, B32 S1: simm16, B16

```
SCC = (S0.i ≤ signext(SIMM16)).
```

Flags: OPF_WRSCC

s_cmpk_le_u32 *ssrc*, *simm16*
S0: sdst, B32 S1: simm16, B16

```
SCC = (S0.u ≤ SIMM16).
```

Flags: OPF_WRSCC

s_cmpk_lg_i32 *ssrc*, *simm16*
S0: sdst, B32 S1: simm16, B16

```
SCC = (S0.i ≠ signext(SIMM16)).
```

Flags: OPF_WRSCC

s_cmpk_lg_u32 *ssrc*, *simm16*
S0: sdst, B32 S1: simm16, B16

```
SCC = (S0.u ≠ SIMM16).
```

Flags: OPF_WRSCC

s_cmpk_lt_i32 *ssrc*, *simm16*
S0: sdst, B32 S1: simm16, B16

```
SCC = (S0.i < signext(SIMM16)).
```

Flags: OPF_WRSCC

s_cmpk_lt_u32 *ssrc*, *simm16*
S0: sdst, B32 S1: simm16, B16

```
SCC = (S0.u < SIMM16).
```

Flags: OPF_WRSCC

s_getreg_b32 *sdst*, *simm16*
D0: sdst, B32 S0: simm16, B16

D.u = hardware-reg. Read some or all of a hardware register into the LSBs of D.

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

In sp3 you may use the builtin function hwreg(reg, offset, size) to generate the simm16 operand.

s_movk_i32 *sdst*, *simm16*
D0: sdst, B32 S0: simm16, B16

```
D.i = signext(SIMM16).
```

Sign extension from a 16-bit constant.

s_mulk_i32 *sdst*, *simm16*
D0: sdst, ↔, B32 S0: simm16, B16

```
D.i = D.i * signext(SIMM16).
```

Flags: OPF_DACCUM

s_setreg_b32 *simm16*, *ssrc*

D0: simm16, B16 S0: sdst, B32

hardware-reg = S0.u. Write some or all of the LSBs of D into a hardware register.

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

In sp3 you may use the builtin function hwreg(reg, offset, size) to generate the simm16 operand.

s_setreg_imm32_b32 *simm16*, *simm32*

D0: simm16, B16 S0: simm32, B32

Write some or all of the LSBs of IMM32 into a hardware register; this instruction requires a 32-bit literal constant.

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

6 Encoding SOP2

Scalar ALU operations with one destination and two sources.

s_absdiff_i32 *sdst*, *ssrc_0*, *ssrc_1*
D0: sdst, I32 S0: ssrc, I32 S1: ssrc, 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
```

Flags: OPF_WRSCC

---

s_add_i32 *sdst*, *ssrc_0*, *ssrc_1*
D0: sdst, I32 S0: ssrc, I32 S1: ssrc, I32

```
D.i = S0.i + S1.i;
SCC = (S0.u[31] == S1.u[31] && S0.u[31] ≠ D.u[31]). // signed overflow.
```

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

Flags: OPF_WRSCC

---

s_add_u32 *sdst*, *ssrc_0*, *ssrc_1*
D0: sdst, U32 S0: ssrc, U32 S1: ssrc, U32

```
D.u = S0.u + S1.u;
SCC = (S0.u + S1.u ≥ 0x100000000ULL ? 1 : 0). // unsigned overflow/carry-out, S_ADDC_U32
```

Flags: OPF_WRSCC

---

s_addc_u32 *sdst*, *ssrc_0*, *ssrc_1*
D0: sdst, U32 S0: ssrc, U32 S1: ssrc, U32

```
D.u = S0.u + S1.u + SCC;
SCC = (S0.u + S1.u + SCC ≥ 0x100000000ULL ? 1 : 0). // unsigned overflow.
```

Flags: OPF_RDSCC, OPF_WRSCC

---

s_and_b32 *sdst*, *ssrc_0*, *ssrc_1*
D0: sdst, U32 S0: ssrc, U32 S1: ssrc, U32
s_and_b64 *sdst*[2], *ssrc_0*[2], *ssrc_1*[2]
D0: sdst, U64 S0: ssrc, U64 S1: ssrc, U64

```
D = S0 & S1;
SCC = (D ≠ 0).
```

Flags: OPF_WRSCC

s_andn2_b32 *sdst*, *ssrc_0*, *ssrc_1*
D0: sdst, U32 S0: ssrc, U32 S1: ssrc, U32
s_andn2_b64 *sdst*[2], *ssrc_0*[2], *ssrc_1*[2]
D0: sdst, U64 S0: ssrc, U64 S1: ssrc, U64

```
D = S0 & ~S1;
SCC = (D ≠ 0).
```

Flags: OPF_WRSCC

s_ashr_i32 *sdst*, *ssrc_0*, *ssrc_1*
D0: sdst, I32 S0: ssrc, I32 S1: ssrc, U32

```
D.i = signext(S0.i) >> S1.u[4:0];
SCC = (D.i ≠ 0).
```

Flags: OPF_WRSCC

s_ashr_i64 *sdst*[2], *ssrc_0*[2], *ssrc_1*
D0: sdst, I64 S0: ssrc, I64 S1: ssrc, U32

```
D.i64 = signext(S0.i64) >> S1.u[5:0];
SCC = (D.i64 ≠ 0).
```

Flags: OPF_WRSCC

s_bfe_i32 *sdst*, *ssrc_0*, *ssrc_1*
D0: sdst, I32 S0: ssrc, I32 S1: ssrc, U32

```
D.i = signext((S0.i >> S1.u[4:0]) & ((1 << 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.

Flags: OPF_WRSCC

s_bfe_i64 *sdst*[2], *ssrc_0*[2], *ssrc_1*
D0: sdst, I64 S0: ssrc, I64 S1: ssrc, U32

```
D.i64 = signext((S0.i64 >> S1.u[5:0]) & ((1 << 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.

Flags: OPF_WRSCC

s_bfe_u32 *sdst*, *ssrc_0*, *ssrc_1*
D0: sdst, U32 S0: ssrc, U32 S1: ssrc, U32

```
D.u = (S0.u >> S1.u[4:0]) & ((1 << 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.

Flags: OPF_WRSCC

s_bfe_u64 *sdst*[2], *ssrc_0*[2], *ssrc_1*
D0: sdst, U64 S0: ssrc, U64 S1: ssrc, U32

```
D.u64 = (S0.u64 >> S1.u[5:0]) & ((1 << 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.

Flags: OPF_WRSCC

s_bfm_b32 *sdst*, *ssrc_0*, *ssrc_1*
D0: sdst, U32 S0: ssrc, U32 S1: ssrc, U32

```
D.u = ((1 << S0.u[4:0]) - 1) << S1.u[4:0].
```

Bitfield mask.

s_bfm_b64 *sdst*[2], *ssrc_0*, *ssrc_1*
D0: sdst, U64 S0: ssrc, U32 S1: ssrc, U32

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

Bitfield mask.

s_cbranch_g_fork *ssrc_0*[2], *ssrc_1*[2]
S0: ssrc_nolit, B64 S1: ssrc_nolit, B64

```
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.

Flags: OPF_WRPC

s_cselect_b32 *sdst*, *ssrc_0*, *ssrc_1*
D0: sdst, B32 S0: ssrc, B32 S1: ssrc, B32

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

Conditional select.

Flags: OPF_RDSCC

s_cselect_b64 *sdst*[2], *ssrc_0*[2], *ssrc_1*[2]
D0: sdst, B64 S0: ssrc, B64 S1: ssrc, B64

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

Conditional select.

Flags: OPF_RDSCC

s_lshl1_add_u32 *sdst*, *ssrc_0*, *ssrc_1*
D0: sdst, U32 S0: ssrc, U32 S1: ssrc, U32

```
D.u = (S0.u << 1) + S1.u;
SCC = (((S0.u << 1) + S1.u) ≥ 0x100000000ULL ? 1 : 0). // unsigned overflow.
```

Flags: OPF_WRSCC

s_lshl2_add_u32 *sdst*, *ssrc_0*, *ssrc_1*
D0: sdst, U32 S0: ssrc, U32 S1: ssrc, U32

```
D.u = (S0.u << 2) + S1.u;
SCC = (((S0.u << 2) + S1.u) ≥ 0x100000000ULL ? 1 : 0). // unsigned overflow.
```

Flags: OPF_WRSCC

s_lshl3_add_u32 *sdst*, *ssrc_0*, *ssrc_1*
D0: sdst, U32 S0: ssrc, U32 S1: ssrc, U32

```
D.u = (S0.u << 3) + S1.u;
SCC = (((S0.u << 3) + S1.u) ≥ 0x100000000ULL ? 1 : 0). // unsigned overflow.
```

Flags: OPF_WRSCC

s_lshl4_add_u32 *sdst*, *ssrc_0*, *ssrc_1*
D0: sdst, U32 S0: ssrc, U32 S1: ssrc, U32

```
D.u = (S0.u << 4) + S1.u;
SCC = (((S0.u << 4) + S1.u) ≥ 0x100000000ULL ? 1 : 0). // unsigned overflow.
```

Flags: OPF_WRSCC

s_lshl_b32 *sdst*, *ssrc_0*, *ssrc_1*
D0: sdst, U32 S0: ssrc, U32 S1: ssrc, U32

```
D.u = S0.u << S1.u[4:0];
SCC = (D.u ≠ 0).
```

Flags: OPF_WRSCC

s_lshl_b64 *sdst*[2], *ssrc_0*[2], *ssrc_1*
D0: sdst, U64 S0: ssrc, U64 S1: ssrc, U32

```
D.u64 = S0.u64 << S1.u[5:0];
SCC = (D.u64 ≠ 0).
```

Flags: OPF_WRSCC

s_lshr_b32 *sdst*, *ssrc_0*, *ssrc_1*
D0: sdst, U32 S0: ssrc, U32 S1: ssrc, U32

```
D.u = S0.u >> S1.u[4:0];
SCC = (D.u ≠ 0).
```

Flags: OPF_WRSCC

s_lshr_b64 *sdst*[2], *ssrc_0*[2], *ssrc_1*
D0: sdst, U64 S0: ssrc, U64 S1: ssrc, U32

```
D.u64 = S0.u64 >> S1.u[5:0];
SCC = (D.u64 ≠ 0).
```

Flags: OPF_WRSCC

s_max_i32 *sdst*, *ssrc_0*, *ssrc_1*
D0: sdst, I32 S0: ssrc, I32 S1: ssrc, I32

```
D.i = (S0.i > S1.i) ? S0.i : S1.i;
SCC = (S0.i > S1.i).
```

Flags: OPF_WRSCC

s_max_u32 *sdst*, *ssrc_0*, *ssrc_1*
D0: sdst, U32 S0: ssrc, U32 S1: ssrc, U32

```
D.u = (S0.u > S1.u) ? S0.u : S1.u;
SCC = (S0.u > S1.u).
```

Flags: OPF_WRSCC

s_min_i32 *sdst*, *ssrc_0*, *ssrc_1*
D0: sdst, I32 S0: ssrc, I32 S1: ssrc, I32

```
D.i = (S0.i < S1.i) ? S0.i : S1.i;
SCC = (S0.i < S1.i).
```

Flags: OPF_WRSCC

s_min_u32 *sdst*, *ssrc_0*, *ssrc_1*
D0: sdst, U32 S0: ssrc, U32 S1: ssrc, U32

```
D.u = (S0.u < S1.u) ? S0.u : S1.u;
SCC = (S0.u < S1.u).
```

Flags: OPF_WRSCC

s_mul_hi_i32 *sdst*, *ssrc_0*, *ssrc_1*
D0: sdst, I32 S0: ssrc, I32 S1: ssrc, I32

```
D.i = (S0.i * S1.i) >> 32.
```

s_mul_hi_u32 *sdst*, *ssrc_0*, *ssrc_1*
D0: sdst, U32 S0: ssrc, U32 S1: ssrc, U32

```
D.u = (S0.u * S1.u) >> 32.
```

s_mul_i32 *sdst*, *ssrc_0*, *ssrc_1*
D0: sdst, I32 S0: ssrc, I32 S1: ssrc, I32

```
D.i = S0.i * S1.i.
```

s_nand_b32 *sdst*, *ssrc_0*, *ssrc_1*
D0: sdst, U32 S0: ssrc, U32 S1: ssrc, U32
s_nand_b64 *sdst*[2], *ssrc_0*[2], *ssrc_1*[2]
D0: sdst, U64 S0: ssrc, U64 S1: ssrc, U64

```
D = ∼(S0 & S1);
SCC = (D ≠ 0).
```

Flags: OPF_WRSCC

s_nor_b32 *sdst*, *ssrc_0*, *ssrc_1*
D0: sdst, U32 S0: ssrc, U32 S1: ssrc, U32
s_nor_b64 *sdst*[2], *ssrc_0*[2], *ssrc_1*[2]
D0: sdst, U64 S0: ssrc, U64 S1: ssrc, U64

```
D = ∼(S0 | S1);
SCC = (D ≠ 0).
```

Flags: OPF_WRSCC

s_or_b32 *sdst*, *ssrc_0*, *ssrc_1*
D0: sdst, U32 S0: ssrc, U32 S1: ssrc, U32
s_or_b64 *sdst*[2], *ssrc_0*[2], *ssrc_1*[2]
D0: sdst, U64 S0: ssrc, U64 S1: ssrc, U64

```
D = S0 | S1;
SCC = (D ≠ 0).
```

Flags: OPF_WRSCC

s_orn2_b32 *sdst*, *ssrc_0*, *ssrc_1*
D0: sdst, U32 S0: ssrc, U32 S1: ssrc, U32
s_orn2_b64 *sdst*[2], *ssrc_0*[2], *ssrc_1*[2]
D0: sdst, U64 S0: ssrc, U64 S1: ssrc, U64

```
D = S0 | ~S1;
SCC = (D ≠ 0).
```

Flags: OPF_WRSCC

s_pack_hh_b32_b16 *sdst*, *ssrc_0*, *ssrc_1*
D0: sdst, B32 S0: ssrc, B32 S1: ssrc, B32

```
D.u[31:0] = { S1.u[31:16], S0.u[31:16] }.
```

s_pack_lh_b32_b16 *sdst*, *ssrc_0*, *ssrc_1*
D0: sdst, B32 S0: ssrc, B16 S1: ssrc, B32

```
D.u[31:0] = { S1.u[31:16], S0.u[15:0] }.
```

s_pack_ll_b32_b16 *sdst*, *ssrc_0*, *ssrc_1*
D0: sdst, B32 S0: ssrc, B16 S1: ssrc, B16

```
D.u[31:0] = { S1.u[15:0], S0.u[15:0] }.
```

s_rfe_restore_b64 *ssrc_0*[2], *ssrc_1*
S0: ssrc, B64 S1: ssrc, B32

```
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 should always use S_RFE_B64. The second argument is ignored.

Flags: SEN_NODST , OPF_BREAK_ISTREAM, OPF_WRPC

s_sub_i32 *sdst*, *ssrc_0*, *ssrc_1*
D0: sdst, I32 S0: ssrc, I32 S1: ssrc, I32

```
D.i = S0.i - S1.i;
SCC = (S0.u[31] ≠ S1.u[31] && S0.u[31] ≠ D.u[31]). // signed overflow.
```

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

Flags: OPF_WRSCC

s_sub_u32 *sdst*, *ssrc_0*, *ssrc_1*
D0: sdst, U32 S0: ssrc, U32 S1: ssrc, U32

```
D.u = S0.u - S1.u;
SCC = (S1.u > S0.u ? 1 : 0). // unsigned overflow or carry-out for S_SUBB_U32.
```

Flags: OPF_WRSCC

s_subb_u32 *sdst*, *ssrc_0*, *ssrc_1*
D0: sdst, U32 S0: ssrc, U32 S1: ssrc, U32

```
D.u = S0.u - S1.u - SCC;
SCC = (S1.u + SCC > S0.u ? 1 : 0). // unsigned overflow.
```

Flags: OPF_RDSCC, OPF_WRSCC

s_xnor_b32 *sdst*, *ssrc_0*, *ssrc_1*
D0: sdst, U32 S0: ssrc, U32 S1: ssrc, U32
s_xnor_b64 *sdst*[2], *ssrc_0*[2], *ssrc_1*[2]
D0: sdst, U64 S0: ssrc, U64 S1: ssrc, U64

```
D = ~(S0 ^ S1);
SCC = (D ≠ 0).
```

Flags: OPF_WRSCC

s_xor_b32 *sdst*, *ssrc_0*, *ssrc_1*
D0: sdst, U32 S0: ssrc, U32 S1: ssrc, U32
s_xor_b64 *sdst*[2], *ssrc_0*[2], *ssrc_1*[2]
D0: sdst, U64 S0: ssrc, U64 S1: ssrc, U64

```
D = S0 ^ S1;
SCC = (D ≠ 0).
```

Flags: OPF_WRSCC

7 Encoding SMEM

Scalar memory operations.

s_atc_probe *perm_rwx*, *sgpr_base*[2], *offset*
S0: simm8, B8 S1: sreg, BUF S2: smem_offset, B32

Probe or prefetch an address into the SQC data cache.

Flags: SEN_ATCPROBE

s_atc_probe_buffer *perm_rwx*, *sgpr_base*[4], *offset*
S0: simm8, B8 S1: sreg, RSRC S2: smem_offset, B32

Probe or prefetch an address into the SQC data cache.

Flags: SEN_ATCPROBE , OPF_BUFCONST

s_atomic_add *sgpr_data*, *sgpr_base*[2], *offset*
D0: sreg, ↔, B32 S0: sreg, BUF S1: smem_offset, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] += DATA;
RETURN_DATA = tmp.
```

Flags: OPF_MEM_ATOMIC

s_atomic_add_x2 *sgpr_data*[2], *sgpr_base*[2], *offset*
D0: sreg, ↔, B64 S0: sreg, BUF S1: smem_offset, B32

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] += DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_MEM_ATOMIC

s_atomic_and *sgpr_data*, *sgpr_base*[2], *offset*
D0: sreg, ↔, B32 S0: sreg, BUF S1: smem_offset, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] &= DATA;
RETURN_DATA = tmp.
```

Flags: OPF_MEM_ATOMIC

s_atomic_and_x2 *sgpr_data*[2], *sgpr_base*[2], *offset*
D0: sreg, ↔, B64 S0: sreg, BUF S1: smem_offset, B32

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] &= DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_MEM_ATOMIC

s_atomic_cmpswap *sgpr_data*[2], *sgpr_base*[2], *offset*
D0: sreg, ↔, B32 S0: sreg, BUF S1: smem_offset, B32

```
// 32bit
tmp = MEM[ADDR];
src = DATA[0];
cmp = DATA[1];
MEM[ADDR] = (tmp == cmp) ? src : tmp;
RETURN_DATA[0] = tmp.
```

Flags: OPF_ATOMIC_CMPSWAP, OPF_MEM_ATOMIC

s_atomic_cmpswap_x2 *sgpr_data*[4], *sgpr_base*[2], *offset*
D0: sreg, ↔, B64 S0: sreg, BUF S1: smem_offset, B32

```
// 64bit
tmp = MEM[ADDR];
src = DATA[0:1];
cmp = DATA[2:3];
MEM[ADDR] = (tmp == cmp) ? src : tmp;
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_ATOMIC_CMPSWAP, OPF_MEM_ATOMIC

s_atomic_dec *sgpr_data*, *sgpr_base*[2], *offset*
D0: sreg, ↔, B32 S0: sreg, BUF S1: smem_offset, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (tmp == 0 || tmp > DATA) ? DATA : tmp - 1; // unsigned compare
RETURN_DATA = tmp.
```

Flags: OPF_MEM_ATOMIC

s_atomic_dec_x2 *sgpr_data*[2], *sgpr_base*[2], *offset*
D0: sreg, ↔, B64 S0: sreg, BUF S1: smem_offset, B32

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] = (tmp == 0 || tmp > DATA[0:1]) ? DATA[0:1] : tmp - 1; // unsigned compare
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_MEM_ATOMIC

s_atomic_inc *sgpr_data*, *sgpr_base*[2], *offset*
D0: sreg, ↔, B32 S0: sreg, BUF S1: smem_offset, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (tmp ≥ DATA) ? 0 : tmp + 1; // unsigned compare
RETURN_DATA = tmp.
```

Flags: OPF_MEM_ATOMIC

s_atomic_inc_x2 *sgpr_data*[2], *sgpr_base*[2], *offset*
D0: sreg, ↔, B64 S0: sreg, BUF S1: smem_offset, B32

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] = (tmp ≥ DATA[0:1]) ? 0 : tmp + 1; // unsigned compare
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_MEM_ATOMIC

s_atomic_or *sgpr_data*, *sgpr_base*[2], *offset*
D0: sreg, ↔, B32 S0: sreg, BUF S1: smem_offset, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] |= DATA;
RETURN_DATA = tmp.
```

Flags: OPF_MEM_ATOMIC

s_atomic_or_x2 *sgpr_data*[2], *sgpr_base*[2], *offset*
D0: sreg, ↔, B64 S0: sreg, BUF S1: smem_offset, B32

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] |= DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_MEM_ATOMIC

s_atomic_smax *sgpr_data*, *sgpr_base*[2], *offset*
D0: sreg, ↔, B32 S0: sreg, BUF S1: smem_offset, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (DATA > tmp) ? DATA : tmp; // signed compare
RETURN_DATA = tmp.
```

Flags: OPF_MEM_ATOMIC

s_atomic_smax_x2 *sgpr_data*[2], *sgpr_base*[2], *offset*
D0: sreg, ↔, B64 S0: sreg, BUF S1: smem_offset, B32

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] -= (DATA[0:1] > tmp) ? DATA[0:1] : tmp; // signed compare
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_MEM_ATOMIC

s_atomic_smin *sgpr_data*, *sgpr_base*[2], *offset*
D0: sreg, ↔, B32 S0: sreg, BUF S1: smem_offset, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (DATA < tmp) ? DATA : tmp; // signed compare
RETURN_DATA = tmp.
```

Flags: OPF_MEM_ATOMIC

s_atomic_smin_x2 *sgpr_data*[2], *sgpr_base*[2], *offset*
D0: sreg, ↔, B64 S0: sreg, BUF S1: smem_offset, B32

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] -= (DATA[0:1] < tmp) ? DATA[0:1] : tmp; // signed compare
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_MEM_ATOMIC

s_atomic_sub *sgpr_data*, *sgpr_base*[2], *offset*
D0: sreg, ↔, B32 S0: sreg, BUF S1: smem_offset, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] -= DATA;
RETURN_DATA = tmp.
```

Flags: OPF_MEM_ATOMIC

s_atomic_sub_x2 *sgpr_data*[2], *sgpr_base*[2], *offset*
D0: sreg, ↔, B64 S0: sreg, BUF S1: smem_offset, B32

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] -= DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_MEM_ATOMIC

s_atomic_swap *sgpr_data*, *sgpr_base*[2], *offset*
D0: sreg, ↔, B32 S0: sreg, BUF S1: smem_offset, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = DATA;
RETURN_DATA = tmp.
```

Flags: OPF_MEM_ATOMIC

s_atomic_swap_x2 *sgpr_data*[2], *sgpr_base*[2], *offset*
D0: sreg, ↔, B64 S0: sreg, BUF S1: smem_offset, B32

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] = DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_MEM_ATOMIC

s_atomic_umax *sgpr_data*, *sgpr_base*[2], *offset*
D0: sreg, ↔, B32 S0: sreg, BUF S1: smem_offset, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (DATA > tmp) ? DATA : tmp; // unsigned compare
RETURN_DATA = tmp.
```

Flags: OPF_MEM_ATOMIC

s_atomic_umax_x2 *sgpr_data*[2], *sgpr_base*[2], *offset*
D0: sreg, ↔, B64 S0: sreg, BUF S1: smem_offset, B32

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] -= (DATA[0:1] > tmp) ? DATA[0:1] : tmp; // unsigned compare
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_MEM_ATOMIC

s_atomic_umin *sgpr_data*, *sgpr_base*[2], *offset*
D0: sreg, ↔, B32 S0: sreg, BUF S1: smem_offset, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (DATA < tmp) ? DATA : tmp; // unsigned compare
RETURN_DATA = tmp.
```

Flags: OPF_MEM_ATOMIC

s_atomic_umin_x2 *sgpr_data*[2], *sgpr_base*[2], *offset*
D0: sreg, ↔, B64 S0: sreg, BUF S1: smem_offset, B32

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] -= (DATA[0:1] < tmp) ? DATA[0:1] : tmp; // unsigned compare
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_MEM_ATOMIC

s_atomic_xor *sgpr_data*, *sgpr_base*[2], *offset*
D0: sreg, ↔, B32 S0: sreg, BUF S1: smem_offset, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] ^= DATA;
RETURN_DATA = tmp.
```

Flags: OPF_MEM_ATOMIC

s_atomic_xor_x2 *sgpr_data*[2], *sgpr_base*[2], *offset*
D0: sreg, ↔, B64 S0: sreg, BUF S1: smem_offset, B32

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] ^= DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_MEM_ATOMIC

s_buffer_atomic_add *sgpr_data*, *sgpr_base*[4], *offset*
D0: sreg, ↔, B32 S0: sreg, RSRC S1: smem_offset, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] += DATA;
RETURN_DATA = tmp.
```

Flags: OPF_BUFCONST, OPF_MEM_ATOMIC

s_buffer_atomic_add_x2 *sgpr_data*[2], *sgpr_base*[4], *offset*
D0: sreg, ↔, B64 S0: sreg, RSRC S1: smem_offset, B32

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] += DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_BUFCONST, OPF_MEM_ATOMIC

s_buffer_atomic_and *sgpr_data*, *sgpr_base*[4], *offset*
D0: sreg, ↔, B32 S0: sreg, RSRC S1: smem_offset, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] &= DATA;
RETURN_DATA = tmp.
```

Flags: OPF_BUFCONST, OPF_MEM_ATOMIC

s_buffer_atomic_and_x2 *sgpr_data*[2], *sgpr_base*[4], *offset*
D0: sreg, ↔, B64 S0: sreg, RSRC S1: smem_offset, B32

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] &= DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_BUFCONST, OPF_MEM_ATOMIC

s_buffer_atomic_cmpswap *sgpr_data*[2], *sgpr_base*[4], *offset*
D0: sreg, ↔, B32 S0: sreg, RSRC S1: smem_offset, B32

```
// 32bit
tmp = MEM[ADDR];
src = DATA[0];
cmp = DATA[1];
MEM[ADDR] = (tmp == cmp) ? src : tmp;
RETURN_DATA[0] = tmp.
```

Flags: OPF_ATOMIC_CMPSWAP, OPF_BUFCONST, OPF_MEM_ATOMIC

s_buffer_atomic_cmpswap_x2 *sgpr_data*[4], *sgpr_base*[4], *offset*
D0: sreg, ↔, B64 S0: sreg, RSRC S1: smem_offset, B32

```
// 64bit
tmp = MEM[ADDR];
src = DATA[0:1];
cmp = DATA[2:3];
MEM[ADDR] = (tmp == cmp) ? src : tmp;
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_ATOMIC_CMPSWAP, OPF_BUFCONST, OPF_MEM_ATOMIC

s_buffer_atomic_dec *sgpr_data*, *sgpr_base*[4], *offset*
D0: sreg, ↔, B32 S0: sreg, RSRC S1: smem_offset, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (tmp == 0 || tmp > DATA) ? DATA : tmp - 1; // unsigned compare
RETURN_DATA = tmp.
```

Flags: OPF_BUFCONST, OPF_MEM_ATOMIC

s_buffer_atomic_dec_x2 *sgpr_data*[2], *sgpr_base*[4], *offset*
D0: sreg, ↔, B64 S0: sreg, RSRC S1: smem_offset, B32

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] = (tmp == 0 || tmp > DATA[0:1]) ? DATA[0:1] : tmp - 1; // unsigned compare
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_BUFCONST, OPF_MEM_ATOMIC

s_buffer_atomic_inc *sgpr_data*, *sgpr_base*[4], *offset*
D0: sreg, ↔, B32 S0: sreg, RSRC S1: smem_offset, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (tmp ≥ DATA) ? 0 : tmp + 1; // unsigned compare
RETURN_DATA = tmp.
```

Flags: OPF_BUFCONST, OPF_MEM_ATOMIC

s_buffer_atomic_inc_x2 *sgpr_data*[2], *sgpr_base*[4], *offset*
D0: sreg, ↔, B64 S0: sreg, RSRC S1: smem_offset, B32

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] = (tmp ≥ DATA[0:1]) ? 0 : tmp + 1; // unsigned compare
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_BUFCONST, OPF_MEM_ATOMIC

s_buffer_atomic_or *sgpr_data*, *sgpr_base*[4], *offset*
D0: sreg, ↔, B32 S0: sreg, RSRC S1: smem_offset, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] |= DATA;
RETURN_DATA = tmp.
```

Flags: OPF_BUFCONST, OPF_MEM_ATOMIC

s_buffer_atomic_or_x2 *sgpr_data*[2], *sgpr_base*[4], *offset*
D0: sreg, ↔, B64 S0: sreg, RSRC S1: smem_offset, B32

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] |= DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_BUFCONST, OPF_MEM_ATOMIC

s_buffer_atomic_smax *sgpr_data*, *sgpr_base*[4], *offset*
D0: sreg, ↔, B32 S0: sreg, RSRC S1: smem_offset, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (DATA > tmp) ? DATA : tmp; // signed compare
RETURN_DATA = tmp.
```

Flags: OPF_BUFCONST, OPF_MEM_ATOMIC

s_buffer_atomic_smax_x2 *sgpr_data*[2], *sgpr_base*[4], *offset*
D0: sreg, ↔, B64 S0: sreg, RSRC S1: smem_offset, B32

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] -= (DATA[0:1] > tmp) ? DATA[0:1] : tmp; // signed compare
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_BUFCONST, OPF_MEM_ATOMIC

s_buffer_atomic_smin *sgpr_data*, *sgpr_base*[4], *offset*
D0: sreg, ↔, B32 S0: sreg, RSRC S1: smem_offset, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (DATA < tmp) ? DATA : tmp; // signed compare
RETURN_DATA = tmp.
```

Flags: OPF_BUFCONST, OPF_MEM_ATOMIC

s_buffer_atomic_smin_x2 *sgpr_data*[2], *sgpr_base*[4], *offset*
D0: sreg, ↔, B64 S0: sreg, RSRC S1: smem_offset, B32

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] -= (DATA[0:1] < tmp) ? DATA[0:1] : tmp; // signed compare
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_BUFCONST, OPF_MEM_ATOMIC

s_buffer_atomic_sub *sgpr_data*, *sgpr_base*[4], *offset*
D0: sreg, ↔, B32 S0: sreg, RSRC S1: smem_offset, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] -= DATA;
RETURN_DATA = tmp.
```

Flags: OPF_BUFCONST, OPF_MEM_ATOMIC

s_buffer_atomic_sub_x2 *sgpr_data*[2], *sgpr_base*[4], *offset*
D0: sreg, ↔, B64 S0: sreg, RSRC S1: smem_offset, B32

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] -= DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_BUFCONST, OPF_MEM_ATOMIC

s_buffer_atomic_swap *sgpr_data*, *sgpr_base*[4], *offset*
D0: sreg, ↔, B32 S0: sreg, RSRC S1: smem_offset, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = DATA;
RETURN_DATA = tmp.
```

Flags: OPF_BUFCONST, OPF_MEM_ATOMIC

s_buffer_atomic_swap_x2 *sgpr_data*[2], *sgpr_base*[4], *offset*
D0: sreg, ↔, B64 S0: sreg, RSRC S1: smem_offset, B32

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] = DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_BUFCONST, OPF_MEM_ATOMIC

s_buffer_atomic_umax *sgpr_data*, *sgpr_base*[4], *offset*
D0: sreg, ↔, B32 S0: sreg, RSRC S1: smem_offset, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (DATA > tmp) ? DATA : tmp; // unsigned compare
RETURN_DATA = tmp.
```

Flags: OPF_BUFCONST, OPF_MEM_ATOMIC

s_buffer_atomic_umax_x2 *sgpr_data*[2], *sgpr_base*[4], *offset*
D0: sreg, ↔, B64 S0: sreg, RSRC S1: smem_offset, B32

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] -= (DATA[0:1] > tmp) ? DATA[0:1] : tmp; // unsigned compare
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_BUFCONST, OPF_MEM_ATOMIC

s_buffer_atomic_umin *sgpr_data*, *sgpr_base*[4], *offset*
D0: sreg, ↔, B32 S0: sreg, RSRC S1: smem_offset, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (DATA < tmp) ? DATA : tmp; // unsigned compare
RETURN_DATA = tmp.
```

Flags: OPF_BUFCONST, OPF_MEM_ATOMIC

s_buffer_atomic_umin_x2 *sgpr_data*[2], *sgpr_base*[4], *offset*
D0: sreg, ↔, B64 S0: sreg, RSRC S1: smem_offset, B32

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] -= (DATA[0:1] < tmp) ? DATA[0:1] : tmp; // unsigned compare
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_BUFCONST, OPF_MEM_ATOMIC

s_buffer_atomic_xor *sgpr_data*, *sgpr_base*[4], *offset*
D0: sreg, ↔, B32 S0: sreg, RSRC S1: smem_offset, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] ^= DATA;
RETURN_DATA = tmp.
```

Flags: OPF_BUFCONST, OPF_MEM_ATOMIC

s_buffer_atomic_xor_x2 *sgpr_data*[2], *sgpr_base*[4], *offset*
D0: sreg, ↔, B64 S0: sreg, RSRC S1: smem_offset, B32

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] ^= DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_BUFCONST, OPF_MEM_ATOMIC

s_buffer_load_dword *sgpr_dst*, *sgpr_base*[4], *offset*
D0: sreg, B32 S0: sreg, RSRC S1: smem_offset, B32

Read 1 dword from scalar data cache. See S_LOAD_DWORD for details on the offset input.

Flags: OPF_BUFCONST

s_buffer_load_dwordx16 *sgpr_dst*[16], *sgpr_base*[4], *offset*
D0: sreg, B512 S0: sreg, RSRC S1: smem_offset, B32

Read 16 dwords from scalar data cache. See S_LOAD_DWORD for details on the offset input.

Flags: OPF_BUFCONST

s_buffer_load_dwordx2 *sgpr_dst*[2], *sgpr_base*[4], *offset*
D0: sreg, B64 S0: sreg, RSRC S1: smem_offset, B32

Read 2 dwords from scalar data cache. See S_LOAD_DWORD for details on the offset input.

Flags: OPF_BUFCONST

s_buffer_load_dwordx4 *sgpr_dst*[4], *sgpr_base*[4], *offset*
D0: sreg, B128 S0: sreg, RSRC S1: smem_offset, B32

Read 4 dwords from scalar data cache. See S_LOAD_DWORD for details on the offset input.

Flags: OPF_BUFCONST

s_buffer_load_dwordx8 *sgpr_dst*[8], *sgpr_base*[4], *offset*
D0: sreg, B256 S0: sreg, RSRC S1: smem_offset, B32

Read 8 dwords from scalar data cache. See S_LOAD_DWORD for details on the offset input.

Flags: OPF_BUFCONST

s_buffer_store_dword *sgpr_data*, *sgpr_base*[4], *offset*
S0: sreg, B32 S1: sreg, RSRC S2: smem_offset, B32

Write 1 dword to scalar data cache. See S_STORE_DWORD for details on the offset input.

Flags: OPF_BUFCONST, OPF_MEM_STORE

s_buffer_store_dwordx2 *sgpr_data*[2], *sgpr_base*[4], *offset*
S0: sreg, B64 S1: sreg, RSRC S2: smem_offset, B32

Write 2 dwords to scalar data cache. See S_STORE_DWORD for details on the offset input.

Flags: OPF_BUFCONST, OPF_MEM_STORE

s_buffer_store_dwordx4 *sgpr_data*[4], *sgpr_base*[4], *offset*
S0: sreg, B128 S1: sreg, RSRC S2: smem_offset, B32

Write 4 dwords to scalar data cache. See S_STORE_DWORD for details on the offset input.

Flags: OPF_BUFCONST, OPF_MEM_STORE

s_dcache_discard *sgpr_base*[2], *offset*
S0: sreg, BUF S1: smem_offset, B32

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.

s_dcache_discard_x2 *sgpr_base*[2], *offset*
S0: sreg, BUF S1: smem_offset, B32

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.

s_dcache_inv

Invalidate the scalar data cache.

Flags: SEN_NOOPR

s_dcache_inv_vol

Invalidate the scalar data cache volatile lines.

Flags: SEN_NOOPR

s_dcache_wb

Write back dirty data in the scalar data cache.

Flags: SEN_NOOPR

s_dcache_wb_vol

Write back dirty data in the scalar data cache volatile lines.

Flags: SEN_NOOPR

s_load_dword *sgpr_dst*, *sgpr_base*[2], *offset*
D0: sreg, B32 S0: sreg, BUF S1: smem_offset, B32

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.

s_load_dwordx16 *sgpr_dst*[16], *sgpr_base*[2], *offset*
D0: sreg, B512 S0: sreg, BUF S1: smem_offset, B32

Read 16 dwords from scalar data cache. See S_LOAD_DWORD for details on the offset input.

s_load_dwordx2 *sgpr_dst*[2], *sgpr_base*[2], *offset*
D0: sreg, B64 S0: sreg, BUF S1: smem_offset, B32

Read 2 dwords from scalar data cache. See S_LOAD_DWORD for details on the offset input.

s_load_dwordx4 *sgpr_dst*[4], *sgpr_base*[2], *offset*
D0: sreg, B128 S0: sreg, BUF S1: smem_offset, B32

Read 4 dwords from scalar data cache. See S_LOAD_DWORD for details on the offset input.

s_load_dwordx8 *sgpr_dst*[8], *sgpr_base*[2], *offset*
D0: sreg, B256 S0: sreg, BUF S1: smem_offset, B32

Read 8 dwords from scalar data cache. See S_LOAD_DWORD for details on the offset input.

s_memrealtime *sgpr_dst*[2]
D0: sreg, B64

Return current 64-bit RTC.

s_memtime *sgpr_dst*[2]
D0: sreg, B64

Return current 64-bit timestamp.

s_scratch_load_dword *sgpr_dst*, *sgpr_base*[2], *offset*
D0: sreg, B32 S0: sreg, BUF S1: smem_offset, B32

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.

Flags: OPF_SMEM_SCRATCH

s_scratch_load_dwordx2 *sgpr_dst*[2], *sgpr_base*[2], *offset*
D0: sreg, B64 S0: sreg, BUF S1: smem_offset, B32

Read 2 dwords from scalar data cache. See S_SCRATCH_LOAD_DWORD for details on the offset input.

Flags: OPF_SMEM_SCRATCH

s_scratch_load_dwordx4 *sgpr_dst*[4], *sgpr_base*[2], *offset*
D0: sreg, B128 S0: sreg, BUF S1: smem_offset, B32

Read 4 dwords from scalar data cache. See S_SCRATCH_LOAD_DWORD for details on the offset input.

Flags: OPF_SMEM_SCRATCH

s_scratch_store_dword *sgpr_data*, *sgpr_base*[2], *offset*
S0: sreg, B32 S1: sreg, BUF S2: smem_offset, B32

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.

Flags: OPF_MEM_STORE, OPF_SMEM_SCRATCH

s_scratch_store_dwordx2 *sgpr_data*[2], *sgpr_base*[2], *offset*
S0: sreg, B64 S1: sreg, BUF S2: smem_offset, B32

Write 2 dwords from scalar data cache. See S_SCRATCH_STORE_DWORD for details on the offset input.

Flags: OPF_MEM_STORE, OPF_SMEM_SCRATCH

s_scratch_store_dwordx4 *sgpr_data*[4], *sgpr_base*[2], *offset*
S0: sreg, B128 S1: sreg, BUF S2: smem_offset, B32

Write 4 dwords from scalar data cache. See S_SCRATCH_STORE_DWORD for details on the offset input.

Flags: OPF_MEM_STORE, OPF_SMEM_SCRATCH

s_store_dword *sgpr_data*, *sgpr_base*[2], *offset*
S0: sreg, B32 S1: sreg, BUF S2: smem_offset, B32

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.

Flags: OPF_MEM_STORE

s_store_dwordx2 *sgpr_data*[2], *sgpr_base*[2], *offset*
S0: sreg, B64 S1: sreg, BUF S2: smem_offset, B32

Write 2 dwords to scalar data cache. See S_STORE_DWORD for details on the offset input.

Flags: OPF_MEM_STORE

s_store_dwordx4 *sgpr_data*[4], *sgpr_base*[2], *offset*
S0: sreg, B128 S1: sreg, BUF S2: smem_offset, B32

Write 4 dwords to scalar data cache. See S_STORE_DWORD for details on the offset input.

Flags: OPF_MEM_STORE

8 Encoding VOP1

Vector ALU operations with one destination and one source. Instructions in this encoding may be promoted to VOP3 unless otherwise noted.

v_bfrev_b32 *vdst*, *src*
D0: vgpr, B32 S0: src, B32

v_bfrev_b32 *vdst*, *src*
D0: vgpr, B32 S0: src_nolit, B32

```
D.u[31:0] = S0.u[0:31].
```

Bitfield reverse. Input and output modifiers not supported.

Flags: OPF_CACGRP2

---

v_ceil_f16 *vdst*, *src*
D0: vgpr, F16 S0: src, F16

v_ceil_f16 *vdst*, *src*
D0: vgpr, F16 S0: src_nolit, 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.

---

v_ceil_f32 *vdst*, *src*
D0: vgpr, F32 S0: src, F32

v_ceil_f32 *vdst*, *src*
D0: vgpr, F32 S0: src_nolit, 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.

Flags: OPF_CACGRP2

---

v_ceil_f64 *vdst*[2], *src*[2]
D0: vgpr, F64 S0: src, F64

v_ceil_f64 *vdst*[2], *src*[2]
D0: vgpr, F64 S0: src_nolit, 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.

Flags: OPF_NODPP

v_clrexcp
v_clrexcp

Clear wave's exception state in SIMD (SP).

Flags: SEN_NOOPR , OPF_NODPP, OPF_NOSDWA

v_cos_f16	*vdst*,	*src*
	D0: vgpr, F16	S0: src, F16
v_cos_f16	*vdst*,	*src*
	D0: vgpr, F16	S0: src_nolit, F16

```
D.f16 = cos(S0.f16 * 2 * PI).
```

Trigonometric cosine. Denormals are supported.

Examples:

```
V_COS_F16(0xfc00) ⟹ 0xfe00      // cos(-INF) = NAN
V_COS_F16(0xfbff) ⟹ 0x3c00      // Most negative finite FP16
V_COS_F16(0x8000) ⟹ 0x3c00      // cos(-0.0) = 1
V_COS_F16(0x3400) ⟹ 0x0000      // cos(0.25) = 0
V_COS_F16(0x7bff) ⟹ 0x3c00      // Most positive finite FP16
V_COS_F16(0x7c00) ⟹ 0xfe00      // cos(+INF) = NAN
```

Flags: OPF_CACGRP2, OPF_TRANS

v_cos_f32	*vdst*,	*src*
	D0: vgpr, F32	S0: src, F32
v_cos_f32	*vdst*,	*src*
	D0: vgpr, F32	S0: src_nolit, F32

```
D.f = cos(S0.f * 2 * PI).
```

Trigonometric cosine. Denormals are supported.

Examples:

```
V_COS_F32(0xff800000) ⟹ 0xffc00000      // cos(-INF) = NAN
V_COS_F32(0xff7fffff) ⟹ 0x3f800000      // -MaxFloat, finite
V_COS_F32(0x80000000) ⟹ 0x3f800000      // cos(-0.0) = 1
V_COS_F32(0x3e800000) ⟹ 0x00000000      // cos(0.25) = 0
V_COS_F32(0x7f800000) ⟹ 0xffc00000      // cos(+INF) = NAN
```

Flags: OPF_TRANS

v_cvt_f16_f32	*vdst*,	*src*
	D0: vgpr, F16	S0: src, F32
v_cvt_f16_f32	*vdst*,	*src*
	D0: vgpr, F16	S0: src_nolit, F32

```
D.f16 = flt32_to_flt16(S0.f).
```

0.5ULP accuracy, supports input modifiers and creates FP16 denormals when appropriate.

v_cvt_f16_i16 *vdst*, *src*
D0: vgpr, F16 S0: src, I16
v_cvt_f16_i16 *vdst*, *src*
D0: vgpr, F16 S0: src_nolit, I16

```
D.f16 = int16_to_flt16(S.i16).
```

0.5ULP accuracy, supports denormals, rounding, exception flags and saturation.

v_cvt_f16_u16 *vdst*, *src*
D0: vgpr, F16 S0: src, U16
v_cvt_f16_u16 *vdst*, *src*
D0: vgpr, F16 S0: src_nolit, U16

```
D.f16 = uint16_to_flt16(S.u16).
```

0.5ULP accuracy, supports denormals, rounding, exception flags and saturation.

v_cvt_f32_f16 *vdst*, *src*
D0: vgpr, F32 S0: src, F16
v_cvt_f32_f16 *vdst*, *src*
D0: vgpr, F32 S0: src_nolit, F16

```
D.f = flt16_to_flt32(S0.f16).
```

0ULP accuracy, FP16 denormal inputs are always accepted.

Flags: OPF_CACGRP2

v_cvt_f32_f64 *vdst*, *src*[2]
D0: vgpr, F32 S0: src, F64
v_cvt_f32_f64 *vdst*, *src*[2]
D0: vgpr, F32 S0: src_nolit, F64

```
D.f = (float)S0.d.
```

0.5ULP accuracy, denormals are supported.

Flags: OPF_NODPP

v_cvt_f32_i32 *vdst*, *src*
D0: vgpr, F32 S0: src, I32
v_cvt_f32_i32 *vdst*, *src*
D0: vgpr, F32 S0: src_nolit, I32

```
D.f = (float)S0.i.
```

0.5ULP accuracy.

Flags: OPF_CACGRP2

v_cvt_f32_u32 *vdst*, *src*
D0: vgpr, F32 S0: src, U32
v_cvt_f32_u32 *vdst*, *src*
D0: vgpr, F32 S0: src_nolit, U32

```
D.f = (float)S0.u.
```

0.5ULP accuracy.

Flags: OPF_CACGRP2

v_cvt_f32_ubyte0 *vdst*, *src*
D0: vgpr, F32 S0: src, U32
v_cvt_f32_ubyte0 *vdst*, *src*
D0: vgpr, F32 S0: src_nolit, U32

```
D.f = (float)(S0.u[7:0]).
```

Flags: OPF_CACGRP2

v_cvt_f32_ubyte1 *vdst*, *src*
D0: vgpr, F32 S0: src, U32
v_cvt_f32_ubyte1 *vdst*, *src*
D0: vgpr, F32 S0: src_nolit, U32

```
D.f = (float)(S0.u[15:8]).
```

Flags: OPF_CACGRP2

v_cvt_f32_ubyte2 *vdst*, *src*
D0: vgpr, F32 S0: src, U32
v_cvt_f32_ubyte2 *vdst*, *src*
D0: vgpr, F32 S0: src_nolit, U32

```
D.f = (float)(S0.u[23:16]).
```

Flags: OPF_CACGRP2

v_cvt_f32_ubyte3 *vdst*, *src*
D0: vgpr, F32 S0: src, U32
v_cvt_f32_ubyte3 *vdst*, *src*
D0: vgpr, F32 S0: src_nolit, U32

```
D.f = (float)(S0.u[31:24]).
```

Flags: OPF_CACGRP2

v_cvt_f64_f32 *vdst*[2], *src*
D0: vgpr, F64 S0: src, F32
v_cvt_f64_f32 *vdst*[2], *src*
D0: vgpr, F64 S0: src_nolit, F32

```
D.d = (double)S0.f.
```

0ULP accuracy, denormals are supported.

Flags: OPF_NODPP

v_cvt_f64_i32 *vdst*[2], *src*
D0: vgpr, F64 S0: src, I32

v_cvt_f64_i32 *vdst*[2], *src*
D0: vgpr, F64 S0: src_nolit, I32

```
D.d = (double)S0.i.
```

0ULP accuracy.

Flags: OPF_NODPP

v_cvt_f64_u32 *vdst*[2], *src*
D0: vgpr, F64 S0: src, U32

v_cvt_f64_u32 *vdst*[2], *src*
D0: vgpr, F64 S0: src_nolit, U32

```
D.d = (double)S0.u.
```

0ULP accuracy.

Flags: OPF_NODPP

v_cvt_flr_i32_f32 *vdst*, *src*
D0: vgpr, I32 S0: src, F32

v_cvt_flr_i32_f32 *vdst*, *src*
D0: vgpr, I32 S0: src_nolit, F32

```
D.i = (int)floor(S0.f).
```

1ULP accuracy, denormals are supported.

Flags: OPF_CACGRP2

v_cvt_i16_f16 *vdst*, *src*
D0: vgpr, I16 S0: src, F16

v_cvt_i16_f16 *vdst*, *src*
D0: vgpr, I16 S0: src_nolit, F16

```
D.i16 = flt16_to_int16(S.f16).
```

1ULP accuracy, supports rounding, exception flags and saturation. FP16 denormals are accepted. Conversion is done with truncation.

Generation of the INEXACT exception is controlled by the CLAMP bit. INEXACT exceptions are enabled for this conversion iff CLAMP == 1.

v_cvt_i32_f32 *vdst*, *src*
D0: vgpr, I32 S0: src, F32

v_cvt_i32_f32 *vdst*, *src*
D0: vgpr, I32 S0: src_nolit, F32

```
D.i = (int)S0.f.
```

1ULP accuracy, out-of-range floating point values (including infinity) saturate. NaN is converted to 0.

Generation of the INEXACT exception is controlled by the CLAMP bit. INEXACT exceptions are enabled for this conversion iff CLAMP == 1.

Flags: OPF_CACGRP2

v_cvt_i32_f64 *vdst*, *src*[2]
D0: vgpr, I32 S0: src, F64
v_cvt_i32_f64 *vdst*, *src*[2]
D0: vgpr, I32 S0: src_nolit, F64

```
D.i = (int)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.

Flags: OPF_NODPP

v_cvt_norm_i16_f16 *vdst*, *src*
D0: vgpr, I16 S0: src, F16
v_cvt_norm_i16_f16 *vdst*, *src*
D0: vgpr, I16 S0: src_nolit, F16

```
D.i16 = flt16_to_snorm16(S.f16).
```

0.5ULP accuracy, supports rounding, exception flags and saturation, denormals are supported.

v_cvt_norm_u16_f16 *vdst*, *src*
D0: vgpr, U16 S0: src, F16
v_cvt_norm_u16_f16 *vdst*, *src*
D0: vgpr, U16 S0: src_nolit, F16

```
D.u16 = flt16_to_unorm16(S.f16).
```

0.5ULP accuracy, supports rounding, exception flags and saturation, denormals are supported.

v_cvt_off_f32_i4 *vdst*, *src*
D0: vgpr, F32 S0: src, I32
v_cvt_off_f32_i4 *vdst*, *src*
D0: vgpr, F32 S0: src_nolit, I32

4-bit signed int to 32-bit float. Used for interpolation in shader.

Flags: OPF_CACGRP2

v_cvt_rpi_i32_f32 *vdst*, *src*
D0: vgpr, I32 S0: src, F32
v_cvt_rpi_i32_f32 *vdst*, *src*
D0: vgpr, I32 S0: src_nolit, F32

```
D.i = (int)floor(S0.f + 0.5).
```

0.5ULP accuracy, denormals are supported.

Flags: OPF_CACGRP2

v_cvt_u16_f16 *vdst*, *src*
D0: vgpr, U16 S0: src, F16
v_cvt_u16_f16 *vdst*, *src*
D0: vgpr, U16 S0: src_nolit, F16

```
D.u16 = flt16_to_uint16(S.f16).
```

1ULP accuracy, supports rounding, exception flags and saturation. FP16 denormals are accepted. Conversion is done with truncation.

Generation of the INEXACT exception is controlled by the CLAMP bit. INEXACT exceptions are enabled for this conversion iff CLAMP == 1.

v_cvt_u32_f32 *vdst*, *src*
D0: vgpr, U32 S0: src, F32
v_cvt_u32_f32 *vdst*, *src*
D0: vgpr, U32 S0: src_nolit, F32

```
D.u = (unsigned)S0.f.
```

1ULP accuracy, out-of-range floating point values (including infinity) saturate. NaN is converted to 0.

Generation of the INEXACT exception is controlled by the CLAMP bit. INEXACT exceptions are enabled for this conversion iff CLAMP == 1.

Flags: OPF_CACGRP2

v_cvt_u32_f64 *vdst*, *src*[2]
D0: vgpr, U32 S0: src, F64
v_cvt_u32_f64 *vdst*, *src*[2]
D0: vgpr, U32 S0: src_nolit, 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.

Flags: OPF_NODPP

v_exp_f16 *vdst*, *src*
D0: vgpr, F16 S0: src, F16
v_exp_f16 *vdst*, *src*
D0: vgpr, F16 S0: src_nolit, F16

```
D.f16 = pow(2.0, S0.f16).
```

Base 2 exponentiation. 0.51ULP accuracy, denormals are supported.

Examples:

```
V_EXP_F16(0xfc00) ⟹ 0x0000    // exp(-INF) = 0
V_EXP_F16(0x8000) ⟹ 0x3c00    // exp(-0.0) = 1
V_EXP_F16(0x7c00) ⟹ 0x7c00    // exp(+INF) = +INF
```

Flags: OPF_CACGRP2, OPF_TRANS

v_exp_f32 *vdst*, *src*
D0: vgpr, F32 S0: src, F32
v_exp_f32 *vdst*, *src*
D0: vgpr, F32 S0: src_nolit, F32

```
D.f = pow(2.0, S0.f).
```

Base 2 exponentiation. 1ULP accuracy, denormals are flushed.

Examples:

```
V_EXP_F32(0xff800000) ⟹ 0x00000000  // exp(-INF) = 0
V_EXP_F32(0x80000000) ⟹ 0x3f800000  // exp(-0.0) = 1
V_EXP_F32(0x7f800000) ⟹ 0x7f800000  // exp(+INF) = +INF
```

Flags: OPF_CACGRP2, OPF_TRANS

v_exp_legacy_f32 *vdst*, *src*
D0: vgpr, F32 S0: src, F32
v_exp_legacy_f32 *vdst*, *src*
D0: vgpr, F32 S0: src_nolit, F32

```
D.f = pow(2.0, S0.f).
```

Power with legacy semantics.

Flags: ASIC_LEGACY_LOG, OPF_CACGRP2, OPF_TRANS

v_ffbh_i32 *vdst*, *src*
D0: vgpr, I32 S0: src, I32
v_ffbh_i32 *vdst*, *src*
D0: vgpr, I32 S0: src_nolit, I32

```
D.i = -1; // Set if all bits are the same
for i in 1 . . . 31 do
    // 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
```

Flags: OPF_CACGRP2

v_ffbh_u32 *vdst*, *src*
D0: vgpr, I32 S0: src, U32

v_ffbh_u32 *vdst*, *src*
D0: vgpr, I32 S0: src_nolit, U32

```
D.i = -1; // Set if no ones are found
for i in 0 . . . 31 do
        // Note: search is from the MSB
        if S0.u[31 - 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:

V_FFBH_U32(0x00000000) $\Longrightarrow$ 0xffffffff
V_FFBH_U32(0x800000ff) $\Longrightarrow$ 0
V_FFBH_U32(0x100000ff) $\Longrightarrow$ 3
V_FFBH_U32(0x0000ffff) $\Longrightarrow$ 16
V_FFBH_U32(0x00000001) $\Longrightarrow$ 31

Flags: OPF_CACGRP2

v_ffbl_b32 *vdst*, *src*
D0: vgpr, I32 S0: src, U32

v_ffbl_b32 *vdst*, *src*
D0: vgpr, I32 S0: src_nolit, U32

```
D.i = -1; // Set if no ones are found
for i in 0 . . . 31 do // Search from LSB
        if S0.u[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:

V_FFBL_B32(0x00000000) $\Longrightarrow$ 0xffffffff
V_FFBL_B32(0xff000001) $\Longrightarrow$ 0
V_FFBL_B32(0xff000008) $\Longrightarrow$ 3
V_FFBL_B32(0xffff0000) $\Longrightarrow$ 16
V_FFBL_B32(0x80000000) $\Longrightarrow$ 31

Flags: OPF_CACGRP2

v_floor_f16 *vdst*, *src*
D0: vgpr, F16 S0: src, F16
v_floor_f16 *vdst*, *src*
D0: vgpr, F16 S0: src_nolit, 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.

v_floor_f32 *vdst*, *src*
D0: vgpr, F32 S0: src, F32
v_floor_f32 *vdst*, *src*
D0: vgpr, F32 S0: src_nolit, F32

```
D.f = trunc(S0.f);
if(S0.f < 0.0 && S0.f ≠ D.f) then
        D.f += -1.0;
endif.
```

Round down to previous whole integer.

Flags: OPF_CACGRP2

v_floor_f64 *vdst*[2], *src*[2]
D0: vgpr, F64 S0: src, F64
v_floor_f64 *vdst*[2], *src*[2]
D0: vgpr, F64 S0: src_nolit, 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.

Flags: OPF_NODPP

v_fract_f16 *vdst*, *src*
D0: vgpr, F16 S0: src, F16
v_fract_f16 *vdst*, *src*
D0: vgpr, F16 S0: src_nolit, F16

```
D.f16 = S0.f16 + -floor(S0.f16).
```

Return fractional portion of a number. 0.5ULP accuracy, denormals are accepted.

v_fract_f32 *vdst*, *src*
D0: vgpr, F32 S0: src, F32
v_fract_f32 *vdst*, *src*
D0: vgpr, F32 S0: src_nolit, F32

```
D.f = S0.f + -floor(S0.f).
```

Return fractional portion of a number. 0.5ULP accuracy, denormals are accepted.

Flags: OPF_CACGRP2

v_fract_f64 *vdst*[2], *src*[2]
D0: vgpr, F64 S0: src, F64
v_fract_f64 *vdst*[2], *src*[2]
D0: vgpr, F64 S0: src_nolit, F64

```
D.d = S0.d + -floor(S0.d).
```

Return fractional portion of a number. 0.5ULP accuracy, denormals are accepted.

Flags: OPF_CACGRP2, OPF_NODPP

v_frexp_exp_i16_f16 *vdst*, *src*
D0: vgpr, I16 S0: src, F16
v_frexp_exp_i16_f16 *vdst*, *src*
D0: vgpr, I16 S0: src_nolit, 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.

v_frexp_exp_i32_f32 *vdst*, *src*
D0: vgpr, I32 S0: src, F32
v_frexp_exp_i32_f32 *vdst*, *src*
D0: vgpr, I32 S0: src_nolit, F32

```
if(S0.f == ±INF || S0.f == NAN) then
        D.i = 0;
else
        D.i = TwosComplement(Exponent(S0.f) - 127 + 1);
endif.
```

Returns exponent of single precision float input, such that S0.f = significand * (2 ** exponent). See also V_FREXP_MANT_F32, which returns the significand. See the C library function frexp() for more information.

v_frexp_exp_i32_f64 *vdst*, *src*[2]
D0: vgpr, I32 S0: src, F64
v_frexp_exp_i32_f64 *vdst*, *src*[2]
D0: vgpr, I32 S0: src_nolit, 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.

Flags: OPF_NODPP

v_frexp_mant_f16 *vdst*, *src*
D0: vgpr, F16 S0: src, F16
v_frexp_mant_f16 *vdst*, *src*
D0: vgpr, F16 S0: src_nolit, F16

```
if(S0.f16 == ±INF || S0.f16 == NAN) then
        D.f16 = S0.f16;
else
        D.f16 = Mantissa(S0.f16);
endif.
```

Result range is in (-1.0,-0.5][0.5,1.0) in normal cases. Returns binary significand of half precision float input, such that S0.f16 = significand * (2 ** exponent). See also V_FREXP_EXP_I16_F16, which returns integer exponent. See the C library function frexp() for more information.

v_frexp_mant_f32 *vdst*, *src*
D0: vgpr, F32 S0: src, F32
v_frexp_mant_f32 *vdst*, *src*
D0: vgpr, F32 S0: src_nolit, F32

```
if(S0.f == ±INF || S0.f == NAN) then
        D.f = S0.f;
else
        D.f = Mantissa(S0.f);
endif.
```

Result range is in (-1.0,-0.5][0.5,1.0) in normal cases. Returns binary significand of single precision float input, such that S0.f = significand * (2 ** exponent). See also V_FREXP_EXP_I32_F32, which returns integer exponent. See the C library function frexp() for more information.

v_frexp_mant_f64 *vdst*[2], *src*[2]
D0: vgpr, F64 S0: src, F64
v_frexp_mant_f64 *vdst*[2], *src*[2]
D0: vgpr, F64 S0: src_nolit, 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 normal 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.

Flags: OPF_NODPP

v_log_f16 *vdst*, *src*
D0: vgpr, F16 S0: src, F16
v_log_f16 *vdst*, *src*
D0: vgpr, F16 S0: src_nolit, F16

```
D.f16 = log2(S0.f).
```

Base 2 logarithm. 0.51ULP accuracy, denormals are supported.

Examples:

```
V_LOG_F16(0xfc00) ⟹ 0xfe00      // log(-INF) = NAN
V_LOG_F16(0xbc00) ⟹ 0xfe00      // log(-1.0) = NAN
V_LOG_F16(0x8000) ⟹ 0xfc00      // log(-0.0) = -INF
V_LOG_F16(0x0000) ⟹ 0xfc00      // log(+0.0) = -INF
V_LOG_F16(0x3c00) ⟹ 0x0000      // log(+1.0) = 0
V_LOG_F16(0x7c00) ⟹ 0x7c00      // log(+INF) = +INF
```

Flags: OPF_CACGRP2, OPF_TRANS

v_log_f32 *vdst*, *src*
D0: vgpr, F32 S0: src, F32
v_log_f32 *vdst*, *src*
D0: vgpr, F32 S0: src_nolit, F32

```
D.f = log2(S0.f).
```

Base 2 logarithm. 1ULP accuracy, denormals are flushed.

Examples:

```
V_LOG_F32(0xff800000) ⟹ 0xffc00000  // log(-INF) = NAN
V_LOG_F32(0xbf800000) ⟹ 0xffc00000  // log(-1.0) = NAN
V_LOG_F32(0x80000000) ⟹ 0xff800000  // log(-0.0) = -INF
V_LOG_F32(0x00000000) ⟹ 0xff800000  // log(+0.0) = -INF
V_LOG_F32(0x3f800000) ⟹ 0x00000000  // log(+1.0) = 0
V_LOG_F32(0x7f800000) ⟹ 0x7f800000  // log(+INF) = +INF
```

You may safely ignore this paragraph. This paragraph is inserted to ensure this specification fails legal scans, and for tracking purposes. It has nothing to do with the V_LOG_F32 opcode. TRACKJDS20151119, GLADIUS, MILOS, KRYPTOS, ANUBIS.

Flags: OPF_CACGRP2, OPF_TRANS

v_log_legacy_f32 *vdst*, *src*
D0: vgpr, F32 S0: src, F32
v_log_legacy_f32 *vdst*, *src*
D0: vgpr, F32 S0: src_nolit, F32

```
D.f = log2(S0.f).
```

Base 2 logarithm with legacy semantics.

Flags: ASIC_LEGACY_LOG, OPF_CACGRP2, OPF_TRANS

v_mov_b32 *vdst*, *src*
D0: vgpr, B32 S0: src, B32
v_mov_b32 *vdst*, *src*
D0: vgpr, B32 S0: src_nolit, B32

```
D.u = S0.u.
```

Input and output modifiers not supported; this is an untyped operation.

Flags: OPF_CACGRP2, OPF_MOV

v_mov_fed_b32 *vdst*, *src*
D0: vgpr, B32 S0: src, B32
v_mov_fed_b32 *vdst*, *src*
D0: vgpr, B32 S0: src_nolit, B32

```
D.u = Corrupt(S0.u).
```

Introduce EDC double error upon write to dest vgpr without causing an exception.

Input and output modifiers not supported; this is an untyped operation.

Flags: ASIC_FED_INSTRUCTIONS, OPF_CACGRP2, OPF_MOV

v_nop
v_nop

Do nothing.

Flags: SEN_NOOPR

v_not_b32	*vdst*,	*src*
	D0: vgpr, U32	S0: src, U32
v_not_b32	*vdst*,	*src*
	D0: vgpr, U32	S0: src_nolit, U32

```
D.u = ~S0.u.
```

Bitwise negation. Input and output modifiers not supported.

Flags: OPF_CACGRP2

v_rcp_f16	*vdst*,	*src*
	D0: vgpr, F16	S0: src, F16
v_rcp_f16	*vdst*,	*src*
	D0: vgpr, F16	S0: src_nolit, F16

```
D.f16 = 1.0 / S0.f16.
```

Reciprocal with IEEE rules and 0.51ULP accuracy.

Examples:

```
V_RCP_F16(0xfc00) ⟹ 0x8000      // rcp(-INF) = -0
V_RCP_F16(0xc000) ⟹ 0xb800      // rcp(-2.0) = -0.5
V_RCP_F16(0x8000) ⟹ 0xfc00      // rcp(-0.0) = -INF
V_RCP_F16(0x0000) ⟹ 0x7c00      // rcp(+0.0) = +INF
V_RCP_F16(0x7c00) ⟹ 0x0000      // rcp(+INF) = +0
```

Flags: OPF_CACGRP2, OPF_TRANS

v_rcp_f32	*vdst*,	*src*
	D0: vgpr, F32	S0: src, F32
v_rcp_f32	*vdst*,	*src*
	D0: vgpr, F32	S0: src_nolit, F32

```
D.f = 1.0 / S0.f.
```

Reciprocal with IEEE rules and 1ULP accuracy. Accuracy converges to < 0.5ULP when using the Newton-Raphson method and 2 FMA operations. Denormals are flushed.

Examples:

```
V_RCP_F32(0xff800000) ⟹ 0x80000000      // rcp(-INF) = -0
V_RCP_F32(0xc0000000) ⟹ 0xbf000000      // rcp(-2.0) = -0.5
V_RCP_F32(0x80000000) ⟹ 0xff800000      // rcp(-0.0) = -INF
V_RCP_F32(0x00000000) ⟹ 0x7f800000      // rcp(+0.0) = +INF
V_RCP_F32(0x7f800000) ⟹ 0x00000000      // rcp(+INF) = +0
```

Flags: OPF_CACGRP2, OPF_TRANS

v_rcp_f64 *vdst*[2], *src*[2]
D0: vgpr, F64 S0: src, F64
v_rcp_f64 *vdst*[2], *src*[2]
D0: vgpr, F64 S0: src_nolit, F64

```
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.

Flags: OPF_CACGRP2, OPF_DPFP, OPF_NODPP, OPF_TRANS

v_rcp_iflag_f32 *vdst*, *src*
D0: vgpr, F32 S0: src, F32
v_rcp_iflag_f32 *vdst*, *src*
D0: vgpr, F32 S0: src_nolit, F32

```
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_IFLAG_F32
MUL_F32 (2**32 - 1)
CVT_U32_F32

Signed:
CVT_F32_I32
RCP_IFLAG_F32
MUL_F32 (2**31 - 1)
CVT_I32_F32

Flags: OPF_CACGRP2, OPF_TRANS

v_readfirstlane_b32 *sdst*, *vsrc*
D0: sreg_novcc, B32 S0: vgpr_or_lds, B32

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. SQ translates to V_READLANE_B32.

Input and output modifiers not supported; this is an untyped operation.

Flags: OPF_NODPP, OPF_NOSDWA, OPF_NOVOP3

v_rndne_f16 *vdst*, *src*
D0: vgpr, F16 S0: src, F16
v_rndne_f16 *vdst*, *src*
D0: vgpr, F16 S0: src_nolit, 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.

v_rndne_f32 *vdst*, *src*
D0: vgpr, F32 S0: src, F32
v_rndne_f32 *vdst*, *src*
D0: vgpr, F32 S0: src_nolit, F32

```
D.f = floor(S0.f + 0.5);
if(floor(S0.f) is even && fract(S0.f) == 0.5) then
        D.f -= 1.0;
endif.
```

Round-to-nearest-even semantics.

Flags: OPF_CACGRP2

v_rndne_f64 *vdst*[2], *src*[2]
D0: vgpr, F64 S0: src, F64
v_rndne_f64 *vdst*[2], *src*[2]
D0: vgpr, F64 S0: src_nolit, 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.

Flags: OPF_NODPP

v_rsq_f16 *vdst*, *src*
D0: vgpr, F16 S0: src, F16
v_rsq_f16 *vdst*, *src*
D0: vgpr, F16 S0: src_nolit, F16

```
D.f16 = 1.0 / sqrt(S0.f16).
```

Reciprocal square root with IEEE rules. 0.51ULP accuracy, denormals are supported.

Examples:

```
V_RSQ_F16(0xfc00) ⟹ 0xfe00      // rsq(-INF) = NAN
V_RSQ_F16(0x8000) ⟹ 0xfc00      // rsq(-0.0) = -INF
V_RSQ_F16(0x0000) ⟹ 0x7c00      // rsq(+0.0) = +INF
V_RSQ_F16(0x4400) ⟹ 0x3800      // rsq(+4.0) = +0.5
V_RSQ_F16(0x7c00) ⟹ 0x0000      // rsq(+INF) = +0
```

Flags: OPF_CACGRP2, OPF_TRANS

v_rsq_f32 *vdst*, *src*
D0: vgpr, F32 S0: src, F32

v_rsq_f32 *vdst*, *src*
D0: vgpr, F32 S0: src_nolit, F32

```
D.f = 1.0 / sqrt(S0.f).
```

Reciprocal square root with IEEE rules. 1ULP accuracy, denormals are flushed.

Examples:

```
V_RSQ_F32(0xff800000) ⟹ 0xffc00000  // rsq(-INF) = NAN
V_RSQ_F32(0x80000000) ⟹ 0xff800000  // rsq(-0.0) = -INF
V_RSQ_F32(0x00000000) ⟹ 0x7f800000  // rsq(+0.0) = +INF
V_RSQ_F32(0x40800000) ⟹ 0x3f000000  // rsq(+4.0) = +0.5
V_RSQ_F32(0x7f800000) ⟹ 0x00000000  // rsq(+INF) = +0
```

Flags: OPF_CACGRP2, OPF_TRANS

v_rsq_f64 *vdst*[2], *src*[2]
D0: vgpr, F64 S0: src, F64

v_rsq_f64 *vdst*[2], *src*[2]
D0: vgpr, F64 S0: src_nolit, F64

```
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.

Flags: OPF_CACGRP2, OPF_DPFP, OPF_NODPP, OPF_TRANS

v_sat_pk_u8_i16 *vdst*, *src*
D0: vgpr, B16 S0: src, B32

v_sat_pk_u8_i16 *vdst*, *src*
D0: vgpr, B16 S0: src_nolit, B32

```
D.u32 = {16'b0, sat8(S.u[31:16]), sat8(S.u[15:0])}.
```

v_screen_partition_4se_b32 *vdst*, *src*
D0: vgpr, B32 S0: src, B32
v_screen_partition_4se_b32 *vdst*, *src*
D0: vgpr, B32 S0: src_nolit, B32

```
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] ^ y[6], y[5] ^ x[6] }` . 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 — if `((x_max/32 - x_min/32 ≥ 3)` OR `((y_max/32 - y_min/32 ≥ 3)` (tile size of 32) then all backends are covered.

3. Call the opcode with this 8 bit select: `{ x_min[6:5], y_min[6:5], x_max[6:5], y_max[6:5] }` .

4. The opcode will return a 4 bit mask indicating which backends are covered, where bit 0 indicates SE0 is covered and bit 3 indicates SE3 is covered.

Example:

1. The calculated bounding box is (0, 0) to (25, 35).

2. Observe the bounding box is not large enough to trivially cover all backends.

3. Divide by tile size 32 and concatenate bits to produce a selector of binary 00000001.

4. Opcode will return `0x3` which means backend 0 and 1 are covered.

v_sin_f16 *vdst*, *src*
D0: vgpr, F16 S0: src, F16
v_sin_f16 *vdst*, *src*
D0: vgpr, F16 S0: src_nolit, 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
```

Flags: OPF_CACGRP2, OPF_TRANS

v_sin_f32 *vdst*, *src*
D0: vgpr, F32 S0: src, F32
v_sin_f32 *vdst*, *src*
D0: vgpr, F32 S0: src_nolit, F32

```
D.f = sin(S0.f * 2 * PI).
```

Trigonometric sine. Denormals are supported.

Examples:

```
V_SIN_F32(0xff800000) ⟹ 0xffc00000      // sin(-INF) = NAN
V_SIN_F32(0xff7fffff) ⟹ 0x00000000      // -MaxFloat, finite
V_SIN_F32(0x80000000) ⟹ 0x80000000      // sin(-0.0) = -0
V_SIN_F32(0x3e800000) ⟹ 0x3f800000      // sin(0.25) = 1
V_SIN_F32(0x7f800000) ⟹ 0xffc00000      // sin(+INF) = NAN
```

Flags: OPF_CACGRP2, OPF_TRANS

v_sqrt_f16 *vdst*, *src*
D0: vgpr, F16 S0: src, F16
v_sqrt_f16 *vdst*, *src*
D0: vgpr, F16 S0: src_nolit, F16

```
D.f16 = sqrt(S0.f16).
```

Square root. 0.51ULP accuracy, denormals are supported.

Examples:

```
V_SQRT_F16(0xfc00) ⟹ 0xfe00      // sqrt(-INF) = NAN
V_SQRT_F16(0x8000) ⟹ 0x8000      // sqrt(-0.0) = -0
V_SQRT_F16(0x0000) ⟹ 0x0000      // sqrt(+0.0) = +0
V_SQRT_F16(0x4400) ⟹ 0x4000      // sqrt(+4.0) = +2.0
V_SQRT_F16(0x7c00) ⟹ 0x7c00      // sqrt(+INF) = +INF
```

Flags: OPF_CACGRP2, OPF_TRANS

v_sqrt_f32 *vdst*, *src*
D0: vgpr, F32 S0: src, F32
v_sqrt_f32 *vdst*, *src*
D0: vgpr, F32 S0: src_nolit, F32

```
D.f = sqrt(S0.f).
```

Square root. 1ULP accuracy, denormals are flushed.

Examples:

```
V_SQRT_F32(0xff800000) ⟹ 0xffc00000  // sqrt(-INF) = NAN
V_SQRT_F32(0x80000000) ⟹ 0x80000000  // sqrt(-0.0) = -0
V_SQRT_F32(0x00000000) ⟹ 0x00000000  // sqrt(+0.0) = +0
V_SQRT_F32(0x40800000) ⟹ 0x40000000  // sqrt(+4.0) = +2.0
V_SQRT_F32(0x7f800000) ⟹ 0x7f800000  // sqrt(+INF) = +INF
```

Flags: OPF_CACGRP2, OPF_TRANS

v_sqrt_f64 *vdst*[2], *src*[2]
D0: vgpr, F64 S0: src, F64
v_sqrt_f64 *vdst*[2], *src*[2]
D0: vgpr, F64 S0: src_nolit, F64

```
D.d = sqrt(S0.d).
```

Square root with perhaps not the accuracy you were hoping for – (2**29)ULP accuracy. On the upside, denormals are supported.

Flags: OPF_CACGRP2, OPF_DPFP, OPF_NODPP, OPF_TRANS

v_swap_b32 *dst*, *src*
D0: vgpr, ↔, B32 S0: src_vgpr, ↔, B32

```
tmp = D.u;
D.u = S0.u;
S0.u = tmp.
```

Swap operands. Input and output modifiers not supported; this is an untyped operation.

Flags: OPF_CACGRP2, OPF_MOV, OPF_NODPP, OPF_NOSDWA, OPF_NOVOP3

v_trunc_f16 *vdst*, *src*
D0: vgpr, F16 S0: src, F16
v_trunc_f16 *vdst*, *src*
D0: vgpr, F16 S0: src_nolit, F16

```
D.f16 = trunc(S0.f16).
```

Return integer part of S0.f16, round-to-zero semantics.

v_trunc_f32	*vdst*,	*src*
	D0: vgpr, F32	S0: src, F32
v_trunc_f32	*vdst*,	*src*
	D0: vgpr, F32	S0: src_nolit, F32

```
D.f = trunc(S0.f).
```

Return integer part of S0.f, round-to-zero semantics.

Flags: OPF_CACGRP2

v_trunc_f64	*vdst*[2],	*src*[2]
	D0: vgpr, F64	S0: src, F64
v_trunc_f64	*vdst*[2],	*src*[2]
	D0: vgpr, F64	S0: src_nolit, F64

```
D.d = trunc(S0.d).
```

Return integer part of S0.d, round-to-zero semantics.

Flags: OPF_NODPP

8.1 Notes for Encoding VOP1

8.1.1 Input modifiers

Source operands may invoke the negation and absolute-value input modifiers with `-`*src* and `abs(`*src*`)`, respectively. For example,

```
v_add_f32 v0, v1, -v2           // Subtract v2 from v1
v_add_f32 v0, abs(v1), abs(v2)  // Take absolute value of both inputs
```

In general, negation and absolute value are only supported for floating point input operands (operands with a type of F16, F32 or F64); they are not supported for integer or untyped inputs.

8.1.2 Output modifiers

`mul:{1,2,4}`
: Set output modifier to multiply by N. Default 1, which leaves the result unmodified. This operation is only defined when the result is F16, F32 or F64.

`div:{1,2}`
: Set output modifier to divide by N. Default 1, which leaves the result unmodified. This operation is only defined when the result is F16, F32 or F64.

`clamp:{0,1}`
: Clamp (saturate) the output. Default 0. Can also write `noclamp`. Saturation is defined as follows.

  For I16, I32, I64 results: if the result exceeds SHRT_MAX/INT_MAX/LLONG_MAX it will be clamped to the most positive representable value; if the result is below SHRT_MIN/INT_MIN/LLONG_MIN it will be clamped to the most negative representable value.

  For U16, U32, U64 results: if the result exceeds USHRT_MAX/UINT_MAX/ULLONG_MAX it will be clamped to the most positive representable value; if the result is below 0 it will be clamped to 0.

  For F16, F32, F64 results: the result will be clamped to the interval [0.0, 1.0].

8.1.3 Interpolation operands and modifiers

vgpr_dst is a vector GPR to store result in, and use as accumulator source in certain interpolation operations.

vgpr_ij is a vector GPR to read i/j value from.

attr is `attr0.x` through `attr63.w`, parameter attribute and channel to be interpolated.

param is `p10`, `p20` or `p0`.

For 16-bit interpolation it is necessary to specify whether we are operating on the high or low word of the attribute. For this, two new modifiers are provided:

`high`
: Interpolate using the high 16 bits of the attribute.

`low`
: Interpolate using the low 16 bits of the attribute. Default.

8.1.4 Other modifiers

`vop3:{0,1}`
: Force VOP3 encoding even if instruction can be represented in smaller encoding. Default 0. Can also write `novop3`. Note that even if this modifier is not set, an opcode will still use the VOP3 encoding if the operands or modifiers given cannot be expressed in a smaller encoding.

8.1.5 SDWA output modifiers

This only applies to VOP1/VOP2/VOPC encodings.

The SDWA subencoding supports sign-extension of inputs; this may be written as `sext(`*src*`)`, similar to how the `abs` modifier is used.

`src0_sel:`*sdwa_sel*
: Apply to the selected sdwa data bits of src0. Default DWORD.

`src1_sel:`*sdwa_sel*
: Apply to the selected sdwa data bits of src1. Default DWORD.

`dst_sel:`*sdwa_sel*
: Apply to the selected sdwa data bits of dst. Default DWORD.

`dst_unused:`*sdwa_unused*
: Determine what to do with the unused dst bits. Default UNUSED_PAD.

8.1.6 DPP output modifiers

This only applies to VOP1/VOP2/VOPC encodings.

General modifiers Any combination of these modifiers may be applied for DPP instructions.

`bank_mask:[0. . . 0xf]`
: Apply to the selected bank mask bits.

`row_mask:[0. . . 0xf]`
: Apply to the selected row mask bits.

`bound_ctrl:{0,1}`
: If true, then writes to out-of-bounds threads are written as zero. If false, writes to out-of-bounds threads are disabled.

DPP permutation control Only one of the following modifiers may be applied to the instruction.

Each wave of 64 threads $(W_0, \ldots, W_{63})$ is divided into 4 *rows* of 16 threads, $(R_0, \ldots, R_{15})$ and into 16 *quads* of 4 threads (usually pixels), $(P_0, \ldots, P_3)$.

`quad_perm:[pix0,pix1,pix2,pix3]`

DPP permutation: Applied for each pixel within a quad. The `N`th entry in the array specifies that the `N`th pixel in the output will obtain its data from the `quad_perm[N]`th pixel in the input.

The identity transformation is represented as `quad_perm:[0,1,2,3]`.

`quad_perm:[1,2,0,1]` applies the following permutation which includes broadcasting one of the pixels to multiple destinations:

$$\begin{bmatrix} P_0 & P_1 \\ P_2 & P_3 \end{bmatrix} : \qquad \begin{bmatrix} A & B \\ C & D \end{bmatrix} \Rightarrow \begin{bmatrix} B & C \\ A & B \end{bmatrix}$$

To broadcast one pixel (e.g. pixel 0) to all destinations in the quad, use `quad_perm:[0,0,0,0]`:

$$\begin{bmatrix} P_0 & P_1 \\ P_2 & P_3 \end{bmatrix} : \qquad \begin{bmatrix} A & B \\ C & D \end{bmatrix} \Rightarrow \begin{bmatrix} A & A \\ A & A \end{bmatrix}$$

In general the output thread P' is determined by

$$P'_n = P_{\texttt{quad\_perm}[n]}$$

`row_shr:[1. . . 15]`

DPP permutation: Shifts threads in each row to the right by the specified amount.

In general the output thread R' with count C is determined by

$$R'_n = \begin{cases} R_{n-C} & n \geq C \\ \texttt{bound\_ctrl} & \text{otherwise} \end{cases}$$

`row_shl:[1. . . 15]`

DPP permutation: Shifts threads in each row to the left by the specified amount.

In general the output thread R' with count C is determined by

$$R'_n = \begin{cases} R_{n+C} & n + C < 16 \\ \texttt{bound\_ctrl} & \text{otherwise} \end{cases}$$

`row_ror:[1. . . 15]`

DPP permutation: Rotates threads in each row to the right by the specified amount.

In general the output thread R' with count C is determined by

$$R'_n = \begin{cases} R_{n-C} & n \geq C \\ R_{n-C+16} & \text{otherwise} \end{cases}$$

`wave_shr`
DPP permutation: Shifts threads in the entire wave to the right by one.

In general the output thread W' is determined by

$$W'_n = \begin{cases} W_{n-1} & n > 0 \\ \texttt{bound\_ctrl} & \text{otherwise} \end{cases}$$

`wave_shl`
DPP permutation: Shifts threads in the entire wave to the left by one.

In general the output thread W' is determined by

$$W'_n = \begin{cases} W_{n+1} & n < 63 \\ \texttt{bound\_ctrl} & \text{otherwise} \end{cases}$$

`wave_ror`
DPP permutation: Rotates threads in the entire wave to the right by one.

In general the output thread W' is determined by

$$W'_n = \begin{cases} W_{n-1} & n > 0 \\ W_{63} & \text{otherwise} \end{cases}$$

`wave_rol`
DPP permutation: Rotates threads in the entire wave to the left by one.

In general the output thread W' is determined by

$$W'_n = \begin{cases} W_{n+1} & n < 63 \\ W_0 & \text{otherwise} \end{cases}$$

`row_mirror`
DPP permutation: Mirrors threads within each row.

In general the output thread R' is determined by

$$R'_n = R_{15-n}$$

`row_half_mirror`
: DPP permutation: Mirrors threads within each half-row.

  In general the output thread R' is determined by

$$R'_n = \begin{cases} R_{7-n} & n < 8 \\ R_{23-n} & \text{otherwise} \end{cases}$$

`row_bcast:{15, 31}`
: DPP permutation: Broadcast a thread to subsequent rows.

9 Encoding VOPC

Vector ALU comparison operations with two sources. Instructions in this encoding may be promoted to VOP3 unless otherwise noted. Note that vector compare instructions respect the EXEC mask for computing the result in each thread; if EXEC[threadId] is zero, then the comparison for that thread is not performed and the result of VCC[threadId] is always zero.

v_cmp_class_f16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F16 S1: vgpr, F16
v_cmp_class_f16 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, F16 S1: src_simple, 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.

v_cmp_class_f32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F32 S1: vgpr, B32
v_cmp_class_f32 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, F32 S1: src_simple, B32

VCC = IEEE numeric class function specified in S1.u, performed on S0.f

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.

v_cmp_class_f64 *vcc*[2], *src_0*[2], *src_1*
D0: vcc, B64 S0: src, F64 S1: vgpr, B32
v_cmp_class_f64 *sdst*[2], *src_0*[2], *src_1*
D0: sreg, B64 S0: src_nolit, F64 S1: src_simple, B32

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.

Flags: OPF_NODPP

v_cmp_eq_f16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F16 S1: vgpr, F16
v_cmp_eq_f16 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, F16 S1: src_simple, F16
v_cmp_eq_f32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F32 S1: vgpr, F32
v_cmp_eq_f32 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, F32 S1: src_simple, F32

```
D.u64[threadId] = (S0 == S1).
// D = VCC in VOPC encoding.
```

v_cmp_eq_f64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, F64 S1: vgpr, F64
v_cmp_eq_f64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sreg, B64 S0: src_nolit, F64 S1: src_simple, F64

```
D.u64[threadId] = (S0 == S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP

v_cmp_eq_i16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, I16 S1: vgpr, I16
v_cmp_eq_i16 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, I16 S1: src_simple, I16
v_cmp_eq_i32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, I32 S1: vgpr, I32
v_cmp_eq_i32 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, I32 S1: src_simple, I32

```
D.u64[threadId] = (S0 == S1).
// D = VCC in VOPC encoding.
```

v_cmp_eq_i64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, I64 S1: vgpr, I64
v_cmp_eq_i64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sreg, B64 S0: src_nolit, I64 S1: src_simple, I64

```
D.u64[threadId] = (S0 == S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP

v_cmp_eq_u16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, U16 S1: vgpr, U16
v_cmp_eq_u16 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, U16 S1: src_simple, U16
v_cmp_eq_u32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, U32 S1: vgpr, U32
v_cmp_eq_u32 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, U32 S1: src_simple, U32

```
D.u64[threadId] = (S0 == S1).
// D = VCC in VOPC encoding.
```

v_cmp_eq_u64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, U64 S1: vgpr, U64
v_cmp_eq_u64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sreg, B64 S0: src_nolit, U64 S1: src_simple, U64

```
D.u64[threadId] = (S0 == S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP

v_cmp_f_f16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F16 S1: vgpr, F16
v_cmp_f_f16 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, F16 S1: src_simple, F16
v_cmp_f_f32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F32 S1: vgpr, F32
v_cmp_f_f32 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, F32 S1: src_simple, F32

```
D.u64[threadId] = 0.
// D = VCC in VOPC encoding.
```

v_cmp_f_f64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, F64 S1: vgpr, F64
v_cmp_f_f64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sreg, B64 S0: src_nolit, F64 S1: src_simple, F64

```
D.u64[threadId] = 0.
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP

v_cmp_f_i16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, I16 S1: vgpr, I16
v_cmp_f_i16 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, I16 S1: src_simple, I16
v_cmp_f_i32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, I32 S1: vgpr, I32
v_cmp_f_i32 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, I32 S1: src_simple, I32

```
D.u64[threadId] = 0.
// D = VCC in VOPC encoding.
```

---

v_cmp_f_i64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, I64 S1: vgpr, I64
v_cmp_f_i64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sreg, B64 S0: src_nolit, I64 S1: src_simple, I64

```
D.u64[threadId] = 0.
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP

---

v_cmp_f_u16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, U16 S1: vgpr, U16
v_cmp_f_u16 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, U16 S1: src_simple, U16
v_cmp_f_u32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, U32 S1: vgpr, U32
v_cmp_f_u32 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, U32 S1: src_simple, U32

```
D.u64[threadId] = 0.
// D = VCC in VOPC encoding.
```

---

v_cmp_f_u64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, U64 S1: vgpr, U64
v_cmp_f_u64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sreg, B64 S0: src_nolit, U64 S1: src_simple, U64

```
D.u64[threadId] = 0.
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP

---

v_cmp_ge_f16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F16 S1: vgpr, F16
v_cmp_ge_f16 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, F16 S1: src_simple, F16
v_cmp_ge_f32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F32 S1: vgpr, F32
v_cmp_ge_f32 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, F32 S1: src_simple, F32

```
D.u64[threadId] = (S0 ≥ S1).
// D = VCC in VOPC encoding.
```

---

v_cmp_ge_f64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, F64 S1: vgpr, F64
v_cmp_ge_f64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sreg, B64 S0: src_nolit, F64 S1: src_simple, F64

```
D.u64[threadId] = (S0 ≥ S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP

v_cmp_ge_i16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, I16 S1: vgpr, I16
v_cmp_ge_i16 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, I16 S1: src_simple, I16
v_cmp_ge_i32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, I32 S1: vgpr, I32
v_cmp_ge_i32 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, I32 S1: src_simple, I32

```
D.u64[threadId] = (S0 ≥ S1).
// D = VCC in VOPC encoding.
```

v_cmp_ge_i64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, I64 S1: vgpr, I64
v_cmp_ge_i64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sreg, B64 S0: src_nolit, I64 S1: src_simple, I64

```
D.u64[threadId] = (S0 ≥ S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP

v_cmp_ge_u16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, U16 S1: vgpr, U16
v_cmp_ge_u16 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, U16 S1: src_simple, U16
v_cmp_ge_u32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, U32 S1: vgpr, U32
v_cmp_ge_u32 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, U32 S1: src_simple, U32

```
D.u64[threadId] = (S0 ≥ S1).
// D = VCC in VOPC encoding.
```

v_cmp_ge_u64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, U64 S1: vgpr, U64
v_cmp_ge_u64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sreg, B64 S0: src_nolit, U64 S1: src_simple, U64

```
D.u64[threadId] = (S0 ≥ S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP

v_cmp_gt_f16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F16 S1: vgpr, F16
v_cmp_gt_f16 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, F16 S1: src_simple, F16
v_cmp_gt_f32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F32 S1: vgpr, F32
v_cmp_gt_f32 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, F32 S1: src_simple, F32

```
D.u64[threadId] = (S0 > S1).
// D = VCC in VOPC encoding.
```

v_cmp_gt_f64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, F64 S1: vgpr, F64
v_cmp_gt_f64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sreg, B64 S0: src_nolit, F64 S1: src_simple, F64

```
D.u64[threadId] = (S0 > S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP

v_cmp_gt_i16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, I16 S1: vgpr, I16
v_cmp_gt_i16 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, I16 S1: src_simple, I16
v_cmp_gt_i32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, I32 S1: vgpr, I32
v_cmp_gt_i32 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, I32 S1: src_simple, I32

```
D.u64[threadId] = (S0 > S1).
// D = VCC in VOPC encoding.
```

v_cmp_gt_i64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, I64 S1: vgpr, I64
v_cmp_gt_i64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sreg, B64 S0: src_nolit, I64 S1: src_simple, I64

```
D.u64[threadId] = (S0 > S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP

v_cmp_gt_u16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, U16 S1: vgpr, U16
v_cmp_gt_u16 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, U16 S1: src_simple, U16
v_cmp_gt_u32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, U32 S1: vgpr, U32
v_cmp_gt_u32 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, U32 S1: src_simple, U32

```
D.u64[threadId] = (S0 > S1).
// D = VCC in VOPC encoding.
```

v_cmp_gt_u64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, U64 S1: vgpr, U64
v_cmp_gt_u64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sreg, B64 S0: src_nolit, U64 S1: src_simple, U64

```
D.u64[threadId] = (S0 > S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP

v_cmp_le_f16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F16 S1: vgpr, F16
v_cmp_le_f16 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, F16 S1: src_simple, F16
v_cmp_le_f32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F32 S1: vgpr, F32
v_cmp_le_f32 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, F32 S1: src_simple, F32

```
D.u64[threadId] = (S0 ≤ S1).
// D = VCC in VOPC encoding.
```

v_cmp_le_f64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, F64 S1: vgpr, F64
v_cmp_le_f64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sreg, B64 S0: src_nolit, F64 S1: src_simple, F64

```
D.u64[threadId] = (S0 ≤ S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP

v_cmp_le_i16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, I16 S1: vgpr, I16
v_cmp_le_i16 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, I16 S1: src_simple, I16
v_cmp_le_i32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, I32 S1: vgpr, I32
v_cmp_le_i32 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, I32 S1: src_simple, I32

```
D.u64[threadId] = (S0 ≤ S1).
// D = VCC in VOPC encoding.
```

v_cmp_le_i64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, I64 S1: vgpr, I64
v_cmp_le_i64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sreg, B64 S0: src_nolit, I64 S1: src_simple, I64

```
D.u64[threadId] = (S0 ≤ S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP

v_cmp_le_u16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, U16 S1: vgpr, U16
v_cmp_le_u16 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, U16 S1: src_simple, U16
v_cmp_le_u32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, U32 S1: vgpr, U32
v_cmp_le_u32 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, U32 S1: src_simple, U32

```
D.u64[threadId] = (S0 ≤ S1).
// D = VCC in VOPC encoding.
```

v_cmp_le_u64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, U64 S1: vgpr, U64
v_cmp_le_u64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sreg, B64 S0: src_nolit, U64 S1: src_simple, U64

```
D.u64[threadId] = (S0 ≤ S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP

v_cmp_lg_f16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F16 S1: vgpr, F16
v_cmp_lg_f16 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, F16 S1: src_simple, F16
v_cmp_lg_f32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F32 S1: vgpr, F32
v_cmp_lg_f32 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, F32 S1: src_simple, F32

```
D.u64[threadId] = (S0 <> S1).
// D = VCC in VOPC encoding.
```

v_cmp_lg_f64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, F64 S1: vgpr, F64
v_cmp_lg_f64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sreg, B64 S0: src_nolit, F64 S1: src_simple, F64

```
D.u64[threadId] = (S0 <> S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP

v_cmp_lt_f16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F16 S1: vgpr, F16
v_cmp_lt_f16 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, F16 S1: src_simple, F16
v_cmp_lt_f32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F32 S1: vgpr, F32
v_cmp_lt_f32 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, F32 S1: src_simple, F32

```
D.u64[threadId] = (S0 < S1).
// D = VCC in VOPC encoding.
```

v_cmp_lt_f64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, F64 S1: vgpr, F64
v_cmp_lt_f64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sreg, B64 S0: src_nolit, F64 S1: src_simple, F64

```
D.u64[threadId] = (S0 < S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP

v_cmp_lt_i16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, I16 S1: vgpr, I16
v_cmp_lt_i16 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, I16 S1: src_simple, I16
v_cmp_lt_i32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, I32 S1: vgpr, I32
v_cmp_lt_i32 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, I32 S1: src_simple, I32

```
D.u64[threadId] = (S0 < S1).
// D = VCC in VOPC encoding.
```

v_cmp_lt_i64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, I64 S1: vgpr, I64
v_cmp_lt_i64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sreg, B64 S0: src_nolit, I64 S1: src_simple, I64

```
D.u64[threadId] = (S0 < S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP

v_cmp_lt_u16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, U16 S1: vgpr, U16
v_cmp_lt_u16 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, U16 S1: src_simple, U16
v_cmp_lt_u32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, U32 S1: vgpr, U32
v_cmp_lt_u32 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, U32 S1: src_simple, U32

```
D.u64[threadId] = (S0 < S1).
// D = VCC in VOPC encoding.
```

v_cmp_lt_u64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, U64 S1: vgpr, U64
v_cmp_lt_u64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sreg, B64 S0: src_nolit, U64 S1: src_simple, U64

```
D.u64[threadId] = (S0 < S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP

v_cmp_ne_i16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, I16 S1: vgpr, I16
v_cmp_ne_i16 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, I16 S1: src_simple, I16
v_cmp_ne_i32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, I32 S1: vgpr, I32
v_cmp_ne_i32 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, I32 S1: src_simple, I32

```
D.u64[threadId] = (S0 <> S1).
// D = VCC in VOPC encoding.
```

v_cmp_ne_i64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, I64 S1: vgpr, I64
v_cmp_ne_i64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sreg, B64 S0: src_nolit, I64 S1: src_simple, I64

```
D.u64[threadId] = (S0 <> S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP

v_cmp_ne_u16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, U16 S1: vgpr, U16
v_cmp_ne_u16 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, U16 S1: src_simple, U16
v_cmp_ne_u32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, U32 S1: vgpr, U32
v_cmp_ne_u32 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, U32 S1: src_simple, U32

```
D.u64[threadId] = (S0 <> S1).
// D = VCC in VOPC encoding.
```

v_cmp_ne_u64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, U64 S1: vgpr, U64
v_cmp_ne_u64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sreg, B64 S0: src_nolit, U64 S1: src_simple, U64

```
D.u64[threadId] = (S0 <> S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP

v_cmp_neq_f16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F16 S1: vgpr, F16
v_cmp_neq_f16 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, F16 S1: src_simple, F16
v_cmp_neq_f32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F32 S1: vgpr, F32
v_cmp_neq_f32 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, F32 S1: src_simple, F32

```
D.u64[threadId] = !(S0 == S1) // With NAN inputs this is not the same operation as ≠.
// D = VCC in VOPC encoding.
```

v_cmp_neq_f64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, F64 S1: vgpr, F64
v_cmp_neq_f64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sreg, B64 S0: src_nolit, F64 S1: src_simple, F64

```
D.u64[threadId] = !(S0 == S1) // With NAN inputs this is not the same operation as ≠.
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP

v_cmp_nge_f16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F16 S1: vgpr, F16
v_cmp_nge_f16 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, F16 S1: src_simple, F16
v_cmp_nge_f32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F32 S1: vgpr, F32
v_cmp_nge_f32 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, F32 S1: src_simple, F32

```
D.u64[threadId] = !(S0 ≥ S1) // With NAN inputs this is not the same operation as <.
// D = VCC in VOPC encoding.
```

v_cmp_nge_f64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, F64 S1: vgpr, F64
v_cmp_nge_f64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sreg, B64 S0: src_nolit, F64 S1: src_simple, F64

```
D.u64[threadId] = !(S0 ≥ S1) // With NAN inputs this is not the same operation as <.
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP

v_cmp_ngt_f16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F16 S1: vgpr, F16
v_cmp_ngt_f16 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, F16 S1: src_simple, F16
v_cmp_ngt_f32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F32 S1: vgpr, F32
v_cmp_ngt_f32 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, F32 S1: src_simple, F32

```
D.u64[threadId] = !(S0 > S1) // With NAN inputs this is not the same operation as ≤.
// D = VCC in VOPC encoding.
```

v_cmp_ngt_f64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, F64 S1: vgpr, F64
v_cmp_ngt_f64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sreg, B64 S0: src_nolit, F64 S1: src_simple, F64

```
D.u64[threadId] = !(S0 > S1) // With NAN inputs this is not the same operation as ≤.
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP

v_cmp_nle_f16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F16 S1: vgpr, F16
v_cmp_nle_f16 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, F16 S1: src_simple, F16
v_cmp_nle_f32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F32 S1: vgpr, F32
v_cmp_nle_f32 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, F32 S1: src_simple, F32

```
D.u64[threadId] = !(S0 ≤ S1) // With NAN inputs this is not the same operation as >.
// D = VCC in VOPC encoding.
```

v_cmp_nle_f64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, F64 S1: vgpr, F64
v_cmp_nle_f64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sreg, B64 S0: src_nolit, F64 S1: src_simple, F64

```
D.u64[threadId] = !(S0 ≤ S1) // With NAN inputs this is not the same operation as >.
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP

v_cmp_nlg_f16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F16 S1: vgpr, F16
v_cmp_nlg_f16 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, F16 S1: src_simple, F16
v_cmp_nlg_f32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F32 S1: vgpr, F32
v_cmp_nlg_f32 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, F32 S1: src_simple, F32

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

v_cmp_nlg_f64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, F64 S1: vgpr, F64
v_cmp_nlg_f64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sreg, B64 S0: src_nolit, F64 S1: src_simple, F64

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

Flags: OPF_NODPP

v_cmp_nlt_f16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F16 S1: vgpr, F16
v_cmp_nlt_f16 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, F16 S1: src_simple, F16
v_cmp_nlt_f32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F32 S1: vgpr, F32
v_cmp_nlt_f32 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, F32 S1: src_simple, F32

```
D.u64[threadId] = !(S0 < S1) // With NAN inputs this is not the same operation as ≥.
// D = VCC in VOPC encoding.
```

v_cmp_nlt_f64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, F64 S1: vgpr, F64
v_cmp_nlt_f64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sreg, B64 S0: src_nolit, F64 S1: src_simple, F64

```
D.u64[threadId] = !(S0 < S1) // With NAN inputs this is not the same operation as ≥.
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP

v_cmp_o_f16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F16 S1: vgpr, F16
v_cmp_o_f16 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, F16 S1: src_simple, F16
v_cmp_o_f32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F32 S1: vgpr, F32
v_cmp_o_f32 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, F32 S1: src_simple, F32

```
D.u64[threadId] = (!isNan(S0) && !isNan(S1)).
// D = VCC in VOPC encoding.
```

v_cmp_o_f64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, F64 S1: vgpr, F64
v_cmp_o_f64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sreg, B64 S0: src_nolit, F64 S1: src_simple, F64

```
D.u64[threadId] = (!isNan(S0) && !isNan(S1)).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP

v_cmp_t_i16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, I16 S1: vgpr, I16
v_cmp_t_i16 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, I16 S1: src_simple, I16
v_cmp_t_i32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, I32 S1: vgpr, I32
v_cmp_t_i32 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, I32 S1: src_simple, I32

```
D.u64[threadId] = 1.
// D = VCC in VOPC encoding.
```

v_cmp_t_i64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, I64 S1: vgpr, I64
v_cmp_t_i64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sreg, B64 S0: src_nolit, I64 S1: src_simple, I64

```
D.u64[threadId] = 1.
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP

v_cmp_t_u16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, U16 S1: vgpr, U16
v_cmp_t_u16 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, U16 S1: src_simple, U16
v_cmp_t_u32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, U32 S1: vgpr, U32
v_cmp_t_u32 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, U32 S1: src_simple, U32

```
D.u64[threadId] = 1.
// D = VCC in VOPC encoding.
```

v_cmp_t_u64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, U64 S1: vgpr, U64
v_cmp_t_u64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sreg, B64 S0: src_nolit, U64 S1: src_simple, U64

```
D.u64[threadId] = 1.
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP

v_cmp_tru_f16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F16 S1: vgpr, F16
v_cmp_tru_f16 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, F16 S1: src_simple, F16
v_cmp_tru_f32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F32 S1: vgpr, F32
v_cmp_tru_f32 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, F32 S1: src_simple, F32

```
D.u64[threadId] = 1.
// D = VCC in VOPC encoding.
```

v_cmp_tru_f64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, F64 S1: vgpr, F64
v_cmp_tru_f64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sreg, B64 S0: src_nolit, F64 S1: src_simple, F64

```
D.u64[threadId] = 1.
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP

v_cmp_u_f16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F16 S1: vgpr, F16
v_cmp_u_f16 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, F16 S1: src_simple, F16
v_cmp_u_f32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F32 S1: vgpr, F32
v_cmp_u_f32 *sdst*[2], *src_0*, *src_1*
D0: sreg, B64 S0: src_nolit, F32 S1: src_simple, F32

```
D.u64[threadId] = (isNan(S0) || isNan(S1)).
// D = VCC in VOPC encoding.
```

v_cmp_u_f64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, F64 S1: vgpr, F64
v_cmp_u_f64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sreg, B64 S0: src_nolit, F64 S1: src_simple, F64

```
D.u64[threadId] = (isNan(S0) || isNan(S1)).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP

v_cmpx_class_f16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F16 S1: vgpr, F16
v_cmpx_class_f16 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, F16 S1: src_simple, F16

EXEC = 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.

Flags: OPF_SDST, OPF_WREX

v_cmpx_class_f32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F32 S1: vgpr, B32
v_cmpx_class_f32 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, F32 S1: src_simple, B32

EXEC = VCC = IEEE numeric class function specified in S1.u, performed on S0.f

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.

Flags: OPF_SDST, OPF_WREX

v_cmpx_class_f64 *vcc*[2], *src_0*[2], *src_1*
D0: vcc, B64 S0: src, F64 S1: vgpr, B32
v_cmpx_class_f64 *sdst*[2], *src_0*[2], *src_1*
D0: sdst, B64 S0: src_nolit, F64 S1: src_simple, B32

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.

Flags: OPF_NODPP, OPF_SDST, OPF_WREX

v_cmpx_eq_f16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F16 S1: vgpr, F16
v_cmpx_eq_f16 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, F16 S1: src_simple, F16
v_cmpx_eq_f32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F32 S1: vgpr, F32
v_cmpx_eq_f32 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, F32 S1: src_simple, F32

```
EXEC[threadId] = D.u64[threadId] = (S0 == S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_SDST, OPF_WREX

v_cmpx_eq_f64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, F64 S1: vgpr, F64
v_cmpx_eq_f64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sdst, B64 S0: src_nolit, F64 S1: src_simple, F64

```
EXEC[threadId] = D.u64[threadId] = (S0 == S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP, OPF_SDST, OPF_WREX

v_cmpx_eq_i16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, I16 S1: vgpr, I16
v_cmpx_eq_i16 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, I16 S1: src_simple, I16
v_cmpx_eq_i32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, I32 S1: vgpr, I32
v_cmpx_eq_i32 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, I32 S1: src_simple, I32

```
EXEC[threadId] = D.u64[threadId] = (S0 == S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_SDST, OPF_WREX

v_cmpx_eq_i64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, I64 S1: vgpr, I64
v_cmpx_eq_i64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sdst, B64 S0: src_nolit, I64 S1: src_simple, I64

```
EXEC[threadId] = D.u64[threadId] = (S0 == S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP, OPF_SDST, OPF_WREX

v_cmpx_eq_u16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, U16 S1: vgpr, U16
v_cmpx_eq_u16 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, U16 S1: src_simple, U16
v_cmpx_eq_u32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, U32 S1: vgpr, U32
v_cmpx_eq_u32 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, U32 S1: src_simple, U32

```
EXEC[threadId] = D.u64[threadId] = (S0 == S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_SDST, OPF_WREX

v_cmpx_eq_u64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, U64 S1: vgpr, U64
v_cmpx_eq_u64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sdst, B64 S0: src_nolit, U64 S1: src_simple, U64

```
EXEC[threadId] = D.u64[threadId] = (S0 == S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP, OPF_SDST, OPF_WREX

v_cmpx_f_f16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F16 S1: vgpr, F16
v_cmpx_f_f16 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, F16 S1: src_simple, F16
v_cmpx_f_f32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F32 S1: vgpr, F32
v_cmpx_f_f32 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, F32 S1: src_simple, F32

```
EXEC[threadId] = D.u64[threadId] = 0.
// D = VCC in VOPC encoding.
```

Flags: OPF_SDST, OPF_WREX

v_cmpx_f_f64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, F64 S1: vgpr, F64
v_cmpx_f_f64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sdst, B64 S0: src_nolit, F64 S1: src_simple, F64

```
EXEC[threadId] = D.u64[threadId] = 0.
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP, OPF_SDST, OPF_WREX

v_cmpx_f_i16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, I16 S1: vgpr, I16
v_cmpx_f_i16 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, I16 S1: src_simple, I16
v_cmpx_f_i32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, I32 S1: vgpr, I32
v_cmpx_f_i32 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, I32 S1: src_simple, I32

```
EXEC[threadId] = D.u64[threadId] = 0.
// D = VCC in VOPC encoding.
```

Flags: OPF_SDST, OPF_WREX

v_cmpx_f_i64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, I64 S1: vgpr, I64
v_cmpx_f_i64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sdst, B64 S0: src_nolit, I64 S1: src_simple, I64

```
EXEC[threadId] = D.u64[threadId] = 0.
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP, OPF_SDST, OPF_WREX

v_cmpx_f_u16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, U16 S1: vgpr, U16
v_cmpx_f_u16 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, U16 S1: src_simple, U16
v_cmpx_f_u32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, U32 S1: vgpr, U32
v_cmpx_f_u32 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, U32 S1: src_simple, U32

```
EXEC[threadId] = D.u64[threadId] = 0.
// D = VCC in VOPC encoding.
```

Flags: OPF_SDST, OPF_WREX

v_cmpx_f_u64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, U64 S1: vgpr, U64
v_cmpx_f_u64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sdst, B64 S0: src_nolit, U64 S1: src_simple, U64

```
EXEC[threadId] = D.u64[threadId] = 0.
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP, OPF_SDST, OPF_WREX

v_cmpx_ge_f16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F16 S1: vgpr, F16
v_cmpx_ge_f16 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, F16 S1: src_simple, F16
v_cmpx_ge_f32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F32 S1: vgpr, F32
v_cmpx_ge_f32 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, F32 S1: src_simple, F32

```
EXEC[threadId] = D.u64[threadId] = (S0 ≥ S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_SDST, OPF_WREX

v_cmpx_ge_f64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, F64 S1: vgpr, F64
v_cmpx_ge_f64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sdst, B64 S0: src_nolit, F64 S1: src_simple, F64

```
EXEC[threadId] = D.u64[threadId] = (S0 ≥ S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP, OPF_SDST, OPF_WREX

v_cmpx_ge_i16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, I16 S1: vgpr, I16
v_cmpx_ge_i16 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, I16 S1: src_simple, I16
v_cmpx_ge_i32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, I32 S1: vgpr, I32
v_cmpx_ge_i32 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, I32 S1: src_simple, I32

```
EXEC[threadId] = D.u64[threadId] = (S0 ≥ S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_SDST, OPF_WREX

v_cmpx_ge_i64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, I64 S1: vgpr, I64
v_cmpx_ge_i64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sdst, B64 S0: src_nolit, I64 S1: src_simple, I64

```
EXEC[threadId] = D.u64[threadId] = (S0 ≥ S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP, OPF_SDST, OPF_WREX

v_cmpx_ge_u16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, U16 S1: vgpr, U16
v_cmpx_ge_u16 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, U16 S1: src_simple, U16
v_cmpx_ge_u32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, U32 S1: vgpr, U32
v_cmpx_ge_u32 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, U32 S1: src_simple, U32

```
EXEC[threadId] = D.u64[threadId] = (S0 ≥ S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_SDST, OPF_WREX

v_cmpx_ge_u64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, U64 S1: vgpr, U64
v_cmpx_ge_u64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sdst, B64 S0: src_nolit, U64 S1: src_simple, U64

```
EXEC[threadId] = D.u64[threadId] = (S0 ≥ S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP, OPF_SDST, OPF_WREX

v_cmpx_gt_f16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F16 S1: vgpr, F16
v_cmpx_gt_f16 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, F16 S1: src_simple, F16
v_cmpx_gt_f32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F32 S1: vgpr, F32
v_cmpx_gt_f32 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, F32 S1: src_simple, F32

```
EXEC[threadId] = D.u64[threadId] = (S0 > S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_SDST, OPF_WREX

v_cmpx_gt_f64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, F64 S1: vgpr, F64
v_cmpx_gt_f64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sdst, B64 S0: src_nolit, F64 S1: src_simple, F64

```
EXEC[threadId] = D.u64[threadId] = (S0 > S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP, OPF_SDST, OPF_WREX

v_cmpx_gt_i16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, I16 S1: vgpr, I16
v_cmpx_gt_i16 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, I16 S1: src_simple, I16
v_cmpx_gt_i32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, I32 S1: vgpr, I32
v_cmpx_gt_i32 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, I32 S1: src_simple, I32

```
EXEC[threadId] = D.u64[threadId] = (S0 > S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_SDST, OPF_WREX

v_cmpx_gt_i64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, I64 S1: vgpr, I64
v_cmpx_gt_i64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sdst, B64 S0: src_nolit, I64 S1: src_simple, I64

```
EXEC[threadId] = D.u64[threadId] = (S0 > S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP, OPF_SDST, OPF_WREX

v_cmpx_gt_u16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, U16 S1: vgpr, U16
v_cmpx_gt_u16 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, U16 S1: src_simple, U16
v_cmpx_gt_u32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, U32 S1: vgpr, U32
v_cmpx_gt_u32 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, U32 S1: src_simple, U32

```
EXEC[threadId] = D.u64[threadId] = (S0 > S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_SDST, OPF_WREX

v_cmpx_gt_u64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, U64 S1: vgpr, U64
v_cmpx_gt_u64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sdst, B64 S0: src_nolit, U64 S1: src_simple, U64

```
EXEC[threadId] = D.u64[threadId] = (S0 > S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP, OPF_SDST, OPF_WREX

v_cmpx_le_f16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F16 S1: vgpr, F16
v_cmpx_le_f16 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, F16 S1: src_simple, F16
v_cmpx_le_f32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F32 S1: vgpr, F32
v_cmpx_le_f32 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, F32 S1: src_simple, F32

```
EXEC[threadId] = D.u64[threadId] = (S0 ≤ S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_SDST, OPF_WREX

v_cmpx_le_f64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, F64 S1: vgpr, F64
v_cmpx_le_f64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sdst, B64 S0: src_nolit, F64 S1: src_simple, F64

```
EXEC[threadId] = D.u64[threadId] = (S0 ≤ S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP, OPF_SDST, OPF_WREX

v_cmpx_le_i16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, I16 S1: vgpr, I16
v_cmpx_le_i16 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, I16 S1: src_simple, I16
v_cmpx_le_i32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, I32 S1: vgpr, I32
v_cmpx_le_i32 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, I32 S1: src_simple, I32

```
EXEC[threadId] = D.u64[threadId] = (S0 ≤ S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_SDST, OPF_WREX

v_cmpx_le_i64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, I64 S1: vgpr, I64
v_cmpx_le_i64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sdst, B64 S0: src_nolit, I64 S1: src_simple, I64

```
EXEC[threadId] = D.u64[threadId] = (S0 ≤ S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP, OPF_SDST, OPF_WREX

v_cmpx_le_u16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, U16 S1: vgpr, U16
v_cmpx_le_u16 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, U16 S1: src_simple, U16
v_cmpx_le_u32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, U32 S1: vgpr, U32
v_cmpx_le_u32 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, U32 S1: src_simple, U32

```
EXEC[threadId] = D.u64[threadId] = (S0 ≤ S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_SDST, OPF_WREX

v_cmpx_le_u64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, U64 S1: vgpr, U64
v_cmpx_le_u64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sdst, B64 S0: src_nolit, U64 S1: src_simple, U64

```
EXEC[threadId] = D.u64[threadId] = (S0 ≤ S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP, OPF_SDST, OPF_WREX

v_cmpx_lg_f16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F16 S1: vgpr, F16
v_cmpx_lg_f16 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, F16 S1: src_simple, F16
v_cmpx_lg_f32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F32 S1: vgpr, F32
v_cmpx_lg_f32 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, F32 S1: src_simple, F32

```
EXEC[threadId] = D.u64[threadId] = (S0 <> S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_SDST, OPF_WREX

v_cmpx_lg_f64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, F64 S1: vgpr, F64
v_cmpx_lg_f64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sdst, B64 S0: src_nolit, F64 S1: src_simple, F64

```
EXEC[threadId] = D.u64[threadId] = (S0 <> S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP, OPF_SDST, OPF_WREX

v_cmpx_lt_f16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F16 S1: vgpr, F16
v_cmpx_lt_f16 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, F16 S1: src_simple, F16
v_cmpx_lt_f32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F32 S1: vgpr, F32
v_cmpx_lt_f32 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, F32 S1: src_simple, F32

```
EXEC[threadId] = D.u64[threadId] = (S0 < S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_SDST, OPF_WREX

v_cmpx_lt_f64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, F64 S1: vgpr, F64
v_cmpx_lt_f64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sdst, B64 S0: src_nolit, F64 S1: src_simple, F64

```
EXEC[threadId] = D.u64[threadId] = (S0 < S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP, OPF_SDST, OPF_WREX

v_cmpx_lt_i16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, I16 S1: vgpr, I16
v_cmpx_lt_i16 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, I16 S1: src_simple, I16
v_cmpx_lt_i32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, I32 S1: vgpr, I32
v_cmpx_lt_i32 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, I32 S1: src_simple, I32

```
EXEC[threadId] = D.u64[threadId] = (S0 < S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_SDST, OPF_WREX

v_cmpx_lt_i64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, I64 S1: vgpr, I64
v_cmpx_lt_i64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sdst, B64 S0: src_nolit, I64 S1: src_simple, I64

```
EXEC[threadId] = D.u64[threadId] = (S0 < S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP, OPF_SDST, OPF_WREX

v_cmpx_lt_u16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, U16 S1: vgpr, U16
v_cmpx_lt_u16 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, U16 S1: src_simple, U16
v_cmpx_lt_u32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, U32 S1: vgpr, U32
v_cmpx_lt_u32 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, U32 S1: src_simple, U32

```
EXEC[threadId] = D.u64[threadId] = (S0 < S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_SDST, OPF_WREX

v_cmpx_lt_u64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, U64 S1: vgpr, U64
v_cmpx_lt_u64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sdst, B64 S0: src_nolit, U64 S1: src_simple, U64

```
EXEC[threadId] = D.u64[threadId] = (S0 < S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP, OPF_SDST, OPF_WREX

v_cmpx_ne_i16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, I16 S1: vgpr, I16
v_cmpx_ne_i16 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, I16 S1: src_simple, I16
v_cmpx_ne_i32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, I32 S1: vgpr, I32
v_cmpx_ne_i32 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, I32 S1: src_simple, I32

```
EXEC[threadId] = D.u64[threadId] = (S0 <> S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_SDST, OPF_WREX

v_cmpx_ne_i64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, I64 S1: vgpr, I64
v_cmpx_ne_i64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sdst, B64 S0: src_nolit, I64 S1: src_simple, I64

```
EXEC[threadId] = D.u64[threadId] = (S0 <> S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP, OPF_SDST, OPF_WREX

v_cmpx_ne_u16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, U16 S1: vgpr, U16
v_cmpx_ne_u16 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, U16 S1: src_simple, U16
v_cmpx_ne_u32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, U32 S1: vgpr, U32
v_cmpx_ne_u32 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, U32 S1: src_simple, U32

```
EXEC[threadId] = D.u64[threadId] = (S0 <> S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_SDST, OPF_WREX

v_cmpx_ne_u64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, U64 S1: vgpr, U64
v_cmpx_ne_u64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sdst, B64 S0: src_nolit, U64 S1: src_simple, U64

```
EXEC[threadId] = D.u64[threadId] = (S0 <> S1).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP, OPF_SDST, OPF_WREX

v_cmpx_neq_f16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F16 S1: vgpr, F16
v_cmpx_neq_f16 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, F16 S1: src_simple, F16
v_cmpx_neq_f32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F32 S1: vgpr, F32
v_cmpx_neq_f32 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, F32 S1: src_simple, F32

```
EXEC[threadId] = D.u64[threadId] = !(S0 == S1) // With NAN inputs this is not the same operation as ≠.
// D = VCC in VOPC encoding.
```

Flags: OPF_SDST, OPF_WREX

v_cmpx_neq_f64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, F64 S1: vgpr, F64
v_cmpx_neq_f64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sdst, B64 S0: src_nolit, F64 S1: src_simple, F64

```
EXEC[threadId] = D.u64[threadId] = !(S0 == S1) // With NAN inputs this is not the same operation as ≠.
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP, OPF_SDST, OPF_WREX

v_cmpx_nge_f16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F16 S1: vgpr, F16
v_cmpx_nge_f16 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, F16 S1: src_simple, F16
v_cmpx_nge_f32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F32 S1: vgpr, F32
v_cmpx_nge_f32 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, F32 S1: src_simple, F32

```
EXEC[threadId] = D.u64[threadId] = !(S0 ≥ S1) // With NAN inputs this is not the same operation as <.
// D = VCC in VOPC encoding.
```

Flags: OPF_SDST, OPF_WREX

v_cmpx_nge_f64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, F64 S1: vgpr, F64
v_cmpx_nge_f64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sdst, B64 S0: src_nolit, F64 S1: src_simple, F64

```
EXEC[threadId] = D.u64[threadId] = !(S0 ≥ S1) // With NAN inputs this is not the same operation as <.
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP, OPF_SDST, OPF_WREX

v_cmpx_ngt_f16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F16 S1: vgpr, F16
v_cmpx_ngt_f16 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, F16 S1: src_simple, F16
v_cmpx_ngt_f32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F32 S1: vgpr, F32
v_cmpx_ngt_f32 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, F32 S1: src_simple, F32

```
EXEC[threadId] = D.u64[threadId] = !(S0 > S1) // With NAN inputs this is not the same operation as ≤.
// D = VCC in VOPC encoding.
```

Flags: OPF_SDST, OPF_WREX

v_cmpx_ngt_f64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, F64 S1: vgpr, F64
v_cmpx_ngt_f64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sdst, B64 S0: src_nolit, F64 S1: src_simple, F64

```
EXEC[threadId] = D.u64[threadId] = !(S0 > S1) // With NAN inputs this is not the same operation as ≤.
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP, OPF_SDST, OPF_WREX

v_cmpx_nle_f16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F16 S1: vgpr, F16
v_cmpx_nle_f16 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, F16 S1: src_simple, F16
v_cmpx_nle_f32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F32 S1: vgpr, F32
v_cmpx_nle_f32 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, F32 S1: src_simple, F32

```
EXEC[threadId] = D.u64[threadId] = !(S0 ≤ S1) // With NAN inputs this is not the same operation as >.
// D = VCC in VOPC encoding.
```

Flags: OPF_SDST, OPF_WREX

v_cmpx_nle_f64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, F64 S1: vgpr, F64
v_cmpx_nle_f64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sdst, B64 S0: src_nolit, F64 S1: src_simple, F64

```
EXEC[threadId] = D.u64[threadId] = !(S0 ≤ S1) // With NAN inputs this is not the same operation as >.
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP, OPF_SDST, OPF_WREX

v_cmpx_nlg_f16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F16 S1: vgpr, F16
v_cmpx_nlg_f16 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, F16 S1: src_simple, F16
v_cmpx_nlg_f32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F32 S1: vgpr, F32
v_cmpx_nlg_f32 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, F32 S1: src_simple, F32

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

Flags: OPF_SDST, OPF_WREX

v_cmpx_nlg_f64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, F64 S1: vgpr, F64
v_cmpx_nlg_f64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sdst, B64 S0: src_nolit, F64 S1: src_simple, F64

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

Flags: OPF_NODPP, OPF_SDST, OPF_WREX

v_cmpx_nlt_f16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F16 S1: vgpr, F16
v_cmpx_nlt_f16 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, F16 S1: src_simple, F16
v_cmpx_nlt_f32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F32 S1: vgpr, F32
v_cmpx_nlt_f32 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, F32 S1: src_simple, F32

```
EXEC[threadId] = D.u64[threadId] = !(S0 < S1) // With NAN inputs this is not the same operation as ≥.
// D = VCC in VOPC encoding.
```

Flags: OPF_SDST, OPF_WREX

v_cmpx_nlt_f64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, F64 S1: vgpr, F64
v_cmpx_nlt_f64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sdst, B64 S0: src_nolit, F64 S1: src_simple, F64

```
EXEC[threadId] = D.u64[threadId] = !(S0 < S1) // With NAN inputs this is not the same operation as ≥.
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP, OPF_SDST, OPF_WREX

v_cmpx_o_f16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F16 S1: vgpr, F16
v_cmpx_o_f16 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, F16 S1: src_simple, F16
v_cmpx_o_f32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F32 S1: vgpr, F32
v_cmpx_o_f32 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, F32 S1: src_simple, F32

```
EXEC[threadId] = D.u64[threadId] = (!isNan(S0) && !isNan(S1)).
// D = VCC in VOPC encoding.
```

Flags: OPF_SDST, OPF_WREX

v_cmpx_o_f64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, F64 S1: vgpr, F64
v_cmpx_o_f64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sdst, B64 S0: src_nolit, F64 S1: src_simple, F64

```
EXEC[threadId] = D.u64[threadId] = (!isNan(S0) && !isNan(S1)).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP, OPF_SDST, OPF_WREX

v_cmpx_t_i16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, I16 S1: vgpr, I16
v_cmpx_t_i16 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, I16 S1: src_simple, I16
v_cmpx_t_i32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, I32 S1: vgpr, I32
v_cmpx_t_i32 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, I32 S1: src_simple, I32

```
EXEC[threadId] = D.u64[threadId] = 1.
// D = VCC in VOPC encoding.
```

Flags: OPF_SDST, OPF_WREX

v_cmpx_t_i64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, I64 S1: vgpr, I64
v_cmpx_t_i64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sdst, B64 S0: src_nolit, I64 S1: src_simple, I64

```
EXEC[threadId] = D.u64[threadId] = 1.
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP, OPF_SDST, OPF_WREX

v_cmpx_t_u16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, U16 S1: vgpr, U16
v_cmpx_t_u16 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, U16 S1: src_simple, U16
v_cmpx_t_u32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, U32 S1: vgpr, U32
v_cmpx_t_u32 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, U32 S1: src_simple, U32

```
EXEC[threadId] = D.u64[threadId] = 1.
// D = VCC in VOPC encoding.
```

Flags: OPF_SDST, OPF_WREX

v_cmpx_t_u64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, U64 S1: vgpr, U64
v_cmpx_t_u64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sdst, B64 S0: src_nolit, U64 S1: src_simple, U64

```
EXEC[threadId] = D.u64[threadId] = 1.
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP, OPF_SDST, OPF_WREX

v_cmpx_tru_f16 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F16 S1: vgpr, F16
v_cmpx_tru_f16 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, F16 S1: src_simple, F16
v_cmpx_tru_f32 *vcc*[2], *src_0*, *src_1*
D0: vcc, B64 S0: src, F32 S1: vgpr, F32
v_cmpx_tru_f32 *sdst*[2], *src_0*, *src_1*
D0: sdst, B64 S0: src_nolit, F32 S1: src_simple, F32

```
EXEC[threadId] = D.u64[threadId] = 1.
// D = VCC in VOPC encoding.
```

Flags: OPF_SDST, OPF_WREX

v_cmpx_tru_f64 *vcc*[2], *src_0*[2], *src_1*[2]
D0: vcc, B64 S0: src, F64 S1: vgpr, F64
v_cmpx_tru_f64 *sdst*[2], *src_0*[2], *src_1*[2]
D0: sdst, B64 S0: src_nolit, F64 S1: src_simple, F64

```
EXEC[threadId] = D.u64[threadId] = 1.
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP, OPF_SDST, OPF_WREX

v_cmpx_u_f16	*vcc*[2],	*src_0*,	*src_1*
	D0: vcc, B64	S0: src, F16	S1: vgpr, F16
v_cmpx_u_f16	*sdst*[2],	*src_0*,	*src_1*
	D0: sdst, B64	S0: src_nolit, F16	S1: src_simple, F16
v_cmpx_u_f32	*vcc*[2],	*src_0*,	*src_1*
	D0: vcc, B64	S0: src, F32	S1: vgpr, F32
v_cmpx_u_f32	*sdst*[2],	*src_0*,	*src_1*
	D0: sdst, B64	S0: src_nolit, F32	S1: src_simple, F32

```
EXEC[threadId] = D.u64[threadId] = (isNan(S0) || isNan(S1)).
// D = VCC in VOPC encoding.
```

Flags: OPF_SDST, OPF_WREX

v_cmpx_u_f64	*vcc*[2],	*src_0*[2],	*src_1*[2]
	D0: vcc, B64	S0: src, F64	S1: vgpr, F64
v_cmpx_u_f64	*sdst*[2],	*src_0*[2],	*src_1*[2]
	D0: sdst, B64	S0: src_nolit, F64	S1: src_simple, F64

```
EXEC[threadId] = D.u64[threadId] = (isNan(S0) || isNan(S1)).
// D = VCC in VOPC encoding.
```

Flags: OPF_NODPP, OPF_SDST, OPF_WREX

9.1 Notes for Encoding VOPC

9.1.1 Input modifiers

Source operands may invoke the negation and absolute-value input modifiers with `-`*src* and `abs(`*src*`)`, respectively. For example,

```
v_add_f32 v0, v1, -v2           // Subtract v2 from v1
v_add_f32 v0, abs(v1), abs(v2)  // Take absolute value of both inputs
```

In general, negation and absolute value are only supported for floating point input operands (operands with a type of F16, F32 or F64); they are not supported for integer or untyped inputs.

9.1.2 Output modifiers

`mul:{1,2,4}`
: Set output modifier to multiply by N. Default 1, which leaves the result unmodified. This operation is only defined when the result is F16, F32 or F64.

`div:{1,2}`
: Set output modifier to divide by N. Default 1, which leaves the result unmodified. This operation is only defined when the result is F16, F32 or F64.

`clamp:{0,1}`
: Clamp (saturate) the output. Default 0. Can also write `noclamp`. Saturation is defined as follows.

  For I16, I32, I64 results: if the result exceeds SHRT_MAX/INT_MAX/LLONG_MAX it will be clamped to the most positive representable value; if the result is below SHRT_MIN/INT_MIN/LLONG_MIN it will be clamped to the most negative representable value.

  For U16, U32, U64 results: if the result exceeds USHRT_MAX/UINT_MAX/ULLONG_MAX it will be clamped to the most positive representable value; if the result is below 0 it will be clamped to 0.

  For F16, F32, F64 results: the result will be clamped to the interval [0.0, 1.0].

9.1.3 Interpolation operands and modifiers

vgpr_dst is a vector GPR to store result in, and use as accumulator source in certain interpolation operations.

vgpr_ij is a vector GPR to read i/j value from.

attr is `attr0.x` through `attr63.w`, parameter attribute and channel to be interpolated.

param is `p10`, `p20` or `p0`.

For 16-bit interpolation it is necessary to specify whether we are operating on the high or low word of the attribute. For this, two new modifiers are provided:

`high`
: Interpolate using the high 16 bits of the attribute.

`low`
: Interpolate using the low 16 bits of the attribute. Default.

9.1.4 Other modifiers

`vop3:{0,1}`
: Force VOP3 encoding even if instruction can be represented in smaller encoding. Default 0. Can also write `novop3`. Note that even if this modifier is not set, an opcode will still use the VOP3 encoding if the operands or modifiers given cannot be expressed in a smaller encoding.

9.1.5 SDWA output modifiers

This only applies to VOP1/VOP2/VOPC encodings.

The SDWA subencoding supports sign-extension of inputs; this may be written as `sext(`*src*`)`, similar to how the `abs` modifier is used.

`src0_sel:`*sdwa_sel*
: Apply to the selected sdwa data bits of src0. Default DWORD.

`src1_sel:`*sdwa_sel*
Apply to the selected sdwa data bits of src1. Default DWORD.

`dst_sel:`*sdwa_sel*
Apply to the selected sdwa data bits of dst. Default DWORD.

`dst_unused:`*sdwa_unused*
Determine what to do with the unused dst bits. Default UNUSED_PAD.

9.1.6 DPP output modifiers

This only applies to VOP1/VOP2/VOPC encodings.

General modifiers Any combination of these modifiers may be applied for DPP instructions.

`bank_mask:[0. . . 0xf]`
Apply to the selected bank mask bits.

`row_mask:[0. . . 0xf]`
Apply to the selected row mask bits.

`bound_ctrl:{0,1}`
If true, then writes to out-of-bounds threads are written as zero. If false, writes to out-of-bounds threads are disabled.

DPP permutation control Only one of the following modifiers may be applied to the instruction.

Each wave of 64 threads $(W_0, \ldots, W_{63})$ is divided into 4 *rows* of 16 threads, $(R_0, \ldots, R_{15})$ and into 16 *quads* of 4 threads (usually pixels), $(P_0, \ldots, P_3)$.

`quad_perm:[pix0,pix1,pix2,pix3]`

DPP permutation: Applied for each pixel within a quad. The `N`th entry in the array specifies that the `N`th pixel in the output will obtain its data from the `quad_perm[N]`th pixel in the input.

The identity transformation is represented as `quad_perm:[0,1,2,3]`.

`quad_perm:[1,2,0,1]` applies the following permutation which includes broadcasting one of the pixels to multiple destinations:

$$\begin{bmatrix} P_0 & P_1 \\ P_2 & P_3 \end{bmatrix} : \qquad \begin{bmatrix} A & B \\ C & D \end{bmatrix} \Rightarrow \begin{bmatrix} B & C \\ A & B \end{bmatrix}$$

To broadcast one pixel (e.g. pixel 0) to all destinations in the quad, use `quad_perm:[0,0,0,0]`:

$$\begin{bmatrix} P_0 & P_1 \\ P_2 & P_3 \end{bmatrix} : \qquad \begin{bmatrix} A & B \\ C & D \end{bmatrix} \Rightarrow \begin{bmatrix} A & A \\ A & A \end{bmatrix}$$

In general the output thread P' is determined by

$$P'_n = P_{\texttt{quad\_perm}[n]}$$

`row_shr:[1. . . 15]`

DPP permutation: Shifts threads in each row to the right by the specified amount.

In general the output thread R' with count C is determined by

$$R'_n = \begin{cases} R_{n-C} & n \geq C \\ \texttt{bound\_ctrl} & \text{otherwise} \end{cases}$$

`row_shl:[1. . . 15]`

DPP permutation: Shifts threads in each row to the left by the specified amount.

In general the output thread R' with count C is determined by

$$R'_n = \begin{cases} R_{n+C} & n + C < 16 \\ \texttt{bound\_ctrl} & \text{otherwise} \end{cases}$$

`row_ror:[1. . . 15]`

DPP permutation: Rotates threads in each row to the right by the specified amount.

In general the output thread R' with count C is determined by

$$R'_n = \begin{cases} R_{n-C} & n \geq C \\ R_{n-C+16} & \text{otherwise} \end{cases}$$

`wave_shr`
DPP permutation: Shifts threads in the entire wave to the right by one.

In general the output thread W' is determined by

$$W'_n = \begin{cases} W_{n-1} & n > 0 \\ \texttt{bound\_ctrl} & \text{otherwise} \end{cases}$$

`wave_shl`
DPP permutation: Shifts threads in the entire wave to the left by one.

In general the output thread W' is determined by

$$W'_n = \begin{cases} W_{n+1} & n < 63 \\ \texttt{bound\_ctrl} & \text{otherwise} \end{cases}$$

`wave_ror`
DPP permutation: Rotates threads in the entire wave to the right by one.

In general the output thread W' is determined by

$$W'_n = \begin{cases} W_{n-1} & n > 0 \\ W_{63} & \text{otherwise} \end{cases}$$

`wave_rol`
DPP permutation: Rotates threads in the entire wave to the left by one.

In general the output thread W' is determined by

$$W'_n = \begin{cases} W_{n+1} & n < 63 \\ W_0 & \text{otherwise} \end{cases}$$

`row_mirror`
DPP permutation: Mirrors threads within each row.

In general the output thread R' is determined by

$$R'_n = R_{15-n}$$

`row_half_mirror`
: DPP permutation: Mirrors threads within each half-row.

  In general the output thread R' is determined by

$$R'_n = \begin{cases} R_{7-n} & n < 8 \\ R_{23-n} & \text{otherwise} \end{cases}$$

`row_bcast:{15, 31}`
: DPP permutation: Broadcast a thread to subsequent rows.

10 Encoding VOP2

Vector ALU operations with one destination and two sources. Instructions in this encoding may be promoted to VOP3 unless otherwise noted.

v_add_co_u32	*vdst*,	*vcc*[2],	*src_0*,	*src_1*
	D0: vgpr, U32	D1: vcc, B64	S0: src, U32	S1: vgpr, U32
v_add_co_u32	*vdst*,	*carryout*[2],	*src_0*,	*src_1*
	D0: vgpr, U32	D1: sreg, B64	S0: src_nolit, U32	S1: src_simple, 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.

Flags: OPF_CACGRP2, OPF_VCCD

v_add_f16	*vdst*,	*src_0*,	*src_1*
	D0: vgpr, F16	S0: src, F16	S1: vgpr, F16
v_add_f16	*vdst*,	*src_0*,	*src_1*
	D0: vgpr, F16	S0: src_nolit, F16	S1: src_simple, F16

```
D.f16 = S0.f16 + S1.f16.
```

Supports denormals, round mode, exception flags, saturation. 0.5ULP precision, denormals are supported.

v_add_f32	*vdst*,	*src_0*,	*src_1*
	D0: vgpr, F32	S0: src, F32	S1: vgpr, F32
v_add_f32	*vdst*,	*src_0*,	*src_1*
	D0: vgpr, F32	S0: src_nolit, F32	S1: src_simple, F32

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

0.5ULP precision, denormals are supported.

Flags: OPF_CACGRP1

v_add_u16	*vdst*,	*src_0*,	*src_1*
	D0: vgpr, U16	S0: src, U16	S1: vgpr, U16
v_add_u16	*vdst*,	*src_0*,	*src_1*
	D0: vgpr, U16	S0: src_nolit, U16	S1: src_simple, U16

```
D.u16 = S0.u16 + S1.u16.
```

Supports saturation (unsigned 16-bit integer domain).

v_add_u32	*vdst*,	*src_0*,	*src_1*
	D0: vgpr, U32	S0: src, U32	S1: vgpr, U32
v_add_u32	*vdst*,	*src_0*,	*src_1*
	D0: vgpr, U32	S0: src_nolit, U32	S1: src_simple, U32

```
D.u = S0.u + S1.u.
```

Flags: OPF_CACGRP2

v_addc_co_u32	*vdst*,	*vcc*[2],	*src_0*,	*src_1*,
	D0: vgpr, U32	D1: vcc, B64	S0: src, U32	S1: vgpr, U32
...	*vcc*[2]			
	S2: vcc, B64			
v_addc_co_u32	*vdst*,	*carryout*[2],	*src_0*,	*src_1*,
	D0: vgpr, U32	D1: sreg, B64	S0: src_nolit, U32	S1: src_simple, U32
...	*carryin*[2]			
	S2: sreg, B64			

```
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.

Flags: OPF_CACGRP2, OPF_VCCD, OPF_VCCS

v_and_b32	*vdst*,	*src_0*,	*src_1*
	D0: vgpr, U32	S0: src, U32	S1: vgpr, U32
v_and_b32	*vdst*,	*src_0*,	*src_1*
	D0: vgpr, U32	S0: src_nolit, U32	S1: src_simple, U32

```
D.u = S0.u & S1.u.
```

Input and output modifiers not supported.

Flags: OPF_CACGRP2

v_ashrrev_i16	*vdst*,	*src_0*,	*src_1*
	D0: vgpr, I16	S0: src_nolds, U16	S1: vgpr, I16
v_ashrrev_i16	*vdst*,	*src_0*,	*src_1*
	D0: vgpr, I16	S0: src_simple, U16	S1: src_simple, I16

```
D.i[15:0] = signext(S1.i[15:0]) >> S0.i[3:0].
```

SQ translates this to an internal SP opcode.

Flags: OPF_SQXLATE

v_ashrrev_i32	*vdst*,	*src_0*,	*src_1*
	D0: vgpr, I32	S0: src_nolds, U32	S1: vgpr, I32
v_ashrrev_i32	*vdst*,	*src_0*,	*src_1*
	D0: vgpr, I32	S0: src_simple, U32	S1: src_simple, I32

```
D.i = signext(S1.i) >> S0.i[4:0].
```

SQ translates this to an internal SP opcode.

Flags: OPF_CACGRP2, OPF_SQXLATE

v_cndmask_b32 *vdst*, *src_0*, *src_1*, *vcc*[2]
D0: vgpr, B32 S0: src, B32 S1: vgpr, B32 S2: vcc, B64
v_cndmask_b32 *vdst*, *src_0*, *src_1*, *carryin*[2]
D0: vgpr, B32 S0: src_nolit, B32 S1: src_simple, B32 S2: sreg, B64

```
D.u = (VCC[threadId] ? S1.u : S0.u).
```

Conditional mask on each thread. In VOP3 the VCC source may be a scalar GPR specified in S2.u.
Flags: OPF_CACGRP2, OPF_VCCS

v_dot2c_f32_f16 *vdst*, *src_0*, *src_1*
D0: vgpr, ↔, F32 S0: src, F16 S1: vgpr, F16
v_dot2c_f32_f16 *vdst*, *src_0*, *src_1*
D0: vgpr, ↔, F32 S0: src_nolit, F16 S1: src_simple, F16

```
D.f32 = S0.f16[0] * S1.f16[0] +
        S0.f16[1] * S1.f16[1] + D.f32.
```

Deep learning. Dot product of low-precision values with high-precision accumulator.
Flags: ASIC_DEEP_LEARNING, OPF_DACCUM

v_dot2c_i32_i16 *vdst*, *src_0*, *src_1*
D0: vgpr, ↔, I32 S0: src, I16 S1: vgpr, I16
v_dot2c_i32_i16 *vdst*, *src_0*, *src_1*
D0: vgpr, ↔, I32 S0: src_nolit, I16 S1: src_simple, I16

```
D.i32 = S0.i16[0] * S1.i16[0] +
        S0.i16[1] * S1.i16[1] + D.i32.
```

Deep learning. Dot product of low-precision values with high-precision accumulator.
Flags: ASIC_DEEP_LEARNING, OPF_DACCUM

v_dot4c_i32_i8 *vdst*, *src_0*, *src_1*
D0: vgpr, ↔, I32 S0: src, B32 S1: vgpr, B32
v_dot4c_i32_i8 *vdst*, *src_0*, *src_1*
D0: vgpr, ↔, I32 S0: src_nolit, B32 S1: src_simple, B32

```
D.i32 = S0.i8[0] * S1.i8[0] +
        S0.i8[1] * S1.i8[1] +
        S0.i8[2] * S1.i8[2] +
        S0.i8[3] * S1.i8[3] + D.i32.
```

Deep learning. Dot product of low-precision values with high-precision accumulator.
Flags: ASIC_DEEP_LEARNING, OPF_DACCUM, OPF_S0_I8, OPF_S1_I8

v_dot8c_i32_i4 *vdst*, *src_0*, *src_1*
D0: vgpr, ↔, I32 S0: src, B32 S1: vgpr, B32
v_dot8c_i32_i4 *vdst*, *src_0*, *src_1*
D0: vgpr, ↔, I32 S0: src_nolit, B32 S1: src_simple, B32

```
D.i32 = S0.i4[0] * S1.i4[0] +
        S0.i4[1] * S1.i4[1] +
        S0.i4[2] * S1.i4[2] +
        S0.i4[3] * S1.i4[3] +
        S0.i4[4] * S1.i4[4] +
        S0.i4[5] * S1.i4[5] +
        S0.i4[6] * S1.i4[6] +
        S0.i4[7] * S1.i4[7] + D.i32.
```

Deep learning. Dot product of low-precision values with high-precision accumulator.

Flags: ASIC_DEEP_LEARNING, OPF_DACCUM, OPF_S0_I8, OPF_S1_I8

v_fmac_f32 *vdst*, *src_0*, *src_1*
D0: vgpr, ↔, F32 S0: src, F32 S1: vgpr, F32
v_fmac_f32 *vdst*, *src_0*, *src_1*
D0: vgpr, ↔, F32 S0: src_nolit, F32 S1: src_simple, F32

```
D.f32 = S0.f32 * S1.f32 + D.f32.
```

VOP2 version of V_FMA_F32 with 3rd src VGPR address is the vDst.

Flags: ASIC_DEEP_LEARNING, OPF_DACCUM

v_ldexp_f16 *vdst*, *src_0*, *src_1*
D0: vgpr, F16 S0: src, F16 S1: vgpr, F16
v_ldexp_f16 *vdst*, *src_0*, *src_1*
D0: vgpr, F16 S0: src_nolit, F16 S1: src_simple, F16

```
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
```

Flags: SEN_VOP2

v_lshlrev_b16 *vdst*, *src_0*, *src_1*
D0: vgpr, U16 S0: src_nolds, U16 S1: vgpr, U16
v_lshlrev_b16 *vdst*, *src_0*, *src_1*
D0: vgpr, U16 S0: src_simple, U16 S1: src_simple, U16

```
D.u[15:0] = S1.u[15:0] << S0.u[3:0].
```

SQ translates this to an internal SP opcode.

Flags: OPF_SQXLATE

v_lshlrev_b32 *vdst*, *src_0*, *src_1*
D0: vgpr, U32 S0: src_nolds, U32 S1: vgpr, U32
v_lshlrev_b32 *vdst*, *src_0*, *src_1*
D0: vgpr, U32 S0: src_simple, U32 S1: src_simple, U32

```
D.u = S1.u << S0.u[4:0].
```

SQ translates this to an internal SP opcode.

Flags: OPF_CACGRP2, OPF_SQXLATE

v_lshrrev_b16 *vdst*, *src_0*, *src_1*
D0: vgpr, U16 S0: src_nolds, U16 S1: vgpr, U16
v_lshrrev_b16 *vdst*, *src_0*, *src_1*
D0: vgpr, U16 S0: src_simple, U16 S1: src_simple, U16

```
D.u[15:0] = S1.u[15:0] >> S0.u[3:0].
```

SQ translates this to an internal SP opcode.

Flags: OPF_SQXLATE

v_lshrrev_b32 *vdst*, *src_0*, *src_1*
D0: vgpr, U32 S0: src_nolds, U32 S1: vgpr, U32
v_lshrrev_b32 *vdst*, *src_0*, *src_1*
D0: vgpr, U32 S0: src_simple, U32 S1: src_simple, U32

```
D.u = S1.u >> S0.u[4:0].
```

SQ translates this to an internal SP opcode.

Flags: OPF_CACGRP2, OPF_SQXLATE

v_mac_f16 *vdst*, *src_0*, *src_1*
D0: vgpr, ↔, F16 S0: src, F16 S1: vgpr, F16
v_mac_f16 *vdst*, *src_0*, *src_1*
D0: vgpr, ↔, F16 S0: src_nolit, F16 S1: src_simple, F16

```
D.f16 = S0.f16 * S1.f16 + D.f16.
```

Supports round mode, exception flags, saturation. SQ translates this to V_MAD_LEGACY_F16.

Flags: OPF_CACGRP2, OPF_DACCUM, OPF_NOSDWA, OPF_SQXLATE

v_mac_f32 *vdst*, *src_0*, *src_1*
D0: vgpr, ↔, F32 S0: src, F32 S1: vgpr, F32
v_mac_f32 *vdst*, *src_0*, *src_1*
D0: vgpr, ↔, F32 S0: src_nolit, F32 S1: src_simple, F32

```
D.f = S0.f * S1.f + D.f.
```

SQ translates to V_MAD_F32.

Flags: OPF_CACGRP1, OPF_DACCUM, OPF_NOSDWA, OPF_SQXLATE

v_madak_f16 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, F16 S0: src, F16 S1: vgpr, F16 S2: simm32, F16

```
D.f16 = S0.f16 * S1.f16 + K.f16.
```
// K is a 16-bit literal constant stored in the following literal DWORD.

This opcode cannot use the VOP3 encoding and cannot use input/output modifiers. Supports round mode, exception flags, saturation. SQ translates this to V_MAD_LEGACY_F16.

Flags: OPF_CACGRP2, OPF_IMPLIED_LITERAL, OPF_NODPP, OPF_NOSDWA, OPF_NOVOP3, OPF_SQXLATE

v_madak_f32 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, F32 S0: src, F32 S1: vgpr, F32 S2: simm32, 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. SQ translates to V_MAD_F32.

Flags: OPF_CACGRP1, OPF_IMPLIED_LITERAL, OPF_NODPP, OPF_NOSDWA, OPF_NOVOP3, OPF_SQXLATE

v_madmk_f16 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, F16 S0: src, F16 S1: simm32, F16 S2: vgpr, F16

```
D.f16 = S0.f16 * K.f16 + S1.f16.
```
// K is a 16-bit literal constant stored in the following literal DWORD.

This opcode cannot use the VOP3 encoding and cannot use input/output modifiers. Supports round mode, exception flags, saturation. SQ translates this to V_MAD_LEGACY_F16.

Flags: OPF_CACGRP2, OPF_IMPLIED_LITERAL, OPF_NODPP, OPF_NOSDWA, OPF_NOVOP3, OPF_SQXLATE

v_madmk_f32 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, F32 S0: src, F32 S1: simm32, F32 S2: vgpr, 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. SQ translates to V_MAD_F32.

Flags: OPF_CACGRP1, OPF_IMPLIED_LITERAL, OPF_NODPP, OPF_NOSDWA, OPF_NOVOP3, OPF_SQXLATE

v_max_f16	*vdst*,	*src_0*,	*src_1*
	D0: vgpr, F16	S0: src, F16	S1: vgpr, F16
v_max_f16	*vdst*,	*src_0*,	*src_1*
	D0: vgpr, F16	S0: src_nolit, F16	S1: src_simple, 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);
else
        D.f16 = (S0.f16 > S1.f16 ? S0.f16 : S1.f16);
endif.
```

IEEE compliant. Supports denormals, round mode, exception flags, saturation.

v_max_f32	*vdst*,	*src_0*,	*src_1*
	D0: vgpr, F32	S0: src, F32	S1: vgpr, F32
v_max_f32	*vdst*,	*src_0*,	*src_1*
	D0: vgpr, F32	S0: src_nolit, F32	S1: src_simple, 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 = S0.f;
else if (S0.f == -0.0 && S1.f == +0.0)
        D.f = S1.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.
```

Flags: OPF_CACGRP2

v_max_i16 *vdst*, *src_0*, *src_1*
D0: vgpr, I16 S0: src, I16 S1: vgpr, I16
v_max_i16 *vdst*, *src_0*, *src_1*
D0: vgpr, I16 S0: src_nolit, I16 S1: src_simple, I16

```
D.i16 = (S0.i16 ≥ S1.i16 ? S0.i16 : S1.i16).
```

v_max_i32 *vdst*, *src_0*, *src_1*
D0: vgpr, I32 S0: src, I32 S1: vgpr, I32
v_max_i32 *vdst*, *src_0*, *src_1*
D0: vgpr, I32 S0: src_nolit, I32 S1: src_simple, I32

```
D.i = (S0.i ≥ S1.i ? S0.i : S1.i).
```

Flags: OPF_CACGRP2

v_max_u16 *vdst*, *src_0*, *src_1*
D0: vgpr, U16 S0: src, U16 S1: vgpr, U16
v_max_u16 *vdst*, *src_0*, *src_1*
D0: vgpr, U16 S0: src_nolit, U16 S1: src_simple, U16

```
D.u16 = (S0.u16 ≥ S1.u16 ? S0.u16 : S1.u16).
```

v_max_u32 *vdst*, *src_0*, *src_1*
D0: vgpr, U32 S0: src, U32 S1: vgpr, U32
v_max_u32 *vdst*, *src_0*, *src_1*
D0: vgpr, U32 S0: src_nolit, U32 S1: src_simple, U32

```
D.u = (S0.u ≥ S1.u ? S0.u : S1.u).
```

Flags: OPF_CACGRP2

v_min_f16 *vdst*, *src_0*, *src_1*
D0: vgpr, F16 S0: src, F16 S1: vgpr, F16
v_min_f16 *vdst*, *src_0*, *src_1*
D0: vgpr, F16 S0: src_nolit, F16 S1: src_simple, 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 = S1.f16;
else if (S0.f16 == -0.0 && S1.f16 == +0.0)
        D.f16 = S0.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.

v_min_f32 *vdst*, *src_0*, *src_1*
D0: vgpr, F32 S0: src, F32 S1: vgpr, F32
v_min_f32 *vdst*, *src_0*, *src_1*
D0: vgpr, F32 S0: src_nolit, F32 S1: src_simple, 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.
```

Flags: OPF_CACGRP2

v_min_i16 *vdst*, *src_0*, *src_1*
D0: vgpr, I16 S0: src, I16 S1: vgpr, I16
v_min_i16 *vdst*, *src_0*, *src_1*
D0: vgpr, I16 S0: src_nolit, I16 S1: src_simple, I16

```
D.i16 = (S0.i16 < S1.i16 ? S0.i16 : S1.i16).
```

v_min_i32 *vdst*, *src_0*, *src_1*
D0: vgpr, I32 S0: src, I32 S1: vgpr, I32
v_min_i32 *vdst*, *src_0*, *src_1*
D0: vgpr, I32 S0: src_nolit, I32 S1: src_simple, I32

```
D.i = (S0.i < S1.i ? S0.i : S1.i).
```

Flags: OPF_CACGRP2

v_min_u16 *vdst*, *src_0*, *src_1*
D0: vgpr, U16 S0: src, U16 S1: vgpr, U16
v_min_u16 *vdst*, *src_0*, *src_1*
D0: vgpr, U16 S0: src_nolit, U16 S1: src_simple, U16

```
D.u16 = (S0.u16 < S1.u16 ? S0.u16 : S1.u16).
```

v_min_u32 *vdst*, *src_0*, *src_1*
D0: vgpr, U32 S0: src, U32 S1: vgpr, U32
v_min_u32 *vdst*, *src_0*, *src_1*
D0: vgpr, U32 S0: src_nolit, U32 S1: src_simple, U32

```
D.u = (S0.u < S1.u ? S0.u : S1.u).
```

Flags: OPF_CACGRP2

v_mul_f16 *vdst*, *src_0*, *src_1*
D0: vgpr, F16 S0: src, F16 S1: vgpr, F16
v_mul_f16 *vdst*, *src_0*, *src_1*
D0: vgpr, F16 S0: src_nolit, F16 S1: src_simple, F16

```
D.f16 = S0.f16 * S1.f16.
```

Supports denormals, round mode, exception flags, saturation. 0.5ULP precision, denormals are supported.

v_mul_f32 *vdst*, *src_0*, *src_1*
D0: vgpr, F32 S0: src, F32 S1: vgpr, F32
v_mul_f32 *vdst*, *src_0*, *src_1*
D0: vgpr, F32 S0: src_nolit, F32 S1: src_simple, F32

```
D.f = S0.f * S1.f.
```

0.5ULP precision, denormals are supported.

Flags: OPF_CACGRP1

v_mul_hi_i32_i24 *vdst*, *src_0*, *src_1*
D0: vgpr, I32 S0: src, I32 S1: vgpr, I32
v_mul_hi_i32_i24 *vdst*, *src_0*, *src_1*
D0: vgpr, I32 S0: src_nolit, I32 S1: src_simple, I32

```
D.i = (S0.i[23:0] * S1.i[23:0])>>32.
```

Flags: OPF_CACGRP1

v_mul_hi_u32_u24 *vdst*, *src_0*, *src_1*
D0: vgpr, U32 S0: src, U32 S1: vgpr, U32
v_mul_hi_u32_u24 *vdst*, *src_0*, *src_1*
D0: vgpr, U32 S0: src_nolit, U32 S1: src_simple, U32

```
D.i = (S0.u[23:0] * S1.u[23:0])>>32.
```

Flags: OPF_CACGRP1

v_mul_i32_i24 *vdst*, *src_0*, *src_1*
D0: vgpr, I32 S0: src, I32 S1: vgpr, I32
v_mul_i32_i24 *vdst*, *src_0*, *src_1*
D0: vgpr, I32 S0: src_nolit, I32 S1: src_simple, I32

```
D.i = S0.i[23:0] * S1.i[23:0].
```

Flags: OPF_CACGRP1

v_mul_legacy_f32 *vdst*, *src_0*, *src_1*
D0: vgpr, F32 S0: src, F32 S1: vgpr, F32
v_mul_legacy_f32 *vdst*, *src_0*, *src_1*
D0: vgpr, F32 S0: src_nolit, F32 S1: src_simple, F32

```
D.f = S0.f * S1.f. // DX9 rules, 0.0*x = 0.0
```

Flags: OPF_CACGRP1

v_mul_lo_u16 *vdst*, *src_0*, *src_1*
D0: vgpr, U16 S0: src, U16 S1: vgpr, U16
v_mul_lo_u16 *vdst*, *src_0*, *src_1*
D0: vgpr, U16 S0: src_nolit, U16 S1: src_simple, U16

```
D.u16 = S0.u16 * S1.u16.
```

Supports saturation (unsigned 16-bit integer domain).

v_mul_u32_u24 *vdst*, *src_0*, *src_1*
D0: vgpr, U32 S0: src, U32 S1: vgpr, U32
v_mul_u32_u24 *vdst*, *src_0*, *src_1*
D0: vgpr, U32 S0: src_nolit, U32 S1: src_simple, U32

```
D.u = S0.u[23:0] * S1.u[23:0].
```

Flags: OPF_CACGRP1

v_or_b32 *vdst*, *src_0*, *src_1*
D0: vgpr, U32 S0: src, U32 S1: vgpr, U32
v_or_b32 *vdst*, *src_0*, *src_1*
D0: vgpr, U32 S0: src_nolit, U32 S1: src_simple, U32

```
D.u = S0.u | S1.u.
```

Input and output modifiers not supported.

Flags: OPF_CACGRP2

v_pk_fmac_f16 *vdst*, *src_0*, *src_1*
D0: vgpr, ↔, F16 S0: src, F16 S1: vgpr, F16
v_pk_fmac_f16 *vdst*, *src_0*, *src_1*
D0: vgpr, ↔, F16 S0: src_nolit, F16 S1: src_simple, F16

```
D.f16[0] = S0.f16[0] * S1.f16[0] + S2.f16[0];
D.f16[1] = S0.f16[1] * S1.f16[1] + S2.f16[1].
```

VOP2 version of V_PK_FMA_F16 with 3rd src VGPR address is the vDst.

Flags: ASIC_DEEP_LEARNING, OPF_DACCUM

v_sub_co_u32 *vdst*, *vcc*[2], *src_0*, *src_1*
D0: vgpr, U32 D1: vcc, B64 S0: src, U32 S1: vgpr, U32
v_sub_co_u32 *vdst*, *carryout*[2], *src_0*, *src_1*
D0: vgpr, U32 D1: sreg, B64 S0: src_nolit, U32 S1: src_simple, 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.

Flags: OPF_CACGRP2, OPF_VCCD

v_sub_f16 *vdst*, *src_0*, *src_1*
D0: vgpr, F16 S0: src, F16 S1: vgpr, F16
v_sub_f16 *vdst*, *src_0*, *src_1*
D0: vgpr, F16 S0: src_nolit, F16 S1: src_simple, F16

```
D.f16 = S0.f16 - S1.f16.
```

Supports denormals, round mode, exception flags, saturation. SQ translates to V_ADD_F16.

Flags: OPF_SQXLATE

v_sub_f32 *vdst*, *src_0*, *src_1*
D0: vgpr, F32 S0: src, F32 S1: vgpr, F32
v_sub_f32 *vdst*, *src_0*, *src_1*
D0: vgpr, F32 S0: src_nolit, F32 S1: src_simple, F32

```
D.f = S0.f - S1.f.
```

SQ translates to V_ADD_F32.

Flags: OPF_CACGRP1, OPF_SQXLATE

v_sub_u16 *vdst*, *src_0*, *src_1*
D0: vgpr, U16 S0: src, U16 S1: vgpr, U16
v_sub_u16 *vdst*, *src_0*, *src_1*
D0: vgpr, U16 S0: src_nolit, U16 S1: src_simple, U16

```
D.u16 = S0.u16 - S1.u16.
```

Supports saturation (unsigned 16-bit integer domain).

v_sub_u32 *vdst*, *src_0*, *src_1*
D0: vgpr, U32 S0: src, U32 S1: vgpr, U32
v_sub_u32 *vdst*, *src_0*, *src_1*
D0: vgpr, U32 S0: src_nolit, U32 S1: src_simple, U32

```
D.u = S0.u - S1.u.
```

Flags: OPF_CACGRP2

v_subb_co_u32 *vdst*, *vcc*[2], *src_0*, *src_1*,
D0: vgpr, U32 D1: vcc, B64 S0: src, U32 S1: vgpr, U32
... *vcc*[2]
S2: vcc, B64
v_subb_co_u32 *vdst*, *carryout*[2], *src_0*, *src_1*,
D0: vgpr, U32 D1: sreg, B64 S0: src_nolit, U32 S1: src_simple, U32
... *carryin*[2]
S2: sreg, B64

```
D.u = S0.u - S1.u - VCC[threadId];
VCC[threadId] = (S1.u + VCC[threadId] > S0.u ? 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.

Flags: OPF_CACGRP2, OPF_VCCD, OPF_VCCS

v_subbrev_co_u32 *vdst*, *vcc*[2], *src_0*, *src_1*,
D0: vgpr, U32 D1: vcc, B64 S0: src_nolds, U32 S1: vgpr, U32
... *vcc*[2]
S2: vcc, B64
v_subbrev_co_u32 *vdst*, *carryout*[2], *src_0*, *src_1*,
D0: vgpr, U32 D1: sreg, B64 S0: src_simple, U32 S1: src_simple, U32
... *carryin*[2]
S2: sreg, B64

```
D.u = S1.u - S0.u - VCC[threadId];
VCC[threadId] = (S1.u + VCC[threadId] > S0.u ? 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. SQ translates to V_SUBB_CO_U32. SQ translates this to V_SUBREV_U32 with reversed operands.

Flags: OPF_CACGRP2, OPF_SQXLATE, OPF_VCCD, OPF_VCCS

v_subrev_co_u32 *vdst*, *vcc*[2], *src_0*, *src_1*
D0: vgpr, U32 D1: vcc, B64 S0: src_nolds, U32 S1: vgpr, U32
v_subrev_co_u32 *vdst*, *carryout*[2], *src_0*, *src_1*
D0: vgpr, U32 D1: sreg, B64 S0: src_simple, U32 S1: src_simple, 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. SQ translates this to V_SUB_CO_U32 with reversed operands.

Flags: OPF_CACGRP2, OPF_SQXLATE, OPF_VCCD

v_subrev_f16 *vdst*, *src_0*, *src_1*
D0: vgpr, F16 S0: src_nolds, F16 S1: vgpr, F16
v_subrev_f16 *vdst*, *src_0*, *src_1*
D0: vgpr, F16 S0: src_simple, F16 S1: src_simple, F16

```
D.f16 = S1.f16 - S0.f16.
```

Supports denormals, round mode, exception flags, saturation. SQ translates to V_ADD_F16.

Flags: OPF_SQXLATE

v_subrev_f32 *vdst*, *src_0*, *src_1*
D0: vgpr, F32 S0: src, F32 S1: vgpr, F32
v_subrev_f32 *vdst*, *src_0*, *src_1*
D0: vgpr, F32 S0: src_nolit, F32 S1: src_simple, F32

```
D.f = S1.f - S0.f.
```

SQ translates to V_ADD_F32.

Flags: OPF_CACGRP1, OPF_SQXLATE

v_subrev_u16	*vdst*,	*src_0*,	*src_1*
	D0: vgpr, U16	S0: src_nolds, U16	S1: vgpr, U16
v_subrev_u16	*vdst*,	*src_0*,	*src_1*
	D0: vgpr, U16	S0: src_simple, U16	S1: src_simple, U16

```
D.u16 = S1.u16 - S0.u16.
```

Supports saturation (unsigned 16-bit integer domain). SQ translates this to V_SUB_U16 with reversed operands.

Flags: OPF_SQXLATE

v_subrev_u32	*vdst*,	*src_0*,	*src_1*
	D0: vgpr, U32	S0: src_nolds, U32	S1: vgpr, U32
v_subrev_u32	*vdst*,	*src_0*,	*src_1*
	D0: vgpr, U32	S0: src_simple, U32	S1: src_simple, U32

```
D.u = S1.u - S0.u.
```

SQ translates this to V_SUB_U32 with reversed operands.

Flags: OPF_CACGRP2, OPF_SQXLATE

v_xnor_b32	*vdst*,	*src_0*,	*src_1*
	D0: vgpr, B32	S0: src, B32	S1: vgpr, B32
v_xnor_b32	*vdst*,	*src_0*,	*src_1*
	D0: vgpr, B32	S0: src_nolit, B32	S1: src_simple, B32

```
D.b32 = S0.b32 XNOR S1.b32.
```

Flags: ASIC_DEEP_LEARNING

v_xor_b32	*vdst*,	*src_0*,	*src_1*
	D0: vgpr, U32	S0: src, U32	S1: vgpr, U32
v_xor_b32	*vdst*,	*src_0*,	*src_1*
	D0: vgpr, U32	S0: src_nolit, U32	S1: src_simple, U32

```
D.u = S0.u ^ S1.u.
```

Input and output modifiers not supported.

Flags: OPF_CACGRP2

10.1 Notes for Encoding VOP2

10.1.1 Input modifiers

Source operands may invoke the negation and absolute-value input modifiers with `-`*src* and `abs(`*src*`)`, respectively. For example,

```
v_add_f32 v0, v1, -v2           // Subtract v2 from v1
v_add_f32 v0, abs(v1), abs(v2)  // Take absolute value of both inputs
```

In general, negation and absolute value are only supported for floating point input operands (operands with a type of F16, F32 or F64); they are not supported for integer or untyped inputs.

10.1.2 Output modifiers

`mul:{1,2,4}`
Set output modifier to multiply by N. Default 1, which leaves the result unmodified. This operation is only defined when the result is F16, F32 or F64.

`div:{1,2}`
Set output modifier to divide by N. Default 1, which leaves the result unmodified. This operation is only defined when the result is F16, F32 or F64.

`clamp:{0,1}`
Clamp (saturate) the output. Default 0. Can also write `noclamp`. Saturation is defined as follows.

For I16, I32, I64 results: if the result exceeds SHRT_MAX/INT_MAX/LLONG_MAX it will be clamped to the most positive representable value; if the result is below SHRT_MIN/INT_MIN/LLONG_MIN it will be clamped to the most negative representable value.

For U16, U32, U64 results: if the result exceeds USHRT_MAX/UINT_MAX/ULLONG_MAX it will be clamped to the most positive representable value; if the result is below 0 it will be clamped to 0.

For F16, F32, F64 results: the result will be clamped to the interval [0.0, 1.0].

10.1.3 Interpolation operands and modifiers

vgpr_dst is a vector GPR to store result in, and use as accumulator source in certain interpolation operations.

vgpr_ij is a vector GPR to read i/j value from.

attr is `attr0.x` through `attr63.w`, parameter attribute and channel to be interpolated.

param is `p10`, `p20` or `p0`.

For 16-bit interpolation it is necessary to specify whether we are operating on the high or low word of the attribute. For this, two new modifiers are provided:

`high`
Interpolate using the high 16 bits of the attribute.

`low`
Interpolate using the low 16 bits of the attribute. Default.

10.1.4 Other modifiers

`vop3:{0,1}`
Force VOP3 encoding even if instruction can be represented in smaller encoding. Default 0. Can also write `novop3`. Note that even if this modifier is not set, an opcode will still use the VOP3 encoding if the operands or modifiers given cannot be expressed in a smaller encoding.

10.1.5 SDWA output modifiers

This only applies to VOP1/VOP2/VOPC encodings.

The SDWA subencoding supports sign-extension of inputs; this may be written as `sext(`*src*`)`, similar to how the `abs` modifier is used.

`src0_sel:`*sdwa_sel*
: Apply to the selected sdwa data bits of src0. Default DWORD.

`src1_sel:`*sdwa_sel*
: Apply to the selected sdwa data bits of src1. Default DWORD.

`dst_sel:`*sdwa_sel*
: Apply to the selected sdwa data bits of dst. Default DWORD.

`dst_unused:`*sdwa_unused*
: Determine what to do with the unused dst bits. Default UNUSED_PAD.

10.1.6 DPP output modifiers

This only applies to VOP1/VOP2/VOPC encodings.

General modifiers Any combination of these modifiers may be applied for DPP instructions.

`bank_mask:[0. . . 0xf]`
: Apply to the selected bank mask bits.

`row_mask:[0. . . 0xf]`
: Apply to the selected row mask bits.

`bound_ctrl:{0,1}`
: If true, then writes to out-of-bounds threads are written as zero. If false, writes to out-of-bounds threads are disabled.

DPP permutation control Only one of the following modifiers may be applied to the instruction.

Each wave of 64 threads $(W_0, \ldots, W_{63})$ is divided into 4 *rows* of 16 threads, $(R_0, \ldots, R_{15})$ and into 16 *quads* of 4 threads (usually pixels), $(P_0, \ldots, P_3)$.

`quad_perm:[pix0,pix1,pix2,pix3]`

DPP permutation: Applied for each pixel within a quad. The `N`th entry in the array specifies that the `N`th pixel in the output will obtain its data from the `quad_perm[N]`th pixel in the input.

The identity transformation is represented as `quad_perm:[0,1,2,3]`.

`quad_perm:[1,2,0,1]` applies the following permutation which includes broadcasting one of the pixels to multiple destinations:

$$\begin{bmatrix} P_0 & P_1 \\ P_2 & P_3 \end{bmatrix} : \qquad \begin{bmatrix} A & B \\ C & D \end{bmatrix} \Rightarrow \begin{bmatrix} B & C \\ A & B \end{bmatrix}$$

To broadcast one pixel (e.g. pixel 0) to all destinations in the quad, use `quad_perm:[0,0,0,0]`:

$$\begin{bmatrix} P_0 & P_1 \\ P_2 & P_3 \end{bmatrix} : \qquad \begin{bmatrix} A & B \\ C & D \end{bmatrix} \Rightarrow \begin{bmatrix} A & A \\ A & A \end{bmatrix}$$

In general the output thread P' is determined by

$$P'_n = P_{\texttt{quad\_perm}[n]}$$

`row_shr:[1. . . 15]`

DPP permutation: Shifts threads in each row to the right by the specified amount.

In general the output thread R' with count C is determined by

$$R'_n = \begin{cases} R_{n-C} & n \geq C \\ \texttt{bound\_ctrl} & \text{otherwise} \end{cases}$$

`row_shl:[1. . . 15]`

DPP permutation: Shifts threads in each row to the left by the specified amount.

In general the output thread R' with count C is determined by

$$R'_n = \begin{cases} R_{n+C} & n + C < 16 \\ \texttt{bound\_ctrl} & \text{otherwise} \end{cases}$$

`row_ror:[1. . . 15]`

DPP permutation: Rotates threads in each row to the right by the specified amount.

In general the output thread R' with count C is determined by

$$R'_n = \begin{cases} R_{n-C} & n \geq C \\ R_{n-C+16} & \text{otherwise} \end{cases}$$

`wave_shr`
DPP permutation: Shifts threads in the entire wave to the right by one.

In general the output thread W' is determined by

$$W'_n = \begin{cases} W_{n-1} & n > 0 \\ \texttt{bound\_ctrl} & \text{otherwise} \end{cases}$$

`wave_shl`
DPP permutation: Shifts threads in the entire wave to the left by one.

In general the output thread W' is determined by

$$W'_n = \begin{cases} W_{n+1} & n < 63 \\ \texttt{bound\_ctrl} & \text{otherwise} \end{cases}$$

`wave_ror`
DPP permutation: Rotates threads in the entire wave to the right by one.

In general the output thread W' is determined by

$$W'_n = \begin{cases} W_{n-1} & n > 0 \\ W_{63} & \text{otherwise} \end{cases}$$

`wave_rol`
DPP permutation: Rotates threads in the entire wave to the left by one.

In general the output thread W' is determined by

$$W'_n = \begin{cases} W_{n+1} & n < 63 \\ W_0 & \text{otherwise} \end{cases}$$

`row_mirror`
DPP permutation: Mirrors threads within each row.

In general the output thread R' is determined by

$$R'_n = R_{15-n}$$

`row_half_mirror`
: DPP permutation: Mirrors threads within each half-row.

  In general the output thread R' is determined by

$$R'_n = \begin{cases} R_{7-n} & n < 8 \\ R_{23-n} & \text{otherwise} \end{cases}$$

`row_bcast:{15, 31}`
: DPP permutation: Broadcast a thread to subsequent rows.

11 Encoding VINTRP

Vector ALU interpolation. Instructions in this encoding may be promoted to VOP3 unless otherwise noted.

v_interp_mov_f32 *vgpr_dst*, *param*, *attr*
D0: vgpr, F32 S0: param S1: attr
v_interp_mov_f32 *vgpr_dst*, *param*, *attr*
D0: vgpr, F32 S0: param S1: attr

```
D.f = {P10,P20,P0}[S.u].
```

Parameter load. Used for custom interpolation in the shader.

Flags: OPF_INTERP, OPF_RDM0

v_interp_p1_f32 *vgpr_dst*, *vgpr_ij*, *attr*
D0: vgpr, F32 S0: vgpr, F32 S1: attr
v_interp_p1_f32 *vgpr_dst*, *vgpr_ij*, *attr*
D0: vgpr, F32 S0: src_vgpr, F32 S1: attr

```
D.f = P10 * S.f + P0.
```

Parameter interpolation (SQ translates to V_MAD_F32 for SP).

CAUTION: when in HALF_LDS mode, D must not be the same GPR as S; if D == S then data corruption will occur.

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.

Flags: OPF_CACGRP1, OPF_INTERP, OPF_RDM0

v_interp_p2_f32 *vgpr_dst*, *vgpr_ij*, *attr*
D0: vgpr, ↔, F32 S0: vgpr, F32 S1: attr
v_interp_p2_f32 *vgpr_dst*, *vgpr_ij*, *attr*
D0: vgpr, ↔, F32 S0: src_vgpr, F32 S1: attr

```
D.f = P20 * S.f + D.f.
```

Parameter interpolation (SQ translates to V_MAD_F32 for SP).

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.

Flags: OPF_CACGRP1, OPF_DACCUM, OPF_INTERP, OPF_RDM0

11.1 Notes for Encoding VINTRP

11.1.1 Input modifiers

Source operands may invoke the negation and absolute-value input modifiers with `-`*src* and `abs(`*src*`)`, respectively. For example,

```
v_add_f32 v0, v1, -v2           // Subtract v2 from v1
v_add_f32 v0, abs(v1), abs(v2)  // Take absolute value of both inputs
```

In general, negation and absolute value are only supported for floating point input operands (operands with a type of F16, F32 or F64); they are not supported for integer or untyped inputs.

11.1.2 Output modifiers

`mul:{1,2,4}`
: Set output modifier to multiply by N. Default 1, which leaves the result unmodified. This operation is only defined when the result is F16, F32 or F64.

`div:{1,2}`
: Set output modifier to divide by N. Default 1, which leaves the result unmodified. This operation is only defined when the result is F16, F32 or F64.

`clamp:{0,1}`
: Clamp (saturate) the output. Default 0. Can also write `noclamp`. Saturation is defined as follows.

  For I16, I32, I64 results: if the result exceeds SHRT_MAX/INT_MAX/LLONG_MAX it will be clamped to the most positive representable value; if the result is below SHRT_MIN/INT_MIN/LLONG_MIN it will be clamped to the most negative representable value.

  For U16, U32, U64 results: if the result exceeds USHRT_MAX/UINT_MAX/ULLONG_MAX it will be clamped to the most positive representable value; if the result is below 0 it will be clamped to 0.

  For F16, F32, F64 results: the result will be clamped to the interval [0.0, 1.0].

11.1.3 Interpolation operands and modifiers

vgpr_dst is a vector GPR to store result in, and use as accumulator source in certain interpolation operations.

vgpr_ij is a vector GPR to read i/j value from.

attr is `attr0.x` through `attr63.w`, parameter attribute and channel to be interpolated.

param is `p10`, `p20` or `p0`.

For 16-bit interpolation it is necessary to specify whether we are operating on the high or low word of the attribute. For this, two new modifiers are provided:

`high`
: Interpolate using the high 16 bits of the attribute.

`low`
: Interpolate using the low 16 bits of the attribute. Default.

11.1.4 Other modifiers

`vop3:{0,1}`
: Force VOP3 encoding even if instruction can be represented in smaller encoding. Default 0. Can also write `novop3`. Note that even if this modifier is not set, an opcode will still use the VOP3 encoding if the operands or modifiers given cannot be expressed in a smaller encoding.

12 Encoding VOP3P

Vector ALU operations requiring two or three sources and performing two 16 bit ops in one instruction

v_dot2_f32_f16 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, F32 S0: src_nolit, F16 S1: src_simple, F16 S2: src_simple, F32
D.f32 = S0.f16[0] * S1.f16[0] + S0.f16[1] * S1.f16[1] + S2.f32
Flags: ASIC_DEEP_LEARNING

v_dot2_i32_i16 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, I32 S0: src_nolit, I16 S1: src_simple, I16 S2: src_simple, I32
D.i32 = S0.i16[0] * S1.i16[0] + S0.i16[1] * S1.i16[1] + S2.i32
Flags: ASIC_DEEP_LEARNING

v_dot2_i32_i16_i8 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, I32 S0: src_nolit, B32 S1: src_simple, I16 S2: src_simple, I32
D.i32 = S0.i8[0] * S1.i16[0] + S0.i8[1] * S1.i16[1] + S2.i32
Flags: ASIC_DEEP_LEARNING, OPF_S0_I8

v_dot2_u32_u16 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, U32 S0: src_nolit, U16 S1: src_simple, U16 S2: src_simple, U32
D.u32 = S0.u16[0] * S1.u16[0] + S0.u16[1] * S1.u16[1] + S2.u32
Flags: ASIC_DEEP_LEARNING

v_dot2_u32_u16_u8 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, U32 S0: src_nolit, B32 S1: src_simple, U16 S2: src_simple, U32
D.u32 = S0.u8[0] * S1.u16[0] + S0.u8[1] * S1.u16[1] + S2.u32
Flags: ASIC_DEEP_LEARNING, OPF_S0_U8

v_dot4_i32_i8 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, I32 S0: src_nolit, B32 S1: src_simple, B32 S2: src_simple, I32
D.i32 = S0.i8[0] * S1.i8[0] + S0.i8[1] * S1.i8[1] + S0.i8[2] * S1.i8[2] + S0.i8[3] * S1.i8[3] + S2.i32
Flags: ASIC_DEEP_LEARNING, OPF_S0_I8, OPF_S1_I8

v_dot4_u32_u8 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, U32 S0: src_nolit, B32 S1: src_simple, B32 S2: src_simple, U32
D.u32 = S0.u8[0] * S1.u8[0] + S0.u8[1] * S1.u8[1] + S0.u8[2] * S1.u8[2] + S0.u8[3] * S1.u8[3] + S2.u32
Flags: ASIC_DEEP_LEARNING, OPF_S0_U8, OPF_S1_U8

v_dot8_i32_i4 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, I32 S0: src_nolit, B32 S1: src_simple, B32 S2: src_simple, I32
D.i32 = S0.i4[0] * S1.i4[0] + S0.i4[1] * S1.i4[1] + S0.i4[2] * S1.i4[2] + S0.i4[3] * S1.i4[3] + S0.i4[4] * S1.i4[4] + S0.i4[5] * S1.i4[5] + S0.i4[6] * S1.i4[6] + S0.i4[7] * S1.i4[7] + S2.i32
Flags: ASIC_DEEP_LEARNING, OPF_S0_I4, OPF_S1_I4

v_dot8_u32_u4 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, U32 S0: src_nolit, B32 S1: src_simple, B32 S2: src_simple, U32

D.u32 = S0.u4[0] * S1.u4[0] + S0.u4[1] * S1.u4[1] + S0.u4[2] * S1.u4[2] + S0.u4[3] * S1.u4[3] + S0.u4[4] * S1.u4[4] + S0.u4[5] * S1.u4[5] + S0.u4[6] * S1.u4[6] + S0.u4[7] * S1.u4[7] + S2.u32

Flags: ASIC_DEEP_LEARNING, OPF_S0_U4, OPF_S1_U4

v_mad_mix_f32 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, B32 S0: src_nolit, B32 S1: src_simple, B32 S2: src_simple, B32

D.f[31:0] = S0.f * S1.f + S2.f. Size and location of S0, S1 and S2 controlled by OPSEL.

Flags: OPF_CACGRP1, OPF_OPSEL_VOP3P

v_mad_mixhi_f16 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, B32 S0: src_nolit, B32 S1: src_simple, B32 S2: src_simple, B32

D.f[31:16] = S0.f * S1.f + S2.f. Size and location of S0, S1 and S2 controlled by OPSEL.

Flags: OPF_CACGRP1, OPF_OPSEL_VOP3P

v_mad_mixlo_f16 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, B32 S0: src_nolit, B32 S1: src_simple, B32 S2: src_simple, B32

D.f[15:0] = S0.f * S1.f + S2.f. Size and location of S0, S1 and S2 controlled by OPSEL.

Flags: OPF_CACGRP1, OPF_OPSEL_VOP3P

v_pk_add_f16 *vdst*, *src_0*, *src_1*
D0: vgpr, F16 S0: src_nolit, F16 S1: src_simple, F16

D.f[31:16] = S0.f[31:16] + S1.f[31:16] . D.f[15:0] = S0.f[15:0] + S1.f[15:0] .

Flags: SEN_VOP2 , OPF_CACGRP1, OPF_OPSEL_VOP3P

v_pk_add_i16 *vdst*, *src_0*, *src_1*
D0: vgpr, B16 S0: src_nolit, B16 S1: src_simple, B16

D.i[31:16] = S0.i[31:16] + S1.i[31:16] . D.i[15:0] = S0.i[15:0] + S1.i[15:0] .

Flags: SEN_VOP2 , OPF_CACGRP2, OPF_OPSEL_VOP3P

v_pk_add_u16 *vdst*, *src_0*, *src_1*
D0: vgpr, B16 S0: src_nolit, B16 S1: src_simple, B16

D.u[31:16] = S0.u[31:16] + S1.u[31:16] . D.u[15:0] = S0.u[15:0] + S1.u[15:0] .

Flags: SEN_VOP2 , OPF_CACGRP2, OPF_OPSEL_VOP3P

v_pk_ashrrev_i16 *vdst*, *src_0*, *src_1*
D0: vgpr, B32 S0: src_simple, B32 S1: src_simple, B32

D.i[31:16] = S1.i[31:16] >> S0.i[19:16] . D.i[15:0] = S1.i[15:0] >> S0.i[3:0] .

SQ translates this to an internal SP opcode.

Flags: SEN_VOP2 , OPF_CACGRP2, OPF_OPSEL_VOP3P, OPF_SQXLATE

v_pk_fma_f16 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, F16 S0: src_nolit, F16 S1: src_simple, F16 S2: src_simple, F16

D.f[31:16] = S0.f[31:16] * S1.f[31:16] + S2.f[31:16] . D.f[15:0] = S0.f[15:0] * S1.f[15:0] + S2.f[15:0] .

Fused half-precision multiply add.

Flags: OPF_CACGRP0, OPF_OPSEL_VOP3P

v_pk_lshlrev_b16 *vdst*, *src_0*, *src_1*
D0: vgpr, B32 S0: src_simple, B32 S1: src_simple, B32

D.u[31:16] = S1.u[31:16] << S0.u[19:16] . D.u[15:0] = S1.u[15:0] << S0.u[3:0] .

SQ translates this to an internal SP opcode.

Flags: SEN_VOP2 , OPF_CACGRP2, OPF_OPSEL_VOP3P, OPF_SQXLATE

v_pk_lshrrev_b16 *vdst*, *src_0*, *src_1*
D0: vgpr, B32 S0: src_simple, B32 S1: src_simple, B32

D.u[31:16] = S1.u[31:16] >> S0.u[19:16] . D.u[15:0] = S1.u[15:0] >> S0.u[3:0] .

SQ translates this to an internal SP opcode.

Flags: SEN_VOP2 , OPF_CACGRP2, OPF_OPSEL_VOP3P, OPF_SQXLATE

v_pk_mad_i16 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, B16 S0: src_nolit, B16 S1: src_simple, B16 S2: src_simple, B16

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] .

Flags: OPF_CACGRP1, OPF_OPSEL_VOP3P

v_pk_mad_u16 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, B16 S0: src_nolit, B16 S1: src_simple, B16 S2: src_simple, B16

D.u[31:16] = S0.u[31:16] * S1.u[31:16] + S2.u[31:16] . D.u[15:0] = S0.u[15:0] * S1.u[15:0] + S2.u[15:0] .

Flags: OPF_CACGRP1, OPF_OPSEL_VOP3P

v_pk_max_f16 *vdst*, *src_0*, *src_1*
D0: vgpr, F16 S0: src_nolit, F16 S1: src_simple, F16

D.f[31:16] = max(S0.f[31:16], S1.f[31:16]) . D.f[15:0] = max(S0.f[15:0], S1.f[15:0]) .

Flags: SEN_VOP2 , OPF_CACGRP2, OPF_OPSEL_VOP3P

v_pk_max_i16 *vdst*, *src_0*, *src_1*
D0: vgpr, B16 S0: src_nolit, B16 S1: src_simple, B16

D.i[31:16] = (S0.i[31:16] $\geq$ S1.i[31:16]) ? S0.i[31:16] : S1.i[31:16] . D.i[15:0] = (S0.i[15:0] $\geq$ S1.i[15:0]) ? S0.i[15:0] : S1.i[15:0] .

Flags: SEN_VOP2 , OPF_CACGRP2, OPF_OPSEL_VOP3P

v_pk_max_u16 *vdst*, *src_0*, *src_1*
D0: vgpr, B16 S0: src_nolit, B16 S1: src_simple, B16

D.u[31:16] = (S0.u[31:16] $\geq$ S1.u[31:16]) ? S0.u[31:16] : S1.u[31:16] . D.u[15:0] = (S0.u[15:0] $\geq$ S1.u[15:0]) ? S0.u[15:0] : S1.u[15:0] .

Flags: SEN_VOP2 , OPF_CACGRP2, OPF_OPSEL_VOP3P

v_pk_min_f16 *vdst*, *src_0*, *src_1*
D0: vgpr, F16 S0: src_nolit, F16 S1: src_simple, F16
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]) .
Flags: SEN_VOP2 , OPF_CACGRP2, OPF_OPSEL_VOP3P

v_pk_min_i16 *vdst*, *src_0*, *src_1*
D0: vgpr, B16 S0: src_nolit, B16 S1: src_simple, B16
D.i[31:16] = (S0.i[31:16] < S1.i[31:16]) ? S0.i[31:16] : S1.i[31:16] . D.i[15:0] = (S0.i[15:0] < S1.i[15:0]) ? S0.i[15:0] : S1.i[15:0]
Flags: SEN_VOP2 , OPF_CACGRP2, OPF_OPSEL_VOP3P

v_pk_min_u16 *vdst*, *src_0*, *src_1*
D0: vgpr, B16 S0: src_nolit, B16 S1: src_simple, B16
D.u[31:16] = (S0.u[31:16] < S1.u[31:16]) ? S0.u[31:16] : S1.u[31:16] . D.u[15:0] = (S0.u[15:0] < S1.u[15:0]) ? S0.u[15:0] : S1.u[15:0] .
Flags: SEN_VOP2 , OPF_CACGRP2, OPF_OPSEL_VOP3P

v_pk_mul_f16 *vdst*, *src_0*, *src_1*
D0: vgpr, F16 S0: src_nolit, F16 S1: src_simple, F16
D.f[31:16] = S0.f[31:16] * S1.f[31:16] . D.f[15:0] = S0.f[15:0] * S1.f[15:0] .
Flags: SEN_VOP2 , OPF_CACGRP1, OPF_OPSEL_VOP3P

v_pk_mul_lo_u16 *vdst*, *src_0*, *src_1*
D0: vgpr, B16 S0: src_nolit, B16 S1: src_simple, B16
D.u[31:16] = S0.u[31:16] * S1.u[31:16] . D.u[15:0] = S0.u[15:0] * S1.u[15:0] .
Flags: SEN_VOP2 , OPF_CACGRP1, OPF_OPSEL_VOP3P

v_pk_sub_i16 *vdst*, *src_0*, *src_1*
D0: vgpr, B16 S0: src_nolit, B16 S1: src_simple, B16
D.i[31:16] = S0.i[31:16] - S1.i[31:16] . D.i[15:0] = S0.i[15:0] - S1.i[15:0] .
Flags: SEN_VOP2 , OPF_CACGRP2, OPF_OPSEL_VOP3P

v_pk_sub_u16 *vdst*, *src_0*, *src_1*
D0: vgpr, B16 S0: src_nolit, B16 S1: src_simple, B16
D.u[31:16] = S0.u[31:16] - S1.u[31:16] . D.u[15:0] = S0.u[15:0] - S1.u[15:0] .
Flags: SEN_VOP2 , OPF_CACGRP2, OPF_OPSEL_VOP3P

12.1 Notes for Encoding VOP3P

`op_sel:[src0, src1, src2, dst]`
Also available in VOP3 encoding. Controls if high (1) or low (0) bits of operands are used for lower half of destination. dst input is only required for vop3 ops, and src2 is not required in vop2 sub-encoded ops. Default 0.

`op_sel_hi:[src0, src1, src2]`
: Also available in VOP3 encoding. Controls if high (1) or low (0) bits of operands are used for upper half of destination. src2 is not required in vop2 sub-encoded ops. Default 1.

`neg_lo:[src0, src1, src2]`
: Controls if operands are negated for calculation of lower half of destination. src2 is not required in vop2 sub-encoded ops. Default 0.

`neg_hi:[src0, src1, src2]`
: Controls if operands are negated for calculation of upper half of destination. src2 is not required in vop2 sub-encoded ops. Default 0.

`clamp:{0,1}`
: Clamp output. Default 0. Can also write `noclamp`.

;

13 Encoding VOP3

Vector ALU operations requiring three sources or special modifiers; also some obscure opcodes. Almost all VOP1, VOP2, VOPC and VINTRP opcodes can be promoted to VOP3 to use input and output modifiers.

v_add3_u32 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, U32 S0: src_nolit, U32 S1: src_simple, U32 S2: src_simple, U32

```
D.u = S0.u + S1.u + S2.u.
```

Flags: OPF_CACGRP1

v_add_f64 *vdst*[2], *src_0*[2], *src_1*[2]
D0: vgpr, F64 S0: src_nolit, F64 S1: src_simple, F64

```
D.d = S0.d + S1.d.
```

0.5ULP precision, denormals are supported.

Flags: SEN_VOP2 , OPF_CACGRP2, OPF_DPFP

v_add_i16 *vdst*, *src_0*, *src_1*
D0: vgpr, I16 S0: src_nolit, I16 S1: src_simple, I16

```
D.i16 = S0.i16 + S1.i16.
```

Supports saturation (signed 16-bit integer domain).

Flags: SEN_VOP2 , OPF_OPSEL

v_add_i32 *vdst*, *src_0*, *src_1*
D0: vgpr, I32 S0: src_nolit, I32 S1: src_simple, I32

```
D.i = S0.i + S1.i.
```

Supports saturation (signed 32-bit integer domain).

Flags: SEN_VOP2 , OPF_CACGRP2

v_add_lshl_u32 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, U32 S0: src_nolit, U32 S1: src_simple, U32 S2: src_simple, U32

```
D.u = (S0.u + S1.u) << S2.u[4:0].
```

Flags: OPF_CACGRP1

v_alignbit_b32 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, U32 S0: src_nolit, U32 S1: src_simple, U32 S2: src_simple, U32

```
D.u = ({S0,S1} >> S2.u[4:0]) & 0xffffffff.
```

Flags: OPF_CACGRP1, OPF_OPSEL

v_alignbyte_b32 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, U32 S0: src_nolit, U32 S1: src_simple, U32 S2: src_simple, U32

```
D.u = ({S0,S1} >> (8*S2.u[4:0])) & 0xffffffff.
```

Flags: OPF_CACGRP1, OPF_OPSEL

v_and_or_b32 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, U32 S0: src_nolit, U32 S1: src_simple, U32 S2: src_simple, U32

```
D.u = (S0.u & S1.u) | S2.u.
```

Flags: OPF_CACGRP2

v_ashrrev_i64 *vdst*[2], *src_0*, *src_1*[2]
D0: vgpr, I64 S0: src_simple, U32 S1: src_simple, I64

```
D.u64 = signext(S1.u64) >> S0.u[5:0].
```

SQ translates this to an internal SP opcode.

Flags: SEN_VOP2 , OPF_CACGRP2, OPF_SQXLATE

v_bcnt_u32_b32 *vdst*, *src_0*, *src_1*
D0: vgpr, U32 S0: src_nolit, U32 S1: src_simple, U32

```
D.u = 0;
for i in 0 . . .  31 do
       D.u += (S0.u[i] == 1 ? 1 : 0);
endfor.
```

Bit count.

Flags: SEN_VOP2 , OPF_CACGRP2

v_bfe_i32 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, I32 S0: src_nolit, I32 S1: src_simple, U32 S2: src_simple, U32

```
D.i = (S0.i >> S1.u[4:0]) & ((1 << S2.u[4:0]) - 1).
```

Bitfield extract with S0 = data, S1 = field_offset, S2 = field_width.

Flags: OPF_CACGRP2

v_bfe_u32 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, U32 S0: src_nolit, U32 S1: src_simple, U32 S2: src_simple, 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.

Flags: OPF_CACGRP2

v_bfi_b32 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, U32 S0: src_nolit, U32 S1: src_simple, U32 S2: src_simple, U32

```
D.u = (S0.u & S1.u) | (~S0.u & S2.u).
```

Bitfield insert.

Flags: OPF_CACGRP2

v_bfm_b32 *vdst*, *src_0*, *src_1*
D0: vgpr, U32 S0: src_nolit, U32 S1: src_simple, U32

```
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.

Flags: SEN_VOP2 , OPF_CACGRP2

v_cubeid_f32 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, F32 S0: src_nolit, F32 S1: src_simple, F32 S2: src_simple, F32

D.f = cubemap face ID ({0.0, 1.0, . . . , 5.0}). XYZ coordinate is given in (S0.f, S1.f, S2.f).

Flags: OPF_CACGRP2

v_cubema_f32 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, F32 S0: src_nolit, F32 S1: src_simple, F32 S2: src_simple, F32

D.f = 2.0 * cubemap major axis. XYZ coordinate is given in (S0.f, S1.f, S2.f).

Flags: OPF_CACGRP2

v_cubesc_f32 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, F32 S0: src_nolit, F32 S1: src_simple, F32 S2: src_simple, F32

D.f = cubemap S coordinate. XYZ coordinate is given in (S0.f, S1.f, S2.f).

Flags: OPF_CACGRP2

v_cubetc_f32 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, F32 S0: src_nolit, F32 S1: src_simple, F32 S2: src_simple, F32

D.f = cubemap T coordinate. XYZ coordinate is given in (S0.f, S1.f, S2.f).

Flags: OPF_CACGRP2

v_cvt_pk_i16_i32 *vdst*, *src_0*, *src_1*
D0: vgpr, B32 S0: src_nolit, B32 S1: src_simple, B32

```
D = {int32_to_int16(S1.i), int32_to_int16(S0.i)}.
```

Flags: SEN_VOP2 , OPF_CACGRP2

v_cvt_pk_u16_u32 *vdst*, *src_0*, *src_1*
D0: vgpr, B32 S0: src_nolit, B32 S1: src_simple, B32

```
D = {uint32_to_uint16(S1.u), uint32_to_uint16(S0.u)}.
```

Flags: SEN_VOP2 , OPF_CACGRP2

v_cvt_pk_u8_f32 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, B32 S0: src_nolit, F32 S1: src_simple, B32 S2: src_simple, B32

```
D.u = (S2.u & ~(0xff << (8 * S1.u[1:0])));
D.u = D.u | ((flt32_to_uint8(S0.f) & 0xff) << (8 * S1.u[1:0])).
```

Convert floating point value S0 to 8-bit unsigned integer and pack the result into byte S1 of dword S2.

Flags: OPF_CACGRP2

v_cvt_pkaccum_u8_f32 *vdst*, *src_0*, *src_1*
D0: vgpr, ↔, B32 S0: src_nolit, F32 S1: src_simple, B32

```
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. SQ translates to V_CVT_PK_U8_F32. Note: this opcode uses src_c to pass destination in as a source.

Flags: SEN_VOP2 , OPF_CACGRP2, OPF_DACCUM, OPF_SQXLATE

v_cvt_pknorm_i16_f16 *vdst*, *src_0*, *src_1*
D0: vgpr, B32 S0: src_nolit, F16 S1: src_simple, F16

```
D = {(snorm)S1.f16, (snorm)S0.f16}.
```

Flags: SEN_VOP2 , OPF_CACGRP2, OPF_OPSEL

v_cvt_pknorm_i16_f32 *vdst*, *src_0*, *src_1*
D0: vgpr, B32 S0: src_nolit, F32 S1: src_simple, F32

```
D = {(snorm)S1.f, (snorm)S0.f}.
```

Flags: SEN_VOP2 , OPF_CACGRP2

v_cvt_pknorm_u16_f16 *vdst*, *src_0*, *src_1*
D0: vgpr, B32 S0: src_nolit, F16 S1: src_simple, F16

```
D = {(unorm)S1.f16, (unorm)S0.f16}.
```

Flags: SEN_VOP2 , OPF_CACGRP2, OPF_OPSEL

v_cvt_pknorm_u16_f32 *vdst*, *src_0*, *src_1*
D0: vgpr, B32 S0: src_nolit, F32 S1: src_simple, F32

```
D = {(unorm)S1.f, (unorm)S0.f}.
```

Flags: SEN_VOP2 , OPF_CACGRP2

v_cvt_pkrtz_f16_f32 *vdst*, *src_0*, *src_1*
D0: vgpr, B32 S0: src_nolit, F32 S1: src_simple, 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.

Flags: SEN_VOP2 , OPF_CACGRP2

v_div_fixup_f16 *vdst*, *src_0*, *src_1*, *src_2*

D0: vgpr, F16 S0: src_nolit, F16 S1: src_simple, F16 S2: src_simple, 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.

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.

Flags: OPF_OPSEL

v_div_fixup_f32 *vdst*, *src_0*, *src_1*, *src_2*

D0: vgpr, F32 S0: src_nolit, F32 S1: src_simple, F32 S2: src_simple, F32

```
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 == 0 || abs(S2.f) == ±INF)
       // 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.

Flags: OPF_CACGRP2

v_div_fixup_f64 *vdst*[2], *src_0*[2], *src_1*[2], *src_2*[2]
D0: vgpr, F64 S0: src_nolit, F64 S1: src_simple, F64 S2: src_simple, F64

```
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.

Flags: OPF_CACGRP2, OPF_DPFP

v_div_fixup_legacy_f16 *vdst*, *src_0*, *src_1*, *src_2*

D0: vgpr, F16 S0: src_nolit, F16 S1: src_simple, F16 S2: src_simple, 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.

Flags: OPF_OPSEL

v_div_fmas_f32 *vdst*, *src_0*, *src_1*, *src_2*

D0: vgpr, F32 S0: src_nolit, F32 S1: src_simple, F32 S2: src_simple, 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 never flushed, but output flushing is allowed.

Flags: OPF_CACGRP1, OPF_FMAS, OPF_RDVCC

v_div_fmas_f64 *vdst*[2], *src_0*[2], *src_1*[2], *src_2*[2]
D0: vgpr, F64 S0: src_nolit, F64 S1: src_simple, F64 S2: src_simple, 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 never flushed, but output flushing is allowed.

Flags: OPF_CACGRP0, OPF_DPFP, OPF_FMAS, OPF_RDVCC

v_div_scale_f32 *vdst*, *vcc*[2], *src_0*, *src_1*, *src_2*

D0: vgpr, F32 D1: vcc, B64 S0: src_nolit, F32 S1: src_simple, F32 S2: src_simple, 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).

Flags: OPF_CACGRP2, OPF_VCCD

v_div_scale_f64 *vdst*[2], *vcc*[2], *src_0*[2], *src_1*[2],
D0: vgpr, F64 D1: vcc, B64 S0: src_nolit, F64 S1: src_simple, F64
... *src_2*[2]
S2: src_simple, F64

```
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 producses a VCC flag for post-scaling of the quotient (using V_DIV_FMAS_F64).

Flags: OPF_CACGRP2, OPF_DPFP, OPF_VCCD

v_fma_f16 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, F16 S0: src_nolit, F16 S1: src_simple, F16 S2: src_simple, F16

```
D.f16 = S0.f16 * 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.

Flags: OPF_CACGRP2, OPF_OPSEL

v_fma_f32 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, F32 S0: src_nolit, F32 S1: src_simple, F32 S2: src_simple, F32

```
D.f = S0.f * S1.f + S2.f.
```

Fused single precision multiply add. 0.5ULP accuracy, denormals are supported.

Flags: OPF_CACGRP1

v_fma_f64 *vdst*[2], *src_0*[2], *src_1*[2], *src_2*[2]
D0: vgpr, F64 S0: src_nolit, F64 S1: src_simple, F64 S2: src_simple, F64

```
D.d = S0.d * S1.d + S2.d.
```

Fused double precision multiply add. 0.5ULP precision, denormals are supported.

Flags: OPF_CACGRP0, OPF_DPFP

v_fma_legacy_f16 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, F16 S0: src_nolit, F16 S1: src_simple, F16 S2: src_simple, F16

```
D.f16 = S0.f16 * S1.f16 + S2.f16.
```

Fused half precision multiply add.

Flags: OPF_CACGRP2, OPF_OPSEL

v_interp_p1ll_f16 *vgpr_dst*, *vgpr_ij*, *attr*
D0: vgpr, F32 S0: src_vgpr, F32 S1: attr, F16

```
D.f32 = P10.f16 * S0.f32 + P0.f16.
```

'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.

Flags: ASIC_32BANK_LDS, OPF_CACGRP1, OPF_INTERP

v_interp_p1lv_f16 *vgpr_dst*, *vgpr_ij*, *attr*, *vgpr_add*
D0: vgpr, F32 S0: src_vgpr, F32 S1: attr, F16 S2: src_vgpr, F16

```
D.f32 = P10.f16 * S0.f32 + (S2.u32 >> (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.

Flags: OPF_CACGRP1, OPF_INTERP

v_interp_p2_f16 *vgpr_dst*, *vgpr_ij*, *attr*, *vgpr_add*
D0: vgpr, F16 S0: src_vgpr, F32 S1: attr, F16 S2: src_vgpr, F32

```
D.f16 = P20.f16 * S0.f32 + 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.

Flags: OPF_CACGRP1, OPF_INTERP, OPF_OPSEL

v_interp_p2_legacy_f16 *vgpr_dst*, *vgpr_ij*, *attr*, *vgpr_add*
D0: vgpr, F16 S0: src_vgpr, F32 S1: attr, F16 S2: src_vgpr, F32

```
D.f16 = P20.f16 * S0.f32 + S2.f32.
```

Final computation. attr_word selects LDS high or low 16bits. Used for both 16- and 32-bank LDS. Result is always 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.

Flags: OPF_CACGRP1, OPF_INTERP, OPF_OPSEL

v_ldexp_f32 *vdst*, *src_0*, *src_1*
D0: vgpr, F32 S0: src_nolit, F32 S1: src_simple, I32

```
D.f = S0.f * (2 ** S1.i).
```

Flags: SEN_VOP2 , OPF_CACGRP2

v_ldexp_f64 *vdst*[2], *src_0*[2], *src_1*
D0: vgpr, F64 S0: src_nolit, F64 S1: src_simple, I32

```
D.d = S0.d * (2 ** S1.i).
```

Flags: SEN_VOP2 , OPF_CACGRP2

v_lerp_u8 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, U32 S0: src_nolit, B32 S1: src_simple, B32 S2: src_simple, B32

```
D.u = ((S0.u[31:24] + S1.u[31:24] + S2.u[24]) >> 1) << 24
D.u += ((S0.u[23:16] + S1.u[23:16] + S2.u[16]) >> 1) << 16;
D.u += ((S0.u[15:8] + S1.u[15:8] + S2.u[8]) >> 1) << 8;
D.u += ((S0.u[7:0] + S1.u[7:0] + S2.u[0]) >> 1).
```

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.

Flags: OPF_CACGRP2

v_lshl_add_u32 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, U32 S0: src_nolit, U32 S1: src_simple, U32 S2: src_simple, U32

```
D.u = (S0.u << S1.u[4:0]) + S2.u.
```

Flags: OPF_CACGRP1

v_lshl_or_b32 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, U32 S0: src_nolit, U32 S1: src_simple, U32 S2: src_simple, U32

```
D.u = (S0.u << S1.u[4:0]) | S2.u.
```

Flags: OPF_CACGRP2

v_lshlrev_b64 *vdst*[2], *src_0*, *src_1*[2]
D0: vgpr, U64 S0: src_simple, U32 S1: src_simple, U64

```
D.u64 = S1.u64 << S0.u[5:0].
```

SQ translates this to an internal SP opcode.

Flags: SEN_VOP2 , OPF_CACGRP2, OPF_SQXLATE

v_lshrrev_b64 *vdst*[2], *src_0*, *src_1*[2]
D0: vgpr, U64 S0: src_simple, U32 S1: src_simple, U64

```
D.u64 = S1.u64 >> S0.u[5:0].
```

SQ translates this to an internal SP opcode.

Flags: SEN_VOP2 , OPF_CACGRP2, OPF_SQXLATE

v_mad_f16 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, F16 S0: src_nolit, F16 S1: src_simple, F16 S2: src_simple, F16

```
D.f16 = S0.f16 * S1.f16 + S2.f16.
```

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.

Flags: OPF_CACGRP2, OPF_OPSEL

v_mad_f32 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, F32 S0: src_nolit, F32 S1: src_simple, F32 S2: src_simple, F32

```
D.f = S0.f * S1.f + S2.f.
```

1ULP accuracy, denormals are flushed.

Flags: OPF_CACGRP1

v_mad_i16 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, I16 S0: src_nolit, I16 S1: src_simple, I16 S2: src_simple, I16

```
D.i16 = S0.i16 * 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.

Flags: OPF_CACGRP2, OPF_OPSEL

v_mad_i32_i16 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, I32 S0: src_nolit, I16 S1: src_simple, I16 S2: src_simple, I32

```
D.i32 = S0.i16 * S1.i16 + S2.i32.
```

Flags: OPF_CACGRP2, OPF_OPSEL

v_mad_i32_i24 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, I32 S0: src_nolit, I32 S1: src_simple, I32 S2: src_simple, I32

```
D.i = S0.i[23:0] * S1.i[23:0] + S2.i.
```

Flags: OPF_CACGRP1

v_mad_i64_i32 *vdst*[2], *carryout*[2], *src_0*, *src_1*, ...
D0: vgpr, B32 D1: sreg, B64 S0: src_nolit, B32 S1: src_simple, B32
src_2[2]
S2: src_simple, B32

```
{vcc_out,D.i64} = S0.i32 * S1.i32 + S2.i64.
```

Flags: OPF_CACGRP1, OPF_VCCD

v_mad_legacy_f16 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, F16 S0: src_nolit, F16 S1: src_simple, F16 S2: src_simple, 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.

Flags: OPF_CACGRP2, OPF_OPSEL

v_mad_legacy_f32 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, F32 S0: src_nolit, F32 S1: src_simple, F32 S2: src_simple, F32

```
D.f = S0.f * S1.f + S2.f. // DX9 rules, 0.0 * x = 0.0
```

Flags: OPF_CACGRP1

v_mad_legacy_i16 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, B16 S0: src_nolit, B16 S1: src_simple, B16 S2: src_simple, B16

```
D.i16 = S0.i16 * 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 written as 0 (this is different from V_MAD_I16).
If op_sel[3] is 1 Result is written to 16 MSBs of destination VGPR and lo 16 bits are preserved.

Flags: OPF_CACGRP2, OPF_OPSEL

v_mad_legacy_u16 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, B16 S0: src_nolit, B16 S1: src_simple, B16 S2: src_simple, B16

```
D.u16 = S0.u16 * 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 written as 0 (this is different from V_MAD_U16).
If op_sel[3] is 1 Result is written to 16 MSBs of destination VGPR and lo 16 bits are preserved.

Flags: OPF_CACGRP2, OPF_OPSEL

v_mad_u16 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, U16 S0: src_nolit, U16 S1: src_simple, U16 S2: src_simple, U16

```
D.u16 = S0.u16 * 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.

Flags: OPF_CACGRP2, OPF_OPSEL

v_mad_u32_u16 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, U32 S0: src_nolit, U16 S1: src_simple, U16 S2: src_simple, U32

```
D.u32 = S0.u16 * S1.u16 + S2.u32.
```

Flags: OPF_CACGRP2, OPF_OPSEL

v_mad_u32_u24 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, U32 S0: src_nolit, U32 S1: src_simple, U32 S2: src_simple, U32

```
D.u = S0.u[23:0] * S1.u[23:0] + S2.u.
```

Flags: OPF_CACGRP1

v_mad_u64_u32 *vdst*[2], *carryout*[2], *src_0*, *src_1*,
D0: vgpr, B32 D1: sreg, B64 S0: src_nolit, B32 S1: src_simple, B32
... *src_2*[2]
S2: src_simple, B32

```
{vcc_out,D.u64} = S0.u32 * S1.u32 + S2.u64.
```

Flags: OPF_CACGRP1, OPF_VCCD

v_max3_f16 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, F16 S0: src_nolit, F16 S1: src_simple, F16 S2: src_simple, F16

```
D.f16 = V_MAX_F16(V_MAX_F16(S0.f16, S1.f16), S2.f16).
```

Flags: OPF_CACGRP2, OPF_OPSEL

v_max3_f32 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, F32 S0: src_nolit, F32 S1: src_simple, F32 S2: src_simple, F32

```
D.f = V_MAX_F32(V_MAX_F32(S0.f, S1.f), S2.f).
```

Flags: OPF_CACGRP1

v_max3_i16 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, I16 S0: src_nolit, I16 S1: src_simple, I16 S2: src_simple, I16

```
D.i16 = V_MAX_I16(V_MAX_I16(S0.i16, S1.i16), S2.i16).
```

Flags: OPF_CACGRP2, OPF_OPSEL

v_max3_i32 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, I32 S0: src_nolit, I32 S1: src_simple, I32 S2: src_simple, I32

```
D.i = V_MAX_I32(V_MAX_I32(S0.i, S1.i), S2.i).
```

Flags: OPF_CACGRP1

v_max3_u16 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, U16 S0: src_nolit, U16 S1: src_simple, U16 S2: src_simple, U16

```
D.u16 = V_MAX_U16(V_MAX_U16(S0.u16, S1.u16), S2.u16).
```

Flags: OPF_CACGRP2, OPF_OPSEL

v_max3_u32 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, U32 S0: src_nolit, U32 S1: src_simple, U32 S2: src_simple, U32

```
D.u = V_MAX_U32(V_MAX_U32(S0.u, S1.u), S2.u).
```

Flags: OPF_CACGRP1

v_max_f64 *vdst*[2], *src_0*[2], *src_1*[2]
D0: vgpr, F64 S0: src_nolit, F64 S1: src_simple, 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 = S0.d;
else if (S0.d == -0.0 && S1.d == +0.0)
        D.d = S1.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.
```

Flags: SEN_VOP2 , OPF_CACGRP2

v_mbcnt_hi_u32_b32 *vdst*, *src_0*, *src_1*
D0: vgpr, U32 S0: src_nolit, U32 S1: src_simple, U32

```
ThreadMask = (1LL << ThreadPosition) - 1;
MaskedValue = (S0.u & 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.

Example to compute each thread's position in 0..63:

```
v_mbcnt_lo_u32_b32 v0, -1, 0
v_mbcnt_hi_u32_b32 v0, -1, v0
// v0 now contains ThreadPosition
```

Flags: SEN_VOP2 , OPF_CACGRP2

v_mbcnt_lo_u32_b32 *vdst*, *src_0*, *src_1*
D0: vgpr, U32 S0: src_nolit, U32 S1: src_simple, U32

```
ThreadMask = (1LL << ThreadPosition) - 1;
MaskedValue = (S0.u & 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.

Flags: SEN_VOP2 , OPF_CACGRP2

v_med3_f16 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, F16 S0: src_nolit, F16 S1: src_simple, F16 S2: src_simple, F16

```
if (isNan(S0.f16) || isNan(S1.f16) || isNan(S2.f16))
        D.f16 = V_MIN3_F16(S0.f16, S1.f16, S2.f16);
else if (V_MAX3_F16(S0.f16, S1.f16, S2.f16) == S0.f16)
        D.f16 = V_MAX_F16(S1.f16, S2.f16);
else if (V_MAX3_F16(S0.f16, S1.f16, S2.f16) == S1.f16)
        D.f16 = V_MAX_F16(S0.f16, S2.f16);
else
        D.f16 = V_MAX_F16(S0.f16, S1.f16);
endif.
```

Flags: OPF_CACGRP2, OPF_OPSEL

v_med3_f32 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, F32 S0: src_nolit, F32 S1: src_simple, F32 S2: src_simple, F32

```
if (isNan(S0.f) || isNan(S1.f) || isNan(S2.f))
        D.f = V_MIN3_F32(S0.f, S1.f, S2.f);
else if (V_MAX3_F32(S0.f, S1.f, S2.f) == S0.f)
        D.f = V_MAX_F32(S1.f, S2.f);
else if (V_MAX3_F32(S0.f, S1.f, S2.f) == S1.f)
        D.f = V_MAX_F32(S0.f, S2.f);
else
        D.f = V_MAX_F32(S0.f, S1.f);
endif.
```

Flags: OPF_CACGRP1

v_med3_i16 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, I16 S0: src_nolit, I16 S1: src_simple, I16 S2: src_simple, I16

```
if (V_MAX3_I16(S0.i16, S1.i16, S2.i16) == S0.i16)
        D.i16 = V_MAX_I16(S1.i16, S2.i16);
else if (V_MAX3_I16(S0.i16, S1.i16, S2.i16) == S1.i16)
        D.i16 = V_MAX_I16(S0.i16, S2.i16);
else
        D.i16 = V_MAX_I16(S0.i16, S1.i16);
endif.
```

Flags: OPF_CACGRP2, OPF_OPSEL

v_med3_i32 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, I32 S0: src_nolit, I32 S1: src_simple, I32 S2: src_simple, I32

```
if (V_MAX3_I32(S0.i, S1.i, S2.i) == S0.i)
        D.i = V_MAX_I32(S1.i, S2.i);
else if (V_MAX3_I32(S0.i, S1.i, S2.i) == S1.i)
        D.i = V_MAX_I32(S0.i, S2.i);
else
        D.i = V_MAX_I32(S0.i, S1.i);
endif.
```

Flags: OPF_CACGRP1

v_med3_u16 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, U16 S0: src_nolit, U16 S1: src_simple, U16 S2: src_simple, U16

```
if (V_MAX3_U16(S0.u16, S1.u16, S2.u16) == S0.u16)
       D.u16 = V_MAX_U16(S1.u16, S2.u16);
else if (V_MAX3_U16(S0.u16, S1.u16, S2.u16) == S1.u16)
       D.u16 = V_MAX_U16(S0.u16, S2.u16);
else
       D.u16 = V_MAX_U16(S0.u16, S1.u16);
endif.
```

Flags: OPF_CACGRP2, OPF_OPSEL

v_med3_u32 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, U32 S0: src_nolit, U32 S1: src_simple, U32 S2: src_simple, U32

```
if (V_MAX3_U32(S0.u, S1.u, S2.u) == S0.u)
       D.u = V_MAX_U32(S1.u, S2.u);
else if (V_MAX3_U32(S0.u, S1.u, S2.u) == S1.u)
       D.u = V_MAX_U32(S0.u, S2.u);
else
       D.u = V_MAX_U32(S0.u, S1.u);
endif.
```

Flags: OPF_CACGRP1

v_min3_f16 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, F16 S0: src_nolit, F16 S1: src_simple, F16 S2: src_simple, F16

```
D.f16 = V_MIN_F16(V_MIN_F16(S0.f16, S1.f16), S2.f16).
```

Flags: OPF_CACGRP2, OPF_OPSEL

v_min3_f32 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, F32 S0: src_nolit, F32 S1: src_simple, F32 S2: src_simple, F32

```
D.f = V_MIN_F32(V_MIN_F32(S0.f, S1.f), S2.f).
```

Flags: OPF_CACGRP1

v_min3_i16 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, I16 S0: src_nolit, I16 S1: src_simple, I16 S2: src_simple, I16

```
D.i16 = V_MIN_I16(V_MIN_I16(S0.i16, S1.i16), S2.i16).
```

Flags: OPF_CACGRP2, OPF_OPSEL

v_min3_i32 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, I32 S0: src_nolit, I32 S1: src_simple, I32 S2: src_simple, I32

```
D.i = V_MIN_I32(V_MIN_I32(S0.i, S1.i), S2.i).
```

Flags: OPF_CACGRP1

v_min3_u16 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, U16 S0: src_nolit, U16 S1: src_simple, U16 S2: src_simple, U16

```
D.u16 = V_MIN_U16(V_MIN_U16(S0.u16, S1.u16), S2.u16).
```

Flags: OPF_CACGRP2, OPF_OPSEL

v_min3_u32 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, U32 S0: src_nolit, U32 S1: src_simple, U32 S2: src_simple, U32

```
D.u = V_MIN_U32(V_MIN_U32(S0.u, S1.u), S2.u).
```

Flags: OPF_CACGRP1

v_min_f64 *vdst*[2], *src_0*[2], *src_1*[2]
D0: vgpr, F64 S0: src_nolit, F64 S1: src_simple, 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.
```

Flags: SEN_VOP2 , OPF_CACGRP2

v_mqsad_pk_u16_u8 *vdst*[2], *src_0*[2], *src_1*, *src_2*[2]
D0: vgpr, B64 S0: src_nolit, B64 S1: src_simple, B32 S2: src_simple, B64

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])

Flags: OPF_CACGRP1

v_mqsad_u32_u8 *vdst*[4], *src_0*[2], *src_1*, *src_2*[4]
D0: vgpr, B128 S0: src_nolit, B64 S1: src_simple, B32 S2: src_vgpr, B128

D.u128 = Masked Quad-Byte SAD with 32-bit accum_lo/hi(S0.u[63:0], S1.u[31:0], S2.u[127:0])

Flags: OPF_CACGRP1

v_msad_u8 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, U32 S0: src_nolit, B32 S1: src_simple, B32 S2: src_simple, B32

D.u = Masked Byte SAD with accum_lo(S0.u, S1.u, S2.u).

Flags: OPF_CACGRP1

v_mul_f64 *vdst*[2], *src_0*[2], *src_1*[2]
D0: vgpr, F64 S0: src_nolit, F64 S1: src_simple, F64

```
D.d = S0.d * S1.d.
```

0.5ULP precision, denormals are supported.

Flags: SEN_VOP2 , OPF_CACGRP0, OPF_DPFP

v_mul_hi_i32 *vdst*, *src_0*, *src_1*
D0: vgpr, I32 S0: src_nolit, I32 S1: src_simple, I32

```
D.i = (S0.i * S1.i) >> 32.
```

Flags: SEN_VOP2 , OPF_CACGRP2

v_mul_hi_u32 *vdst*, *src_0*, *src_1*
D0: vgpr, U32 S0: src_nolit, U32 S1: src_simple, U32

```
D.u = (S0.u * S1.u) >> 32.
```

Flags: SEN_VOP2 , OPF_CACGRP2

v_mul_lo_u32 *vdst*, *src_0*, *src_1*
D0: vgpr, U32 S0: src_nolit, U32 S1: src_simple, U32

```
D.u = S0.u * S1.u.
```

Flags: SEN_VOP2 , OPF_CACGRP2

v_or3_b32 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, U32 S0: src_nolit, U32 S1: src_simple, U32 S2: src_simple, U32

```
D.u = S0.u | S1.u | S2.u.
```

Flags: OPF_CACGRP2

v_pack_b32_f16 *vdst*, *src_0*, *src_1*
D0: vgpr, B32 S0: src_nolit, F16 S1: src_simple, F16

```
D[31:16].f16 = S1.f16;
D[15:0].f16 = S0.f16.
```

Flags: SEN_VOP2 , OPF_CACGRP2, OPF_OPSEL

v_perm_b32 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, B32 S0: src_nolit, B32 S1: src_simple, B32 S2: src_simple, B32

```
D.u[31:24] = byte_permute({S0.u, S1.u}, S2.u[31:24]);
D.u[23:16] = byte_permute({S0.u, S1.u}, S2.u[23:16]);
D.u[15:8] = byte_permute({S0.u, S1.u}, S2.u[15:8]);
D.u[7:0] = byte_permute({S0.u, S1.u}, S2.u[7:0]);

byte permute(byte in[8], byte sel) {
       if(sel≥13) then return 0xff;
       elsif(sel==12) then return 0x00;
       elsif(sel==11) then return in[7][7] * 0xff;
       elsif(sel==10) then return in[5][7] * 0xff;
       elsif(sel==9) then return in[3][7] * 0xff;
       elsif(sel==8) then return in[1][7] * 0xff;
       else return in[sel];
}
```

Byte permute.

Flags: OPF_CACGRP2

v_qsad_pk_u16_u8 *vdst*[2], *src_0*[2], *src_1*, *src_2*[2]
D0: vgpr, B64 S0: src_nolit, B64 S1: src_simple, B32 S2: src_simple, B64

D.u = Quad-Byte SAD with 16-bit packed accum_lo/hi(S0.u[63:0], S1.u[31:0], S2.u[63:0])

Flags: OPF_CACGRP1

v_readlane_b32 *sdst*, *vsrc_0*, *ssrc_1*
D0: sreg_novcc, B32 S0: vgpr_or_lds, B32 S1: ssrc_lanesel, 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.

v_sad_hi_u8 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, U32 S0: src_nolit, B32 S1: src_simple, B32 S2: src_simple, U32

```
D.u = (SAD_U8(S0, S1, 0) << 16) + S2.u.
```

Sum of absolute differences with accumulation, overflow is lost.

Flags: OPF_CACGRP1

v_sad_u16 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, U32 S0: src_nolit, B32 S1: src_simple, B32 S2: src_simple, U32

```
D.u = abs(S0.i[31:16] - S1.i[31:16]) + abs(S0.i[15:0] - S1.i[15:0]) + S2.u.
```

Word SAD with accumulation.

Flags: OPF_CACGRP1

v_sad_u32 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, U32 S0: src_nolit, I32 S1: src_simple, I32 S2: src_simple, U32

```
D.u = abs(S0.i - S1.i) + S2.u.
```

Dword SAD with accumulation.

Flags: OPF_CACGRP1

v_sad_u8 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, U32 S0: src_nolit, B32 S1: src_simple, B32 S2: src_simple, B32

```
D.u = abs(S0.i[31:24] - S1.i[31:24]);
D.u += abs(S0.i[23:16] - S1.i[23:16]);
D.u += abs(S0.i[15:8] - S1.i[15:8]);
D.u += abs(S0.i[7:0] - S1.i[7:0]) + S2.u.
```

Sum of absolute differences with accumulation, overflow into upper bits is allowed.

Flags: OPF_CACGRP1

v_sub_i16 *vdst*, *src_0*, *src_1*
D0: vgpr, I16 S0: src_nolit, I16 S1: src_simple, I16

```
D.i16 = S0.i16 - S1.i16.
```

Supports saturation (signed 16-bit integer domain).

Flags: SEN_VOP2 , OPF_OPSEL

v_sub_i32 *vdst*, *src_0*, *src_1*
D0: vgpr, I32 S0: src_nolit, I32 S1: src_simple, I32

```
D.i = S0.i - S1.i.
```

Supports saturation (signed 32-bit integer domain).

Flags: SEN_VOP2 , OPF_CACGRP2

v_trig_preop_f64 *vdst*[2], *src_0*[2], *src_1*
D0: vgpr, F64 S0: src_nolit, F64 S1: src_simple, B32

```
shift = S1.u * 53;
if exponent(S0.d) > 1077 then
       shift += exponent(S0.d) - 1077;
endif
result = (double) ((2/PI[1200:0] << shift) & 0x1fffff_ffffffff);
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 is always round-to-zero. Large inputs (exp > 1968) are scaled to avoid loss of precision through denormalization.

Flags: SEN_VOP2

v_writelane_b32 *vdst*, *ssrc_0*, *ssrc_1*
D0: vgpr, ↔, B32 S0: ssrc_nolit, B32 S1: ssrc_lanesel, B32

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. SQ translates to V_MOV_B32.

Flags: OPF_CACGRP2, OPF_SQXLATE

v_xad_u32 *vdst*, *src_0*, *src_1*, *src_2*
D0: vgpr, U32 S0: src_nolit, U32 S1: src_simple, U32 S2: src_simple, U32

```
D.u32 = (S0.u32 ^ S1.u32) + S2.u32.
```

No carryin/carryout and no saturation. This opcode exists to accelerate the SHA256 hash algorithm.

Flags: OPF_CACGRP2

13.1 Notes for Encoding VOP3

13.1.1 Input modifiers

Source operands may invoke the negation and absolute-value input modifiers with `-`*src* and `abs(`*src*`)`, respectively. For example,

```
v_add_f32 v0, v1, -v2           // Subtract v2 from v1
v_add_f32 v0, abs(v1), abs(v2)  // Take absolute value of both inputs
```

In general, negation and absolute value are only supported for floating point input operands (operands with a type of F16, F32 or F64); they are not supported for integer or untyped inputs.

13.1.2 Output modifiers

`mul:{1,2,4}`
: Set output modifier to multiply by N. Default 1, which leaves the result unmodified. This operation is only defined when the result is F16, F32 or F64.

`div:{1,2}`
: Set output modifier to divide by N. Default 1, which leaves the result unmodified. This operation is only defined when the result is F16, F32 or F64.

`clamp:{0,1}`
: Clamp (saturate) the output. Default 0. Can also write `noclamp`. Saturation is defined as follows.

  For I16, I32, I64 results: if the result exceeds SHRT_MAX/INT_MAX/LLONG_MAX it will be clamped to the most positive representable value; if the result is below SHRT_MIN/INT_MIN/LLONG_MIN it will be clamped to the most negative representable value.

  For U16, U32, U64 results: if the result exceeds USHRT_MAX/UINT_MAX/ULLONG_MAX it will be clamped to the most positive representable value; if the result is below 0 it will be clamped to 0.

  For F16, F32, F64 results: the result will be clamped to the interval [0.0, 1.0].

13.1.3 Interpolation operands and modifiers

vgpr_dst is a vector GPR to store result in, and use as accumulator source in certain interpolation operations.

vgpr_ij is a vector GPR to read i/j value from.

attr is `attr0.x` through `attr63.w`, parameter attribute and channel to be interpolated.

param is `p10`, `p20` or `p0`.

For 16-bit interpolation it is necessary to specify whether we are operating on the high or low word of the attribute. For this, two new modifiers are provided:

`high`
: Interpolate using the high 16 bits of the attribute.

`low`
: Interpolate using the low 16 bits of the attribute. Default.

13.1.4 Other modifiers

`vop3:{0,1}`
: Force VOP3 encoding even if instruction can be represented in smaller encoding. Default 0. Can also write `novop3`. Note that even if this modifier is not set, an opcode will still use the VOP3 encoding if the operands or modifiers given cannot be expressed in a smaller encoding.

14 Encoding DS

Local and global data share operations.

ds_add_f32 *vgpr_a*, *vgpr_d0*
S0: vgpr, B32 S1: vgpr, F32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] += DATA;
RETURN_DATA = tmp.
```

Floating point add that handles NaN/INF/denormal values.

Flags: OPF_DS1A, OPF_DS1D, OPF_DS32FLT, OPF_MEM_ATOMIC

ds_add_rtn_f32 *vgpr_rtn*, *vgpr_a*, *vgpr_d0*
D0: vgpr, F32 S0: vgpr, B32 S1: vgpr, F32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] += DATA;
RETURN_DATA = tmp.
```

Floating point add that handles NaN/INF/denormal values.

Flags: OPF_DS1A, OPF_DS1D, OPF_DS32FLT, OPF_DSRTN, OPF_MEM_ATOMIC

ds_add_rtn_u32 *vgpr_rtn*, *vgpr_a*, *vgpr_d0*
D0: vgpr, B32 S0: vgpr, B32 S1: vgpr, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] += DATA;
RETURN_DATA = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_DSRTN, OPF_MEM_ATOMIC

ds_add_rtn_u64 *vgpr_rtn*[2], *vgpr_a*, *vgpr_d0*[2]
D0: vgpr, B64 S0: vgpr, B32 S1: vgpr, B64

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] += DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_DS64BIT, OPF_DSRTN, OPF_MEM_ATOMIC

ds_add_src2_f32 *vgpr_a*
S0: vgpr, B32

```
// 32bit
A = ADDR_BASE;
B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0});
MEM[A] = MEM[B] + MEM[A].

Float, handles NaN/INF/denorm.
```

Flags: OPF_DS1A, OPF_DS32FLT, OPF_MEM_ATOMIC

ds_add_src2_u32 *vgpr_a*
S0: vgpr, B32

```
// 32bit
A = ADDR_BASE;
B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0});
MEM[A] = MEM[A] + MEM[B].
```

Flags: OPF_DS1A, OPF_MEM_ATOMIC

ds_add_src2_u64 *vgpr_a*
S0: vgpr, B32

```
// 64bit
A = ADDR_BASE;
B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0});
MEM[A] = MEM[A] + MEM[B].
```

Flags: OPF_DS1A, OPF_DS64BIT, OPF_MEM_ATOMIC

ds_add_u32 *vgpr_a*, *vgpr_d0*
S0: vgpr, B32 S1: vgpr, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] += DATA;
RETURN_DATA = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_MEM_ATOMIC

ds_add_u64 *vgpr_a*, *vgpr_d0*[2]
S0: vgpr, B32 S1: vgpr, B64

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] += DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_DS64BIT, OPF_MEM_ATOMIC

ds_and_b32 *vgpr_a*, *vgpr_d0*
S0: vgpr, B32 S1: vgpr, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] &= DATA;
RETURN_DATA = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_MEM_ATOMIC

ds_and_b64 *vgpr_a*, *vgpr_d0*[2]
S0: vgpr, B32 S1: vgpr, B64

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] &= DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_DS64BIT, OPF_MEM_ATOMIC

ds_and_rtn_b32 *vgpr_rtn*, *vgpr_a*, *vgpr_d0*
D0: vgpr, B32 S0: vgpr, B32 S1: vgpr, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] &= DATA;
RETURN_DATA = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_DSRTN, OPF_MEM_ATOMIC

ds_and_rtn_b64 *vgpr_rtn*[2], *vgpr_a*, *vgpr_d0*[2]
D0: vgpr, B64 S0: vgpr, B32 S1: vgpr, B64

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] &= DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_DS64BIT, OPF_DSRTN, OPF_MEM_ATOMIC

ds_and_src2_b32 *vgpr_a*
S0: vgpr, B32

```
// 32bit
A = ADDR_BASE;
B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0});
MEM[A] = MEM[A] & MEM[B].
```

Flags: OPF_DS1A, OPF_MEM_ATOMIC

ds_and_src2_b64 *vgpr_a*
S0: vgpr, B32

```
// 64bit
A = ADDR_BASE;
B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0});
MEM[A] = MEM[A] & MEM[B].
```

Flags: OPF_DS1A, OPF_DS64BIT, OPF_MEM_ATOMIC

ds_append *vgpr_rtn*
D0: vgpr, B32

LDS & GDS. Add (count_bits(exec_mask)) to the value stored in DS memory at (M0.base + instr_offset). Return the pre-operation value to VGPRs.

Flags: OPF_DSRTN, OPF_MEM_ATOMIC, OPF_RDM0

ds_bpermute_b32 *vgpr_rtn*, *vgpr_a*, *vgpr_d0*
D0: vgpr, B32 S0: vgpr, B32 S1: vgpr, B32

```
// VGPR[index][thread_id] is the VGPR RAM
// VDST, ADDR and DATA0 are from the microcode DS encoding
tmp[0..63] = 0
for i in 0..63 do
        // ADDR needs to be divided by 4.
        // High-order bits are ignored.
        src_lane = floor((VGPR[ADDR][i] + OFFSET) / 4) mod 64
        // EXEC is applied to the source VGPR reads.
        next if !EXEC[src_lane]
        tmp[i] = VGPR[DATA0][src_lane]
endfor
// Copy data into destination VGPRs. Some source
// data may be broadcast to multiple lanes.
for i in 0..63 do
        next if !EXEC[i]
        VGPR[VDST][i] = tmp[i]
endfor
```

Backward permute. This does not access LDS memory and may be called even if no LDS memory is allocated to the wave. It uses LDS hardware to implement an arbitrary swizzle across threads in a wavefront.

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

Note that EXEC mask is applied to both VGPR read and write. If src_lane selects a disabled thread, zero will be returned.

See also DS_PERMUTE_B32.

Examples (simplified 4-thread wavefronts):

VGPR[SRC0] = { A, B, C, D }
VGPR[ADDR] = { 0, 0, 12, 4 }
EXEC = `0xF`, OFFSET = 0
VGPR[VDST] := { A, A, D, B }

VGPR[SRC0] = { A, B, C, D }
VGPR[ADDR] = { 0, 0, 12, 4 }
EXEC = `0xA`, OFFSET = 0
VGPR[VDST] := { -, 0, -, B }

Flags: OPF_DS1A, OPF_DS1D, OPF_DSRTN, OPF_MEM_ATOMIC

ds_cmpst_b32 *vgpr_a*, *vgpr_d0*, *vgpr_d1*
S0: vgpr, B32 S1: vgpr, B32 S2: vgpr, 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.

Flags: OPF_DS1A, OPF_DS2D, OPF_MEM_ATOMIC

ds_cmpst_b64 *vgpr_a*, *vgpr_d0*[2], *vgpr_d1*[2]
S0: vgpr, B32 S1: vgpr, B64 S2: vgpr, 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.

Flags: OPF_DS1A, OPF_DS2D, OPF_DS64BIT, OPF_MEM_ATOMIC

ds_cmpst_f32 *vgpr_a*, *vgpr_d0*, *vgpr_d1*
S0: vgpr, B32 S1: vgpr, F32 S2: vgpr, 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. Caution, the order of src and cmp are the *opposite* of the BUFFER_ATOMIC_FCMPSWAP opcode.

Flags: OPF_DS1A, OPF_DS2D, OPF_DS32FLT, OPF_MEM_ATOMIC

ds_cmpst_f64 *vgpr_a*, *vgpr_d0*[2], *vgpr_d1*[2]
S0: vgpr, B32 S1: vgpr, F64 S2: vgpr, F64

```
// 64bit
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. Caution, the order of src and cmp are the *opposite* of the BUFFER_ATOMIC_FCMPSWAP_X2 opcode.

Flags: OPF_DS1A, OPF_DS2D, OPF_DS64FLT, OPF_MEM_ATOMIC

ds_cmpst_rtn_b32 *vgpr_rtn*, *vgpr_a*, *vgpr_d0*, *vgpr_d1*
D0: vgpr, B32 S0: vgpr, B32 S1: vgpr, B32 S2: vgpr, 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.

Flags: OPF_DS1A, OPF_DS2D, OPF_DSRTN, OPF_MEM_ATOMIC

ds_cmpst_rtn_b64 *vgpr_rtn*[2], *vgpr_a*, *vgpr_d0*[2], *vgpr_d1*[2]
D0: vgpr, B64 S0: vgpr, B32 S1: vgpr, B64 S2: vgpr, 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.

Flags: OPF_DS1A, OPF_DS2D, OPF_DS64BIT, OPF_DSRTN, OPF_MEM_ATOMIC

ds_cmpst_rtn_f32 *vgpr_rtn*, *vgpr_a*, *vgpr_d0*, *vgpr_d1*
D0: vgpr, F32 S0: vgpr, B32 S1: vgpr, F32 S2: vgpr, 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. Caution, the order of src and cmp are the *opposite* of the BUFFER_ATOMIC_FCMPSWAP opcode.

Flags: OPF_DS1A, OPF_DS2D, OPF_DS32FLT, OPF_DSRTN, OPF_MEM_ATOMIC

ds_cmpst_rtn_f64 *vgpr_rtn*[2], *vgpr_a*, *vgpr_d0*[2], *vgpr_d1*[2]
D0: vgpr, F64 S0: vgpr, B32 S1: vgpr, F64 S2: vgpr, F64

```
// 64bit
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. Caution, the order of src and cmp are the *opposite* of the BUFFER_ATOMIC_FCMPSWAP_X2 opcode.

Flags: OPF_DS1A, OPF_DS2D, OPF_DS64FLT, OPF_DSRTN, OPF_MEM_ATOMIC

ds_condxchg32_rtn_b64 *vgpr_rtn*[2], *vgpr_a*, *vgpr_d0*[2]
D0: vgpr, B64 S0: vgpr, B32 S1: vgpr, B64

Conditional write exchange.

Flags: OPF_DS1A, OPF_DS1D, OPF_DS64BIT, OPF_DSRTN, OPF_MEM_ATOMIC

ds_consume *vgpr_rtn*
D0: vgpr, B32

LDS & GDS. Subtract (count_bits(exec_mask)) from the value stored in DS memory at (M0.base + instr_offset). Return the pre-operation value to VGPRs.

Flags: OPF_DSRTN, OPF_MEM_ATOMIC, OPF_RDM0

ds_dec_rtn_u32 *vgpr_rtn*, *vgpr_a*, *vgpr_d0*
D0: vgpr, B32 S0: vgpr, B32 S1: vgpr, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (tmp == 0 || tmp > DATA) ? DATA : tmp - 1; // unsigned compare
RETURN_DATA = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_DSRTN, OPF_MEM_ATOMIC

ds_dec_rtn_u64 *vgpr_rtn*[2], *vgpr_a*, *vgpr_d0*[2]
D0: vgpr, B64 S0: vgpr, B32 S1: vgpr, B64

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] = (tmp == 0 || tmp > DATA[0:1]) ? DATA[0:1] : tmp - 1; // unsigned compare
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_DS64BIT, OPF_DSRTN, OPF_MEM_ATOMIC

ds_dec_src2_u32 *vgpr_a*
S0: vgpr, B32

```
// 32bit
A = ADDR_BASE;
B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0});
MEM[A] = (MEM[A] == 0 || MEM[A] > MEM[B] ? MEM[B] : MEM[A] - 1).

Uint decrement.
```

Flags: OPF_DS1A, OPF_MEM_ATOMIC

ds_dec_src2_u64 *vgpr_a*
S0: vgpr, B32

```
// 64bit
A = ADDR_BASE;
B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0});
MEM[A] = (MEM[A] == 0 || MEM[A] > MEM[B] ? MEM[B] : MEM[A] - 1).

Uint decrement.
```

Flags: OPF_DS1A, OPF_DS64BIT, OPF_MEM_ATOMIC

ds_dec_u32 *vgpr_a*, *vgpr_d0*
S0: vgpr, B32 S1: vgpr, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (tmp == 0 || tmp > DATA) ? DATA : tmp - 1; // unsigned compare
RETURN_DATA = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_MEM_ATOMIC

ds_dec_u64 *vgpr_a*, *vgpr_d0*[2]
S0: vgpr, B32 S1: vgpr, B64

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] = (tmp == 0 || tmp > DATA[0:1]) ? DATA[0:1] : tmp - 1; // unsigned compare
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_DS64BIT, OPF_MEM_ATOMIC

ds_gws_barrier *vgpr_d0*
S0: vgpr, B32

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.

```
// 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);
value = DATA[thread: index];

// Input Decision Machine
state.type[rid] = BARRIER;
if(state[rid].counter ≤ 0) then
        thread[rid].flag = state[rid].flag;
        ENQUEUE;
        state[rid].flag = !state.flag;
        state[rid].counter = value;
        return rd_done;
else
        state[rid].counter -= 1;
        thread.flag = state[rid].flag;
        ENQUEUE;
endif.
```

Since the waves deliver the count for the next barrier, this function can have a different size barrier for each occurrence.

```
// Release Machine
if(state.type == BARRIER) then
        if(state.flag ≠ thread.flag) then
                return rd_done;
        endif;
endif.
```

Flags: OPF_DS0A, OPF_DS1D, OPF_GDSONLY, OPF_RDM0

ds_gws_init *vgpr_d0*
S0: vgpr, B32

GDS Only: Initialize a barrier or semaphore resource.

```
// 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
```

Flags: OPF_DS0A, OPF_DS1D, OPF_GDSONLY, OPF_RDM0

ds_gws_sema_br *vgpr_d0*

S0: vgpr, B32

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.

```
// 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.

Flags: OPF_DS0A, OPF_DS1D, OPF_GDSONLY, OPF_RDM0

ds_gws_sema_p

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.

```
// Determine the GWS resource to work on
rid[5:0] = SH_SX_EXPCMD.gds_base[5:0] + offset0[5:0];
state.type = SEMAPHORE;
ENQUEUE until(state[rid].counter > 0)
state[rid].counter -= 1;
return rd_done;
```

Flags: OPF_DS0A, OPF_GDSONLY, OPF_RDM0

ds_gws_sema_release_all

GDS Only: The GWS resource (rid) indicated will process this opcode by updating the counter and labeling the specified resource as a semaphore.

```
// 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] = state.wave_in_queue;
state.type = SEMAPHORE;
return rd_done; // release calling wave
```

This action will release ALL queued waves; it Will have no effect if no waves are present.

Flags: OPF_DS0A, OPF_GDSONLY, OPF_RDM0

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.

```
// 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 waved if any are queued in this resource.

Flags: OPF_DS0A, OPF_GDSONLY, OPF_RDM0

ds_inc_rtn_u32 *vgpr_rtn*, *vgpr_a*, *vgpr_d0*
D0: vgpr, B32 S0: vgpr, B32 S1: vgpr, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (tmp ≥ DATA) ? 0 : tmp + 1; // unsigned compare
RETURN_DATA = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_DSRTN, OPF_MEM_ATOMIC

ds_inc_rtn_u64 *vgpr_rtn*[2], *vgpr_a*, *vgpr_d0*[2]
D0: vgpr, B64 S0: vgpr, B32 S1: vgpr, B64

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] = (tmp ≥ DATA[0:1]) ? 0 : tmp + 1; // unsigned compare
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_DS64BIT, OPF_DSRTN, OPF_MEM_ATOMIC

ds_inc_src2_u32 *vgpr_a*
S0: vgpr, B32

```
// 32bit
A = ADDR_BASE;
B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0});
MEM[A] = (MEM[A] ≥ MEM[B] ? 0 : MEM[A] + 1).
```

Flags: OPF_DS1A, OPF_MEM_ATOMIC

ds_inc_src2_u64 *vgpr_a*
S0: vgpr, B32

```
// 64bit
A = ADDR_BASE;
B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0});
MEM[A] = (MEM[A] ≥ MEM[B] ? 0 : MEM[A] + 1).
```

Flags: OPF_DS1A, OPF_DS64BIT, OPF_MEM_ATOMIC

ds_inc_u32 *vgpr_a*, *vgpr_d0*
S0: vgpr, B32 S1: vgpr, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (tmp ≥ DATA) ? 0 : tmp + 1; // unsigned compare
RETURN_DATA = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_MEM_ATOMIC

ds_inc_u64 *vgpr_a*, *vgpr_d0*[2]
S0: vgpr, B32 S1: vgpr, B64

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] = (tmp ≥ DATA[0:1]) ? 0 : tmp + 1; // unsigned compare
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_DS64BIT, OPF_MEM_ATOMIC

ds_max_f32 *vgpr_a*, *vgpr_d0*
S0: vgpr, B32 S1: vgpr, F32

```
// 32bit
tmp = MEM[ADDR];
src = DATA;
cmp = DATA2;
MEM[ADDR] = (tmp > cmp) ? src : tmp.
```

Floating point maximum that handles NaN/INF/denormal values. Note that this opcode is slightly more general-purpose than BUFFER_ATOMIC_FMAX.

Flags: OPF_DS1A, OPF_DS1D, OPF_DS32FLT, OPF_MEM_ATOMIC

ds_max_f64 *vgpr_a*, *vgpr_d0*[2]
S0: vgpr, B32 S1: vgpr, F64

```
// 64bit
tmp = MEM[ADDR];
src = DATA;
cmp = DATA2;
MEM[ADDR] = (tmp > cmp) ? src : tmp.
```

Floating point maximum that handles NaN/INF/denormal values. Note that this opcode is slightly more general-purpose than BUFFER_ATOMIC_FMAX_X2.

Flags: OPF_DS1A, OPF_DS1D, OPF_DS64FLT, OPF_MEM_ATOMIC

ds_max_i32 *vgpr_a*, *vgpr_d0*
S0: vgpr, B32 S1: vgpr, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (DATA > tmp) ? DATA : tmp; // signed compare
RETURN_DATA = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_MEM_ATOMIC

ds_max_i64 *vgpr_a*, *vgpr_d0*[2]
S0: vgpr, B32 S1: vgpr, B64

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] -= (DATA[0:1] > tmp) ? DATA[0:1] : tmp; // signed compare
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_DS64BIT, OPF_MEM_ATOMIC

ds_max_rtn_f32 *vgpr_rtn*, *vgpr_a*, *vgpr_d0*
D0: vgpr, F32 S0: vgpr, B32 S1: vgpr, F32

```
// 32bit
tmp = MEM[ADDR];
src = DATA;
cmp = DATA2;
MEM[ADDR] = (tmp > cmp) ? src : tmp.
```

Floating point maximum that handles NaN/INF/denormal values. Note that this opcode is slightly more general-purpose than BUFFER_ATOMIC_FMAX.

Flags: OPF_DS1A, OPF_DS1D, OPF_DS32FLT, OPF_DSRTN, OPF_MEM_ATOMIC

ds_max_rtn_f64 *vgpr_rtn*[2], *vgpr_a*, *vgpr_d0*[2]
D0: vgpr, F64 S0: vgpr, B32 S1: vgpr, F64

```
// 64bit
tmp = MEM[ADDR];
src = DATA;
cmp = DATA2;
MEM[ADDR] = (tmp > cmp) ? src : tmp.
```

Floating point maximum that handles NaN/INF/denormal values. Note that this opcode is slightly more general-purpose than BUFFER_ATOMIC_FMAX_X2.

Flags: OPF_DS1A, OPF_DS1D, OPF_DS64FLT, OPF_DSRTN, OPF_MEM_ATOMIC

ds_max_rtn_i32 *vgpr_rtn*, *vgpr_a*, *vgpr_d0*
D0: vgpr, B32 S0: vgpr, B32 S1: vgpr, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (DATA > tmp) ? DATA : tmp; // signed compare
RETURN_DATA = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_DSRTN, OPF_MEM_ATOMIC

ds_max_rtn_i64 *vgpr_rtn*[2], *vgpr_a*, *vgpr_d0*[2]
D0: vgpr, B64 S0: vgpr, B32 S1: vgpr, B64

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] -= (DATA[0:1] > tmp) ? DATA[0:1] : tmp; // signed compare
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_DS64BIT, OPF_DSRTN, OPF_MEM_ATOMIC

ds_max_rtn_u32 *vgpr_rtn*, *vgpr_a*, *vgpr_d0*
D0: vgpr, B32 S0: vgpr, B32 S1: vgpr, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (DATA > tmp) ? DATA : tmp; // unsigned compare
RETURN_DATA = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_DSRTN, OPF_MEM_ATOMIC

ds_max_rtn_u64 *vgpr_rtn*[2], *vgpr_a*, *vgpr_d0*[2]
D0: vgpr, B64 S0: vgpr, B32 S1: vgpr, B64

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] -= (DATA[0:1] > tmp) ? DATA[0:1] : tmp; // unsigned compare
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_DS64BIT, OPF_DSRTN, OPF_MEM_ATOMIC

ds_max_src2_f32 *vgpr_a*
S0: vgpr, B32

```
// 32bit
A = ADDR_BASE;
B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0});
MEM[A] = (MEM[B] > MEM[A]) ? MEM[B] : MEM[A].

Float, handles NaN/INF/denorm.
```

Flags: OPF_DS1A, OPF_DS32FLT, OPF_MEM_ATOMIC

ds_max_src2_f64 *vgpr_a*
S0: vgpr, B32

```
// 64bit
A = ADDR_BASE;
B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0});
MEM[A] = (MEM[B] > MEM[A]) ? MEM[B] : MEM[A].

Float, handles NaN/INF/denorm.
```

Flags: OPF_DS1A, OPF_DS64FLT, OPF_MEM_ATOMIC

ds_max_src2_i32 *vgpr_a*
S0: vgpr, B32

```
// 32bit
A = ADDR_BASE;
B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0});
MEM[A] = max(MEM[A], MEM[B]).
```

Flags: OPF_DS1A, OPF_MEM_ATOMIC

ds_max_src2_i64 *vgpr_a*
S0: vgpr, B32

```
// 64bit
A = ADDR_BASE;
B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0});
MEM[A] = max(MEM[A], MEM[B]).
```

Flags: OPF_DS1A, OPF_DS64BIT, OPF_MEM_ATOMIC

ds_max_src2_u32 *vgpr_a*
S0: vgpr, B32

```
// 32bit
A = ADDR_BASE;
B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0});
MEM[A] = max(MEM[A], MEM[B]).
```

Flags: OPF_DS1A, OPF_MEM_ATOMIC

ds_max_src2_u64 *vgpr_a*
S0: vgpr, B32

```
// 64bit
A = ADDR_BASE;
B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0});
MEM[A] = max(MEM[A], MEM[B]).
```

Flags: OPF_DS1A, OPF_DS64BIT, OPF_MEM_ATOMIC

ds_max_u32 *vgpr_a*, *vgpr_d0*
S0: vgpr, B32 S1: vgpr, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (DATA > tmp) ? DATA : tmp; // unsigned compare
RETURN_DATA = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_MEM_ATOMIC

ds_max_u64 *vgpr_a*, *vgpr_d0*[2]
S0: vgpr, B32 S1: vgpr, B64

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] -= (DATA[0:1] > tmp) ? DATA[0:1] : tmp; // unsigned compare
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_DS64BIT, OPF_MEM_ATOMIC

ds_min_f32 *vgpr_a*, *vgpr_d0*
S0: vgpr, B32 S1: vgpr, F32

```
// 32bit
tmp = MEM[ADDR];
src = DATA;
cmp = DATA2;
MEM[ADDR] = (cmp < tmp) ? src : tmp.
```

Floating point minimum that handles NaN/INF/denormal values. Note that this opcode is slightly more general-purpose than BUFFER_ATOMIC_FMIN.

Flags: OPF_DS1A, OPF_DS1D, OPF_DS32FLT, OPF_MEM_ATOMIC

ds_min_f64 *vgpr_a*, *vgpr_d0*[2]
S0: vgpr, B32 S1: vgpr, F64

```
// 64bit
tmp = MEM[ADDR];
src = DATA;
cmp = DATA2;
MEM[ADDR] = (cmp < tmp) ? src : tmp.
```

Floating point minimum that handles NaN/INF/denormal values. Note that this opcode is slightly more general-purpose than BUFFER_ATOMIC_FMIN_X2.

Flags: OPF_DS1A, OPF_DS1D, OPF_DS64FLT, OPF_MEM_ATOMIC

ds_min_i32 *vgpr_a*, *vgpr_d0*
S0: vgpr, B32 S1: vgpr, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (DATA < tmp) ? DATA : tmp; // signed compare
RETURN_DATA = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_MEM_ATOMIC

ds_min_i64 *vgpr_a*, *vgpr_d0*[2]
S0: vgpr, B32 S1: vgpr, B64

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] -= (DATA[0:1] < tmp) ? DATA[0:1] : tmp; // signed compare
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_DS64BIT, OPF_MEM_ATOMIC

ds_min_rtn_f32 *vgpr_rtn*, *vgpr_a*, *vgpr_d0*
D0: vgpr, F32 S0: vgpr, B32 S1: vgpr, F32

```
// 32bit
tmp = MEM[ADDR];
src = DATA;
cmp = DATA2;
MEM[ADDR] = (cmp < tmp) ? src : tmp.
```

Floating point minimum that handles NaN/INF/denormal values. Note that this opcode is slightly more general-purpose than BUFFER_ATOMIC_FMIN.

Flags: OPF_DS1A, OPF_DS1D, OPF_DS32FLT, OPF_DSRTN, OPF_MEM_ATOMIC

ds_min_rtn_f64 *vgpr_rtn*[2], *vgpr_a*, *vgpr_d0*[2]
D0: vgpr, F64 S0: vgpr, B32 S1: vgpr, F64

```
// 64bit
tmp = MEM[ADDR];
src = DATA;
cmp = DATA2;
MEM[ADDR] = (cmp < tmp) ? src : tmp.
```

Floating point minimum that handles NaN/INF/denormal values. Note that this opcode is slightly more general-purpose than BUFFER_ATOMIC_FMIN_X2.

Flags: OPF_DS1A, OPF_DS1D, OPF_DS64FLT, OPF_DSRTN, OPF_MEM_ATOMIC

ds_min_rtn_i32 *vgpr_rtn*, *vgpr_a*, *vgpr_d0*
D0: vgpr, B32 S0: vgpr, B32 S1: vgpr, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (DATA < tmp) ? DATA : tmp; // signed compare
RETURN_DATA = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_DSRTN, OPF_MEM_ATOMIC

ds_min_rtn_i64 *vgpr_rtn*[2], *vgpr_a*, *vgpr_d0*[2]
D0: vgpr, B64 S0: vgpr, B32 S1: vgpr, B64

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] -= (DATA[0:1] < tmp) ? DATA[0:1] : tmp; // signed compare
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_DS64BIT, OPF_DSRTN, OPF_MEM_ATOMIC

ds_min_rtn_u32 *vgpr_rtn*, *vgpr_a*, *vgpr_d0*
D0: vgpr, B32 S0: vgpr, B32 S1: vgpr, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (DATA < tmp) ? DATA : tmp; // unsigned compare
RETURN_DATA = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_DSRTN, OPF_MEM_ATOMIC

ds_min_rtn_u64 *vgpr_rtn*[2], *vgpr_a*, *vgpr_d0*[2]
D0: vgpr, B64 S0: vgpr, B32 S1: vgpr, B64

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] -= (DATA[0:1] < tmp) ? DATA[0:1] : tmp; // unsigned compare
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_DS64BIT, OPF_DSRTN, OPF_MEM_ATOMIC

ds_min_src2_f32 *vgpr_a*
S0: vgpr, B32

```
// 32bit
A = ADDR_BASE;
B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0});
MEM[A] = (MEM[B] < MEM[A]) ? MEM[B] : MEM[A].

Float, handles NaN/INF/denorm.
```

Flags: OPF_DS1A, OPF_DS32FLT, OPF_MEM_ATOMIC

ds_min_src2_f64 *vgpr_a*
S0: vgpr, B32

```
// 64bit
A = ADDR_BASE;
B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0});
MEM[A] = (MEM[B] < MEM[A]) ? MEM[B] : MEM[A].

Float, handles NaN/INF/denorm.
```

Flags: OPF_DS1A, OPF_DS64FLT, OPF_MEM_ATOMIC

ds_min_src2_i32 *vgpr_a*
S0: vgpr, B32

```
// 32bit
A = ADDR_BASE;
B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0});
MEM[A] = min(MEM[A], MEM[B]).
```

Flags: OPF_DS1A, OPF_MEM_ATOMIC

ds_min_src2_i64 *vgpr_a*
S0: vgpr, B32

```
// 64bit
A = ADDR_BASE;
B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0});
MEM[A] = min(MEM[A], MEM[B]).
```

Flags: OPF_DS1A, OPF_DS64BIT, OPF_MEM_ATOMIC

ds_min_src2_u32 *vgpr_a*
S0: vgpr, B32

```
// 32bit
A = ADDR_BASE;
B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0});
MEM[A] = min(MEM[A], MEM[B]).
```

Flags: OPF_DS1A, OPF_MEM_ATOMIC

ds_min_src2_u64 *vgpr_a*
S0: vgpr, B32

```
// 64bit
A = ADDR_BASE;
B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0});
MEM[A] = min(MEM[A], MEM[B]).
```

Flags: OPF_DS1A, OPF_DS64BIT, OPF_MEM_ATOMIC

ds_min_u32 *vgpr_a*, *vgpr_d0*
S0: vgpr, B32 S1: vgpr, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (DATA < tmp) ? DATA : tmp; // unsigned compare
RETURN_DATA = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_MEM_ATOMIC

ds_min_u64 *vgpr_a*, *vgpr_d0*[2]
S0: vgpr, B32 S1: vgpr, B64

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] -= (DATA[0:1] < tmp) ? DATA[0:1] : tmp; // unsigned compare
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_DS64BIT, OPF_MEM_ATOMIC

ds_mskor_b32 *vgpr_a*, *vgpr_d0*, *vgpr_d1*
S0: vgpr, B32 S1: vgpr, B32 S2: vgpr, 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.

Flags: OPF_DS1A, OPF_DS2D, OPF_MEM_ATOMIC

ds_mskor_b64 *vgpr_a*, *vgpr_d0*[2], *vgpr_d1*[2]
S0: vgpr, B32 S1: vgpr, B64 S2: vgpr, 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.

Flags: OPF_DS1A, OPF_DS2D, OPF_DS64BIT, OPF_MEM_ATOMIC

ds_mskor_rtn_b32 *vgpr_rtn*, *vgpr_a*, *vgpr_d0*, *vgpr_d1*
D0: vgpr, B32 S0: vgpr, B32 S1: vgpr, B32 S2: vgpr, 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.

Flags: OPF_DS1A, OPF_DS2D, OPF_DSRTN, OPF_MEM_ATOMIC

ds_mskor_rtn_b64 *vgpr_rtn*[2], *vgpr_a*, *vgpr_d0*[2], *vgpr_d1*[2]
D0: vgpr, B64 S0: vgpr, B32 S1: vgpr, B64 S2: vgpr, 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.

Flags: OPF_DS1A, OPF_DS2D, OPF_DS64BIT, OPF_DSRTN, OPF_MEM_ATOMIC

ds_nop

Do nothing.

Flags: OPF_DS0A

ds_or_b32 *vgpr_a*, *vgpr_d0*
S0: vgpr, B32 S1: vgpr, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] |= DATA;
RETURN_DATA = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_MEM_ATOMIC

ds_or_b64 *vgpr_a*, *vgpr_d0*[2]
S0: vgpr, B32 S1: vgpr, B64

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] |= DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_DS64BIT, OPF_MEM_ATOMIC

ds_or_rtn_b32 *vgpr_rtn*, *vgpr_a*, *vgpr_d0*
D0: vgpr, B32 S0: vgpr, B32 S1: vgpr, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] |= DATA;
RETURN_DATA = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_DSRTN, OPF_MEM_ATOMIC

ds_or_rtn_b64 *vgpr_rtn*[2], *vgpr_a*, *vgpr_d0*[2]
D0: vgpr, B64 S0: vgpr, B32 S1: vgpr, B64

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] |= DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_DS64BIT, OPF_DSRTN, OPF_MEM_ATOMIC

ds_or_src2_b32 *vgpr_a*
S0: vgpr, B32

```
// 32bit
A = ADDR_BASE;
B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0});
MEM[A] = MEM[A] | MEM[B].
```

Flags: OPF_DS1A, OPF_MEM_ATOMIC

ds_or_src2_b64 *vgpr_a*
S0: vgpr, B32

```
// 64bit
A = ADDR_BASE;
B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0});
MEM[A] = MEM[A] | MEM[B].
```

Flags: OPF_DS1A, OPF_DS64BIT, OPF_MEM_ATOMIC

ds_ordered_count *vgpr_rtn*, *vgpr_a*
D0: vgpr, B32 S0: vgpr, B32

GDS-only. Add (count_bits(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'.

Flags: OPF_DS1A, OPF_DSRTN, OPF_GDSONLY, OPF_RDM0

ds_permute_b32 *vgpr_rtn*, *vgpr_a*, *vgpr_d0*
D0: vgpr, B32 S0: vgpr, B32 S1: vgpr, B32

```
// VGPR[index][thread_id] is the VGPR RAM
// VDST, ADDR and DATA0 are from the microcode DS encoding
tmp[0..63] = 0
for i in 0..63 do
        // If a source thread is disabled, it will not propagate data.
        next if !EXEC[i]
        // ADDR needs to be divided by 4.
        // High-order bits are ignored.
        dst_lane = floor((VGPR[ADDR][i] + OFFSET) / 4) mod 64
        tmp[dst_lane] = VGPR[DATA0][i]
endfor
// Copy data into destination VGPRs. If multiple sources
// select the same destination thread, the highest-numbered
// source thread wins.
for i in 0..63 do
        next if !EXEC[i]
        VGPR[VDST][i] = tmp[i]
endfor
```

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):

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 }

Flags: OPF_DS1A, OPF_DS1D, OPF_DSRTN, OPF_MEM_ATOMIC

ds_read2_b32 *vgpr_rtn*[2], *vgpr_a*
D0: vgpr, B32 S0: vgpr, B32

```
RETURN_DATA[0] = MEM[ADDR_BASE + OFFSET0 * 4];
RETURN_DATA[1] = MEM[ADDR_BASE + OFFSET1 * 4].
```

Read 2 dwords.

Flags: OPF_DS2A, OPF_DSRTN

ds_read2_b64 *vgpr_rtn*[4], *vgpr_a*
D0: vgpr, B64 S0: vgpr, B32

```
RETURN_DATA[0] = MEM[ADDR_BASE + OFFSET0 * 8];
RETURN_DATA[1] = MEM[ADDR_BASE + OFFSET1 * 8].
```

Read 2 qwords.

Flags: OPF_DS2A, OPF_DS64BIT, OPF_DSRTN

ds_read2st64_b32 *vgpr_rtn*[2], *vgpr_a*
D0: vgpr, B32 S0: vgpr, B32

```
RETURN_DATA[0] = MEM[ADDR_BASE + OFFSET0 * 4 * 64];
RETURN_DATA[1] = MEM[ADDR_BASE + OFFSET1 * 4 * 64].
```

Read 2 dwords.

Flags: OPF_DS2A, OPF_DSRTN

ds_read2st64_b64 *vgpr_rtn*[4], *vgpr_a*
D0: vgpr, B64 S0: vgpr, B32

```
RETURN_DATA[0] = MEM[ADDR_BASE + OFFSET0 * 8 * 64];
RETURN_DATA[1] = MEM[ADDR_BASE + OFFSET1 * 8 * 64].
```

Read 2 qwords.

Flags: OPF_DS2A, OPF_DS64BIT, OPF_DSRTN

ds_read_addtid_b32 *vgpr_rtn*
D0: vgpr, B32

```
RETURN_DATA = MEM[ADDR_BASE + OFFSET + M0.OFFSET + TID*4].
```

Dword read.

Flags: OPF_DS0A, OPF_DSRTN, OPF_DS_ADDTID, OPF_RDM0

ds_read_b128 *vgpr_rtn*[4], *vgpr_a*
D0: vgpr, B128 S0: vgpr, B32

Quad-dword read.

Flags: ASIC_LARGE_DS_READ, OPF_DS128BIT, OPF_DS1A, OPF_DSRTN

ds_read_b32 *vgpr_rtn*, *vgpr_a*
D0: vgpr, B32 S0: vgpr, B32

```
RETURN_DATA = MEM[ADDR].
```

Dword read.

Flags: OPF_DS1A, OPF_DSRTN

ds_read_b64 *vgpr_rtn*[2], *vgpr_a*
D0: vgpr, B64 S0: vgpr, B32

```
RETURN_DATA = MEM[ADDR].
```

Read 1 qword.

Flags: OPF_DS1A, OPF_DS64BIT, OPF_DSRTN

ds_read_b96 *vgpr_rtn*[3], *vgpr_a*
D0: vgpr, B96 S0: vgpr, B32

Tri-dword read.

Flags: ASIC_LARGE_DS_READ, OPF_DS1A, OPF_DS96BIT, OPF_DSRTN

ds_read_i16 *vgpr_rtn*, *vgpr_a*
D0: vgpr, B16 S0: vgpr, B32

```
RETURN_DATA = signext(MEM[ADDR][15:0]).
```

Signed short read.

Flags: OPF_DS16BIT, OPF_DS1A, OPF_DSRTN

ds_read_i8 *vgpr_rtn*, *vgpr_a*
D0: vgpr, B32 S0: vgpr, B32

```
RETURN_DATA = signext(MEM[ADDR][7:0]).
```

Signed byte read.

Flags: OPF_DS1A, OPF_DSRTN

ds_read_i8_d16 *vgpr_rtn*, *vgpr_a*
D0: vgpr, B8 S0: vgpr, B32

```
RETURN_DATA[15:0] = signext(MEM[ADDR][7:0]).
```

Signed byte read with masked return to lower word.

Flags: OPF_DS1A, OPF_DS8BIT, OPF_DSRTN

ds_read_i8_d16_hi *vgpr_rtn*, *vgpr_a*
D0: vgpr, B8 S0: vgpr, B32

```
RETURN_DATA[31:16] = signext(MEM[ADDR][7:0]).
```

Signed byte read with masked return to upper word.

Flags: OPF_DS1A, OPF_DS8BIT, OPF_DSRTN

ds_read_u16 *vgpr_rtn*, *vgpr_a*
D0: vgpr, B16 S0: vgpr, B32

```
RETURN_DATA = {16'h0,MEM[ADDR][15:0]}.
```

Unsigned short read.

Flags: OPF_DS16BIT, OPF_DS1A, OPF_DSRTN

ds_read_u16_d16 *vgpr_rtn*, *vgpr_a*
D0: vgpr, B16 S0: vgpr, B32

```
RETURN_DATA[15:0] = MEM[ADDR][15:0].
```

Unsigned short read with masked return to lower word.

Flags: OPF_DS16BIT, OPF_DS1A, OPF_DSRTN

ds_read_u16_d16_hi *vgpr_rtn*, *vgpr_a*
D0: vgpr, B16 S0: vgpr, B32

```
RETURN_DATA[31:0] = MEM[ADDR][15:0].
```

Unsigned short read with masked return to upper word.

Flags: OPF_DS16BIT, OPF_DS1A, OPF_DSRTN

ds_read_u8 *vgpr_rtn*, *vgpr_a*
D0: vgpr, B8 S0: vgpr, B32

```
RETURN_DATA = {24'h0,MEM[ADDR][7:0]}.
```

Unsigned byte read.

Flags: OPF_DS1A, OPF_DS8BIT, OPF_DSRTN

ds_read_u8_d16 *vgpr_rtn*, *vgpr_a*
D0: vgpr, B8 S0: vgpr, B32

```
RETURN_DATA[15:0] = {8'h0,MEM[ADDR][7:0]}.
```

Unsigned byte read with masked return to lower word.

Flags: OPF_DS1A, OPF_DS8BIT, OPF_DSRTN

ds_read_u8_d16_hi *vgpr_rtn*, *vgpr_a*
D0: vgpr, B8 S0: vgpr, B32

```
RETURN_DATA[31:16] = {8'h0,MEM[ADDR][7:0]}.
```

Unsigned byte read with masked return to upper word.

Flags: OPF_DS1A, OPF_DS8BIT, OPF_DSRTN

ds_rsub_rtn_u32 *vgpr_rtn*, *vgpr_a*, *vgpr_d0*
D0: vgpr, B32 S0: vgpr, B32 S1: vgpr, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = DATA - MEM[ADDR];
RETURN_DATA = tmp.
```

Subtraction with reversed operands.

Flags: OPF_DS1A, OPF_DS1D, OPF_DSRTN, OPF_MEM_ATOMIC

ds_rsub_rtn_u64 *vgpr_rtn*[2], *vgpr_a*, *vgpr_d0*[2]
D0: vgpr, B64 S0: vgpr, B32 S1: vgpr, B64

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] = DATA - MEM[ADDR];
RETURN_DATA = tmp.
```

Subtraction with reversed operands.

Flags: OPF_DS1A, OPF_DS1D, OPF_DS64BIT, OPF_DSRTN, OPF_MEM_ATOMIC

ds_rsub_src2_u32 *vgpr_a*
S0: vgpr, B32

```
// 32bit
A = ADDR_BASE;
B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0});
MEM[A] = MEM[B] - MEM[A].
```

Flags: OPF_DS1A, OPF_MEM_ATOMIC

ds_rsub_src2_u64 *vgpr_a*
S0: vgpr, B32

```
// 64bit
A = ADDR_BASE;
B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0});
MEM[A] = MEM[B] - MEM[A].
```

Flags: OPF_DS1A, OPF_DS64BIT, OPF_MEM_ATOMIC

ds_rsub_u32 *vgpr_a*, *vgpr_d0*
S0: vgpr, B32 S1: vgpr, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = DATA - MEM[ADDR];
RETURN_DATA = tmp.
```

Subtraction with reversed operands.

Flags: OPF_DS1A, OPF_DS1D, OPF_MEM_ATOMIC

ds_rsub_u64 *vgpr_a*, *vgpr_d0*[2]
S0: vgpr, B32 S1: vgpr, B64

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] = DATA - MEM[ADDR];
RETURN_DATA = tmp.
```

Subtraction with reversed operands.

Flags: OPF_DS1A, OPF_DS1D, OPF_DS64BIT, OPF_MEM_ATOMIC

ds_sub_rtn_u32 *vgpr_rtn*, *vgpr_a*, *vgpr_d0*
D0: vgpr, B32 S0: vgpr, B32 S1: vgpr, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] -= DATA;
RETURN_DATA = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_DSRTN, OPF_MEM_ATOMIC

ds_sub_rtn_u64 *vgpr_rtn*[2], *vgpr_a*, *vgpr_d0*[2]
D0: vgpr, B64 S0: vgpr, B32 S1: vgpr, B64

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] -= DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_DS64BIT, OPF_DSRTN, OPF_MEM_ATOMIC

ds_sub_src2_u32 *vgpr_a*
S0: vgpr, B32

```
// 32bit
A = ADDR_BASE;
B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0});
MEM[A] = MEM[A] - MEM[B].
```

Flags: OPF_DS1A, OPF_MEM_ATOMIC

ds_sub_src2_u64 *vgpr_a*
S0: vgpr, B32

```
// 64bit
A = ADDR_BASE;
B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0});
MEM[A] = MEM[A] - MEM[B].
```

Flags: OPF_DS1A, OPF_DS64BIT, OPF_MEM_ATOMIC

ds_sub_u32 *vgpr_a*, *vgpr_d0*
S0: vgpr, B32 S1: vgpr, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] -= DATA;
RETURN_DATA = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_MEM_ATOMIC

ds_sub_u64 *vgpr_a*, *vgpr_d0*[2]
S0: vgpr, B32 S1: vgpr, B64

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] -= DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_DS64BIT, OPF_MEM_ATOMIC

ds_swizzle_b32 *vgpr_rtn*, *vgpr_a*
D0: vgpr, B32 S0: vgpr, B32

```
RETURN_DATA = swizzle(vgpr_data, offset1:offset0).
```

Dword swizzle, no data is written to LDS memory; See ds_opcodes.docx for details.

Flags: OPF_DS1A, OPF_DSRTN, OPF_MEM_ATOMIC

ds_wrap_rtn_b32 *vgpr_rtn*, *vgpr_a*, *vgpr_d0*, *vgpr_d1*
D0: vgpr, B32 S0: vgpr, B32 S1: vgpr, B32 S2: vgpr, B32

```
tmp = MEM[ADDR];
MEM[ADDR] = (tmp ≥ DATA) ? tmp - DATA : tmp + DATA2;
RETURN_DATA = tmp.
```

Flags: OPF_DS1A, OPF_DS2D, OPF_DSRTN, OPF_MEM_ATOMIC

ds_write2_b32 *vgpr_a*, *vgpr_d0*, *vgpr_d1*
S0: vgpr, B32 S1: vgpr, B32 S2: vgpr, B32

```
// 32bit
MEM[ADDR_BASE + OFFSET0 * 4] = DATA;
MEM[ADDR_BASE + OFFSET1 * 4] = DATA2.
```

Write 2 dwords.

Flags: OPF_DS2A, OPF_DS2D, OPF_MEM_STORE

ds_write2_b64 *vgpr_a*, *vgpr_d0*[2], *vgpr_d1*[2]
S0: vgpr, B32 S1: vgpr, B64 S2: vgpr, B64

```
// 64bit
MEM[ADDR_BASE + OFFSET0 * 8] = DATA;
MEM[ADDR_BASE + OFFSET1 * 8] = DATA2.
```

Write 2 qwords.

Flags: OPF_DS2A, OPF_DS2D, OPF_DS64BIT, OPF_MEM_STORE

ds_write2st64_b32 *vgpr_a*, *vgpr_d0*, *vgpr_d1*
S0: vgpr, B32 S1: vgpr, B32 S2: vgpr, B32

```
// 32bit
MEM[ADDR_BASE + OFFSET0 * 4 * 64] = DATA;
MEM[ADDR_BASE + OFFSET1 * 4 * 64] = DATA2.
```

Write 2 dwords.

Flags: OPF_DS2A, OPF_DS2D, OPF_MEM_STORE

ds_write2st64_b64 *vgpr_a*, *vgpr_d0*[2], *vgpr_d1*[2]
S0: vgpr, B32 S1: vgpr, B64 S2: vgpr, B64

```
// 64bit
MEM[ADDR_BASE + OFFSET0 * 8 * 64] = DATA;
MEM[ADDR_BASE + OFFSET1 * 8 * 64] = DATA2.
```

Write 2 qwords.

Flags: OPF_DS2A, OPF_DS2D, OPF_DS64BIT, OPF_MEM_STORE

ds_write_addtid_b32 *vgpr_d0*
S0: vgpr, B32

```
// 32bit
MEM[ADDR_BASE + OFFSET + M0.OFFSET + TID*4] = DATA.
```

Write dword.

Flags: OPF_DS0A, OPF_DS1D, OPF_DS_ADDTID, OPF_MEM_STORE, OPF_RDM0

ds_write_b128 *vgpr_a*, *vgpr_d0*[4]
S0: vgpr, B32 S1: vgpr, B128

```
{MEM[ADDR + 12], MEM[ADDR + 8], MEM[ADDR + 4], MEM[ADDR]} = DATA[127:0].
```

Quad-dword write.

Flags: OPF_DS128BIT, OPF_DS1A, OPF_DS1D, OPF_MEM_STORE

ds_write_b16 *vgpr_a*, *vgpr_d0*
S0: vgpr, B32 S1: vgpr, B16

```
MEM[ADDR] = DATA[15:0].
```

Short write.

Flags: OPF_DS16BIT, OPF_DS1A, OPF_DS1D, OPF_MEM_STORE

ds_write_b16_d16_hi *vgpr_a*, *vgpr_d0*
S0: vgpr, B32 S1: vgpr, B16

```
MEM[ADDR] = DATA[31:16].
```

Short write in to high word.

Flags: OPF_DS16BIT, OPF_DS1A, OPF_DS1D, OPF_MEM_STORE

ds_write_b32 *vgpr_a*, *vgpr_d0*
S0: vgpr, B32 S1: vgpr, B32

```
// 32bit
MEM[ADDR] = DATA.
```

Write dword.

Flags: OPF_DS1A, OPF_DS1D, OPF_MEM_STORE

ds_write_b64 *vgpr_a*, *vgpr_d0*[2]
S0: vgpr, B32 S1: vgpr, B64

```
// 64bit
MEM[ADDR] = DATA.
```

Write qword.

Flags: OPF_DS1A, OPF_DS1D, OPF_DS64BIT, OPF_MEM_STORE

ds_write_b8 *vgpr_a*, *vgpr_d0*
S0: vgpr, B32 S1: vgpr, B8

```
MEM[ADDR] = DATA[7:0].
```

Byte write.

Flags: OPF_DS1A, OPF_DS1D, OPF_DS8BIT, OPF_MEM_STORE

ds_write_b8_d16_hi *vgpr_a*, *vgpr_d0*
S0: vgpr, B32 S1: vgpr, B8

```
MEM[ADDR] = DATA[23:16].
```

Byte write in to high word.

Flags: OPF_DS1A, OPF_DS1D, OPF_DS8BIT, OPF_MEM_STORE

ds_write_b96 *vgpr_a*, *vgpr_d0*[3]
S0: vgpr, B32 S1: vgpr, B96

```
{MEM[ADDR + 8], MEM[ADDR + 4], MEM[ADDR]} = DATA[95:0].
```

Tri-dword write.

Flags: OPF_DS1A, OPF_DS1D, OPF_DS96BIT, OPF_MEM_STORE

ds_write_src2_b32 *vgpr_a*
S0: vgpr, B32

```
// 32bit
A = ADDR_BASE;
B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0});
MEM[A] = MEM[B].

Write dword.
```

Flags: OPF_DS1A, OPF_MEM_ATOMIC

ds_write_src2_b64 *vgpr_a*
S0: vgpr, B32

```
// 64bit
A = ADDR_BASE;
B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0});
MEM[A] = MEM[B].

Write qword.
```

Flags: OPF_DS1A, OPF_DS64BIT, OPF_MEM_ATOMIC

ds_wrxchg2_rtn_b32 *vgpr_rtn*[2], *vgpr_a*, *vgpr_d0*, *vgpr_d1*
D0: vgpr, B32 S0: vgpr, B32 S1: vgpr, B32 S2: vgpr, B32

Write-exchange 2 separate dwords.

Flags: OPF_DS2A, OPF_DS2D, OPF_DSRTN, OPF_MEM_ATOMIC

ds_wrxchg2_rtn_b64 *vgpr_rtn*[4], *vgpr_a*, *vgpr_d0*[2], *vgpr_d1*[2]
D0: vgpr, B64 S0: vgpr, B32 S1: vgpr, B64 S2: vgpr, B64

Write-exchange 2 separate qwords.

Flags: OPF_DS2A, OPF_DS2D, OPF_DS64BIT, OPF_DSRTN, OPF_MEM_ATOMIC

ds_wrxchg2st64_rtn_b32 *vgpr_rtn*[2], *vgpr_a*, *vgpr_d0*, *vgpr_d1*
D0: vgpr, B32 S0: vgpr, B32 S1: vgpr, B32 S2: vgpr, B32

Write-exchange 2 separate dwords with a stride of 64 dwords.

Flags: OPF_DS2A, OPF_DS2D, OPF_DSRTN, OPF_MEM_ATOMIC

ds_wrxchg2st64_rtn_b64 *vgpr_rtn*[4], *vgpr_a*, *vgpr_d0*[2], *vgpr_d1*[2]
D0: vgpr, B64 S0: vgpr, B32 S1: vgpr, B64 S2: vgpr, B64

Write-exchange 2 qwords with a stride of 64 qwords.

Flags: OPF_DS2A, OPF_DS2D, OPF_DS64BIT, OPF_DSRTN, OPF_MEM_ATOMIC

ds_wrxchg_rtn_b32 *vgpr_rtn*, *vgpr_a*, *vgpr_d0*
D0: vgpr, B32 S0: vgpr, B32 S1: vgpr, B32

```
tmp = MEM[ADDR];
MEM[ADDR] = DATA;
RETURN_DATA = tmp.
```

Write-exchange operation.

Flags: OPF_DS1A, OPF_DS1D, OPF_DSRTN, OPF_MEM_ATOMIC

ds_wrxchg_rtn_b64 *vgpr_rtn*[2], *vgpr_a*, *vgpr_d0*[2]
D0: vgpr, B64 S0: vgpr, B32 S1: vgpr, B64

```
tmp = MEM[ADDR];
MEM[ADDR] = DATA;
RETURN_DATA = tmp.
```

Write-exchange operation.

Flags: OPF_DS1A, OPF_DS1D, OPF_DS64BIT, OPF_DSRTN, OPF_MEM_ATOMIC

ds_xor_b32 *vgpr_a*, *vgpr_d0*
S0: vgpr, B32 S1: vgpr, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] ^= DATA;
RETURN_DATA = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_MEM_ATOMIC

ds_xor_b64 *vgpr_a*, *vgpr_d0*[2]
S0: vgpr, B32 S1: vgpr, B64

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] ^= DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_DS64BIT, OPF_MEM_ATOMIC

ds_xor_rtn_b32 *vgpr_rtn*, *vgpr_a*, *vgpr_d0*
D0: vgpr, B32 S0: vgpr, B32 S1: vgpr, B32

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] ^= DATA;
RETURN_DATA = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_DSRTN, OPF_MEM_ATOMIC

ds_xor_rtn_b64 *vgpr_rtn*[2], *vgpr_a*, *vgpr_d0*[2]
D0: vgpr, B64 S0: vgpr, B32 S1: vgpr, B64

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] ^= DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_DS1A, OPF_DS1D, OPF_DS64BIT, OPF_DSRTN, OPF_MEM_ATOMIC

ds_xor_src2_b32 *vgpr_a*
S0: vgpr, B32

```
// 32bit
A = ADDR_BASE;
B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0});
MEM[A] = MEM[A] ^ MEM[B].
```

Flags: OPF_DS1A, OPF_MEM_ATOMIC

ds_xor_src2_b64 *vgpr_a*
S0: vgpr, B32

```
// 64bit
A = ADDR_BASE;
B = A + 4*(offset1[7] ? {A[31],A[31:17]} : {offset1[6],offset1[6:0],offset0});
MEM[A] = MEM[A] ^ MEM[B].
```

Flags: OPF_DS1A, OPF_DS64BIT, OPF_MEM_ATOMIC

14.1 Notes for Encoding DS

14.1.1 Operands

vgpr_rtn is the vector GPR to return the previous value of data-share memory to. In instruction descriptions this is referred to as RETURN_DATA. Some instructions do not return any data to VGPRs.

vgpr_a is the vector GPR that contains addresses of data-share memory to access. It is combined with the instruction offset as appropriate. In instruction descriptions the resulting byte address that we access is referred to as:

- ADDR_BASE = *vgpr_a*
- ADDR = *vgpr_a* + *offset*

ADDR.

vgpr_d0 is the vector GPR containing the first data value used by the instruction. In instruction descriptions this is referred to as DATA.

vgpr_d1 is the vector GPR containing the second data value used by the instruction. In instruction descriptions this is referred to as DATA2.

The data share memory itself is referred to as MEM[*a*], with byte address *a*.

14.1.2 Modifiers

`offset0:[0. . . 255]`
: Specify first 8-bit offset. Default 0.

`offset1:[0. . . 255]`
: Specify second 8-bit offset. Default 0.

`offset:[0. . . 65535]`
: Specify single 16-bit offset that spans both 8-bit offsets. Default 0.

`gds:{0,1}`
: If true, this is a GDS operation. Default 0. Can also be written `nogds`.

14.1.3 Additional References

For more details on these opcodes please see the LDS/GDS Opcodes specification in ds_opcodes.docx.

15 Encoding MUBUF

Untyped vector memory buffer operations.

buffer_atomic_add *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr, ↔ S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] += DATA;
RETURN_DATA = tmp.
```

Flags: OPF_MEM_ATOMIC

buffer_atomic_add_x2 *vgpr_d*[2], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr, ↔ S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] += DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_MEM_ATOMIC

buffer_atomic_and *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr, ↔ S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] &= DATA;
RETURN_DATA = tmp.
```

Flags: OPF_MEM_ATOMIC

buffer_atomic_and_x2 *vgpr_d*[2], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr, ↔ S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] &= DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_MEM_ATOMIC

buffer_atomic_cmpswap *vgpr_d*[2], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr, ↔ S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

```
// 32bit
tmp = MEM[ADDR];
src = DATA[0];
cmp = DATA[1];
MEM[ADDR] = (tmp == cmp) ? src : tmp;
RETURN_DATA[0] = tmp.
```

Flags: OPF_ATOMIC_CMPSWAP, OPF_MEM_ATOMIC

buffer_atomic_cmpswap_x2 *vgpr_d*[4], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr, ↔ S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

```
// 64bit
tmp = MEM[ADDR];
src = DATA[0:1];
cmp = DATA[2:3];
MEM[ADDR] = (tmp == cmp) ? src : tmp;
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_ATOMIC_CMPSWAP, OPF_MEM_ATOMIC

buffer_atomic_dec *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr, ↔ S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (tmp == 0 || tmp > DATA) ? DATA : tmp - 1; // unsigned compare
RETURN_DATA = tmp.
```

Flags: OPF_MEM_ATOMIC

buffer_atomic_dec_x2 *vgpr_d*[2], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr, ↔ S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] = (tmp == 0 || tmp > DATA[0:1]) ? DATA[0:1] : tmp - 1; // unsigned compare
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_MEM_ATOMIC

buffer_atomic_inc *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr, ↔ S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (tmp ≥ DATA) ? 0 : tmp + 1; // unsigned compare
RETURN_DATA = tmp.
```

Flags: OPF_MEM_ATOMIC

buffer_atomic_inc_x2 *vgpr_d*[2], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr, ↔ S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] = (tmp ≥ DATA[0:1]) ? 0 : tmp + 1; // unsigned compare
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_MEM_ATOMIC

buffer_atomic_or *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr, ↔ S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] |= DATA;
RETURN_DATA = tmp.
```

Flags: OPF_MEM_ATOMIC

buffer_atomic_or_x2 *vgpr_d*[2], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr, ↔ S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] |= DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_MEM_ATOMIC

buffer_atomic_smax *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr, ↔ S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (DATA > tmp) ? DATA : tmp; // signed compare
RETURN_DATA = tmp.
```

Flags: OPF_MEM_ATOMIC

buffer_atomic_smax_x2 *vgpr_d*[2], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr, ↔ S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] -= (DATA[0:1] > tmp) ? DATA[0:1] : tmp; // signed compare
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_MEM_ATOMIC

buffer_atomic_smin *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr, ↔ S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (DATA < tmp) ? DATA : tmp; // signed compare
RETURN_DATA = tmp.
```

Flags: OPF_MEM_ATOMIC

buffer_atomic_smin_x2 *vgpr_d*[2], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr, ↔ S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] -= (DATA[0:1] < tmp) ? DATA[0:1] : tmp; // signed compare
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_MEM_ATOMIC

buffer_atomic_sub *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr, ↔ S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] -= DATA;
RETURN_DATA = tmp.
```

Flags: OPF_MEM_ATOMIC

buffer_atomic_sub_x2 *vgpr_d*[2], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr, ↔ S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] -= DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_MEM_ATOMIC

buffer_atomic_swap *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr, ↔ S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = DATA;
RETURN_DATA = tmp.
```

Flags: OPF_MEM_ATOMIC

buffer_atomic_swap_x2 *vgpr_d*[2], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr, ↔ S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] = DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_MEM_ATOMIC

buffer_atomic_umax *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr, ↔ S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (DATA > tmp) ? DATA : tmp; // unsigned compare
RETURN_DATA = tmp.
```

Flags: OPF_MEM_ATOMIC

buffer_atomic_umax_x2 *vgpr_d*[2], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr, ↔ S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] -= (DATA[0:1] > tmp) ? DATA[0:1] : tmp; // unsigned compare
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_MEM_ATOMIC

buffer_atomic_umin *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr, ↔ S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (DATA < tmp) ? DATA : tmp; // unsigned compare
RETURN_DATA = tmp.
```

Flags: OPF_MEM_ATOMIC

buffer_atomic_umin_x2 *vgpr_d*[2], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr, ↔ S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] -= (DATA[0:1] < tmp) ? DATA[0:1] : tmp; // unsigned compare
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_MEM_ATOMIC

buffer_atomic_xor *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr, ↔ S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] ^= DATA;
RETURN_DATA = tmp.
```

Flags: OPF_MEM_ATOMIC

buffer_atomic_xor_x2 *vgpr_d*[2], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr, ↔ S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] ^= DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: OPF_MEM_ATOMIC

buffer_load_dword *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

Untyped buffer load dword.

Flags: OPF_ALLOW_RTN_TO_LDS

buffer_load_dwordx2 *vgpr_d*[2], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

Untyped buffer load 2 dwords.

buffer_load_dwordx3 *vgpr_d*[3], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

Untyped buffer load 3 dwords.

buffer_load_dwordx4 *vgpr_d*[4], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

Untyped buffer load 4 dwords.

buffer_load_format_d16_hi_x *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

```
D0[31:16] = MEM[ADDR].
```

Untyped buffer load 1 dword with format conversion.

Flags: OPF_D16, OPF_MEMFMT

buffer_load_format_d16_x *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

Untyped buffer load 1 dword with format conversion. D0[15:0] = {8'h0, MEM[ADDR]}.

Flags: OPF_D16, OPF_D16_CH_1, OPF_MEMFMT

buffer_load_format_d16_xy *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

Untyped buffer load 1 dword with format conversion.

Flags: OPF_D16, OPF_MEMFMT

buffer_load_format_d16_xyz *vgpr_d*[2], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

Untyped buffer load 2 dwords with format conversion.

Flags: OPF_D16, OPF_D16_CH_3, OPF_MEMFMT

buffer_load_format_d16_xyzw *vgpr_d*[2], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

Untyped buffer load 2 dwords with format conversion.

Flags: OPF_D16, OPF_MEMFMT

buffer_load_format_x *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

Untyped buffer load 1 dword with format conversion.

Flags: OPF_ALLOW_RTN_TO_LDS, OPF_MEMFMT

buffer_load_format_xy *vgpr_d*[2], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

Untyped buffer load 2 dwords with format conversion.

Flags: OPF_MEMFMT

buffer_load_format_xyz *vgpr_d*[3], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

Untyped buffer load 3 dwords with format conversion.

Flags: OPF_MEMFMT

buffer_load_format_xyzw *vgpr_d*[4], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

Untyped buffer load 4 dwords with format conversion.

Flags: OPF_MEMFMT

buffer_load_sbyte *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

Untyped buffer load signed byte (sign extend to VGPR destination).

Flags: OPF_ALLOW_RTN_TO_LDS

buffer_load_sbyte_d16 *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

```
D0[15:0] = {8'h0, MEM[ADDR]}.
```

Untyped buffer load signed byte.

Flags: OPF_D16

buffer_load_sbyte_d16_hi *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

```
D0[31:16] = {8'h0, MEM[ADDR]}.
```

Untyped buffer load signed byte.

Flags: OPF_D16

buffer_load_short_d16 *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

```
D0[15:0] = MEM[ADDR].
```

Untyped buffer load short.

Flags: OPF_D16

buffer_load_short_d16_hi *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

```
D0[31:16] = MEM[ADDR].
```

Untyped buffer load short.

Flags: OPF_D16

buffer_load_sshort *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

Untyped buffer load signed short (sign extend to VGPR destination).

Flags: OPF_ALLOW_RTN_TO_LDS

buffer_load_ubyte *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

Untyped buffer load unsigned byte (zero extend to VGPR destination).

Flags: OPF_ALLOW_RTN_TO_LDS

buffer_load_ubyte_d16 *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

```
D0[15:0] = {8'h0, MEM[ADDR]}.
```

Untyped buffer load unsigned byte.

Flags: OPF_D16

buffer_load_ubyte_d16_hi *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

```
D0[31:16] = {8'h0, MEM[ADDR]}.
```

Untyped buffer load unsigned byte.

Flags: OPF_D16

buffer_load_ushort *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

Untyped buffer load unsigned short (zero extend to VGPR destination).

Flags: OPF_ALLOW_RTN_TO_LDS

buffer_store_byte *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
S0: vgpr S1: vgpr S2: sreg, RSRC S3: ssrc_nolit

Untyped buffer store byte. Stores S0[7:0].

Flags: OPF_MEM_STORE

buffer_store_byte_d16_hi *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
S0: vgpr S1: vgpr S2: sreg, RSRC S3: ssrc_nolit

Untyped buffer store byte. Stores S0[23:16].

Flags: OPF_D16, OPF_MEM_STORE

buffer_store_dword *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
S0: vgpr S1: vgpr S2: sreg, RSRC S3: ssrc_nolit

Untyped buffer store dword.

Flags: OPF_MEM_STORE

buffer_store_dwordx2 *vgpr_d*[2], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
S0: vgpr S1: vgpr S2: sreg, RSRC S3: ssrc_nolit

Untyped buffer store 2 dwords.

Flags: OPF_MEM_STORE

buffer_store_dwordx3 *vgpr_d*[3], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
S0: vgpr S1: vgpr S2: sreg, RSRC S3: ssrc_nolit

Untyped buffer store 3 dwords.

Flags: OPF_MEM_STORE

buffer_store_dwordx4 *vgpr_d*[4], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
S0: vgpr S1: vgpr S2: sreg, RSRC S3: ssrc_nolit

Untyped buffer store 4 dwords.

Flags: OPF_MEM_STORE

buffer_store_format_d16_hi_x *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
S0: vgpr S1: vgpr S2: sreg, RSRC S3: ssrc_nolit

Untyped buffer store 1 dword with format conversion.

Flags: OPF_D16, OPF_MEMFMT, OPF_MEM_STORE

buffer_store_format_d16_x *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
S0: vgpr S1: vgpr S2: sreg, RSRC S3: ssrc_nolit

Untyped buffer store 1 dword with format conversion.

Flags: OPF_D16, OPF_D16_CH_1, OPF_MEMFMT, OPF_MEM_STORE

buffer_store_format_d16_xy *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
S0: vgpr S1: vgpr S2: sreg, RSRC S3: ssrc_nolit

Untyped buffer store 1 dword with format conversion.

Flags: OPF_D16, OPF_MEMFMT, OPF_MEM_STORE

buffer_store_format_d16_xyz *vgpr_d*[2], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
S0: vgpr S1: vgpr S2: sreg, RSRC S3: ssrc_nolit

Untyped buffer store 2 dwords with format conversion.

Flags: OPF_D16, OPF_D16_CH_3, OPF_MEMFMT, OPF_MEM_STORE

buffer_store_format_d16_xyzw *vgpr_d*[2], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
S0: vgpr S1: vgpr S2: sreg, RSRC S3: ssrc_nolit

Untyped buffer store 2 dwords with format conversion.

Flags: OPF_D16, OPF_MEMFMT, OPF_MEM_STORE

buffer_store_format_x *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
S0: vgpr S1: vgpr S2: sreg, RSRC S3: ssrc_nolit

Untyped buffer store 1 dword with format conversion.

Flags: OPF_MEMFMT, OPF_MEM_STORE

buffer_store_format_xy *vgpr_d*[2], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
S0: vgpr S1: vgpr S2: sreg, RSRC S3: ssrc_nolit

Untyped buffer store 2 dwords with format conversion.

Flags: OPF_MEMFMT, OPF_MEM_STORE

buffer_store_format_xyz *vgpr_d*[3], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
S0: vgpr S1: vgpr S2: sreg, RSRC S3: ssrc_nolit

Untyped buffer store 3 dwords with format conversion.

Flags: OPF_MEMFMT, OPF_MEM_STORE

buffer_store_format_xyzw *vgpr_d*[4], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
S0: vgpr S1: vgpr S2: sreg, RSRC S3: ssrc_nolit

Untyped buffer store 4 dwords with format conversion.

Flags: OPF_MEMFMT, OPF_MEM_STORE

buffer_store_lds_dword *sgpr_r*[4], *sgpr_o*
S0: sreg, RSRC S1: ssrc_nolit

Store one DWORD from LDS memory to system memory without utilizing VGPRs.

Flags: OPF_MEM_STORE_LDS

buffer_store_short *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
S0: vgpr S1: vgpr S2: sreg, RSRC S3: ssrc_nolit

Untyped buffer store short. Stores S0[15:0].

Flags: OPF_MEM_STORE

buffer_store_short_d16_hi *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
S0: vgpr S1: vgpr S2: sreg, RSRC S3: ssrc_nolit

Untyped buffer store short. Stores S0[31:16].

Flags: OPF_D16, OPF_MEM_STORE

buffer_wbinvl1

Write back and invalidate the shader L1. Always returns ACK to shader.

Flags: SEN_NOOPR

buffer_wbinvl1_vol

Write back and invalidate the shader L1 only for lines that are marked volatile. Always returns ACK to shader.

Flags: SEN_NOOPR

15.1 Notes for Encoding MUBUF

15.1.1 Operands

vgpr_d is a vector GPR to read data from/store result in. In atomic instruction descriptions it is shown as DATA or DATA[*n*] when reading the *n*'th VGPR and as RETURN_DATA or RETURN_DATA[*n*] when writing the *n*'th VGPR. Atomic instructions will only write back data to the VGPRs if the **glc** bit is set.

vgpr_a is a vector GPR containing address information. If two address components are required, the first VGPR is the *index* and the second VGPR is the *offset*.

sgpr_r is a scalar GPR containing the buffer resource constant.

sgpr_o is a scalar GPR with offset to apply to the base address in the buffer resource constant.

In atomic instruction descriptions the final byte memory address generated from *vgpr_a*, *sgpr_r* and *sgpr_o* is shown as ADDR.

15.1.2 Modifiers

`offset:[0...4095]`

Constant offset to include in address calculation.

`offen:{0,1}`

If true, enable offset from VGPR. Default 0. Can also write `nooffen`.

`idxen:{0,1}`
: If true, enable index from VGPR. Default 0. Can also write `noidxen`.

`glc:{0,1}`
: If true, operation is globally coherent. Default 0. Can also write `noglc`.

`slc:{0,1}`
: If true, operation is system coherent. Default 0. Can also write `noslc`.

`lds:{0,1}`
: If true, LDS memory operation. Default 0. Can also write `nolds`.

`tfe:{0,1}`
: TFE bit. Default 0.

16 Encoding MTBUF

Typed vector memory buffer operations.

tbuffer_load_format_d16_x *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

Typed buffer load 1 dword with format conversion.

Flags: OPF_D16, OPF_D16_CH_1, OPF_MEMFMT

tbuffer_load_format_d16_xy *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

Typed buffer load 1 dword with format conversion.

Flags: OPF_D16, OPF_MEMFMT

tbuffer_load_format_d16_xyz *vgpr_d*[2], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

Typed buffer load 2 dwords with format conversion.

Flags: OPF_D16, OPF_D16_CH_3, OPF_MEMFMT

tbuffer_load_format_d16_xyzw *vgpr_d*[2], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

Typed buffer load 2 dwords with format conversion.

Flags: OPF_D16, OPF_MEMFMT

tbuffer_load_format_x *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

Typed buffer load 1 dword with format conversion.

Flags: OPF_MEMFMT

tbuffer_load_format_xy *vgpr_d*[2], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

Typed buffer load 2 dwords with format conversion.

Flags: OPF_MEMFMT

tbuffer_load_format_xyz *vgpr_d*[3], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

Typed buffer load 3 dwords with format conversion.

Flags: OPF_MEMFMT

tbuffer_load_format_xyzw *vgpr_d*[4], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
D0: vgpr S0: vgpr S1: sreg, RSRC S2: ssrc_nolit

Typed buffer load 4 dwords with format conversion.

Flags: OPF_MEMFMT

tbuffer_store_format_d16_x *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
S0: vgpr S1: vgpr S2: sreg, RSRC S3: ssrc_nolit

Typed buffer store 1 dword with format conversion.

Flags: OPF_D16, OPF_D16_CH_1, OPF_MEMFMT, OPF_MEM_STORE

tbuffer_store_format_d16_xy *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
S0: vgpr S1: vgpr S2: sreg, RSRC S3: ssrc_nolit

Typed buffer store 1 dword with format conversion.

Flags: OPF_D16, OPF_MEMFMT, OPF_MEM_STORE

tbuffer_store_format_d16_xyz *vgpr_d*[2], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
S0: vgpr S1: vgpr S2: sreg, RSRC S3: ssrc_nolit

Typed buffer store 2 dwords with format conversion.

Flags: OPF_D16, OPF_D16_CH_3, OPF_MEMFMT, OPF_MEM_STORE

tbuffer_store_format_d16_xyzw *vgpr_d*[2], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
S0: vgpr S1: vgpr S2: sreg, RSRC S3: ssrc_nolit

Typed buffer store 2 dwords with format conversion.

Flags: OPF_D16, OPF_MEMFMT, OPF_MEM_STORE

tbuffer_store_format_x *vgpr_d*, *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
S0: vgpr S1: vgpr S2: sreg, RSRC S3: ssrc_nolit

Typed buffer store 1 dword with format conversion.

Flags: OPF_MEMFMT, OPF_MEM_STORE

tbuffer_store_format_xy *vgpr_d*[2], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
S0: vgpr S1: vgpr S2: sreg, RSRC S3: ssrc_nolit

Typed buffer store 2 dwords with format conversion.

Flags: OPF_MEMFMT, OPF_MEM_STORE

tbuffer_store_format_xyz *vgpr_d*[3], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
S0: vgpr S1: vgpr S2: sreg, RSRC S3: ssrc_nolit

Typed buffer store 3 dwords with format conversion.

Flags: OPF_MEMFMT, OPF_MEM_STORE

tbuffer_store_format_xyzw *vgpr_d*[4], *vgpr_a*[2], *sgpr_r*[4], *sgpr_o*
S0: vgpr S1: vgpr S2: sreg, RSRC S3: ssrc_nolit

Typed buffer store 4 dwords with format conversion.

Flags: OPF_MEMFMT, OPF_MEM_STORE

16.1 Notes for Encoding MTBUF

16.1.1 Operands

vgpr_d is a vector GPR to read data from/store result in. In atomic instruction descriptions it is shown as DATA or DATA[*n*] when reading the *n*'th VGPR and as RETURN_DATA or RETURN_DATA[*n*] when writing the *n*'th VGPR. Atomic instructions will only write back data to the VGPRs if the **glc** bit is set.

vgpr_a is a vector GPR containing address information. If two address components are required, the first VGPR is the *index* and the second VGPR is the *offset*.

sgpr_r is a scalar GPR containing the buffer resource constant.

sgpr_o is a scalar GPR with offset to apply to the base address in the buffer resource constant.

16.1.2 Modifiers

`offset:[0. . . 4095]`
: Constant offset to include in address calculation.

`offen:{0,1}`
: If true, enable offset from VGPR. Default 0. Can also write `nooffen`.

`idxen:{0,1}`
: If true, enable index from VGPR. Default 0. Can also write `noidxen`.

`glc:{0,1}`
: If true, operation is globally coherent. Default 0. Can also write `noglc`.

`slc:{0,1}`
: If true, operation is system coherent. Default 0. Can also write `noslc`.

`format:`*format_vector*
: Format for operation. A vector containing an NFMT entry and a DFMT entry.

`tfe:{0,1}`
: TFE bit. Default 0.

17 Encoding MIMG

Memory image (texture) operations.

image_atomic_add *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8]
D0: vgpr, ↔ S0: vgpr S1: sreg, IMG

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] += DATA;
RETURN_DATA = tmp.
```

Flags: OPF_MEM_ATOMIC

image_atomic_and *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8]
D0: vgpr, ↔ S0: vgpr S1: sreg, IMG

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] &= DATA;
RETURN_DATA = tmp.
```

Flags: OPF_MEM_ATOMIC

image_atomic_cmpswap *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8]
D0: vgpr, ↔ S0: vgpr S1: sreg, IMG

```
// 32bit
tmp = MEM[ADDR];
src = DATA[0];
cmp = DATA[1];
MEM[ADDR] = (tmp == cmp) ? src : tmp;
RETURN_DATA[0] = tmp.
```

Flags: OPF_ATOMIC_CMPSWAP, OPF_MEM_ATOMIC

image_atomic_dec *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8]
D0: vgpr, ↔ S0: vgpr S1: sreg, IMG

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (tmp == 0 || tmp > DATA) ? DATA : tmp - 1; // unsigned compare
RETURN_DATA = tmp.
```

Flags: OPF_MEM_ATOMIC

image_atomic_inc *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8]
D0: vgpr, ↔ S0: vgpr S1: sreg, IMG

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (tmp ≥ DATA) ? 0 : tmp + 1; // unsigned compare
RETURN_DATA = tmp.
```

Flags: OPF_MEM_ATOMIC

image_atomic_or *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8]
D0: vgpr, ↔ S0: vgpr S1: sreg, IMG

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] |= DATA;
RETURN_DATA = tmp.
```

Flags: OPF_MEM_ATOMIC

image_atomic_smax *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8]
D0: vgpr, ↔ S0: vgpr S1: sreg, IMG

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (DATA > tmp) ? DATA : tmp; // signed compare
RETURN_DATA = tmp.
```

Flags: OPF_MEM_ATOMIC

image_atomic_smin *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8]
D0: vgpr, ↔ S0: vgpr S1: sreg, IMG

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (DATA < tmp) ? DATA : tmp; // signed compare
RETURN_DATA = tmp.
```

Flags: OPF_MEM_ATOMIC

image_atomic_sub *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8]
D0: vgpr, ↔ S0: vgpr S1: sreg, IMG

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] -= DATA;
RETURN_DATA = tmp.
```

Flags: OPF_MEM_ATOMIC

image_atomic_swap *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8]
D0: vgpr, ↔ S0: vgpr S1: sreg, IMG

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = DATA;
RETURN_DATA = tmp.
```

Flags: OPF_MEM_ATOMIC

image_atomic_umax *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8]
D0: vgpr, ↔ S0: vgpr S1: sreg, IMG

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (DATA > tmp) ? DATA : tmp; // unsigned compare
RETURN_DATA = tmp.
```

Flags: OPF_MEM_ATOMIC

image_atomic_umin *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8]
D0: vgpr, ↔ S0: vgpr S1: sreg, IMG

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (DATA < tmp) ? DATA : tmp; // unsigned compare
RETURN_DATA = tmp.
```

Flags: OPF_MEM_ATOMIC

image_atomic_xor *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8]
D0: vgpr, ↔ S0: vgpr S1: sreg, IMG

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] ^= DATA;
RETURN_DATA = tmp.
```

Flags: OPF_MEM_ATOMIC

image_gather4 *vgpr_d*[4], *vgpr_a*[3], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP

gather 4 single component elements (2x2).

Flags: OPF_D16, OPF_GATHER4, OPF_SAMPLE

image_gather4_a *vgpr_d*[4], *vgpr_a*[3], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP

gather 4 single component elements (2x2).

Flags: OPF_D16, OPF_GATHER4, OPF_GRADADJUST, OPF_SAMPLE

image_gather4_b *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP

gather 4 single component elements (2x2) with user bias.

Flags: OPF_BIAS, OPF_D16, OPF_GATHER4, OPF_SAMPLE

image_gather4_b_a *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP

gather 4 single component elements (2x2) with user bias.

Flags: OPF_BIAS, OPF_D16, OPF_GATHER4, OPF_GRADADJUST, OPF_SAMPLE

image_gather4_b_cl *vgpr_d*[4], *vgpr_a*[5], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP

gather 4 single component elements (2x2) with user bias and clamp.

Flags: OPF_ACNT, OPF_BIAS, OPF_CLAMP, OPF_D16, OPF_GATHER4, OPF_SAMPLE

image_gather4_b_cl_a *vgpr_d*[4], *vgpr_a*[5], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP

gather 4 single component elements (2x2) with user bias and clamp.

Flags: OPF_ACNT, OPF_BIAS, OPF_CLAMP, OPF_D16, OPF_GATHER4, OPF_GRADADJUST, OPF_SAMPLE

image_gather4_b_cl_o *vgpr_d*[4], *vgpr_a*[6], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
GATHER4_B_CL, with user offsets.
Flags: OPF_ACNT, OPF_BIAS, OPF_CLAMP, OPF_D16, OPF_GATHER4, OPF_OFFSET, OPF_SAMPLE

image_gather4_b_cl_o_a *vgpr_d*[4], *vgpr_a*[6], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
GATHER4_B_CL, with user offsets.
Flags: OPF_ACNT, OPF_BIAS, OPF_CLAMP, OPF_D16, OPF_GATHER4, OPF_GRADADJUST, OPF_OFFSET, OPF_SAMPLE

image_gather4_b_o *vgpr_d*[4], *vgpr_a*[5], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
GATHER4_B, with user offsets.
Flags: OPF_BIAS, OPF_D16, OPF_GATHER4, OPF_OFFSET, OPF_SAMPLE

image_gather4_b_o_a *vgpr_d*[4], *vgpr_a*[5], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
GATHER4_B, with user offsets.
Flags: OPF_BIAS, OPF_D16, OPF_GATHER4, OPF_GRADADJUST, OPF_OFFSET, OPF_SAMPLE

image_gather4_c *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
gather 4 single component elements (2x2) with PCF.
Flags: OPF_COMP, OPF_D16, OPF_GATHER4, OPF_SAMPLE

image_gather4_c_a *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
gather 4 single component elements (2x2) with PCF.
Flags: OPF_COMP, OPF_D16, OPF_GATHER4, OPF_GRADADJUST, OPF_SAMPLE

image_gather4_c_b *vgpr_d*[4], *vgpr_a*[5], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
gather 4 single component elements (2x2) with user bias and PCF.
Flags: OPF_BIAS, OPF_COMP, OPF_D16, OPF_GATHER4, OPF_SAMPLE

image_gather4_c_b_a *vgpr_d*[4], *vgpr_a*[5], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
gather 4 single component elements (2x2) with user bias and PCF.
Flags: OPF_BIAS, OPF_COMP, OPF_D16, OPF_GATHER4, OPF_GRADADJUST, OPF_SAMPLE

image_gather4_c_b_cl *vgpr_d*[4], *vgpr_a*[6], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
gather 4 single component elements (2x2) with user bias, clamp and PCF.
Flags: OPF_ACNT, OPF_BIAS, OPF_CLAMP, OPF_COMP, OPF_D16, OPF_GATHER4, OPF_SAMPLE

image_gather4_c_b_cl_a *vgpr_d*[4], *vgpr_a*[6], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
gather 4 single component elements (2x2) with user bias, clamp and PCF.
Flags: OPF_ACNT, OPF_BIAS, OPF_CLAMP, OPF_COMP, OPF_D16, OPF_GATHER4, OPF_GRADADJUST, OPF_SAMPLE

image_gather4_c_b_cl_o *vgpr_d*[4], *vgpr_a*[7], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
GATHER4_B_CL, with user offsets.
Flags: OPF_ACNT, OPF_BIAS, OPF_CLAMP, OPF_COMP, OPF_D16, OPF_GATHER4, OPF_OFFSET, OPF_SAMPLE

image_gather4_c_b_cl_o_a *vgpr_d*[4], *vgpr_a*[7], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
GATHER4_B_CL, with user offsets.
Flags: OPF_ACNT, OPF_BIAS, OPF_CLAMP, OPF_COMP, OPF_D16, OPF_GATHER4, OPF_GRADADJUST, OPF_OFFSET, OPF_SAMPLE

image_gather4_c_b_o *vgpr_d*[4], *vgpr_a*[6], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
GATHER4_B, with user offsets.
Flags: OPF_BIAS, OPF_COMP, OPF_D16, OPF_GATHER4, OPF_OFFSET, OPF_SAMPLE

image_gather4_c_b_o_a *vgpr_d*[4], *vgpr_a*[6], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
GATHER4_B, with user offsets.
Flags: OPF_BIAS, OPF_COMP, OPF_D16, OPF_GATHER4, OPF_GRADADJUST, OPF_OFFSET, OPF_SAMPLE

image_gather4_c_cl *vgpr_d*[4], *vgpr_a*[5], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
gather 4 single component elements (2x2) with user LOD clamp and PCF.
Flags: OPF_ACNT, OPF_CLAMP, OPF_COMP, OPF_D16, OPF_GATHER4, OPF_SAMPLE

image_gather4_c_cl_a *vgpr_d*[4], *vgpr_a*[5], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
gather 4 single component elements (2x2) with user LOD clamp and PCF.
Flags: OPF_ACNT, OPF_CLAMP, OPF_COMP, OPF_D16, OPF_GATHER4, OPF_GRADADJUST, OPF_SAMPLE

image_gather4_c_cl_o *vgpr_d*[4], *vgpr_a*[6], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
GATHER4_C_CL, with user offsets.
Flags: OPF_ACNT, OPF_CLAMP, OPF_COMP, OPF_D16, OPF_GATHER4, OPF_OFFSET, OPF_SAMPLE

image_gather4_c_cl_o_a *vgpr_d*[4], *vgpr_a*[6], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
GATHER4_C_CL, with user offsets.
Flags: OPF_ACNT, OPF_CLAMP, OPF_COMP, OPF_D16, OPF_GATHER4, OPF_GRADADJUST, OPF_OFFSET, OPF_SAMPLE

image_gather4_c_l *vgpr_d*[4], *vgpr_a*[5], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
gather 4 single component elements (2x2) with user LOD and PCF.
Flags: OPF_ACNT, OPF_COMP, OPF_D16, OPF_GATHER4, OPF_LOD, OPF_SAMPLE

image_gather4_c_l_o *vgpr_d*[4], *vgpr_a*[6], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
GATHER4_C_L, with user offsets.
Flags: OPF_ACNT, OPF_COMP, OPF_D16, OPF_GATHER4, OPF_LOD, OPF_OFFSET, OPF_SAMPLE

image_gather4_c_lz *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
gather 4 single component elements (2x2) at level 0, with PCF.
Flags: OPF_COMP, OPF_D16, OPF_GATHER4, OPF_LZ, OPF_SAMPLE

image_gather4_c_lz_o *vgpr_d*[4], *vgpr_a*[5], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
GATHER4_C_LZ, with user offsets.
Flags: OPF_COMP, OPF_D16, OPF_GATHER4, OPF_LZ, OPF_OFFSET, OPF_SAMPLE

image_gather4_c_o *vgpr_d*[4], *vgpr_a*[5], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
GATHER4_C, with user offsets.
Flags: OPF_COMP, OPF_D16, OPF_GATHER4, OPF_OFFSET, OPF_SAMPLE

image_gather4_c_o_a *vgpr_d*[4], *vgpr_a*[5], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
GATHER4_C, with user offsets.
Flags: OPF_COMP, OPF_D16, OPF_GATHER4, OPF_GRADADJUST, OPF_OFFSET, OPF_SAMPLE

image_gather4_cl *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
gather 4 single component elements (2x2) with user LOD clamp.
Flags: OPF_ACNT, OPF_CLAMP, OPF_D16, OPF_GATHER4, OPF_SAMPLE

image_gather4_cl_a *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
gather 4 single component elements (2x2) with user LOD clamp.
Flags: OPF_ACNT, OPF_CLAMP, OPF_D16, OPF_GATHER4, OPF_GRADADJUST, OPF_SAMPLE

image_gather4_cl_o *vgpr_d*[4], *vgpr_a*[5], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
GATHER4_CL, with user offsets.
Flags: OPF_ACNT, OPF_CLAMP, OPF_D16, OPF_GATHER4, OPF_OFFSET, OPF_SAMPLE

image_gather4_cl_o_a *vgpr_d*[4], *vgpr_a*[5], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
GATHER4_CL, with user offsets.
Flags: OPF_ACNT, OPF_CLAMP, OPF_D16, OPF_GATHER4, OPF_GRADADJUST, OPF_OFFSET, OPF_SAMPLE

image_gather4_l *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
gather 4 single component elements (2x2) with user LOD.
Flags: OPF_ACNT, OPF_D16, OPF_GATHER4, OPF_LOD, OPF_SAMPLE

image_gather4_l_o *vgpr_d*[4], *vgpr_a*[5], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
GATHER4_L, with user offsets.
Flags: OPF_ACNT, OPF_D16, OPF_GATHER4, OPF_LOD, OPF_OFFSET, OPF_SAMPLE

image_gather4_lz *vgpr_d*[4], *vgpr_a*[3], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
gather 4 single component elements (2x2) at level 0.
Flags: OPF_D16, OPF_GATHER4, OPF_LZ, OPF_SAMPLE

image_gather4_lz_o *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
GATHER4_LZ, with user offsets.
Flags: OPF_D16, OPF_GATHER4, OPF_LZ, OPF_OFFSET, OPF_SAMPLE

image_gather4_o *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
GATHER4, with user offsets.
Flags: OPF_D16, OPF_GATHER4, OPF_OFFSET, OPF_SAMPLE

image_gather4_o_a *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
GATHER4, with user offsets.
Flags: OPF_D16, OPF_GATHER4, OPF_GRADADJUST, OPF_OFFSET, OPF_SAMPLE

image_gather4h *vgpr_d*[4], *vgpr_a*[3], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
Same as Gather4, but fetches one component per texel, from a 4x1 group of texels.
Flags: OPF_D16, OPF_GATHER4, OPF_GATHERH, OPF_SAMPLE

image_gather4h_pck *vgpr_d*[4], *vgpr_a*[3], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
Same as GATHER4H, but fetched elements are treated as a single component and packed into GPR(s).
Flags: OPF_GATHER4, OPF_GATHERH, OPF_SAMPLE

image_gather8h_pck *vgpr_d*[4], *vgpr_a*[3], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP

Simliar to GATHER4H_PCK, but packs eight elements from a 8x1 group of texels.

Flags: OPF_GATHER8, OPF_GATHERH, OPF_SAMPLE

image_get_lod *vgpr_d*[4], *vgpr_a*[3], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP

Return calculated LOD. Vdata gets 2 32bit integer values: { rawLOD, clampedLOD }.

Flags: OPF_SAMPLE

image_get_resinfo *vgpr_d*[4], *vgpr_a*, *sgpr_r*[8]
D0: vgpr S0: vgpr S1: sreg, IMG

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}.

Flags: OPF_ACNT, OPF_MIPID

image_load *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8]
D0: vgpr S0: vgpr S1: sreg, IMG

Image memory load with format conversion specified in T#. No sampler.

Flags: OPF_D16

image_load_mip *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8]
D0: vgpr S0: vgpr S1: sreg, IMG

Image memory load with user-supplied mip level. No sampler.

Flags: OPF_ACNT, OPF_D16, OPF_MIPID

image_load_mip_pck *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8]
D0: vgpr S0: vgpr S1: sreg, IMG

Image memory load with user-supplied mip level, no format conversion. No sampler.

Flags: OPF_ACNT, OPF_MIPID

image_load_mip_pck_sgn *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8]
D0: vgpr S0: vgpr S1: sreg, IMG

Image memory load with user-supplied mip level, no format conversion and with sign extension. No sampler.

Flags: OPF_ACNT, OPF_MIPID

image_load_pck *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8]
D0: vgpr S0: vgpr S1: sreg, IMG

Image memory load with no format conversion. No sampler.

image_load_pck_sgn *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8]
D0: vgpr S0: vgpr S1: sreg, IMG

Image memory load with with no format conversion and sign extension. No sampler.

image_sample *vgpr_d*[4], *vgpr_a*[3], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP

sample texture map.

Flags: OPF_D16, OPF_SAMPLE

image_sample_a *vgpr_d*[4], *vgpr_a*[3], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP

sample texture map.

Flags: OPF_D16, OPF_GRADADJUST, OPF_SAMPLE

image_sample_b *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP

sample texture map, with lod bias.

Flags: OPF_BIAS, OPF_D16, OPF_SAMPLE

image_sample_b_a *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP

sample texture map, with lod bias.

Flags: OPF_BIAS, OPF_D16, OPF_GRADADJUST, OPF_SAMPLE

image_sample_b_cl *vgpr_d*[4], *vgpr_a*[5], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP

sample texture map, with LOD clamp specified in shader, with lod bias.

Flags: OPF_ACNT, OPF_BIAS, OPF_CLAMP, OPF_D16, OPF_SAMPLE

image_sample_b_cl_a *vgpr_d*[4], *vgpr_a*[5], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP

sample texture map, with LOD clamp specified in shader, with lod bias.

Flags: OPF_ACNT, OPF_BIAS, OPF_CLAMP, OPF_D16, OPF_GRADADJUST, OPF_SAMPLE

image_sample_b_cl_o *vgpr_d*[4], *vgpr_a*[6], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP

SAMPLE_O, with LOD clamp specified in shader, with lod bias.

Flags: OPF_ACNT, OPF_BIAS, OPF_CLAMP, OPF_D16, OPF_OFFSET, OPF_SAMPLE

image_sample_b_cl_o_a *vgpr_d*[4], *vgpr_a*[6], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP

SAMPLE_O, with LOD clamp specified in shader, with lod bias.

Flags: OPF_ACNT, OPF_BIAS, OPF_CLAMP, OPF_D16, OPF_GRADADJUST, OPF_OFFSET, OPF_SAMPLE

image_sample_b_o *vgpr_d*[4], *vgpr_a*[5], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP

SAMPLE_O, with lod bias.

Flags: OPF_BIAS, OPF_D16, OPF_OFFSET, OPF_SAMPLE

image_sample_b_o_a *vgpr_d*[4], *vgpr_a*[5], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
SAMPLE_O, with lod bias.
Flags: OPF_BIAS, OPF_D16, OPF_GRADADJUST, OPF_OFFSET, OPF_SAMPLE

image_sample_c *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
sample texture map, with PCF.
Flags: OPF_COMP, OPF_D16, OPF_SAMPLE

image_sample_c_a *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
sample texture map, with PCF.
Flags: OPF_COMP, OPF_D16, OPF_GRADADJUST, OPF_SAMPLE

image_sample_c_b *vgpr_d*[4], *vgpr_a*[5], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
SAMPLE_C, with lod bias.
Flags: OPF_BIAS, OPF_COMP, OPF_D16, OPF_SAMPLE

image_sample_c_b_a *vgpr_d*[4], *vgpr_a*[5], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
SAMPLE_C, with lod bias.
Flags: OPF_BIAS, OPF_COMP, OPF_D16, OPF_GRADADJUST, OPF_SAMPLE

image_sample_c_b_cl *vgpr_d*[4], *vgpr_a*[6], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
SAMPLE_C, with LOD clamp specified in shader, with lod bias.
Flags: OPF_ACNT, OPF_BIAS, OPF_CLAMP, OPF_COMP, OPF_D16, OPF_SAMPLE

image_sample_c_b_cl_a *vgpr_d*[4], *vgpr_a*[6], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
SAMPLE_C, with LOD clamp specified in shader, with lod bias.
Flags: OPF_ACNT, OPF_BIAS, OPF_CLAMP, OPF_COMP, OPF_D16, OPF_GRADADJUST, OPF_SAMPLE

image_sample_c_b_cl_o *vgpr_d*[4], *vgpr_a*[7], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
SAMPLE_C_O, with LOD clamp specified in shader, with lod bias.
Flags: OPF_ACNT, OPF_BIAS, OPF_CLAMP, OPF_COMP, OPF_D16, OPF_OFFSET, OPF_SAMPLE

image_sample_c_b_cl_o_a *vgpr_d*[4], *vgpr_a*[7], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
SAMPLE_C_O, with LOD clamp specified in shader, with lod bias.
Flags: OPF_ACNT, OPF_BIAS, OPF_CLAMP, OPF_COMP, OPF_D16, OPF_GRADADJUST, OPF_OFFSET, OPF_SAMPLE

image_sample_c_b_o *vgpr_d*[4], *vgpr_a*[6], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
SAMPLE_C_O, with lod bias.
Flags: OPF_BIAS, OPF_COMP, OPF_D16, OPF_OFFSET, OPF_SAMPLE

image_sample_c_b_o_a *vgpr_d*[4], *vgpr_a*[6], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
SAMPLE_C_O, with lod bias.
Flags: OPF_BIAS, OPF_COMP, OPF_D16, OPF_GRADADJUST, OPF_OFFSET, OPF_SAMPLE

image_sample_c_cd *vgpr_d*[4], *vgpr_a*[10], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
SAMPLE_C, with user derivatives (LOD per quad).
Flags: OPF_COMP, OPF_D16, OPF_DERIV, OPF_SAMPLE

image_sample_c_cd_cl *vgpr_d*[4], *vgpr_a*[11], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
SAMPLE_C, with LOD clamp specified in shader, with user derivatives (LOD per quad).
Flags: OPF_ACNT, OPF_CLAMP, OPF_COMP, OPF_D16, OPF_DERIV, OPF_SAMPLE

image_sample_c_cd_cl_o *vgpr_d*[4], *vgpr_a*[12], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
SAMPLE_C_O, with LOD clamp specified in shader, with user derivatives (LOD per quad).
Flags: OPF_ACNT, OPF_CLAMP, OPF_COMP, OPF_D16, OPF_DERIV, OPF_OFFSET, OPF_SAMPLE

image_sample_c_cd_o *vgpr_d*[4], *vgpr_a*[11], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
SAMPLE_C_O, with user derivatives (LOD per quad).
Flags: OPF_COMP, OPF_D16, OPF_DERIV, OPF_OFFSET, OPF_SAMPLE

image_sample_c_cl *vgpr_d*[4], *vgpr_a*[5], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
SAMPLE_C, with LOD clamp specified in shader.
Flags: OPF_ACNT, OPF_CLAMP, OPF_COMP, OPF_D16, OPF_SAMPLE

image_sample_c_cl_a *vgpr_d*[4], *vgpr_a*[5], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
SAMPLE_C, with LOD clamp specified in shader.
Flags: OPF_ACNT, OPF_CLAMP, OPF_COMP, OPF_D16, OPF_GRADADJUST, OPF_SAMPLE

image_sample_c_cl_o *vgpr_d*[4], *vgpr_a*[6], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
SAMPLE_C_O, with LOD clamp specified in shader.
Flags: OPF_ACNT, OPF_CLAMP, OPF_COMP, OPF_D16, OPF_OFFSET, OPF_SAMPLE

image_sample_c_cl_o_a *vgpr_d*[4], *vgpr_a*[6], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
SAMPLE_C_O, with LOD clamp specified in shader.
Flags: OPF_ACNT, OPF_CLAMP, OPF_COMP, OPF_D16, OPF_GRADADJUST, OPF_OFFSET, OPF_SAMPLE

image_sample_c_d *vgpr_d*[4], *vgpr_a*[10], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
SAMPLE_C, with user derivatives.
Flags: OPF_COMP, OPF_D16, OPF_DERIV, OPF_SAMPLE

image_sample_c_d_cl *vgpr_d*[4], *vgpr_a*[11], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
SAMPLE_C, with LOD clamp specified in shader, with user derivatives.
Flags: OPF_ACNT, OPF_CLAMP, OPF_COMP, OPF_D16, OPF_DERIV, OPF_SAMPLE

image_sample_c_d_cl_o *vgpr_d*[4], *vgpr_a*[12], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
SAMPLE_C_O, with LOD clamp specified in shader, with user derivatives.
Flags: OPF_ACNT, OPF_CLAMP, OPF_COMP, OPF_D16, OPF_DERIV, OPF_OFFSET, OPF_SAMPLE

image_sample_c_d_o *vgpr_d*[4], *vgpr_a*[11], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
SAMPLE_C_O, with user derivatives.
Flags: OPF_COMP, OPF_D16, OPF_DERIV, OPF_OFFSET, OPF_SAMPLE

image_sample_c_l *vgpr_d*[4], *vgpr_a*[5], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
SAMPLE_C, with user LOD.
Flags: OPF_ACNT, OPF_COMP, OPF_D16, OPF_LOD, OPF_SAMPLE

image_sample_c_l_o *vgpr_d*[4], *vgpr_a*[6], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
SAMPLE_C_O, with user LOD.
Flags: OPF_ACNT, OPF_COMP, OPF_D16, OPF_LOD, OPF_OFFSET, OPF_SAMPLE

image_sample_c_lz *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
SAMPLE_C, from level 0.
Flags: OPF_COMP, OPF_D16, OPF_LZ, OPF_SAMPLE

image_sample_c_lz_o *vgpr_d*[4], *vgpr_a*[5], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
SAMPLE_C_O, from level 0.
Flags: OPF_COMP, OPF_D16, OPF_LZ, OPF_OFFSET, OPF_SAMPLE

image_sample_c_o *vgpr_d*[4], *vgpr_a*[5], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
SAMPLE_C with user specified offsets.
Flags: OPF_COMP, OPF_D16, OPF_OFFSET, OPF_SAMPLE

image_sample_c_o_a *vgpr_d*[4], *vgpr_a*[5], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
SAMPLE_C with user specified offsets.
Flags: OPF_COMP, OPF_D16, OPF_GRADADJUST, OPF_OFFSET, OPF_SAMPLE

image_sample_cd *vgpr_d*[4], *vgpr_a*[9], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
sample texture map, with user derivatives (LOD per quad)
Flags: OPF_D16, OPF_DERIV, OPF_SAMPLE

image_sample_cd_cl *vgpr_d*[4], *vgpr_a*[10], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
sample texture map, with LOD clamp specified in shader, with user derivatives (LOD per quad).
Flags: OPF_ACNT, OPF_CLAMP, OPF_D16, OPF_DERIV, OPF_SAMPLE

image_sample_cd_cl_o *vgpr_d*[4], *vgpr_a*[11], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
SAMPLE_O, with LOD clamp specified in shader, with user derivatives (LOD per quad).
Flags: OPF_ACNT, OPF_CLAMP, OPF_D16, OPF_DERIV, OPF_OFFSET, OPF_SAMPLE

image_sample_cd_o *vgpr_d*[4], *vgpr_a*[10], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
SAMPLE_O, with user derivatives (LOD per quad).
Flags: OPF_D16, OPF_DERIV, OPF_OFFSET, OPF_SAMPLE

image_sample_cl *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
sample texture map, with LOD clamp specified in shader.
Flags: OPF_ACNT, OPF_CLAMP, OPF_D16, OPF_SAMPLE

image_sample_cl_a *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
sample texture map, with LOD clamp specified in shader.
Flags: OPF_ACNT, OPF_CLAMP, OPF_D16, OPF_GRADADJUST, OPF_SAMPLE

image_sample_cl_o *vgpr_d*[4], *vgpr_a*[5], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
SAMPLE_O with LOD clamp specified in shader.
Flags: OPF_ACNT, OPF_CLAMP, OPF_D16, OPF_OFFSET, OPF_SAMPLE

image_sample_cl_o_a *vgpr_d*[4], *vgpr_a*[5], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
SAMPLE_O with LOD clamp specified in shader.
Flags: OPF_ACNT, OPF_CLAMP, OPF_D16, OPF_GRADADJUST, OPF_OFFSET, OPF_SAMPLE

image_sample_d *vgpr_d*[4], *vgpr_a*[9], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
sample texture map, with user derivatives
Flags: OPF_D16, OPF_DERIV, OPF_SAMPLE

image_sample_d_cl *vgpr_d*[4], *vgpr_a*[10], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
sample texture map, with LOD clamp specified in shader, with user derivatives.
Flags: OPF_ACNT, OPF_CLAMP, OPF_D16, OPF_DERIV, OPF_SAMPLE

image_sample_d_cl_o *vgpr_d*[4], *vgpr_a*[11], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
SAMPLE_O, with LOD clamp specified in shader, with user derivatives.
Flags: OPF_ACNT, OPF_CLAMP, OPF_D16, OPF_DERIV, OPF_OFFSET, OPF_SAMPLE

image_sample_d_o *vgpr_d*[4], *vgpr_a*[10], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
SAMPLE_O, with user derivatives.
Flags: OPF_D16, OPF_DERIV, OPF_OFFSET, OPF_SAMPLE

image_sample_l *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
sample texture map, with user LOD.
Flags: OPF_ACNT, OPF_D16, OPF_LOD, OPF_SAMPLE

image_sample_l_o *vgpr_d*[4], *vgpr_a*[5], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
SAMPLE_O, with user LOD.
Flags: OPF_ACNT, OPF_D16, OPF_LOD, OPF_OFFSET, OPF_SAMPLE

image_sample_lz *vgpr_d*[4], *vgpr_a*[3], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
sample texture map, from level 0.
Flags: OPF_D16, OPF_LZ, OPF_SAMPLE

image_sample_lz_o *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP
SAMPLE_O, from level 0.
Flags: OPF_D16, OPF_LZ, OPF_OFFSET, OPF_SAMPLE

image_sample_o *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP

sample texture map, with user offsets.

Flags: OPF_D16, OPF_OFFSET, OPF_SAMPLE

image_sample_o_a *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8], *sgpr_s*[4]
D0: vgpr S0: vgpr, F32 S1: sreg, IMG S2: sreg, SAMP

sample texture map, with user offsets.

Flags: OPF_D16, OPF_GRADADJUST, OPF_OFFSET, OPF_SAMPLE

image_store *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8]
S0: vgpr S1: vgpr S2: sreg, IMG

Image memory store with format conversion specified in T#. No sampler.

Flags: OPF_D16, OPF_MEM_STORE

image_store_mip *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8]
S0: vgpr S1: vgpr S2: sreg, IMG

Image memory store with format conversion specified in T# to user specified mip level. No sampler.

Flags: OPF_ACNT, OPF_D16, OPF_MEM_STORE, OPF_MIPID

image_store_mip_pck *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8]
S0: vgpr S1: vgpr S2: sreg, IMG

Image memory store of packed data without format conversion to user-supplied mip level. No sampler.

Flags: OPF_ACNT, OPF_MEM_STORE, OPF_MIPID

image_store_pck *vgpr_d*[4], *vgpr_a*[4], *sgpr_r*[8]
S0: vgpr S1: vgpr S2: sreg, IMG

Image memory store of packed data without format conversion . No sampler.

Flags: OPF_MEM_STORE

17.1 Notes for Encoding MIMG

17.1.1 Operands

vgpr_d is a vector GPR to read data from/store result in. In atomic instruction descriptions it is shown as DATA or DATA[*n*] when reading the *n*'th VGPR and as RETURN_DATA or RETURN_DATA[*n*] when writing the *n*'th VGPR. Atomic instructions will only write back data to the VGPRs if the **glc** bit is set.

vgpr_a is a vector GPR with address in memory.

sgpr_r is a scalar GPR with resource information.

sgpr_s is a scalar GPR with sampler information.

17.1.2 Modifiers

`dmask:[1. . . 15]`
: Mask of data components. Default 1.

`glc:{0,1}`
: If true, operation is globally coherent. Default 0. Can also write `noglc`.

`slc:{0,1}`
: If true, operation is system coherent. Default 0. Can also write `noslc`.

`unorm:{0,1}`
: If true, force unnormalized data. Default 0. Can also write `nounorm`.

`tfe:{0,1}`
: TFE bit. Default 0.

`lwe:{0,1}`
: LWE bit. Default 0.

`da:{0,1}`
: If true, declare an array. Default 0. Can also write `noda`.

`a16:{0,1}`
: If true, all address components are 16bits. Default 0.

`d16:{0,1}`
: If true, convert data load to 16bits before VGPR store. Default 0.

17.1.3 Address VGPR

Address VGPR arguments are packed and appear in this order:

`offset`
: Coordinate offsets; packed bitfield (`sample_o` opcodes with OPF_OFFSET set)

`lodBias`
: LOD bias; float (`sample_b` opcodes with OPF_BIAS set)

`zpcf`
: Percentage closest filtering (`sample_c` opcodes with OPF_COMP set)

`dX/dH`
: Derivative (`sample_d` opcodes with OPF_DERIV set; 1D+)

`dY/dH`
: Derivative (`sample_d` opcodes with OPF_DERIV set; 2D+)

`dZ/dH`
: Derivative (`sample_d` opcodes with OPF_DERIV set; 3D only)

`dX/dV`
: Derivative (`sample_d` opcodes with OPF_DERIV set; 1D+)

`dY/dV`
: Derivative (`sample_d` opcodes with OPF_DERIV set; 2D+)

`dZ/dV`
: Derivative (`sample_d` opcodes with OPF_DERIV set; 3D only)

`texelX`
: X coordinate; integer for non-sampler opcodes (1D+)

`texelY`
: Y coordinate; integer for non-sampler opcodes (2D+)

`texelZ`
: Z coordinate; integer for non-sampler opcodes (3D image types only)

`field`
: Field select (only if interlaced mode is selected)

`cubeFace`
: Cubemap face (CUBE image types only)

`index`
: Array index (only if `da` bit set in instruction)

`mipid`
: Mipmap ID (`_mip` opcodes with OPF_MIPID set)

`lod`
: LOD (`sample_l` opcodes with OPF_LOD set)

`lodClamp`
: LOD clamp value (`sample_cl` opcodes with OPF_CLAMP set)

18 Encoding EXP

Graphics export operations.

exp *tgt*, *vgpr_0*, *vgpr_1*, *vgpr_2*, *vgpr_3*
D0: tgt S0: vgpr S1: vgpr S2: vgpr S3: vgpr
Export through SX.

18.1 Notes for Encoding EXP

18.1.1 tgt Values

`mrt0. . . mrt7`
Colour buffer export (PS only).

`mrtz`
Depth buffer export (PS only).

`param0. . . param31`
Parameter export (VS only).

`pos0. . . pos3`
Position export (VS only).

`null`
Null export.

18.1.2 vgpr Values

`v0. . . v255`
Vector GPR to export for this component.

`off`
To indicate no export for this component.

18.1.3 Modifiers

`done:{0,1}`
If true, last export of this type. Default 0. Can also write `nodone`.

`compr:{0,1}`
If true, export as float16 data. Default 0. Can also write `nocompr`.

`vm:{0,1}`
: If true, set VM bit. Default 0. Can also write `novm`.

19 Encoding FLAT

Flat (DUA) addressing instructions.

flat_atomic_add *vgpr_dst*, *vgpr_addr*[2], *vgpr_src*
D0: vgpr S0: vgpr S1: vgpr

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] += DATA;
RETURN_DATA = tmp.
```

Flags: SEN_FLAT , OPF_MEM_ATOMIC, OPF_RDFLAT, OPF_RDM0

flat_atomic_add_x2 *vgpr_dst*[2], *vgpr_addr*[2], *vgpr_src*[2]
D0: vgpr S0: vgpr S1: vgpr

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] += DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: SEN_FLAT , OPF_MEM_ATOMIC, OPF_RDFLAT, OPF_RDM0

flat_atomic_and *vgpr_dst*, *vgpr_addr*[2], *vgpr_src*
D0: vgpr S0: vgpr S1: vgpr

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] &= DATA;
RETURN_DATA = tmp.
```

Flags: SEN_FLAT , OPF_MEM_ATOMIC, OPF_RDFLAT, OPF_RDM0

flat_atomic_and_x2 *vgpr_dst*[2], *vgpr_addr*[2], *vgpr_src*[2]
D0: vgpr S0: vgpr S1: vgpr

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] &= DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: SEN_FLAT , OPF_MEM_ATOMIC, OPF_RDFLAT, OPF_RDM0

flat_atomic_cmpswap *vgpr_dst*[2], *vgpr_addr*[2], *vgpr_src*[2]
D0: vgpr S0: vgpr S1: vgpr

```
// 32bit
tmp = MEM[ADDR];
src = DATA[0];
cmp = DATA[1];
MEM[ADDR] = (tmp == cmp) ? src : tmp;
RETURN_DATA[0] = tmp.
```

Flags: SEN_FLAT , OPF_ATOMIC_CMPSWAP, OPF_MEM_ATOMIC, OPF_RDFLAT, OPF_RDM0

flat_atomic_cmpswap_x2 *vgpr_dst*[4], *vgpr_addr*[2], *vgpr_src*[4]
D0: vgpr S0: vgpr S1: vgpr

```
// 64bit
tmp = MEM[ADDR];
src = DATA[0:1];
cmp = DATA[2:3];
MEM[ADDR] = (tmp == cmp) ? src : tmp;
RETURN_DATA[0:1] = tmp.
```

Flags: SEN_FLAT , OPF_ATOMIC_CMPSWAP, OPF_MEM_ATOMIC, OPF_RDFLAT, OPF_RDM0

flat_atomic_dec *vgpr_dst*, *vgpr_addr*[2], *vgpr_src*
D0: vgpr S0: vgpr S1: vgpr

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (tmp == 0 || tmp > DATA) ? DATA : tmp - 1; // unsigned compare
RETURN_DATA = tmp.
```

Flags: SEN_FLAT , OPF_MEM_ATOMIC, OPF_RDFLAT, OPF_RDM0

flat_atomic_dec_x2 *vgpr_dst*[2], *vgpr_addr*[2], *vgpr_src*[2]
D0: vgpr S0: vgpr S1: vgpr

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] = (tmp == 0 || tmp > DATA[0:1]) ? DATA[0:1] : tmp - 1; // unsigned compare
RETURN_DATA[0:1] = tmp.
```

Flags: SEN_FLAT , OPF_MEM_ATOMIC, OPF_RDFLAT, OPF_RDM0

flat_atomic_inc *vgpr_dst*, *vgpr_addr*[2], *vgpr_src*
D0: vgpr S0: vgpr S1: vgpr

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (tmp ≥ DATA) ? 0 : tmp + 1; // unsigned compare
RETURN_DATA = tmp.
```

Flags: SEN_FLAT , OPF_MEM_ATOMIC, OPF_RDFLAT, OPF_RDM0

flat_atomic_inc_x2 *vgpr_dst*[2], *vgpr_addr*[2], *vgpr_src*[2]
D0: vgpr S0: vgpr S1: vgpr

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] = (tmp ≥ DATA[0:1]) ? 0 : tmp + 1; // unsigned compare
RETURN_DATA[0:1] = tmp.
```

Flags: SEN_FLAT , OPF_MEM_ATOMIC, OPF_RDFLAT, OPF_RDM0

flat_atomic_or *vgpr_dst*, *vgpr_addr*[2], *vgpr_src*
D0: vgpr S0: vgpr S1: vgpr

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] |= DATA;
RETURN_DATA = tmp.
```

Flags: SEN_FLAT , OPF_MEM_ATOMIC, OPF_RDFLAT, OPF_RDM0

flat_atomic_or_x2 *vgpr_dst*[2], *vgpr_addr*[2], *vgpr_src*[2]
D0: vgpr S0: vgpr S1: vgpr

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] |= DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: SEN_FLAT , OPF_MEM_ATOMIC, OPF_RDFLAT, OPF_RDM0

flat_atomic_smax *vgpr_dst*, *vgpr_addr*[2], *vgpr_src*
D0: vgpr S0: vgpr S1: vgpr

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (DATA > tmp) ? DATA : tmp; // signed compare
RETURN_DATA = tmp.
```

Flags: SEN_FLAT , OPF_MEM_ATOMIC, OPF_RDFLAT, OPF_RDM0

flat_atomic_smax_x2 *vgpr_dst*[2], *vgpr_addr*[2], *vgpr_src*[2]
D0: vgpr S0: vgpr S1: vgpr

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] -= (DATA[0:1] > tmp) ? DATA[0:1] : tmp; // signed compare
RETURN_DATA[0:1] = tmp.
```

Flags: SEN_FLAT , OPF_MEM_ATOMIC, OPF_RDFLAT, OPF_RDM0

flat_atomic_smin *vgpr_dst*, *vgpr_addr*[2], *vgpr_src*
D0: vgpr S0: vgpr S1: vgpr

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (DATA < tmp) ? DATA : tmp; // signed compare
RETURN_DATA = tmp.
```

Flags: SEN_FLAT , OPF_MEM_ATOMIC, OPF_RDFLAT, OPF_RDM0

flat_atomic_smin_x2 *vgpr_dst*[2], *vgpr_addr*[2], *vgpr_src*[2]
D0: vgpr S0: vgpr S1: vgpr

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] -= (DATA[0:1] < tmp) ? DATA[0:1] : tmp; // signed compare
RETURN_DATA[0:1] = tmp.
```

Flags: SEN_FLAT , OPF_MEM_ATOMIC, OPF_RDFLAT, OPF_RDM0

flat_atomic_sub *vgpr_dst*, *vgpr_addr*[2], *vgpr_src*
D0: vgpr S0: vgpr S1: vgpr

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] -= DATA;
RETURN_DATA = tmp.
```

Flags: SEN_FLAT , OPF_MEM_ATOMIC, OPF_RDFLAT, OPF_RDM0

flat_atomic_sub_x2 *vgpr_dst*[2], *vgpr_addr*[2], *vgpr_src*[2]
D0: vgpr S0: vgpr S1: vgpr

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] -= DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: SEN_FLAT , OPF_MEM_ATOMIC, OPF_RDFLAT, OPF_RDM0

flat_atomic_swap *vgpr_dst*, *vgpr_addr*[2], *vgpr_src*
D0: vgpr S0: vgpr S1: vgpr

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = DATA;
RETURN_DATA = tmp.
```

Flags: SEN_FLAT , OPF_MEM_ATOMIC, OPF_RDFLAT, OPF_RDM0

flat_atomic_swap_x2 *vgpr_dst*[2], *vgpr_addr*[2], *vgpr_src*[2]
D0: vgpr S0: vgpr S1: vgpr

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] = DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: SEN_FLAT , OPF_MEM_ATOMIC, OPF_RDFLAT, OPF_RDM0

flat_atomic_umax *vgpr_dst*, *vgpr_addr*[2], *vgpr_src*
D0: vgpr S0: vgpr S1: vgpr

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (DATA > tmp) ? DATA : tmp; // unsigned compare
RETURN_DATA = tmp.
```

Flags: SEN_FLAT , OPF_MEM_ATOMIC, OPF_RDFLAT, OPF_RDM0

flat_atomic_umax_x2 *vgpr_dst*[2], *vgpr_addr*[2], *vgpr_src*[2]
D0: vgpr S0: vgpr S1: vgpr

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] -= (DATA[0:1] > tmp) ? DATA[0:1] : tmp; // unsigned compare
RETURN_DATA[0:1] = tmp.
```

Flags: SEN_FLAT , OPF_MEM_ATOMIC, OPF_RDFLAT, OPF_RDM0

flat_atomic_umin *vgpr_dst*, *vgpr_addr*[2], *vgpr_src*
D0: vgpr S0: vgpr S1: vgpr

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (DATA < tmp) ? DATA : tmp; // unsigned compare
RETURN_DATA = tmp.
```

Flags: SEN_FLAT , OPF_MEM_ATOMIC, OPF_RDFLAT, OPF_RDM0

flat_atomic_umin_x2 *vgpr_dst*[2], *vgpr_addr*[2], *vgpr_src*[2]
D0: vgpr S0: vgpr S1: vgpr

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] -= (DATA[0:1] < tmp) ? DATA[0:1] : tmp; // unsigned compare
RETURN_DATA[0:1] = tmp.
```

Flags: SEN_FLAT , OPF_MEM_ATOMIC, OPF_RDFLAT, OPF_RDM0

flat_atomic_xor *vgpr_dst*, *vgpr_addr*[2], *vgpr_src*
D0: vgpr S0: vgpr S1: vgpr

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] ^= DATA;
RETURN_DATA = tmp.
```

Flags: SEN_FLAT , OPF_MEM_ATOMIC, OPF_RDFLAT, OPF_RDM0

flat_atomic_xor_x2 *vgpr_dst*[2], *vgpr_addr*[2], *vgpr_src*[2]
D0: vgpr S0: vgpr S1: vgpr

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] ^= DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: SEN_FLAT , OPF_MEM_ATOMIC, OPF_RDFLAT, OPF_RDM0

flat_load_dword *vgpr_dst*, *vgpr_addr*[2]
D0: vgpr S0: vgpr

Untyped buffer load dword.

Flags: SEN_FLAT , OPF_RDFLAT, OPF_RDM0

flat_load_dwordx2 *vgpr_dst*[2], *vgpr_addr*[2]
D0: vgpr S0: vgpr

Untyped buffer load 2 dwords.

Flags: SEN_FLAT , OPF_RDFLAT, OPF_RDM0

flat_load_dwordx3 *vgpr_dst*[3], *vgpr_addr*[2]
D0: vgpr S0: vgpr

Untyped buffer load 3 dwords.

Flags: SEN_FLAT , OPF_RDFLAT, OPF_RDM0

flat_load_dwordx4 *vgpr_dst*[4], *vgpr_addr*[2]
D0: vgpr S0: vgpr

Untyped buffer load 4 dwords.

Flags: SEN_FLAT , OPF_RDFLAT, OPF_RDM0

flat_load_sbyte *vgpr_dst*, *vgpr_addr*[2]
D0: vgpr S0: vgpr

Untyped buffer load signed byte (sign extend to VGPR destination).

Flags: SEN_FLAT , OPF_RDFLAT, OPF_RDM0

flat_load_sbyte_d16 *vgpr_dst*, *vgpr_addr*[2]
D0: vgpr S0: vgpr

```
D0[15:0] = {8'h0, MEM[ADDR]}.
```

Untyped buffer load signed byte.

Flags: SEN_FLAT , OPF_D16, OPF_RDFLAT, OPF_RDM0

flat_load_sbyte_d16_hi *vgpr_dst*, *vgpr_addr*[2]
D0: vgpr S0: vgpr

```
D0[31:16] = {8'h0, MEM[ADDR]}.
```

Untyped buffer load signed byte.

Flags: SEN_FLAT , OPF_D16, OPF_RDFLAT, OPF_RDM0

flat_load_short_d16 *vgpr_dst*, *vgpr_addr*[2]
D0: vgpr S0: vgpr

```
D0[15:0] = MEM[ADDR].
```

Untyped buffer load short.

Flags: SEN_FLAT , OPF_D16, OPF_RDFLAT, OPF_RDM0

flat_load_short_d16_hi *vgpr_dst*, *vgpr_addr*[2]
D0: vgpr S0: vgpr

```
D0[31:16] = MEM[ADDR].
```

Untyped buffer load short.

Flags: SEN_FLAT , OPF_D16, OPF_RDFLAT, OPF_RDM0

flat_load_sshort *vgpr_dst*, *vgpr_addr*[2]
D0: vgpr S0: vgpr

Untyped buffer load signed short (sign extend to VGPR destination).

Flags: SEN_FLAT , OPF_RDFLAT, OPF_RDM0

flat_load_ubyte *vgpr_dst*, *vgpr_addr*[2]
D0: vgpr S0: vgpr

Untyped buffer load unsigned byte (zero extend to VGPR destination).

Flags: SEN_FLAT , OPF_RDFLAT, OPF_RDM0

flat_load_ubyte_d16 *vgpr_dst*, *vgpr_addr*[2]
D0: vgpr S0: vgpr

```
D0[15:0] = {8'h0, MEM[ADDR]}.
```

Untyped buffer load unsigned byte.

Flags: SEN_FLAT , OPF_D16, OPF_RDFLAT, OPF_RDM0

flat_load_ubyte_d16_hi *vgpr_dst*, *vgpr_addr*[2]
D0: vgpr S0: vgpr

```
D0[31:16] = {8'h0, MEM[ADDR]}.
```

Untyped buffer load unsigned byte.

Flags: SEN_FLAT , OPF_D16, OPF_RDFLAT, OPF_RDM0

flat_load_ushort *vgpr_dst*, *vgpr_addr*[2]
D0: vgpr S0: vgpr

Untyped buffer load unsigned short (zero extend to VGPR destination).

Flags: SEN_FLAT , OPF_RDFLAT, OPF_RDM0

flat_store_byte *vgpr_addr*[2], *vgpr_src*
S0: vgpr S1: vgpr

Untyped buffer store byte. Stores S0[7:0].

Flags: SEN_FLAT , OPF_MEM_STORE, OPF_RDFLAT, OPF_RDM0

flat_store_byte_d16_hi *vgpr_addr*[2], *vgpr_src*
S0: vgpr S1: vgpr

Untyped buffer store byte. Stores S0[23:16].

Flags: SEN_FLAT , OPF_D16, OPF_MEM_STORE, OPF_RDFLAT, OPF_RDM0

flat_store_dword *vgpr_addr*[2], *vgpr_src*
S0: vgpr S1: vgpr

Untyped buffer store dword.

Flags: SEN_FLAT , OPF_MEM_STORE, OPF_RDFLAT, OPF_RDM0

flat_store_dwordx2 *vgpr_addr*[2], *vgpr_src*[2]
S0: vgpr S1: vgpr

Untyped buffer store 2 dwords.

Flags: SEN_FLAT , OPF_MEM_STORE, OPF_RDFLAT, OPF_RDM0

flat_store_dwordx3 *vgpr_addr*[2], *vgpr_src*[3]
S0: vgpr S1: vgpr

Untyped buffer store 3 dwords.

Flags: SEN_FLAT , OPF_MEM_STORE, OPF_RDFLAT, OPF_RDM0

flat_store_dwordx4 *vgpr_addr*[2], *vgpr_src*[4]
S0: vgpr S1: vgpr

Untyped buffer store 4 dwords.

Flags: SEN_FLAT , OPF_MEM_STORE, OPF_RDFLAT, OPF_RDM0

flat_store_short *vgpr_addr*[2], *vgpr_src*
S0: vgpr S1: vgpr

Untyped buffer store short. Stores S0[15:0].

Flags: SEN_FLAT , OPF_MEM_STORE, OPF_RDFLAT, OPF_RDM0

flat_store_short_d16_hi *vgpr_addr*[2], *vgpr_src*
S0: vgpr S1: vgpr

Untyped buffer store short. Stores S0[31:16].

Flags: SEN_FLAT , OPF_D16, OPF_MEM_STORE, OPF_RDFLAT, OPF_RDM0

global_atomic_add *vgpr_dst*, *vgpr_addr*[2], *vgpr_src*, *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: vgpr S2: sreg

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] += DATA;
RETURN_DATA = tmp.
```

Flags: SEN_GLOBAL , OPF_MEM_ATOMIC, OPF_RDM0

global_atomic_add_x2 *vgpr_dst*[2], *vgpr_addr*[2], *vgpr_src*[2], *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: vgpr S2: sreg

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] += DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: SEN_GLOBAL , OPF_MEM_ATOMIC, OPF_RDM0

global_atomic_and *vgpr_dst*, *vgpr_addr*[2], *vgpr_src*, *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: vgpr S2: sreg

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] &= DATA;
RETURN_DATA = tmp.
```

Flags: SEN_GLOBAL , OPF_MEM_ATOMIC, OPF_RDM0

global_atomic_and_x2 *vgpr_dst*[2], *vgpr_addr*[2], *vgpr_src*[2], *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: vgpr S2: sreg

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] &= DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: SEN_GLOBAL , OPF_MEM_ATOMIC, OPF_RDM0

global_atomic_cmpswap *vgpr_dst*[2], *vgpr_addr*[2], *vgpr_src*[2], *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: vgpr S2: sreg

```
// 32bit
tmp = MEM[ADDR];
src = DATA[0];
cmp = DATA[1];
MEM[ADDR] = (tmp == cmp) ? src : tmp;
RETURN_DATA[0] = tmp.
```

Flags: SEN_GLOBAL , OPF_ATOMIC_CMPSWAP, OPF_MEM_ATOMIC, OPF_RDM0

global_atomic_cmpswap_x2 *vgpr_dst*[4], *vgpr_addr*[2], *vgpr_src*[4], *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: vgpr S2: sreg

```
// 64bit
tmp = MEM[ADDR];
src = DATA[0:1];
cmp = DATA[2:3];
MEM[ADDR] = (tmp == cmp) ? src : tmp;
RETURN_DATA[0:1] = tmp.
```

Flags: SEN_GLOBAL , OPF_ATOMIC_CMPSWAP, OPF_MEM_ATOMIC, OPF_RDM0

global_atomic_dec *vgpr_dst*, *vgpr_addr*[2], *vgpr_src*, *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: vgpr S2: sreg

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (tmp == 0 || tmp > DATA) ? DATA : tmp - 1; // unsigned compare
RETURN_DATA = tmp.
```

Flags: SEN_GLOBAL , OPF_MEM_ATOMIC, OPF_RDM0

global_atomic_dec_x2 *vgpr_dst*[2], *vgpr_addr*[2], *vgpr_src*[2], *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: vgpr S2: sreg

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] = (tmp == 0 || tmp > DATA[0:1]) ? DATA[0:1] : tmp - 1; // unsigned compare
RETURN_DATA[0:1] = tmp.
```

Flags: SEN_GLOBAL , OPF_MEM_ATOMIC, OPF_RDM0

global_atomic_inc *vgpr_dst*, *vgpr_addr*[2], *vgpr_src*, *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: vgpr S2: sreg

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (tmp ≥ DATA) ? 0 : tmp + 1; // unsigned compare
RETURN_DATA = tmp.
```

Flags: SEN_GLOBAL , OPF_MEM_ATOMIC, OPF_RDM0

global_atomic_inc_x2 *vgpr_dst*[2], *vgpr_addr*[2], *vgpr_src*[2], *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: vgpr S2: sreg

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] = (tmp ≥ DATA[0:1]) ? 0 : tmp + 1; // unsigned compare
RETURN_DATA[0:1] = tmp.
```

Flags: SEN_GLOBAL , OPF_MEM_ATOMIC, OPF_RDM0

global_atomic_or *vgpr_dst*, *vgpr_addr*[2], *vgpr_src*, *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: vgpr S2: sreg

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] |= DATA;
RETURN_DATA = tmp.
```

Flags: SEN_GLOBAL , OPF_MEM_ATOMIC, OPF_RDM0

global_atomic_or_x2 *vgpr_dst*[2], *vgpr_addr*[2], *vgpr_src*[2], *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: vgpr S2: sreg

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] |= DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: SEN_GLOBAL , OPF_MEM_ATOMIC, OPF_RDM0

global_atomic_smax *vgpr_dst*, *vgpr_addr*[2], *vgpr_src*, *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: vgpr S2: sreg

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (DATA > tmp) ? DATA : tmp; // signed compare
RETURN_DATA = tmp.
```

Flags: SEN_GLOBAL , OPF_MEM_ATOMIC, OPF_RDM0

global_atomic_smax_x2 *vgpr_dst*[2], *vgpr_addr*[2], *vgpr_src*[2], *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: vgpr S2: sreg

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] -= (DATA[0:1] > tmp) ? DATA[0:1] : tmp; // signed compare
RETURN_DATA[0:1] = tmp.
```

Flags: SEN_GLOBAL , OPF_MEM_ATOMIC, OPF_RDM0

global_atomic_smin *vgpr_dst*, *vgpr_addr*[2], *vgpr_src*, *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: vgpr S2: sreg

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (DATA < tmp) ? DATA : tmp; // signed compare
RETURN_DATA = tmp.
```

Flags: SEN_GLOBAL , OPF_MEM_ATOMIC, OPF_RDM0

global_atomic_smin_x2 *vgpr_dst*[2], *vgpr_addr*[2], *vgpr_src*[2], *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: vgpr S2: sreg

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] -= (DATA[0:1] < tmp) ? DATA[0:1] : tmp; // signed compare
RETURN_DATA[0:1] = tmp.
```

Flags: SEN_GLOBAL , OPF_MEM_ATOMIC, OPF_RDM0

global_atomic_sub *vgpr_dst*, *vgpr_addr*[2], *vgpr_src*, *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: vgpr S2: sreg

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] -= DATA;
RETURN_DATA = tmp.
```

Flags: SEN_GLOBAL , OPF_MEM_ATOMIC, OPF_RDM0

global_atomic_sub_x2 *vgpr_dst*[2], *vgpr_addr*[2], *vgpr_src*[2], *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: vgpr S2: sreg

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] -= DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: SEN_GLOBAL , OPF_MEM_ATOMIC, OPF_RDM0

global_atomic_swap *vgpr_dst*, *vgpr_addr*[2], *vgpr_src*, *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: vgpr S2: sreg

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = DATA;
RETURN_DATA = tmp.
```

Flags: SEN_GLOBAL , OPF_MEM_ATOMIC, OPF_RDM0

global_atomic_swap_x2 *vgpr_dst*[2], *vgpr_addr*[2], *vgpr_src*[2], *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: vgpr S2: sreg

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] = DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: SEN_GLOBAL , OPF_MEM_ATOMIC, OPF_RDM0

global_atomic_umax *vgpr_dst*, *vgpr_addr*[2], *vgpr_src*, *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: vgpr S2: sreg

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (DATA > tmp) ? DATA : tmp; // unsigned compare
RETURN_DATA = tmp.
```

Flags: SEN_GLOBAL , OPF_MEM_ATOMIC, OPF_RDM0

global_atomic_umax_x2 *vgpr_dst*[2], *vgpr_addr*[2], *vgpr_src*[2], *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: vgpr S2: sreg

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] -= (DATA[0:1] > tmp) ? DATA[0:1] : tmp; // unsigned compare
RETURN_DATA[0:1] = tmp.
```

Flags: SEN_GLOBAL , OPF_MEM_ATOMIC, OPF_RDM0

global_atomic_umin *vgpr_dst*, *vgpr_addr*[2], *vgpr_src*, *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: vgpr S2: sreg

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] = (DATA < tmp) ? DATA : tmp; // unsigned compare
RETURN_DATA = tmp.
```

Flags: SEN_GLOBAL , OPF_MEM_ATOMIC, OPF_RDM0

global_atomic_umin_x2 *vgpr_dst*[2], *vgpr_addr*[2], *vgpr_src*[2], *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: vgpr S2: sreg

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] -= (DATA[0:1] < tmp) ? DATA[0:1] : tmp; // unsigned compare
RETURN_DATA[0:1] = tmp.
```

Flags: SEN_GLOBAL , OPF_MEM_ATOMIC, OPF_RDM0

global_atomic_xor *vgpr_dst*, *vgpr_addr*[2], *vgpr_src*, *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: vgpr S2: sreg

```
// 32bit
tmp = MEM[ADDR];
MEM[ADDR] ^= DATA;
RETURN_DATA = tmp.
```

Flags: SEN_GLOBAL , OPF_MEM_ATOMIC, OPF_RDM0

global_atomic_xor_x2 *vgpr_dst*[2], *vgpr_addr*[2], *vgpr_src*[2], *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: vgpr S2: sreg

```
// 64bit
tmp = MEM[ADDR];
MEM[ADDR] ^= DATA[0:1];
RETURN_DATA[0:1] = tmp.
```

Flags: SEN_GLOBAL , OPF_MEM_ATOMIC, OPF_RDM0

global_load_dword *vgpr_dst*, *vgpr_addr*[2], *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: sreg

Untyped buffer load dword.

Flags: SEN_GLOBAL , OPF_RDM0

global_load_dwordx2 *vgpr_dst*[2], *vgpr_addr*[2], *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: sreg

Untyped buffer load 2 dwords.

Flags: SEN_GLOBAL , OPF_RDM0

global_load_dwordx3 *vgpr_dst*[3], *vgpr_addr*[2], *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: sreg

Untyped buffer load 3 dwords.

Flags: SEN_GLOBAL , OPF_RDM0

global_load_dwordx4 *vgpr_dst*[4], *vgpr_addr*[2], *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: sreg

Untyped buffer load 4 dwords.

Flags: SEN_GLOBAL , OPF_RDM0

global_load_sbyte *vgpr_dst*, *vgpr_addr*[2], *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: sreg

Untyped buffer load signed byte (sign extend to VGPR destination).

Flags: SEN_GLOBAL , OPF_RDM0

global_load_sbyte_d16 *vgpr_dst*, *vgpr_addr*[2], *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: sreg

```
D0[15:0] = {8'h0, MEM[ADDR]}.
```

Untyped buffer load signed byte.

Flags: SEN_GLOBAL , OPF_D16, OPF_RDM0

global_load_sbyte_d16_hi *vgpr_dst*, *vgpr_addr*[2], *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: sreg

```
D0[31:16] = {8'h0, MEM[ADDR]}.
```

Untyped buffer load signed byte.

Flags: SEN_GLOBAL , OPF_D16, OPF_RDM0

global_load_short_d16 *vgpr_dst*, *vgpr_addr*[2], *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: sreg

```
D0[15:0] = MEM[ADDR].
```

Untyped buffer load short.

Flags: SEN_GLOBAL , OPF_D16, OPF_RDM0

global_load_short_d16_hi *vgpr_dst*, *vgpr_addr*[2], *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: sreg

```
D0[31:16] = MEM[ADDR].
```

Untyped buffer load short.

Flags: SEN_GLOBAL , OPF_D16, OPF_RDM0

global_load_sshort *vgpr_dst*, *vgpr_addr*[2], *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: sreg

Untyped buffer load signed short (sign extend to VGPR destination).

Flags: SEN_GLOBAL , OPF_RDM0

global_load_ubyte *vgpr_dst*, *vgpr_addr*[2], *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: sreg

Untyped buffer load unsigned byte (zero extend to VGPR destination).

Flags: SEN_GLOBAL , OPF_RDM0

global_load_ubyte_d16 *vgpr_dst*, *vgpr_addr*[2], *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: sreg

```
D0[15:0] = {8'h0, MEM[ADDR]}.
```

Untyped buffer load unsigned byte.

Flags: SEN_GLOBAL , OPF_D16, OPF_RDM0

global_load_ubyte_d16_hi *vgpr_dst*, *vgpr_addr*[2], *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: sreg

```
D0[31:16] = {8'h0, MEM[ADDR]}.
```

Untyped buffer load unsigned byte.

Flags: SEN_GLOBAL , OPF_D16, OPF_RDM0

global_load_ushort *vgpr_dst*, *vgpr_addr*[2], *sgpr_saddr*[2]
D0: vgpr S0: vgpr S1: sreg

Untyped buffer load unsigned short (zero extend to VGPR destination).

Flags: SEN_GLOBAL , OPF_RDM0

global_store_byte *vgpr_addr*[2], *vgpr_src*, *sgpr_saddr*[2]
S0: vgpr S1: vgpr S2: sreg

Untyped buffer store byte. Stores S0[7:0].

Flags: SEN_GLOBAL , OPF_MEM_STORE, OPF_RDM0

global_store_byte_d16_hi *vgpr_addr*[2], *vgpr_src*, *sgpr_saddr*[2]
S0: vgpr S1: vgpr S2: sreg

Untyped buffer store byte. Stores S0[23:16].

Flags: SEN_GLOBAL , OPF_D16, OPF_MEM_STORE, OPF_RDM0

global_store_dword *vgpr_addr*[2], *vgpr_src*, *sgpr_saddr*[2]
S0: vgpr S1: vgpr S2: sreg

Untyped buffer store dword.

Flags: SEN_GLOBAL , OPF_MEM_STORE, OPF_RDM0

global_store_dwordx2 *vgpr_addr*[2], *vgpr_src*[2], *sgpr_saddr*[2]
S0: vgpr S1: vgpr S2: sreg

Untyped buffer store 2 dwords.

Flags: SEN_GLOBAL , OPF_MEM_STORE, OPF_RDM0

global_store_dwordx3 *vgpr_addr*[2], *vgpr_src*[3], *sgpr_saddr*[2]
S0: vgpr S1: vgpr S2: sreg

Untyped buffer store 3 dwords.

Flags: SEN_GLOBAL , OPF_MEM_STORE, OPF_RDM0

global_store_dwordx4 *vgpr_addr*[2], *vgpr_src*[4], *sgpr_saddr*[2]
S0: vgpr S1: vgpr S2: sreg

Untyped buffer store 4 dwords.

Flags: SEN_GLOBAL , OPF_MEM_STORE, OPF_RDM0

global_store_short *vgpr_addr*[2], *vgpr_src*, *sgpr_saddr*[2]
S0: vgpr S1: vgpr S2: sreg

Untyped buffer store short. Stores S0[15:0].

Flags: SEN_GLOBAL , OPF_MEM_STORE, OPF_RDM0

global_store_short_d16_hi *vgpr_addr*[2], *vgpr_src*, *sgpr_saddr*[2]
S0: vgpr S1: vgpr S2: sreg

Untyped buffer store short. Stores S0[31:16].

Flags: SEN_GLOBAL , OPF_D16, OPF_MEM_STORE, OPF_RDM0

scratch_load_dword *vgpr_dst*, *vgpr_addr/off*, *sgpr_saddr/off*
D0: vgpr S0: vgpr S1: sreg

Untyped buffer load dword.

Flags: SEN_SCRATCH , OPF_RDFLAT, OPF_RDM0

scratch_load_dwordx2 *vgpr_dst*[2], *vgpr_addr/off*, *sgpr_saddr/off*
D0: vgpr S0: vgpr S1: sreg

Untyped buffer load 2 dwords.

Flags: SEN_SCRATCH , OPF_RDFLAT, OPF_RDM0

scratch_load_dwordx3 *vgpr_dst*[3], *vgpr_addr/off*, *sgpr_saddr/off*
D0: vgpr S0: vgpr S1: sreg

Untyped buffer load 3 dwords.

Flags: SEN_SCRATCH , OPF_RDFLAT, OPF_RDM0

scratch_load_dwordx4 *vgpr_dst*[4], *vgpr_addr/off*, *sgpr_saddr/off*
D0: vgpr S0: vgpr S1: sreg

Untyped buffer load 4 dwords.

Flags: SEN_SCRATCH , OPF_RDFLAT, OPF_RDM0

scratch_load_sbyte *vgpr_dst*, *vgpr_addr/off*, *sgpr_saddr/off*
D0: vgpr S0: vgpr S1: sreg

Untyped buffer load signed byte (sign extend to VGPR destination).

Flags: SEN_SCRATCH , OPF_RDFLAT, OPF_RDM0

scratch_load_sbyte_d16 *vgpr_dst*, *vgpr_addr/off*, *sgpr_saddr/off*
D0: vgpr S0: vgpr S1: sreg

```
D0[15:0] = {8'h0, MEM[ADDR]}.
```

Untyped buffer load signed byte.

Flags: SEN_SCRATCH , OPF_D16, OPF_RDFLAT, OPF_RDM0

scratch_load_sbyte_d16_hi *vgpr_dst*, *vgpr_addr/off*, *sgpr_saddr/off*
D0: vgpr S0: vgpr S1: sreg

```
D0[31:16] = {8'h0, MEM[ADDR]}.
```

Untyped buffer load signed byte.

Flags: SEN_SCRATCH , OPF_D16, OPF_RDFLAT, OPF_RDM0

scratch_load_short_d16 *vgpr_dst*, *vgpr_addr/off*, *sgpr_saddr/off*
D0: vgpr S0: vgpr S1: sreg

```
D0[15:0] = MEM[ADDR].
```

Untyped buffer load short.

Flags: SEN_SCRATCH , OPF_D16, OPF_RDFLAT, OPF_RDM0

scratch_load_short_d16_hi *vgpr_dst*, *vgpr_addr/off*, *sgpr_saddr/off*
D0: vgpr S0: vgpr S1: sreg

```
D0[31:16] = MEM[ADDR].
```

Untyped buffer load short.

Flags: SEN_SCRATCH , OPF_D16, OPF_RDFLAT, OPF_RDM0

scratch_load_sshort *vgpr_dst*, *vgpr_addr/off*, *sgpr_saddr/off*
D0: vgpr S0: vgpr S1: sreg

Untyped buffer load signed short (sign extend to VGPR destination).

Flags: SEN_SCRATCH , OPF_RDFLAT, OPF_RDM0

scratch_load_ubyte *vgpr_dst*, *vgpr_addr/off*, *sgpr_saddr/off*
D0: vgpr S0: vgpr S1: sreg

Untyped buffer load unsigned byte (zero extend to VGPR destination).

Flags: SEN_SCRATCH , OPF_RDFLAT, OPF_RDM0

scratch_load_ubyte_d16 *vgpr_dst*, *vgpr_addr/off*, *sgpr_saddr/off*
D0: vgpr S0: vgpr S1: sreg

```
D0[15:0] = {8'h0, MEM[ADDR]}.
```

Untyped buffer load unsigned byte.

Flags: SEN_SCRATCH , OPF_D16, OPF_RDFLAT, OPF_RDM0

scratch_load_ubyte_d16_hi *vgpr_dst*, *vgpr_addr/off*, *sgpr_saddr/off*
D0: vgpr S0: vgpr S1: sreg

```
D0[31:16] = {8'h0, MEM[ADDR]}.
```

Untyped buffer load unsigned byte.

Flags: SEN_SCRATCH , OPF_D16, OPF_RDFLAT, OPF_RDM0

scratch_load_ushort *vgpr_dst*, *vgpr_addr/off*, *sgpr_saddr/off*
D0: vgpr S0: vgpr S1: sreg

Untyped buffer load unsigned short (zero extend to VGPR destination).

Flags: SEN_SCRATCH , OPF_RDFLAT, OPF_RDM0

scratch_store_byte *vgpr_addr/off*, *vgpr_src*, *sgpr_saddr/off*
S0: vgpr S1: vgpr S2: sreg

Untyped buffer store byte. Stores S0[7:0].

Flags: SEN_SCRATCH , OPF_MEM_STORE, OPF_RDFLAT, OPF_RDM0

scratch_store_byte_d16_hi *vgpr_addr/off*, *vgpr_src*, *sgpr_saddr/off*
S0: vgpr S1: vgpr S2: sreg

Untyped buffer store byte. Stores S0[23:16].

Flags: SEN_SCRATCH , OPF_D16, OPF_MEM_STORE, OPF_RDFLAT, OPF_RDM0

scratch_store_dword *vgpr_addr/off*, *vgpr_src*, *sgpr_saddr/off*
S0: vgpr S1: vgpr S2: sreg

Untyped buffer store dword.

Flags: SEN_SCRATCH , OPF_MEM_STORE, OPF_RDFLAT, OPF_RDM0

scratch_store_dwordx2 *vgpr_addr/off*, *vgpr_src*[2], *sgpr_saddr/off*
S0: vgpr S1: vgpr S2: sreg

Untyped buffer store 2 dwords.

Flags: SEN_SCRATCH , OPF_MEM_STORE, OPF_RDFLAT, OPF_RDM0

scratch_store_dwordx3 *vgpr_addr/off*, *vgpr_src*[3], *sgpr_saddr/off*
S0: vgpr S1: vgpr S2: sreg

Untyped buffer store 3 dwords.

Flags: SEN_SCRATCH , OPF_MEM_STORE, OPF_RDFLAT, OPF_RDM0

scratch_store_dwordx4 *vgpr_addr/off*, *vgpr_src*[4], *sgpr_saddr/off*
S0: vgpr S1: vgpr S2: sreg

Untyped buffer store 4 dwords.

Flags: SEN_SCRATCH , OPF_MEM_STORE, OPF_RDFLAT, OPF_RDM0

scratch_store_short *vgpr_addr/off*, *vgpr_src*, *sgpr_saddr/off*
S0: vgpr S1: vgpr S2: sreg

Untyped buffer store short. Stores S0[15:0].

Flags: SEN_SCRATCH , OPF_MEM_STORE, OPF_RDFLAT, OPF_RDM0

scratch_store_short_d16_hi *vgpr_addr/off*, *vgpr_src*, *sgpr_saddr/off*
S0: vgpr S1: vgpr S2: sreg

Untyped buffer store short. Stores S0[31:16].

Flags: SEN_SCRATCH , OPF_D16, OPF_MEM_STORE, OPF_RDFLAT, OPF_RDM0

19.1 Notes for Encoding FLAT

19.1.1 Operands

vgpr_dst is a vector GPR to store results in. In atomic instruction descriptions it is shown as RETURN_DATA or RETURN_DATA[*n*] when writing the *n*'th VGPR. Atomic instructions will only write back data to the VGPRs if the **glc** bit is set.

vgpr_addr is a vector GPR containing address information. In atomic instruction descriptions it is shown as ADDR.

vgpr_src is a vector GPR to read data from. In atomic instruction descriptions it is shown as DATA or DATA[*n*] when reading the *n*'th VGPR.

sgpr_saddr is a scalar GPR to read from containing address/offset. To disable the instructions from reading this assign it to "off".

19.1.2 Modifiers

`glc:{0,1}`
: If true, operation is globally coherent. Default 0. Can also write `noglc`.

`slc:{0,1}`
: If true, operation is system coherent. Default 0. Can also write `noslc`.

`inst_offset:[0. . . 4095]`
: 12-bit instruction offset. Default 0.

`lds:{0,1}`
: If true, read data from LDS (stores) or write data to LDS (loads). Default 0.

`nv:{0,1}`
: If true, non-volatile memory access. Default 0.

A General Types

The following types appear as operands in this instruction definition. Each type may optionally be suffixed by *_N* to indicate the operand position and/or *[N]* to indicate an operand that takes multiple consecutive DWORDs.

attr ::=

`attr0. . . attr32`	First interpolation attribute. Increment from here for additional attributes. There are SQ_NUM_ATTR attributes in total.

flat_scratch_lohi ::=

`flat_scratch_lo`	{13'd0, size[18:0]}
`flat_scratch_hi`	{8'd0, offset[31:8]}

label ::=

program_label	a program label to jump to

param ::=

`p10`	
`p20`	
`p0`	

sdst ::=

`s0. . . s101`	Scalar GPR 0. Increment from here for additional GPRs. There are NUM_SGPR SGPRs in total.
`xnack_mask_lo`	xnack_mask[31:0], see XNACK replay mechanism for details.
`xnack_mask_hi`	xnack_mask[63:32], see XNACK replay mechanism for details.
flat_scratch_lohi	
`ttmp0`	Trap handler temps (privileged). Increment from here for additional TTMPs. There are NUM_TTMP TTMPs in total. {TTMP1,TTMP0} = PC_save{hi,lo}.
`ttmp1`	Trap handler temps (privileged).
`ttmp2`	Trap handler temps (privileged).
`ttmp3`	Trap handler temps (privileged).
`ttmp4`	Trap handler temps (privileged).
`ttmp5`	Trap handler temps (privileged).
`ttmp6`	Trap handler temps (privileged).
`ttmp7`	Trap handler temps (privileged).
`ttmp8`	Trap handler temps (privileged).
`ttmp9`	Trap handler temps (privileged).
`ttmp10`	Trap handler temps (privileged).
`ttmp11`	Trap handler temps (privileged).
`ttmp12`	Trap handler temps (privileged).
`ttmp13`	Trap handler temps (privileged).
`ttmp14`	Trap handler temps (privileged).
`ttmp15`	Trap handler temps (privileged).
`vcc_lo`	vcc[31:0]
`vcc_hi`	vcc[63:32]
`m0`	Special register used to hold LDS/GDS addresses, relative indices, and send-messsage values.
`exec_lo`	exec[31:0]
`exec_hi`	exec[63:32]

sdst_exec ::=

`exec_lo`	exec[31:0]
`exec_hi`	exec[63:32]

sdst_m0 ::=

`m0`	Special register used to hold LDS/GDS addresses, relative indices, and send-messsage values.

sgpr ::=

`s0. . . s101`	Scalar GPR 0. Increment from here for additional GPRs. There are NUM_SGPR SGPRs in total.

simm16 ::=

number	a 16-bit integer constant

simm32 ::=

number	a 32-bit integer constant

simm4 ::=

simm8 ::=

smem_offset ::=

sgpr	
`xnack_mask_lo`	xnack_mask[31:0], see XNACK replay mechanism for details.
`xnack_mask_hi`	xnack_mask[63:32], see XNACK replay mechanism for details.
flat_scratch_lohi	
`ttmp0`	Trap handler temps (privileged). Increment from here for additional TTMPs. There are NUM_TTMP TTMPs in total. {TTMP1,TTMP0} = PC_save{hi,lo}.
`ttmp1`	Trap handler temps (privileged).
`ttmp2`	Trap handler temps (privileged).
`ttmp3`	Trap handler temps (privileged).
`ttmp4`	Trap handler temps (privileged).
`ttmp5`	Trap handler temps (privileged).
`ttmp6`	Trap handler temps (privileged).
`ttmp7`	Trap handler temps (privileged).
`ttmp8`	Trap handler temps (privileged).
`ttmp9`	Trap handler temps (privileged).
`ttmp10`	Trap handler temps (privileged).
`ttmp11`	Trap handler temps (privileged).
`ttmp12`	Trap handler temps (privileged).
`ttmp13`	Trap handler temps (privileged).
`ttmp14`	Trap handler temps (privileged).
`ttmp15`	Trap handler temps (privileged).
`vcc_lo`	vcc[31:0]
`vcc_hi`	vcc[63:32]
sdst_m0	
number	a 20-bit integer constant

src : :=

sdst	
0	0
1	1 (integer)
2	2 (integer)
3	3 (integer)
4	4 (integer)
5	5 (integer)
6	6 (integer)
7	7 (integer)
8	8 (integer)
9	9 (integer)
10	10 (integer)
11	11 (integer)
12	12 (integer)
13	13 (integer)
14	14 (integer)
15	15 (integer)
16	16 (integer)
17	17 (integer)
18	18 (integer)
19	19 (integer)
20	20 (integer)
21	21 (integer)
22	22 (integer)
23	23 (integer)
24	24 (integer)
25	25 (integer)
26	26 (integer)
27	27 (integer)
28	28 (integer)
29	29 (integer)
30	30 (integer)
31	31 (integer)
32	32 (integer)
33	33 (integer)
34	34 (integer)
35	35 (integer)
36	36 (integer)
37	37 (integer)

src ::= (continued)

38	38 (integer)
39	39 (integer)
40	40 (integer)
41	41 (integer)
42	42 (integer)
43	43 (integer)
44	44 (integer)
45	45 (integer)
46	46 (integer)
47	47 (integer)
48	48 (integer)
49	49 (integer)
50	50 (integer)
51	51 (integer)
52	52 (integer)
53	53 (integer)
54	54 (integer)
55	55 (integer)
56	56 (integer)
57	57 (integer)
58	58 (integer)
59	59 (integer)
60	60 (integer)
61	61 (integer)
62	62 (integer)
63	63 (integer)
64	64 (integer)
-1	-1 (integer)
-2	-2 (integer)
-3	-3 (integer)
-4	-4 (integer)
-5	-5 (integer)
-6	-6 (integer)
-7	-7 (integer)
-8	-8 (integer)
-9	-9 (integer)
-10	-10 (integer)
-11	-11 (integer)
-12	-12 (integer)
-13	-13 (integer)
-14	-14 (integer)
-15	-15 (integer)
-16	-16 (integer)
0.5	0.5
-0.5	-0.5
1.0	1.0
-1.0	-1.0
2.0	2.0

src ::= (continued)

-2.0	-2.0
4.0	4.0
-4.0	-4.0
0.15915494	1 / (2 * PI). 16-bit: 0x3118, 32-bit: 0x3e22f983, 64-bit: 0x3fc45f30_6dc9c882.
src_vccz	True if vector condition code is zero.
src_execz	True if execute mask is zero.
src_scc	Scalar condition code.
src_shared_base	
src_shared_limit	
src_private_base	
src_private_limit	
src_pops_exiting_wave_id	POPS counter (12b). Returns the ID of the most recent wave to complete in-order.
v0. . . v255	Vector GPR 0. Increment from here for additional GPRs. There are NUM_VGPR VGPRs in total. You may use the constant SQ_SRC_VGPR_BIT to set or clear the high order bit for vector GPRs in this operand.
src_lds_direct	Use LDS direct to supply a single 32-bit value to all lanes. A single DWORD is read from LDS memory at ADDR[M0[15:0]], where M0[15:0] is a byte address and is dword-aligned. M0[18:16] specify the data type for the read and may be 0=UBYTE, 1=USHORT, 2=DWORD, 4=SBYTE, 5=SSHORT. Implies OPF_RDM0 when this operand is used.
src_literal	32-bit literal constant follows this instruction. 16-bit operands use only the 16 LSBs of this constant. 64-bit unsigned integer operands zero-extend the constant. 64-bit signed integer operands sign-extend the constant. 64-bit floating-point operands use the constant as the high 32 MSBs of a double-precision floating point value (1 sign bit, 11 exponent bits and 20 mantissa bits).

src_nolds ::=

sdst	
0	0
1	1 (integer)
2	2 (integer)
3	3 (integer)
4	4 (integer)
5	5 (integer)

src_nolds ::= (continued)

6	6 (integer)
7	7 (integer)
8	8 (integer)
9	9 (integer)
10	10 (integer)
11	11 (integer)
12	12 (integer)
13	13 (integer)
14	14 (integer)
15	15 (integer)
16	16 (integer)
17	17 (integer)
18	18 (integer)
19	19 (integer)
20	20 (integer)
21	21 (integer)
22	22 (integer)
23	23 (integer)
24	24 (integer)
25	25 (integer)
26	26 (integer)
27	27 (integer)
28	28 (integer)
29	29 (integer)
30	30 (integer)
31	31 (integer)
32	32 (integer)
33	33 (integer)
34	34 (integer)
35	35 (integer)
36	36 (integer)
37	37 (integer)
38	38 (integer)
39	39 (integer)
40	40 (integer)
41	41 (integer)
42	42 (integer)
43	43 (integer)
44	44 (integer)
45	45 (integer)
46	46 (integer)
47	47 (integer)
48	48 (integer)
49	49 (integer)
50	50 (integer)
51	51 (integer)
52	52 (integer)
53	53 (integer)

src_nolds ::= (continued)

54	54 (integer)
55	55 (integer)
56	56 (integer)
57	57 (integer)
58	58 (integer)
59	59 (integer)
60	60 (integer)
61	61 (integer)
62	62 (integer)
63	63 (integer)
64	64 (integer)
-1	-1 (integer)
-2	-2 (integer)
-3	-3 (integer)
-4	-4 (integer)
-5	-5 (integer)
-6	-6 (integer)
-7	-7 (integer)
-8	-8 (integer)
-9	-9 (integer)
-10	-10 (integer)
-11	-11 (integer)
-12	-12 (integer)
-13	-13 (integer)
-14	-14 (integer)
-15	-15 (integer)
-16	-16 (integer)
0.5	0.5
-0.5	-0.5
1.0	1.0
-1.0	-1.0
2.0	2.0

src_nolds ::= (continued)

-2.0	-2.0
4.0	4.0
-4.0	-4.0
0.15915494	1 / (2 * PI). 16-bit: 0x3118, 32-bit: 0x3e22f983, 64-bit: 0x3fc45f30_6dc9c882.
src_vccz	True if vector condition code is zero.
src_execz	True if execute mask is zero.
src_scc	Scalar condition code.
src_shared_base	
src_shared_limit	
src_private_base	
src_private_limit	
src_pops_exiting_wave_id	POPS counter (12b). Returns the ID of the most recent wave to complete in-order.
v0. . . v255	Vector GPR 0. Increment from here for additional GPRs. There are NUM_VGPR VGPRs in total. You may use the constant SQ_SRC_VGPR_BIT to set or clear the high order bit for vector GPRs in this operand.
src_literal	32-bit literal constant follows this instruction. 16-bit operands use only the 16 LSBs of this constant. 64-bit unsigned integer operands zero-extend the constant. 64-bit signed integer operands sign-extend the constant. 64-bit floating-point operands use the constant as the high 32 MSBs of a double-precision floating point value (1 sign bit, 11 exponent bits and 20 mantissa bits).

src_nolit ::=

sdst	
0	0
1	1 (integer)
2	2 (integer)
3	3 (integer)
4	4 (integer)
5	5 (integer)
6	6 (integer)
7	7 (integer)
8	8 (integer)
9	9 (integer)
10	10 (integer)
11	11 (integer)
12	12 (integer)
13	13 (integer)
14	14 (integer)
15	15 (integer)
16	16 (integer)
17	17 (integer)
18	18 (integer)
19	19 (integer)
20	20 (integer)
21	21 (integer)

src_nolit ::= (continued)

22	22 (integer)
23	23 (integer)
24	24 (integer)
25	25 (integer)
26	26 (integer)
27	27 (integer)
28	28 (integer)
29	29 (integer)
30	30 (integer)
31	31 (integer)
32	32 (integer)
33	33 (integer)
34	34 (integer)
35	35 (integer)
36	36 (integer)
37	37 (integer)
38	38 (integer)
39	39 (integer)
40	40 (integer)
41	41 (integer)
42	42 (integer)
43	43 (integer)
44	44 (integer)
45	45 (integer)
46	46 (integer)
47	47 (integer)
48	48 (integer)
49	49 (integer)
50	50 (integer)
51	51 (integer)
52	52 (integer)
53	53 (integer)
54	54 (integer)
55	55 (integer)
56	56 (integer)
57	57 (integer)
58	58 (integer)
59	59 (integer)
60	60 (integer)
61	61 (integer)
62	62 (integer)
63	63 (integer)
64	64 (integer)
-1	-1 (integer)
-2	-2 (integer)
-3	-3 (integer)
-4	-4 (integer)
-5	-5 (integer)

src_nolit ::= (continued)

-6	-6 (integer)
-7	-7 (integer)
-8	-8 (integer)
-9	-9 (integer)
-10	-10 (integer)
-11	-11 (integer)
-12	-12 (integer)
-13	-13 (integer)
-14	-14 (integer)
-15	-15 (integer)
-16	-16 (integer)
0.5	0.5
-0.5	-0.5
1.0	1.0
-1.0	-1.0
2.0	2.0
-2.0	-2.0
4.0	4.0
-4.0	-4.0
0.15915494	1 / (2 * PI). 16-bit: 0x3118, 32-bit: 0x3e22f983, 64-bit: 0x3fc45f30_6dc9c882.
src_vccz	True if vector condition code is zero.
src_execz	True if execute mask is zero.
src_scc	Scalar condition code.
src_shared_base	
src_shared_limit	
src_private_base	
src_private_limit	
src_pops_exiting_wave_id	POPS counter (12b). Returns the ID of the most recent wave to complete in-order.
v0. . . v255	Vector GPR 0. Increment from here for additional GPRs. There are NUM_VGPR VGPRs in total. You may use the constant SQ_SRC_VGPR_BIT to set or clear the high order bit for vector GPRs in this operand.
src_lds_direct	Use LDS direct to supply a single 32-bit value to all lanes. A single DWORD is read from LDS memory at ADDR[M0[15:0]], where M0[15:0] is a byte address and is dword-aligned. M0[18:16] specify the data type for the read and may be 0=UBYTE, 1=USHORT, 2=DWORD, 4=SBYTE, 5=SSHORT. Implies OPF_RDM0 when this operand is used.

src_simple ::=

sdst	
0	0
1	1 (integer)
2	2 (integer)
3	3 (integer)
4	4 (integer)
5	5 (integer)

src_simple ::= (continued)

6	6 (integer)
7	7 (integer)
8	8 (integer)
9	9 (integer)
10	10 (integer)
11	11 (integer)
12	12 (integer)
13	13 (integer)
14	14 (integer)
15	15 (integer)
16	16 (integer)
17	17 (integer)
18	18 (integer)
19	19 (integer)
20	20 (integer)
21	21 (integer)
22	22 (integer)
23	23 (integer)
24	24 (integer)
25	25 (integer)
26	26 (integer)
27	27 (integer)
28	28 (integer)
29	29 (integer)
30	30 (integer)
31	31 (integer)
32	32 (integer)
33	33 (integer)
34	34 (integer)
35	35 (integer)
36	36 (integer)
37	37 (integer)
38	38 (integer)
39	39 (integer)
40	40 (integer)
41	41 (integer)
42	42 (integer)
43	43 (integer)
44	44 (integer)
45	45 (integer)
46	46 (integer)
47	47 (integer)
48	48 (integer)
49	49 (integer)
50	50 (integer)
51	51 (integer)
52	52 (integer)
53	53 (integer)

src_simple ::= (continued)

54	54 (integer)
55	55 (integer)
56	56 (integer)
57	57 (integer)
58	58 (integer)
59	59 (integer)
60	60 (integer)
61	61 (integer)
62	62 (integer)
63	63 (integer)
64	64 (integer)
-1	-1 (integer)
-2	-2 (integer)
-3	-3 (integer)
-4	-4 (integer)
-5	-5 (integer)
-6	-6 (integer)
-7	-7 (integer)
-8	-8 (integer)
-9	-9 (integer)
-10	-10 (integer)
-11	-11 (integer)
-12	-12 (integer)
-13	-13 (integer)
-14	-14 (integer)
-15	-15 (integer)
-16	-16 (integer)
0.5	0.5
-0.5	-0.5
1.0	1.0
-1.0	-1.0
2.0	2.0
-2.0	-2.0
4.0	4.0
-4.0	-4.0
0.15915494	1 / (2 * PI). 16-bit: 0x3118, 32-bit: 0x3e22f983, 64-bit: 0x3fc45f30_6dc9c882.
src_vccz	True if vector condition code is zero.
src_execz	True if execute mask is zero.
src_scc	Scalar condition code.
src_shared_base	
src_shared_limit	
src_private_base	
src_private_limit	
src_pops_exiting_wave_id	POPS counter (12b). Returns the ID of the most recent wave to complete in-order.
v0. . . v255	Vector GPR 0. Increment from here for additional GPRs. There are NUM_VGPR VGPRs in total. You may use the constant SQ_SRC_VGPR_BIT to set or clear the high order bit for vector GPRs in this operand.

src_vgpr ::=

v0. . . v255	Vector GPR 0. Increment from here for additional GPRs. There are NUM_VGPR VGPRs in total. You may use the constant SQ_SRC_VGPR_BIT to set or clear the high order bit for vector GPRs in this operand.

sreg ::=

sgpr	
xnack_mask_lo	xnack_mask[31:0], see XNACK replay mechanism for details.
xnack_mask_hi	xnack_mask[63:32], see XNACK replay mechanism for details.
flat_scratch_lohi	
ttmp0	Trap handler temps (privileged). Increment from here for additional TTMPs. There are NUM_TTMP TTMPs in total. {TTMP1,TTMP0} = PC_save{hi,lo}.
ttmp1	Trap handler temps (privileged).
ttmp2	Trap handler temps (privileged).
ttmp3	Trap handler temps (privileged).
ttmp4	Trap handler temps (privileged).
ttmp5	Trap handler temps (privileged).
ttmp6	Trap handler temps (privileged).
ttmp7	Trap handler temps (privileged).
ttmp8	Trap handler temps (privileged).
ttmp9	Trap handler temps (privileged).
ttmp10	Trap handler temps (privileged).
ttmp11	Trap handler temps (privileged).
ttmp12	Trap handler temps (privileged).
ttmp13	Trap handler temps (privileged).
ttmp14	Trap handler temps (privileged).
ttmp15	Trap handler temps (privileged).
vcc_lo	vcc[31:0]
vcc_hi	vcc[63:32]

sreg_novcc ::=

sgpr	
xnack_mask_lo	xnack_mask[31:0], see XNACK replay mechanism for details.
xnack_mask_hi	xnack_mask[63:32], see XNACK replay mechanism for details.
flat_scratch_lohi	
ttmp0	Trap handler temps (privileged). Increment from here for additional TTMPs. There are NUM_TTMP TTMPs in total. {TTMP1,TTMP0} = PC_save{hi,lo}.
ttmp1	Trap handler temps (privileged).
ttmp2	Trap handler temps (privileged).
ttmp3	Trap handler temps (privileged).
ttmp4	Trap handler temps (privileged).
ttmp5	Trap handler temps (privileged).
ttmp6	Trap handler temps (privileged).
ttmp7	Trap handler temps (privileged).
ttmp8	Trap handler temps (privileged).
ttmp9	Trap handler temps (privileged).
ttmp10	Trap handler temps (privileged).

sreg_novcc ::= (continued)

ttmp11	Trap handler temps (privileged).
ttmp12	Trap handler temps (privileged).
ttmp13	Trap handler temps (privileged).
ttmp14	Trap handler temps (privileged).
ttmp15	Trap handler temps (privileged).

ssrc ::=

sdst	
0	0
1	1 (integer)
2	2 (integer)
3	3 (integer)
4	4 (integer)
5	5 (integer)
6	6 (integer)
7	7 (integer)
8	8 (integer)
9	9 (integer)
10	10 (integer)
11	11 (integer)
12	12 (integer)
13	13 (integer)
14	14 (integer)
15	15 (integer)
16	16 (integer)
17	17 (integer)
18	18 (integer)
19	19 (integer)
20	20 (integer)
21	21 (integer)
22	22 (integer)
23	23 (integer)
24	24 (integer)
25	25 (integer)
26	26 (integer)
27	27 (integer)
28	28 (integer)
29	29 (integer)
30	30 (integer)
31	31 (integer)
32	32 (integer)
33	33 (integer)
34	34 (integer)
35	35 (integer)
36	36 (integer)
37	37 (integer)

ssrc : := (continued)

38	38 (integer)
39	39 (integer)
40	40 (integer)
41	41 (integer)
42	42 (integer)
43	43 (integer)
44	44 (integer)
45	45 (integer)
46	46 (integer)
47	47 (integer)
48	48 (integer)
49	49 (integer)
50	50 (integer)
51	51 (integer)
52	52 (integer)
53	53 (integer)
54	54 (integer)
55	55 (integer)
56	56 (integer)
57	57 (integer)
58	58 (integer)
59	59 (integer)
60	60 (integer)
61	61 (integer)
62	62 (integer)
63	63 (integer)
64	64 (integer)
-1	-1 (integer)
-2	-2 (integer)
-3	-3 (integer)
-4	-4 (integer)
-5	-5 (integer)
-6	-6 (integer)
-7	-7 (integer)
-8	-8 (integer)
-9	-9 (integer)
-10	-10 (integer)
-11	-11 (integer)
-12	-12 (integer)
-13	-13 (integer)
-14	-14 (integer)
-15	-15 (integer)
-16	-16 (integer)
0.5	0.5
-0.5	-0.5
1.0	1.0
-1.0	-1.0
2.0	2.0

ssrc ::= (continued)

-2.0	-2.0
4.0	4.0
-4.0	-4.0
0.15915494	1 / (2 * PI). 16-bit: 0x3118, 32-bit: 0x3e22f983, 64-bit: 0x3fc45f30_6dc9c882.
src_vccz	True if vector condition code is zero.
src_execz	True if execute mask is zero.
src_scc	Scalar condition code.
src_shared_base	
src_shared_limit	
src_private_base	
src_private_limit	
src_pops_exiting_wave_id	POPS counter (12b). Returns the ID of the most recent wave to complete in-order.
src_literal	32-bit literal constant follows this instruction. 16-bit operands use only the 16 LSBs of this constant. 64-bit unsigned integer operands zero-extend the constant. 64-bit signed integer operands sign-extend the constant. 64-bit floating-point operands use the constant as the high 32 MSBs of a double-precision floating point value (1 sign bit, 11 exponent bits and 20 mantissa bits).

ssrc_0_63_inlines ::=

0	0
1	1 (integer)
2	2 (integer)
3	3 (integer)
4	4 (integer)
5	5 (integer)
6	6 (integer)
7	7 (integer)
8	8 (integer)
9	9 (integer)
10	10 (integer)
11	11 (integer)
12	12 (integer)
13	13 (integer)
14	14 (integer)
15	15 (integer)

ssrc_0_63_inlines ::= (continued)

16	16 (integer)
17	17 (integer)
18	18 (integer)
19	19 (integer)
20	20 (integer)
21	21 (integer)
22	22 (integer)
23	23 (integer)
24	24 (integer)
25	25 (integer)
26	26 (integer)
27	27 (integer)
28	28 (integer)
29	29 (integer)
30	30 (integer)
31	31 (integer)
32	32 (integer)
33	33 (integer)
34	34 (integer)
35	35 (integer)
36	36 (integer)
37	37 (integer)
38	38 (integer)
39	39 (integer)
40	40 (integer)
41	41 (integer)
42	42 (integer)
43	43 (integer)
44	44 (integer)
45	45 (integer)
46	46 (integer)
47	47 (integer)
48	48 (integer)
49	49 (integer)
50	50 (integer)
51	51 (integer)
52	52 (integer)
53	53 (integer)
54	54 (integer)
55	55 (integer)
56	56 (integer)
57	57 (integer)
58	58 (integer)
59	59 (integer)
60	60 (integer)
61	61 (integer)
62	62 (integer)
63	63 (integer)

ssrc_lanesel ::=

smem_offset
ssrc_0_63_inlines

ssrc_nolit ::=

sdst	
ssrc_0_63_inlines	
64	64 (integer)
-1	-1 (integer)
-2	-2 (integer)
-3	-3 (integer)
-4	-4 (integer)
-5	-5 (integer)
-6	-6 (integer)
-7	-7 (integer)
-8	-8 (integer)
-9	-9 (integer)
-10	-10 (integer)
-11	-11 (integer)
-12	-12 (integer)
-13	-13 (integer)
-14	-14 (integer)
-15	-15 (integer)
-16	-16 (integer)
0.5	0.5
-0.5	-0.5
1.0	1.0
-1.0	-1.0
2.0	2.0
-2.0	-2.0
4.0	4.0
-4.0	-4.0
0.15915494	1 / (2 * PI). 16-bit: 0x3118, 32-bit: 0x3e22f983, 64-bit: 0x3fc45f30_6dc9c882.
src_vccz	True if vector condition code is zero.
src_execz	True if execute mask is zero.
src_scc	Scalar condition code.
src_shared_base	
src_shared_limit	
src_private_base	
src_private_limit	
src_pops_exiting_wave_id	POPS counter (12b). Returns the ID of the most recent wave to complete in-order.

ssrc_special_aperture ::=

src_shared_base	
src_shared_limit	
src_private_base	
src_private_limit	

ssrc_special_execz ::=

src_execz	True if execute mask is zero.

ssrc_special_lds ::=

`src_lds_direct`	Use LDS direct to supply a single 32-bit value to all lanes. A single DWORD is read from LDS memory at ADDR[M0[15:0]], where M0[15:0] is a byte address and is dword-aligned. M0[18:16] specify the data type for the read and may be 0=UBYTE, 1=USHORT, 2=DWORD, 4=SBYTE, 5=SSHORT. Implies OPF_RDM0 when this operand is used.

ssrc_special_lit ::=

`src_literal`	32-bit literal constant follows this instruction. 16-bit operands use only the 16 LSBs of this constant. 64-bit unsigned integer operands zero-extend the constant. 64-bit signed integer operands sign-extend the constant. 64-bit floating-point operands use the constant as the high 32 MSBs of a double-precision floating point value (1 sign bit, 11 exponent bits and 20 mantissa bits).

ssrc_special_nolit ::=

ssrc_0_63_inlines	
`64`	64 (integer)
`-1`	-1 (integer)
`-2`	-2 (integer)
`-3`	-3 (integer)
`-4`	-4 (integer)
`-5`	-5 (integer)
`-6`	-6 (integer)
`-7`	-7 (integer)
`-8`	-8 (integer)
`-9`	-9 (integer)
`-10`	-10 (integer)
`-11`	-11 (integer)
`-12`	-12 (integer)
`-13`	-13 (integer)
`-14`	-14 (integer)
`-15`	-15 (integer)
`-16`	-16 (integer)
`0.5`	0.5
`-0.5`	-0.5
`1.0`	1.0
`-1.0`	-1.0
`2.0`	2.0
`-2.0`	-2.0
`4.0`	4.0
`-4.0`	-4.0
`0.15915494`	1 / (2 * PI). 16-bit: `0x3118`, 32-bit: `0x3e22f983`, 64-bit: `0x3fc45f30_6dc9c882`.
`src_vccz`	True if vector condition code is zero.
ssrc_special_execz	
`src_scc`	Scalar condition code.
ssrc_special_aperture	
`src_pops_exiting_wave_id`	POPS counter (12b). Returns the ID of the most recent wave to complete in-order.

ssrc_special_pops_exiting_wave_id ::=

`src_pops_exiting_wave_id`	POPS counter (12b). Returns the ID of the most recent wave to complete in-order.

ssrc_special_scc ::=

`src_scc`	Scalar condition code.

ssrc_special_vccz ::=

`src_vccz`	True if vector condition code is zero.

tgt ::=

`mrt0. . . mrt7`	Output to colour MRT 0. Increment from here for additional MRTs. There are EXP_NUM_MRT MRTs in total.
`mrtz`	Output to Z.
`null`	Output to NULL.
`pos0. . . pos3`	Output to position 0. Increment from here for additional positions. There are EXP_NUM_POS positions in total.
`param0. . . param31`	Output to parameter 0. Increment from here for additional parameters. There are EXP_NUM_PARAM parameters in total.

trap ::=

`ttmp0`	Trap handler temps (privileged). Increment from here for additional TTMPs. There are NUM_TTMP TTMPs in total. {TTMP1,TTMP0} = PC_save{hi,lo}.
`ttmp1`	Trap handler temps (privileged).
`ttmp2`	Trap handler temps (privileged).
`ttmp3`	Trap handler temps (privileged).
`ttmp4`	Trap handler temps (privileged).
`ttmp5`	Trap handler temps (privileged).
`ttmp6`	Trap handler temps (privileged).
`ttmp7`	Trap handler temps (privileged).
`ttmp8`	Trap handler temps (privileged).
`ttmp9`	Trap handler temps (privileged).
`ttmp10`	Trap handler temps (privileged).
`ttmp11`	Trap handler temps (privileged).
`ttmp12`	Trap handler temps (privileged).
`ttmp13`	Trap handler temps (privileged).
`ttmp14`	Trap handler temps (privileged).
`ttmp15`	Trap handler temps (privileged).

vcc ::=

`vcc`	vcc[63:0]

vcc_lohi ::=

`vcc_lo`	vcc[31:0]
`vcc_hi`	vcc[63:32]

vgpr ::=

src_vgpr

vgpr_or_lds ::=

ssrc_special_lds
src_vgpr

xnack_mask_lohi ::=

`xnack_mask_lo`	xnack_mask[31:0], see XNACK replay mechanism for details.
`xnack_mask_hi`	xnack_mask[63:32], see XNACK replay mechanism for details.

B Formats

The following data and number formats are recognized by sp3.

data_format ::=

```
BUF_DATA_FORMAT_INVALID
BUF_DATA_FORMAT_8
BUF_DATA_FORMAT_16
BUF_DATA_FORMAT_8_8
BUF_DATA_FORMAT_32
BUF_DATA_FORMAT_16_16
BUF_DATA_FORMAT_10_11_11
BUF_DATA_FORMAT_11_11_10
BUF_DATA_FORMAT_10_10_10_2
BUF_DATA_FORMAT_2_10_10_10
BUF_DATA_FORMAT_8_8_8_8
BUF_DATA_FORMAT_32_32
BUF_DATA_FORMAT_16_16_16_16
BUF_DATA_FORMAT_32_32_32
BUF_DATA_FORMAT_32_32_32_32
BUF_DATA_FORMAT_RESERVED_15
IMG_DATA_FORMAT_INVALID
IMG_DATA_FORMAT_8
IMG_DATA_FORMAT_16
IMG_DATA_FORMAT_8_8
IMG_DATA_FORMAT_32
IMG_DATA_FORMAT_16_16
IMG_DATA_FORMAT_10_11_11
IMG_DATA_FORMAT_11_11_10
IMG_DATA_FORMAT_10_10_10_2
IMG_DATA_FORMAT_2_10_10_10
IMG_DATA_FORMAT_8_8_8_8
IMG_DATA_FORMAT_32_32
IMG_DATA_FORMAT_16_16_16_16
IMG_DATA_FORMAT_32_32_32
IMG_DATA_FORMAT_32_32_32_32
IMG_DATA_FORMAT_RESERVED_15
IMG_DATA_FORMAT_5_6_5
IMG_DATA_FORMAT_1_5_5_5
IMG_DATA_FORMAT_5_5_5_1
IMG_DATA_FORMAT_4_4_4_4
IMG_DATA_FORMAT_8_24
IMG_DATA_FORMAT_24_8
IMG_DATA_FORMAT_X24_8_32
IMG_DATA_FORMAT_8_AS_8_8_8_8
IMG_DATA_FORMAT_ETC2_RGB
```

data_format ::= (continued)

IMG_DATA_FORMAT_ETC2_RGBA
IMG_DATA_FORMAT_ETC2_R
IMG_DATA_FORMAT_ETC2_RG
IMG_DATA_FORMAT_ETC2_RGBA1
IMG_DATA_FORMAT_RESERVED_29
IMG_DATA_FORMAT_RESERVED_30
IMG_DATA_FORMAT_6E4
IMG_DATA_FORMAT_GB_GR
IMG_DATA_FORMAT_BG_RG
IMG_DATA_FORMAT_5_9_9_9
IMG_DATA_FORMAT_BC1
IMG_DATA_FORMAT_BC2
IMG_DATA_FORMAT_BC3
IMG_DATA_FORMAT_BC4
IMG_DATA_FORMAT_BC5
IMG_DATA_FORMAT_BC6
IMG_DATA_FORMAT_BC7
IMG_DATA_FORMAT_16_AS_32_32
IMG_DATA_FORMAT_16_AS_16_16_16_16
IMG_DATA_FORMAT_16_AS_32_32_32_32
IMG_DATA_FORMAT_FMASK
IMG_DATA_FORMAT_ASTC_2D_LDR
IMG_DATA_FORMAT_ASTC_2D_HDR
IMG_DATA_FORMAT_ASTC_2D_LDR_SRGB
IMG_DATA_FORMAT_ASTC_3D_LDR
IMG_DATA_FORMAT_ASTC_3D_HDR
IMG_DATA_FORMAT_ASTC_3D_LDR_SRGB
IMG_DATA_FORMAT_N_IN_16
IMG_DATA_FORMAT_N_IN_16_16
IMG_DATA_FORMAT_N_IN_16_16_16_16
IMG_DATA_FORMAT_N_IN_16_AS_16_16_16_16
IMG_DATA_FORMAT_RESERVED_56
IMG_DATA_FORMAT_4_4
IMG_DATA_FORMAT_6_5_5
IMG_DATA_FORMAT_S8_16
IMG_DATA_FORMAT_S8_32
IMG_DATA_FORMAT_8_AS_32
IMG_DATA_FORMAT_8_AS_32_32
IMG_DATA_FORMAT_32_AS_32_32_32_32

num_format ::=

BUF_NUM_FORMAT_UNORM
BUF_NUM_FORMAT_SNORM
BUF_NUM_FORMAT_USCALED
BUF_NUM_FORMAT_SSCALED
BUF_NUM_FORMAT_UINT
BUF_NUM_FORMAT_SINT
BUF_NUM_FORMAT_RESERVED_6
BUF_NUM_FORMAT_FLOAT
IMG_NUM_FORMAT_UNORM

num_format ::= (continued)

```
IMG_NUM_FORMAT_SNORM
IMG_NUM_FORMAT_USCALED
IMG_NUM_FORMAT_SSCALED
IMG_NUM_FORMAT_UINT
IMG_NUM_FORMAT_SINT
IMG_NUM_FORMAT_RESERVED_6
IMG_NUM_FORMAT_FLOAT
IMG_NUM_FORMAT_RESERVED_8
IMG_NUM_FORMAT_SRGB
IMG_NUM_FORMAT_UNORM_UINT
IMG_NUM_FORMAT_RESERVED_11
IMG_NUM_FORMAT_RESERVED_12
IMG_NUM_FORMAT_RESERVED_13
IMG_NUM_FORMAT_RESERVED_14
IMG_NUM_FORMAT_RESERVED_15
```

C SDWA and DPP Constants

The following values for SDWA and DPP are recognized by sp3.

sdwa_unused ::=

SDWA_UNUSED_PAD	Pad all unused bits with 0.
SDWA_UNUSED_SEXT	Sign-extend upper bits; pad lower bits with 0.
SDWA_UNUSED_PRESERVE	Preserve existing unused bits; caused a read-modify-write on the destination.

sdwa_sel ::=

SDWA_BYTE_0	Select data[7:0]
SDWA_BYTE_1	Select data[15:8]
SDWA_BYTE_2	Select data[23:16]
SDWA_BYTE_3	Select data[31:24]
SDWA_WORD_0	Select data[15:0]
SDWA_WORD_1	Select data[31:16]
SDWA_DWORD	Select data[31:0]

dpp_bound_ctrl ::=

DPP_BOUND_OFF	Out-of-bounds data disables the write enable for this lane.
DPP_BOUND_ZERO	Out-of-bounds data reads as zero, writes zero.

D Operand Constants

The following constants are available for construction of special instruction operands.

These names for hardware registers may be used with the `hwreg()` built-in function.

hw_reg ::=

```
HW_REG_MODE
HW_REG_STATUS
HW_REG_TRAPSTS
HW_REG_HW_ID
HW_REG_GPR_ALLOC
HW_REG_LDS_ALLOC
HW_REG_IB_STS
HW_REG_PC_LO
HW_REG_PC_HI
HW_REG_INST_DW0
HW_REG_INST_DW1
HW_REG_IB_DBG0
HW_REG_IB_DBG1
HW_REG_FLUSH_IB
HW_REG_SH_MEM_BASES
HW_REG_SQ_SHADER_TBA_LO
HW_REG_SQ_SHADER_TBA_HI
HW_REG_SQ_SHADER_TMA_LO
HW_REG_SQ_SHADER_TMA_HI
```

These names for messages may be used with the `sendmsg()` built-in function.

msg ::=

```
MSG_INTERRUPT
MSG_GS
MSG_GS_DONE
MSG_SAVEWAVE
MSG_STALL_WAVE_GEN
MSG_HALT_WAVES
MSG_ORDERED_PS_DONE
MSG_EARLY_PRIM_DEALLOC
MSG_GS_ALLOC_REQ
MSG_GET_DOORBELL
MSG_SYSMSG
```

These names for GS messages may be used with the `sendmsg()` built-in function.

gs_op ::=

```
GS_OP_NOP
GS_OP_CUT
GS_OP_EMIT
GS_OP_EMIT_CUT
```

E Register Fields

The following register field offsets and sizes are recognized as shader constants by sp3.

`*_OFFSET` constants indicate bit offsets for register fields, in the range $[0, 31]$.

`*_SIZE` constants indicate bit sizes for register fields, in the range $[1, 32]$.

autoreg_fields ::=

`SQ_WAVE_STATUS_ALLOW_REPLAY_OFFSET`
`SQ_WAVE_STATUS_ALLOW_REPLAY_SIZE`
: When 0, the wave will never receive an XNACK from ATC or attempt to replay instructions

`SQ_WAVE_STATUS_COND_DBG_SYS_OFFSET`
`SQ_WAVE_STATUS_COND_DBG_SYS_SIZE`
: Conditional Debug bit for system.

`SQ_WAVE_STATUS_COND_DBG_USER_OFFSET`
`SQ_WAVE_STATUS_COND_DBG_USER_SIZE`
: Conditional Debug bit for user.

`SQ_WAVE_STATUS_ECC_ERR_OFFSET`
`SQ_WAVE_STATUS_ECC_ERR_SIZE`
: An ECC error has occurred for this wave.

`SQ_WAVE_STATUS_EXECZ_OFFSET`
`SQ_WAVE_STATUS_EXECZ_SIZE`
: Set if all EXEC mask bits are zero.

`SQ_WAVE_STATUS_EXPORT_RDY_OFFSET`
`SQ_WAVE_STATUS_EXPORT_RDY_SIZE`
: Color (PS) or Parameter/position (VS) space has been allocated for this wave. Exports stall until this is set.

`SQ_WAVE_STATUS_FATAL_HALT_OFFSET`
`SQ_WAVE_STATUS_FATAL_HALT_SIZE`
: Set if the wave is in a fatal halt state.

`SQ_WAVE_STATUS_HALT_OFFSET`
`SQ_WAVE_STATUS_HALT_SIZE`
: Set if the wave is in a halt state.

`SQ_WAVE_STATUS_IN_BARRIER_OFFSET`
`SQ_WAVE_STATUS_IN_BARRIER_SIZE`
: Set if the wave is currently waiting at a barrier

`SQ_WAVE_STATUS_IN_TG_OFFSET`
`SQ_WAVE_STATUS_IN_TG_SIZE`
: Set if wave is part of a threadgroup.

`SQ_WAVE_STATUS_MUST_EXPORT_OFFSET`
`SQ_WAVE_STATUS_MUST_EXPORT_SIZE`
: Must export before wave finished.

`SQ_WAVE_STATUS_PERF_EN_OFFSET`
`SQ_WAVE_STATUS_PERF_EN_SIZE`
: When set to 1, performance counters are incremented for this wave.

autoreg_fields ::= (continued)

SQ_WAVE_STATUS_PRIV_OFFSET SQ_WAVE_STATUS_PRIV_SIZE	Priveledged bit. Set when wave is in trap handler.
SQ_WAVE_STATUS_SCC_OFFSET SQ_WAVE_STATUS_SCC_SIZE	Scalar condition code
SQ_WAVE_STATUS_SKIP_EXPORT_OFFSET SQ_WAVE_STATUS_SKIP_EXPORT_SIZE	When set to 1, shader hardware skips all export instructions (treats them as NOPs) because this shader will never be allocated export buffer space. (used for old vs_no_alloc).
SQ_WAVE_STATUS_SPI_PRIO_OFFSET SQ_WAVE_STATUS_SPI_PRIO_SIZE	Wave priority value set by SPI. Higher value = higher priority.
SQ_WAVE_STATUS_TRAP_OFFSET SQ_WAVE_STATUS_TRAP_SIZE	Set if the wave needs to execute the trap handler.
SQ_WAVE_STATUS_TRAP_EN_OFFSET SQ_WAVE_STATUS_TRAP_EN_SIZE	Trap handler is enabled for this wave.
SQ_WAVE_STATUS_TTRACE_CU_EN_OFFSET SQ_WAVE_STATUS_TTRACE_CU_EN_SIZE	Set if ttrace details are enabled for this CU.
SQ_WAVE_STATUS_TTRACE_EN_OFFSET SQ_WAVE_STATUS_TTRACE_EN_SIZE	Thread trace is enabled for this wave.
SQ_WAVE_STATUS_USER_PRIO_OFFSET SQ_WAVE_STATUS_USER_PRIO_SIZE	Wave priority value set by shader. Higher value = higher priority. Default is 0.
SQ_WAVE_STATUS_VALID_OFFSET SQ_WAVE_STATUS_VALID_SIZE	Set if the wave is valid (live).
SQ_WAVE_STATUS_VCCZ_OFFSET SQ_WAVE_STATUS_VCCZ_SIZE	Set if all VCC (vector conditio code) bits are zero.
SQ_WAVE_IB_STS_EXP_CNT_OFFSET SQ_WAVE_IB_STS_EXP_CNT_SIZE	Number of export instructions in-flight
SQ_WAVE_IB_STS_FIRST_REPLAY_OFFSET SQ_WAVE_IB_STS_FIRST_REPLAY_SIZE	First flag for replay, allow write to this field only under privilage mode
SQ_WAVE_IB_STS_LGKM_CNT_OFFSET SQ_WAVE_IB_STS_LGKM_CNT_SIZE	Number of outstanding lds/gds/const-fetch/msg related instructions.
SQ_WAVE_IB_STS_RCNT_OFFSET SQ_WAVE_IB_STS_RCNT_SIZE	Replay rewind cnti, allow write to this field only under privileage mode
SQ_WAVE_IB_STS_VALU_CNT_OFFSET SQ_WAVE_IB_STS_VALU_CNT_SIZE	Number of vector alu instructions in-flight

autoreg_fields : := (continued)

SQ_WAVE_IB_STS_VM_CNT_OFFSET
SQ_WAVE_IB_STS_VM_CNT_SIZE
: Number of vector memory read/write instructions in-flight. Lower bits [3:0]

SQ_WAVE_IB_STS_VM_CNT_HI_OFFSET
SQ_WAVE_IB_STS_VM_CNT_HI_SIZE
: Number of vector memory read/write instructions in-flight. Upper bits [5:4]

SQ_WAVE_MODE_CSP_OFFSET
SQ_WAVE_MODE_CSP_SIZE
: Conditional-branch Stack Pointer

SQ_WAVE_MODE_DEBUG_EN_OFFSET
SQ_WAVE_MODE_DEBUG_EN_SIZE
SQ_WAVE_MODE_DISABLE_PERF_OFFSET
SQ_WAVE_MODE_DISABLE_PERF_SIZE
: Disable perfcounters

SQ_WAVE_MODE_DX10_CLAMP_OFFSET
SQ_WAVE_MODE_DX10_CLAMP_SIZE
SQ_WAVE_MODE_EXCP_EN_OFFSET
SQ_WAVE_MODE_EXCP_EN_SIZE
: Exception enables

SQ_WAVE_MODE_FP16_OVFL_OFFSET
SQ_WAVE_MODE_FP16_OVFL_SIZE
: If set, an overflowed FP16 result will be clamped to +/- MAX_FP16 regardless of round mode, while still preserving true INF values.

SQ_WAVE_MODE_FP_DENORM_OFFSET
SQ_WAVE_MODE_FP_DENORM_SIZE
SQ_WAVE_MODE_FP_ROUND_OFFSET
SQ_WAVE_MODE_FP_ROUND_SIZE
SQ_WAVE_MODE_GPR_IDX_EN_OFFSET
SQ_WAVE_MODE_GPR_IDX_EN_SIZE
: Enable GPR indexing for vector ALU operations

SQ_WAVE_MODE_IEEE_OFFSET
SQ_WAVE_MODE_IEEE_SIZE
SQ_WAVE_MODE_LOD_CLAMPED_OFFSET
SQ_WAVE_MODE_LOD_CLAMPED_SIZE
SQ_WAVE_MODE_POPS_PACKER0_OFFSET
SQ_WAVE_MODE_POPS_PACKER0_SIZE
: Wave is associated with packer 0 POPS counter

SQ_WAVE_MODE_POPS_PACKER1_OFFSET
SQ_WAVE_MODE_POPS_PACKER1_SIZE
: Wave is associated with packer 1 POPS counter

SQ_WAVE_MODE_VSKIP_OFFSET
SQ_WAVE_MODE_VSKIP_SIZE
: Skips any vector instructions

autoreg_fields : := (continued)

SQ_RANDOM_WAVE_PRI_RET_OFFSET
SQ_RANDOM_WAVE_PRI_RET_SIZE
: Random Wave Priority Enable Threshold. Disable rondom wave priority when value = 127.

SQ_RANDOM_WAVE_PRI_RNG_OFFSET
SQ_RANDOM_WAVE_PRI_RNG_SIZE
: Random Number Generator. 13 bits, can be set to a seed value. [5:0] as wave priority randomizer. [12:6] as the enable value to compare with the RET.

SQ_RANDOM_WAVE_PRI_RUI_OFFSET
SQ_RANDOM_WAVE_PRI_RUI_SIZE
: Random Number Generator Update Interval: The interval period = 4*2**(value).

SQ_BUF_RSRC_WORD0_BASE_ADDRESS_OFFSET
SQ_BUF_RSRC_WORD0_BASE_ADDRESS_SIZE
: Byte Base Address, bits 31-0

SQ_BUF_RSRC_WORD1_BASE_ADDRESS_HI_OFFSET
SQ_BUF_RSRC_WORD1_BASE_ADDRESS_HI_SIZE
: Byte Base Address, bits 47-32

SQ_BUF_RSRC_WORD1_CACHE_SWIZZLE_OFFSET
SQ_BUF_RSRC_WORD1_CACHE_SWIZZLE_SIZE
: buffer access. optionally swizzle TC L1 cache banks

SQ_BUF_RSRC_WORD1_STRIDE_OFFSET
SQ_BUF_RSRC_WORD1_STRIDE_SIZE
: Stride, in bytes. [0..16383]. See DATA_FORMAT for a way to extend the range of stride.

SQ_BUF_RSRC_WORD1_SWIZZLE_ENABLE_OFFSET
SQ_BUF_RSRC_WORD1_SWIZZLE_ENABLE_SIZE
: Cache Swizzle Array-Of-Structures according to stride, index_stride and element_size; else linear.

autoreg_fields : := (continued)

SQ_BUF_RSRC_WORD2_NUM_RECORDS_OFFSET
SQ_BUF_RSRC_WORD2_NUM_RECORDS_SIZE
: Number of records in buffer. For scalar memory instructions this is in byte units iff STRIDE == 0. For vector memory instructions this is in byte units iff inst.IDXEN == 1. In all other cases this is in units of STRIDE. (Within the set of legal combinations, this is equivalent to: bytes for raw buffers, stride-units for structured buffers, ignored for private/scratch).

SQ_BUF_RSRC_WORD3_ADD_TID_ENABLE_OFFSET
SQ_BUF_RSRC_WORD3_ADD_TID_ENABLE_SIZE
: Add thread ID (0..63) to the index for address calc. mainly for scratch buffer. Setting this to 1 also indicates that data_format will hold stride bits [17:14] used for scratch offset boundary checks instead of data_format

SQ_BUF_RSRC_WORD3_DATA_FORMAT_OFFSET
SQ_BUF_RSRC_WORD3_DATA_FORMAT_SIZE
: Data format (8, 16, 8_8, etc) in most circumstances. When ADD_TID_ENABLE == 1 and this resource is used in a scratch buffer operation (any MUBUF opcode except *_FORMAT_*), this field is instead used to specify an extended stride for scratch offset boundary checks – this holds bits [17:14] of the stride and extends the range of stride to [0, 262143].

SQ_BUF_RSRC_WORD3_DST_SEL_W_OFFSET
SQ_BUF_RSRC_WORD3_DST_SEL_W_SIZE
: Destination data swizzle - W: x,y,z,w,0,1

SQ_BUF_RSRC_WORD3_DST_SEL_X_OFFSET
SQ_BUF_RSRC_WORD3_DST_SEL_X_SIZE
: Destination data swizzle - X: x,y,z,w,0,1

SQ_BUF_RSRC_WORD3_DST_SEL_Y_OFFSET
SQ_BUF_RSRC_WORD3_DST_SEL_Y_SIZE
: Destination data swizzle - Y: x,y,z,w,0,1

SQ_BUF_RSRC_WORD3_DST_SEL_Z_OFFSET
SQ_BUF_RSRC_WORD3_DST_SEL_Z_SIZE
: Destination data swizzle - Z: x,y,z,w,0,1

SQ_BUF_RSRC_WORD3_INDEX_STRIDE_OFFSET
SQ_BUF_RSRC_WORD3_INDEX_STRIDE_SIZE
: Index Stride: 8,16,32 or 64. used for swizzled buffer addressing

SQ_BUF_RSRC_WORD3_NUM_FORMAT_OFFSET
SQ_BUF_RSRC_WORD3_NUM_FORMAT_SIZE
: Numeric format (unorm, snorm, float, etc)

SQ_BUF_RSRC_WORD3_NV_OFFSET
SQ_BUF_RSRC_WORD3_NV_SIZE
: non-volatile bit (1:non-volatile, 0:volatile)

SQ_BUF_RSRC_WORD3_TYPE_OFFSET
SQ_BUF_RSRC_WORD3_TYPE_SIZE
: Resource type: must be BUFFER

SQ_BUF_RSRC_WORD3_USER_VM_ENABLE_OFFSET
SQ_BUF_RSRC_WORD3_USER_VM_ENABLE_SIZE
: Resource is mapped via tiled pool / heap

autoreg_fields ::= (continued)

SQ_BUF_RSRC_WORD3_USER_VM_MODE_OFFSET
SQ_BUF_RSRC_WORD3_USER_VM_MODE_SIZE
: Unmapped behaviour. 0 = NULL (return 0 / drop write), 1 = INVALID (results in error and trap)

SQ_IMG_RSRC_WORD0_BASE_ADDRESS_OFFSET
SQ_IMG_RSRC_WORD0_BASE_ADDRESS_SIZE
: Image base byte adddress, bits 39-8 (bits 7-0 are zero)

SQ_IMG_RSRC_WORD1_BASE_ADDRESS_HI_OFFSET
SQ_IMG_RSRC_WORD1_BASE_ADDRESS_HI_SIZE
: Image base address, bits 47-40

SQ_IMG_RSRC_WORD1_DATA_FORMAT_OFFSET
SQ_IMG_RSRC_WORD1_DATA_FORMAT_SIZE
: Data format (8, 8_8, 16, etc)

SQ_IMG_RSRC_WORD1_META_DIRECT_OFFSET
SQ_IMG_RSRC_WORD1_META_DIRECT_SIZE
: Direct metadata addressing flag

SQ_IMG_RSRC_WORD1_MIN_LOD_OFFSET
SQ_IMG_RSRC_WORD1_MIN_LOD_SIZE
: Minimum LOD, 4.8 format

SQ_IMG_RSRC_WORD1_NUM_FORMAT_OFFSET
SQ_IMG_RSRC_WORD1_NUM_FORMAT_SIZE
: Numeric format (unorm, snorm, float, etc)

SQ_IMG_RSRC_WORD1_NV_OFFSET
SQ_IMG_RSRC_WORD1_NV_SIZE
: non-volatile bit (1:non-volatile, 0:volatile)

SQ_IMG_RSRC_WORD2_HEIGHT_OFFSET
SQ_IMG_RSRC_WORD2_HEIGHT_SIZE
: Image Height. Expressed as 'height-1', so 0 = height of 1.

SQ_IMG_RSRC_WORD2_PERF_MOD_OFFSET
SQ_IMG_RSRC_WORD2_PERF_MOD_SIZE
: Performance modulation (scales sampler's perf_z, perf_mip, aniso_bias lod_bias_sec)

SQ_IMG_RSRC_WORD2_WIDTH_OFFSET
SQ_IMG_RSRC_WORD2_WIDTH_SIZE
: Image width. Expressed as 'width-1', so 0 = width of 1.

SQ_IMG_RSRC_WORD3_BASE_LEVEL_OFFSET
SQ_IMG_RSRC_WORD3_BASE_LEVEL_SIZE
: Base level

SQ_IMG_RSRC_WORD3_DST_SEL_W_OFFSET
SQ_IMG_RSRC_WORD3_DST_SEL_W_SIZE
: Destination data swizzle - W : x,y,z,w,n_bc_1,0,1

SQ_IMG_RSRC_WORD3_DST_SEL_X_OFFSET
SQ_IMG_RSRC_WORD3_DST_SEL_X_SIZE
: Destination data swizzle - X : x,y,z,w,n_bc_1,0,1

SQ_IMG_RSRC_WORD3_DST_SEL_Y_OFFSET
SQ_IMG_RSRC_WORD3_DST_SEL_Y_SIZE
: Destination data swizzle - Y : x,y,z,w,n_bc_1,0,1

SQ_IMG_RSRC_WORD3_DST_SEL_Z_OFFSET
SQ_IMG_RSRC_WORD3_DST_SEL_Z_SIZE
: Destination data swizzle - Z : x,y,z,w,n_bc_1,0,1

autoreg_fields ::= (continued)

SQ_IMG_RSRC_WORD3_LAST_LEVEL_OFFSET
SQ_IMG_RSRC_WORD3_LAST_LEVEL_SIZE
: Last level

SQ_IMG_RSRC_WORD3_SW_MODE_OFFSET
SQ_IMG_RSRC_WORD3_SW_MODE_SIZE
: Swizzle mode

SQ_IMG_RSRC_WORD3_TYPE_OFFSET
SQ_IMG_RSRC_WORD3_TYPE_SIZE
: Resource type: 1d, 2d, 3d, cube, 1d_array, 2d_array, 2d_msaa, 2d_msaa_array.

SQ_IMG_RSRC_WORD4_BC_SWIZZLE_OFFSET
SQ_IMG_RSRC_WORD4_BC_SWIZZLE_SIZE
: Specifies channel ordering for border color data independent of T# dst_sel_*s. Internal xyzw channels will get the following border color channels as stored in memory: 0=xyzw, 1=xwyz, 2=wzyx, 3=wxyz, 4=zyxw, 5=yxwz.

SQ_IMG_RSRC_WORD4_DEPTH_OFFSET
SQ_IMG_RSRC_WORD4_DEPTH_SIZE
: Depth of 3d texture map. Units are 'depth-1', so 0 = 1 slice, 1=2slices.

SQ_IMG_RSRC_WORD4_PITCH_OFFSET
SQ_IMG_RSRC_WORD4_PITCH_SIZE
: Pitch used for linear addressing, in units of 128B, e.g. a value of '8' means a 1024B pitch. Only 13 bits are used, the most significant bit is unused.

SQ_IMG_RSRC_WORD5_ARRAY_PITCH_OFFSET
SQ_IMG_RSRC_WORD5_ARRAY_PITCH_SIZE
: Used for texture quilting. Number of horizontal slices quilt. Encoded as trunc(log2(# horizontal slices)) + 1.

SQ_IMG_RSRC_WORD5_BASE_ARRAY_OFFSET
SQ_IMG_RSRC_WORD5_BASE_ARRAY_SIZE
: Absolute index of first valid array slice to use.

SQ_IMG_RSRC_WORD5_MAX_MIP_OFFSET
SQ_IMG_RSRC_WORD5_MAX_MIP_SIZE
: Resource MipLevels-1. Describes the resource, as opposed to base_level and last_level, which describes the resouce view. For MSAA, holds number of samples.

SQ_IMG_RSRC_WORD5_META_DATA_ADDRESS_OFFSET
SQ_IMG_RSRC_WORD5_META_DATA_ADDRESS_SIZE
: Upper bits of meta-data address.

SQ_IMG_RSRC_WORD5_META_LINEAR_OFFSET
SQ_IMG_RSRC_WORD5_META_LINEAR_SIZE
: Forces metadata surface to be linear.

SQ_IMG_RSRC_WORD5_META_PIPE_ALIGNED_OFFSET
SQ_IMG_RSRC_WORD5_META_PIPE_ALIGNED_SIZE
: Maintains pipe alignment in metadata addressing.

autoreg_fields ::= (continued)

SQ_IMG_RSRC_WORD5_META_RB_ALIGNED_OFFSET
SQ_IMG_RSRC_WORD5_META_RB_ALIGNED_SIZE
: Maintains rb alignment in metadata addressing.

SQ_IMG_RSRC_WORD6_ALPHA_IS_ON_MSB_OFFSET
SQ_IMG_RSRC_WORD6_ALPHA_IS_ON_MSB_SIZE
: DCC: Set to 1 if the surface's component swap is not reversed which is all the time by default.

SQ_IMG_RSRC_WORD6_COLOR_TRANSFORM_OFFSET
SQ_IMG_RSRC_WORD6_COLOR_TRANSFORM_SIZE
: DCC: auto = 0, none = 1.

SQ_IMG_RSRC_WORD6_COMPRESSION_EN_OFFSET
SQ_IMG_RSRC_WORD6_COMPRESSION_EN_SIZE
: Compression enable for DCC or Compressed FMask/Z/Stencil

SQ_IMG_RSRC_WORD6_COUNTER_BANK_ID_OFFSET
SQ_IMG_RSRC_WORD6_COUNTER_BANK_ID_SIZE
SQ_IMG_RSRC_WORD6_LOD_HDW_CNT_EN_OFFSET
SQ_IMG_RSRC_WORD6_LOD_HDW_CNT_EN_SIZE
SQ_IMG_RSRC_WORD6_LOST_ALPHA_BITS_OFFSET
SQ_IMG_RSRC_WORD6_LOST_ALPHA_BITS_SIZE
: DCC: E2M2.

SQ_IMG_RSRC_WORD6_LOST_COLOR_BITS_OFFSET
SQ_IMG_RSRC_WORD6_LOST_COLOR_BITS_SIZE
: DCC: E2M2.

SQ_IMG_RSRC_WORD6_MIN_LOD_WARN_OFFSET
SQ_IMG_RSRC_WORD6_MIN_LOD_WARN_SIZE
: feedback trigger for LOD.

SQ_IMG_RSRC_WORD7_META_DATA_ADDRESS_OFFSET
SQ_IMG_RSRC_WORD7_META_DATA_ADDRESS_SIZE
: DCC: Bits [39:8] of the metadata address.

SQ_IMG_SAMP_WORD0_ANISO_BIAS_OFFSET
SQ_IMG_SAMP_WORD0_ANISO_BIAS_SIZE
: aniso bias, also used in aniso tap vhen available: unsigned 1.5

SQ_IMG_SAMP_WORD0_ANISO_THRESHOLD_OFFSET
SQ_IMG_SAMP_WORD0_ANISO_THRESHOLD_SIZE
: aniso threshold

SQ_IMG_SAMP_WORD0_CLAMP_X_OFFSET
SQ_IMG_SAMP_WORD0_CLAMP_X_SIZE
: clamp/wrap mode

SQ_IMG_SAMP_WORD0_CLAMP_Y_OFFSET
SQ_IMG_SAMP_WORD0_CLAMP_Y_SIZE
: clamp/wrap mode

SQ_IMG_SAMP_WORD0_CLAMP_Z_OFFSET
SQ_IMG_SAMP_WORD0_CLAMP_Z_SIZE
: clamp/wrap mode

SQ_IMG_SAMP_WORD0_COMPAT_MODE_OFFSET
SQ_IMG_SAMP_WORD0_COMPAT_MODE_SIZE
: compatibility mode, typically for Crossfire scenarios: 0=previous (legacy), 1=current (new) generation behavior

autoreg_fields ::= (continued)

SQ_IMG_SAMP_WORD0_DEPTH_COMPARE_FUNC_OFFSET
SQ_IMG_SAMP_WORD0_DEPTH_COMPARE_FUNC_SIZE
: depth compare function

SQ_IMG_SAMP_WORD0_DISABLE_CUBE_WRAP_OFFSET
SQ_IMG_SAMP_WORD0_DISABLE_CUBE_WRAP_SIZE
: disable cubemap wrap

SQ_IMG_SAMP_WORD0_FILTER_MODE_OFFSET
SQ_IMG_SAMP_WORD0_FILTER_MODE_SIZE
: filter mode; normal lerp, min or max filter

SQ_IMG_SAMP_WORD0_FORCE_DEGAMMA_OFFSET
SQ_IMG_SAMP_WORD0_FORCE_DEGAMMA_SIZE
: force degamma on

SQ_IMG_SAMP_WORD0_FORCE_UNNORMALIZED_OFFSET
SQ_IMG_SAMP_WORD0_FORCE_UNNORMALIZED_SIZE
: force address coords to be un-normalized

SQ_IMG_SAMP_WORD0_MAX_ANISO_RATIO_OFFSET
SQ_IMG_SAMP_WORD0_MAX_ANISO_RATIO_SIZE
: maximum aniso ratio

SQ_IMG_SAMP_WORD0_MC_COORD_TRUNC_OFFSET
SQ_IMG_SAMP_WORD0_MC_COORD_TRUNC_SIZE
SQ_IMG_SAMP_WORD0_TRUNC_COORD_OFFSET
SQ_IMG_SAMP_WORD0_TRUNC_COORD_SIZE
: truncate coordinates

SQ_IMG_SAMP_WORD1_MAX_LOD_OFFSET
SQ_IMG_SAMP_WORD1_MAX_LOD_SIZE
: maximum LOD: u4.8

SQ_IMG_SAMP_WORD1_MIN_LOD_OFFSET
SQ_IMG_SAMP_WORD1_MIN_LOD_SIZE
: minimum LOD: u4.8

SQ_IMG_SAMP_WORD1_PERF_MIP_OFFSET
SQ_IMG_SAMP_WORD1_PERF_MIP_SIZE
: perf mip

SQ_IMG_SAMP_WORD1_PERF_Z_OFFSET
SQ_IMG_SAMP_WORD1_PERF_Z_SIZE
: perf z

SQ_IMG_SAMP_WORD2_ANISO_OVERRIDE_OFFSET
SQ_IMG_SAMP_WORD2_ANISO_OVERRIDE_SIZE
: Indicates if anisotropic filtering is allowed when the resource view contains one level. 0=Allow 1=Disallow, HW overrides to isotropic equivalent (AnisoPoint->Point, AnisoLinear->Linear)

SQ_IMG_SAMP_WORD2_BLEND_ZERO_PRT_OFFSET
SQ_IMG_SAMP_WORD2_BLEND_ZERO_PRT_SIZE
: For PRT fetches, zero out texel if not resident

SQ_IMG_SAMP_WORD2_FILTER_PREC_FIX_OFFSET
SQ_IMG_SAMP_WORD2_FILTER_PREC_FIX_SIZE
: Enable rounding in normalization after filter

SQ_IMG_SAMP_WORD2_LOD_BIAS_OFFSET
SQ_IMG_SAMP_WORD2_LOD_BIAS_SIZE
: LOD bias: S5.8

autoreg_fields ::= (continued)

SQ_IMG_SAMP_WORD2_LOD_BIAS_SEC_OFFSET SQ_IMG_SAMP_WORD2_LOD_BIAS_SEC_SIZE	LOD bias secondary: S1.4
SQ_IMG_SAMP_WORD2_MIP_FILTER_OFFSET SQ_IMG_SAMP_WORD2_MIP_FILTER_SIZE	mip-level filter
SQ_IMG_SAMP_WORD2_MIP_POINT_PRECLAMP_OFFSET SQ_IMG_SAMP_WORD2_MIP_POINT_PRECLAMP_SIZE	Add 0.5 before the resource/sampler clamp
SQ_IMG_SAMP_WORD2_XY_MAG_FILTER_OFFSET SQ_IMG_SAMP_WORD2_XY_MAG_FILTER_SIZE	magnification filter
SQ_IMG_SAMP_WORD2_XY_MIN_FILTER_OFFSET SQ_IMG_SAMP_WORD2_XY_MIN_FILTER_SIZE	minification filter
SQ_IMG_SAMP_WORD2_Z_FILTER_OFFSET SQ_IMG_SAMP_WORD2_Z_FILTER_SIZE	depth filter
SQ_IMG_SAMP_WORD3_BORDER_COLOR_PTR_OFFSET SQ_IMG_SAMP_WORD3_BORDER_COLOR_PTR_SIZE	pointer into a table of border colors
SQ_IMG_SAMP_WORD3_BORDER_COLOR_TYPE_OFFSET SQ_IMG_SAMP_WORD3_BORDER_COLOR_TYPE_SIZE	Opaque-black, transparent-black, white or use border color pointer.
SQ_IMG_SAMP_WORD3_SKIP_DEGAMMA_OFFSET SQ_IMG_SAMP_WORD3_SKIP_DEGAMMA_SIZE	For DATA_FORMATs that support NUM_FORMAT=sRGB, disables the conversion from sRGB to Linear Space
SQ_IND_INDEX_AUTO_INCR_OFFSET SQ_IND_INDEX_AUTO_INCR_SIZE	Enable auto-incremented of INDEX after each write or read
SQ_IND_INDEX_FORCE_READ_OFFSET SQ_IND_INDEX_FORCE_READ_SIZE	Forces a read over a register, even if CU is idle. Clocks should be enabled before doing forced reads.
SQ_IND_INDEX_INDEX_OFFSET SQ_IND_INDEX_INDEX_SIZE	Register index determines which register to read or write. This includes all registers in the SQ_IND address space. These registers are: SQ_DEBUG_STS_GLOBAL*, SQ_DEBUG_STS_LOCAL, SQ_DEBUG_CTRL_LOCAL, and SQ_WAVE_*. Indices 0x200-27f correspond to SGPRs, M0 (27c) and EXEC (27e, 27f). Indices 0x400-4ff correspond to VGPRs 0-255 of this wave for the thread (work item) specified in the THREAD_ID field of this register.

autoreg_fields ::= (continued)

SQ_IND_INDEX_READ_TIMEOUT_OFFSET
SQ_IND_INDEX_READ_TIMEOUT_SIZE
: Reports if the last read timed out.

SQ_IND_INDEX_SIMD_ID_OFFSET
SQ_IND_INDEX_SIMD_ID_SIZE
: SIMD ID for per-wave access

SQ_IND_INDEX_THREAD_ID_OFFSET
SQ_IND_INDEX_THREAD_ID_SIZE
: Thread ID for VGPR access

SQ_IND_INDEX_UNINDEXED_OFFSET
SQ_IND_INDEX_UNINDEXED_SIZE
: Indicates that we should do reads/writes from the unindexed address space (GFXDEC,SQDEC,SQCONDEC). Unimplemented.

SQ_IND_INDEX_WAVE_ID_OFFSET
SQ_IND_INDEX_WAVE_ID_SIZE
: Wave ID for per-wave access

SQ_IND_DATA_DATA_OFFSET
SQ_IND_DATA_DATA_SIZE
: Data Dword

SQ_CMD_CHECK_VMID_OFFSET
SQ_CMD_CHECK_VMID_SIZE
: Whether we should limit this operation to a select VMID

SQ_CMD_CMD_OFFSET
SQ_CMD_CMD_SIZE
: The type of command to send.

SQ_CMD_DATA_OFFSET
SQ_CMD_DATA_SIZE

SQ_CMD_MODE_OFFSET
SQ_CMD_MODE_SIZE
: Which Waves should receive the command.

SQ_CMD_QUEUE_ID_OFFSET
SQ_CMD_QUEUE_ID_SIZE
: Queue ID (just for MODE == KILL_PIPELINE)

SQ_CMD_SIMD_ID_OFFSET
SQ_CMD_SIMD_ID_SIZE
: Simd ID of target wave

SQ_CMD_VM_ID_OFFSET
SQ_CMD_VM_ID_SIZE
: Virtual memory ID for use with CHECK_VMID mode

SQ_CMD_WAVE_ID_OFFSET
SQ_CMD_WAVE_ID_SIZE
: Wave ID of target wave

SQ_DEBUG_STS_GLOBAL_BUSY_OFFSET
SQ_DEBUG_STS_GLOBAL_BUSY_SIZE
: Which compute units are busy in shader the shader engine

SQ_DEBUG_STS_GLOBAL_INTERRUPT_MSG_BUSY_OFFSET
SQ_DEBUG_STS_GLOBAL_INTERRUPT_MSG_BUSY_SIZE
: Indicates if instruction or ECC generated interrupts may be pending.

SQ_DEBUG_STS_GLOBAL_WAVE_LEVEL_SH0_OFFSET
SQ_DEBUG_STS_GLOBAL_WAVE_LEVEL_SH0_SIZE
: Number of waves active in SH0

SQ_DEBUG_STS_GLOBAL_WAVE_LEVEL_SH1_OFFSET
SQ_DEBUG_STS_GLOBAL_WAVE_LEVEL_SH1_SIZE
: Number of waves active in SH1

SQ_DEBUG_STS_LOCAL_BUSY_OFFSET
SQ_DEBUG_STS_LOCAL_BUSY_SIZE
: Whether there are active waves

SQ_DEBUG_STS_LOCAL_WAVE_LEVEL_OFFSET
SQ_DEBUG_STS_LOCAL_WAVE_LEVEL_SIZE
: Indicates how many active waves there are in this SQ

autoreg_fields ::= (continued)

```
SQ_EDC_CNT_LDS_D_DED_COUNT_OFFSET
SQ_EDC_CNT_LDS_D_DED_COUNT_SIZE
SQ_EDC_CNT_LDS_D_SEC_COUNT_OFFSET
SQ_EDC_CNT_LDS_D_SEC_COUNT_SIZE
SQ_EDC_CNT_LDS_I_DED_COUNT_OFFSET
SQ_EDC_CNT_LDS_I_DED_COUNT_SIZE
SQ_EDC_CNT_LDS_I_SEC_COUNT_OFFSET
SQ_EDC_CNT_LDS_I_SEC_COUNT_SIZE
SQ_EDC_CNT_SGPR_DED_COUNT_OFFSET
SQ_EDC_CNT_SGPR_DED_COUNT_SIZE
SQ_EDC_CNT_SGPR_SEC_COUNT_OFFSET
SQ_EDC_CNT_SGPR_SEC_COUNT_SIZE
SQ_EDC_CNT_VGPR0_DED_COUNT_OFFSET
SQ_EDC_CNT_VGPR0_DED_COUNT_SIZE
SQ_EDC_CNT_VGPR0_SEC_COUNT_OFFSET
SQ_EDC_CNT_VGPR0_SEC_COUNT_SIZE
SQ_EDC_CNT_VGPR1_DED_COUNT_OFFSET
SQ_EDC_CNT_VGPR1_DED_COUNT_SIZE
SQ_EDC_CNT_VGPR1_SEC_COUNT_OFFSET
SQ_EDC_CNT_VGPR1_SEC_COUNT_SIZE
SQ_EDC_CNT_VGPR2_DED_COUNT_OFFSET
SQ_EDC_CNT_VGPR2_DED_COUNT_SIZE
SQ_EDC_CNT_VGPR2_SEC_COUNT_OFFSET
SQ_EDC_CNT_VGPR2_SEC_COUNT_SIZE
SQ_EDC_CNT_VGPR3_DED_COUNT_OFFSET
SQ_EDC_CNT_VGPR3_DED_COUNT_SIZE
SQ_EDC_CNT_VGPR3_SEC_COUNT_OFFSET
SQ_EDC_CNT_VGPR3_SEC_COUNT_SIZE
SQC_EDC_CNT_DATA_CU0_UTCL1_LFIFO_DED_COUNT_OFFSET
SQC_EDC_CNT_DATA_CU0_UTCL1_LFIFO_DED_COUNT_SIZE
SQC_EDC_CNT_DATA_CU0_UTCL1_LFIFO_SEC_COUNT_OFFSET
SQC_EDC_CNT_DATA_CU0_UTCL1_LFIFO_SEC_COUNT_SIZE
SQC_EDC_CNT_DATA_CU0_WRITE_DATA_BUF_DED_COUNT_OFFSET
SQC_EDC_CNT_DATA_CU0_WRITE_DATA_BUF_DED_COUNT_SIZE
SQC_EDC_CNT_DATA_CU0_WRITE_DATA_BUF_SEC_COUNT_OFFSET
SQC_EDC_CNT_DATA_CU0_WRITE_DATA_BUF_SEC_COUNT_SIZE
SQC_EDC_CNT_DATA_CU1_UTCL1_LFIFO_DED_COUNT_OFFSET
SQC_EDC_CNT_DATA_CU1_UTCL1_LFIFO_DED_COUNT_SIZE
SQC_EDC_CNT_DATA_CU1_UTCL1_LFIFO_SEC_COUNT_OFFSET
SQC_EDC_CNT_DATA_CU1_UTCL1_LFIFO_SEC_COUNT_SIZE
SQC_EDC_CNT_DATA_CU1_WRITE_DATA_BUF_DED_COUNT_OFFSET
SQC_EDC_CNT_DATA_CU1_WRITE_DATA_BUF_DED_COUNT_SIZE
SQC_EDC_CNT_DATA_CU1_WRITE_DATA_BUF_SEC_COUNT_OFFSET
SQC_EDC_CNT_DATA_CU1_WRITE_DATA_BUF_SEC_COUNT_SIZE
SQC_EDC_CNT_DATA_CU2_UTCL1_LFIFO_DED_COUNT_OFFSET
SQC_EDC_CNT_DATA_CU2_UTCL1_LFIFO_DED_COUNT_SIZE
SQC_EDC_CNT_DATA_CU2_UTCL1_LFIFO_SEC_COUNT_OFFSET
SQC_EDC_CNT_DATA_CU2_UTCL1_LFIFO_SEC_COUNT_SIZE
```

autoreg_fields : := (continued)

SQC_EDC_CNT_DATA_CU2_WRITE_DATA_BUF_DED_COUNT_OFFSET
SQC_EDC_CNT_DATA_CU2_WRITE_DATA_BUF_DED_COUNT_SIZE
SQC_EDC_CNT_DATA_CU2_WRITE_DATA_BUF_SEC_COUNT_OFFSET
SQC_EDC_CNT_DATA_CU2_WRITE_DATA_BUF_SEC_COUNT_SIZE
SQC_EDC_CNT_DATA_CU3_UTCL1_LFIFO_DED_COUNT_OFFSET
SQC_EDC_CNT_DATA_CU3_UTCL1_LFIFO_DED_COUNT_SIZE
SQC_EDC_CNT_DATA_CU3_UTCL1_LFIFO_SEC_COUNT_OFFSET
SQC_EDC_CNT_DATA_CU3_UTCL1_LFIFO_SEC_COUNT_SIZE
SQC_EDC_CNT_DATA_CU3_WRITE_DATA_BUF_DED_COUNT_OFFSET
SQC_EDC_CNT_DATA_CU3_WRITE_DATA_BUF_DED_COUNT_SIZE
SQC_EDC_CNT_DATA_CU3_WRITE_DATA_BUF_SEC_COUNT_OFFSET
SQC_EDC_CNT_DATA_CU3_WRITE_DATA_BUF_SEC_COUNT_SIZE
SQC_EDC_CNT2_DATA_BANKA_BANK_RAM_DED_COUNT_OFFSET
SQC_EDC_CNT2_DATA_BANKA_BANK_RAM_DED_COUNT_SIZE
SQC_EDC_CNT2_DATA_BANKA_BANK_RAM_SEC_COUNT_OFFSET
SQC_EDC_CNT2_DATA_BANKA_BANK_RAM_SEC_COUNT_SIZE
SQC_EDC_CNT2_DATA_BANKA_DIRTY_BIT_RAM_SED_COUNT_OFFSET
SQC_EDC_CNT2_DATA_BANKA_DIRTY_BIT_RAM_SED_COUNT_SIZE
SQC_EDC_CNT2_DATA_BANKA_HIT_FIFO_SED_COUNT_OFFSET
SQC_EDC_CNT2_DATA_BANKA_HIT_FIFO_SED_COUNT_SIZE
SQC_EDC_CNT2_DATA_BANKA_MISS_FIFO_SED_COUNT_OFFSET
SQC_EDC_CNT2_DATA_BANKA_MISS_FIFO_SED_COUNT_SIZE
SQC_EDC_CNT2_DATA_BANKA_TAG_RAM_DED_COUNT_OFFSET
SQC_EDC_CNT2_DATA_BANKA_TAG_RAM_DED_COUNT_SIZE
SQC_EDC_CNT2_DATA_BANKA_TAG_RAM_SEC_COUNT_OFFSET
SQC_EDC_CNT2_DATA_BANKA_TAG_RAM_SEC_COUNT_SIZE
SQC_EDC_CNT2_INST_BANKA_BANK_RAM_DED_COUNT_OFFSET
SQC_EDC_CNT2_INST_BANKA_BANK_RAM_DED_COUNT_SIZE
SQC_EDC_CNT2_INST_BANKA_BANK_RAM_SEC_COUNT_OFFSET
SQC_EDC_CNT2_INST_BANKA_BANK_RAM_SEC_COUNT_SIZE
SQC_EDC_CNT2_INST_BANKA_MISS_FIFO_SED_COUNT_OFFSET
SQC_EDC_CNT2_INST_BANKA_MISS_FIFO_SED_COUNT_SIZE
SQC_EDC_CNT2_INST_BANKA_TAG_RAM_DED_COUNT_OFFSET
SQC_EDC_CNT2_INST_BANKA_TAG_RAM_DED_COUNT_SIZE
SQC_EDC_CNT2_INST_BANKA_TAG_RAM_SEC_COUNT_OFFSET
SQC_EDC_CNT2_INST_BANKA_TAG_RAM_SEC_COUNT_SIZE
SQC_EDC_CNT2_INST_BANKA_UTCL1_MISS_FIFO_SED_COUNT_OFFSET
SQC_EDC_CNT2_INST_BANKA_UTCL1_MISS_FIFO_SED_COUNT_SIZE
SQC_EDC_CNT2_INST_UTCL1_LFIFO_DED_COUNT_OFFSET
SQC_EDC_CNT2_INST_UTCL1_LFIFO_DED_COUNT_SIZE
SQC_EDC_CNT2_INST_UTCL1_LFIFO_SEC_COUNT_OFFSET
SQC_EDC_CNT2_INST_UTCL1_LFIFO_SEC_COUNT_SIZE
SQC_EDC_CNT3_DATA_BANKB_BANK_RAM_DED_COUNT_OFFSET
SQC_EDC_CNT3_DATA_BANKB_BANK_RAM_DED_COUNT_SIZE
SQC_EDC_CNT3_DATA_BANKB_BANK_RAM_SEC_COUNT_OFFSET
SQC_EDC_CNT3_DATA_BANKB_BANK_RAM_SEC_COUNT_SIZE
SQC_EDC_CNT3_DATA_BANKB_DIRTY_BIT_RAM_SED_COUNT_OFFSET
SQC_EDC_CNT3_DATA_BANKB_DIRTY_BIT_RAM_SED_COUNT_SIZE

autoreg_fields : := (continued)

SQC_EDC_CNT3_DATA_BANKB_HIT_FIFO_SED_COUNT_OFFSET
SQC_EDC_CNT3_DATA_BANKB_HIT_FIFO_SED_COUNT_SIZE
SQC_EDC_CNT3_DATA_BANKB_MISS_FIFO_SED_COUNT_OFFSET
SQC_EDC_CNT3_DATA_BANKB_MISS_FIFO_SED_COUNT_SIZE
SQC_EDC_CNT3_DATA_BANKB_TAG_RAM_DED_COUNT_OFFSET
SQC_EDC_CNT3_DATA_BANKB_TAG_RAM_DED_COUNT_SIZE
SQC_EDC_CNT3_DATA_BANKB_TAG_RAM_SEC_COUNT_OFFSET
SQC_EDC_CNT3_DATA_BANKB_TAG_RAM_SEC_COUNT_SIZE
SQC_EDC_CNT3_INST_BANKB_BANK_RAM_DED_COUNT_OFFSET
SQC_EDC_CNT3_INST_BANKB_BANK_RAM_DED_COUNT_SIZE
SQC_EDC_CNT3_INST_BANKB_BANK_RAM_SEC_COUNT_OFFSET
SQC_EDC_CNT3_INST_BANKB_BANK_RAM_SEC_COUNT_SIZE
SQC_EDC_CNT3_INST_BANKB_MISS_FIFO_SED_COUNT_OFFSET
SQC_EDC_CNT3_INST_BANKB_MISS_FIFO_SED_COUNT_SIZE
SQC_EDC_CNT3_INST_BANKB_TAG_RAM_DED_COUNT_OFFSET
SQC_EDC_CNT3_INST_BANKB_TAG_RAM_DED_COUNT_SIZE
SQC_EDC_CNT3_INST_BANKB_TAG_RAM_SEC_COUNT_OFFSET
SQC_EDC_CNT3_INST_BANKB_TAG_RAM_SEC_COUNT_SIZE
SQC_EDC_CNT3_INST_BANKB_UTCL1_MISS_FIFO_SED_COUNT_OFFSET
SQC_EDC_CNT3_INST_BANKB_UTCL1_MISS_FIFO_SED_COUNT_SIZE
SQ_EDC_FUE_CNTL_BLOCK_FUE_FLAGS_OFFSET
SQ_EDC_FUE_CNTL_BLOCK_FUE_FLAGS_SIZE
: Bit 0: SIMD0 Bit 1: SIMD1 Bit 2: SIMD2 Bit 3: SIMD3 Bit 4: SQ Bit 5: LDS Bit 6: TD Bit 7: TA Bit 8: TCP
SQ_EDC_FUE_CNTL_FUE_INTERRUPT_ENABLES_OFFSET
SQ_EDC_FUE_CNTL_FUE_INTERRUPT_ENABLES_SIZE
: Bit 0: SIMD0 Bit 1: SIMD1 Bit 2: SIMD2 Bit 3: SIMD3 Bit 4: SQ Bit 5: LDS Bit 6: TD Bit 7: TA Bit 8: TCP
SQC_EDC_FUE_CNTL_BLOCK_FUE_FLAGS_OFFSET
SQC_EDC_FUE_CNTL_BLOCK_FUE_FLAGS_SIZE
: Bit 0: SQC, bit 1: TCIW
SQC_EDC_FUE_CNTL_FUE_INTERRUPT_ENABLES_OFFSET
SQC_EDC_FUE_CNTL_FUE_INTERRUPT_ENABLES_SIZE
: Bit 0: SQC, bit 1: TCIW
SQ_INTERRUPT_WORD_CMN_HI_ENCODING_OFFSET
SQ_INTERRUPT_WORD_CMN_HI_ENCODING_SIZE
: Which encoding is used for this interrupt
SQ_INTERRUPT_WORD_CMN_HI_SE_ID_OFFSET
SQ_INTERRUPT_WORD_CMN_HI_SE_ID_SIZE
: Shader engine ID
SQ_INTERRUPT_WORD_AUTO_LO_CMD_TIMESTAMP_OFFSET
SQ_INTERRUPT_WORD_AUTO_LO_CMD_TIMESTAMP_SIZE
: Indicates the SQ_CMD_TIMESTAMP register has been updated.
SQ_INTERRUPT_WORD_AUTO_LO_HOST_CMD_OVERFLOW_OFFSET
SQ_INTERRUPT_WORD_AUTO_LO_HOST_CMD_OVERFLOW_SIZE
: Host command fifo has overflowed
SQ_INTERRUPT_WORD_AUTO_LO_HOST_REG_OVERFLOW_OFFSET
SQ_INTERRUPT_WORD_AUTO_LO_HOST_REG_OVERFLOW_SIZE
: Host register fifo has overflowed
SQ_INTERRUPT_WORD_AUTO_LO_IMMED_OVERFLOW_OFFSET
SQ_INTERRUPT_WORD_AUTO_LO_IMMED_OVERFLOW_SIZE
: Immediate register fifo has overflowed

autoreg_fields ::= (continued)

SQ_INTERRUPT_WORD_AUTO_LO_REG_TIMESTAMP_OFFSET
SQ_INTERRUPT_WORD_AUTO_LO_REG_TIMESTAMP_SIZE
: Indicates the SQ_REG_TIMESTAMP register has been updated.

SQ_INTERRUPT_WORD_AUTO_LO_THREAD_TRACE_OFFSET
SQ_INTERRUPT_WORD_AUTO_LO_THREAD_TRACE_SIZE
: Set when thread trace generates an interrupt do to thread trace finish.

SQ_INTERRUPT_WORD_AUTO_LO_THREAD_TRACE_BUF_FULL_OFFSET
SQ_INTERRUPT_WORD_AUTO_LO_THREAD_TRACE_BUF_FULL_SIZE
: Indicates a thread trace buffer has been filled.

SQ_INTERRUPT_WORD_AUTO_LO_THREAD_TRACE_UTC_ERROR_OFFSET
SQ_INTERRUPT_WORD_AUTO_LO_THREAD_TRACE_UTC_ERROR_SIZE
: Indicates a thread trace buffer has encountered an UTC error.

SQ_INTERRUPT_WORD_AUTO_LO_WLT_OFFSET
SQ_INTERRUPT_WORD_AUTO_LO_WLT_SIZE
: Set when wave lifetime is exceeded and generates an interrupt.

SQ_INTERRUPT_WORD_AUTO_HI_ENCODING_OFFSET
SQ_INTERRUPT_WORD_AUTO_HI_ENCODING_SIZE
: Which encoding is used for this interrupt

SQ_INTERRUPT_WORD_AUTO_HI_SE_ID_OFFSET
SQ_INTERRUPT_WORD_AUTO_HI_SE_ID_SIZE
: Shader engine ID

SQ_INTERRUPT_WORD_WAVE_LO_DATA_OFFSET
SQ_INTERRUPT_WORD_WAVE_LO_DATA_SIZE
: User-supplied data from M0[23:0]

SQ_INTERRUPT_WORD_WAVE_LO_PRIV_OFFSET
SQ_INTERRUPT_WORD_WAVE_LO_PRIV_SIZE
: Set if interrupt was generated by the trap handler.

SQ_INTERRUPT_WORD_WAVE_LO_SH_ID_OFFSET
SQ_INTERRUPT_WORD_WAVE_LO_SH_ID_SIZE
: Shader array ID

SQ_INTERRUPT_WORD_WAVE_LO_SIMD_ID_OFFSET
SQ_INTERRUPT_WORD_WAVE_LO_SIMD_ID_SIZE
: SIMD number

SQ_INTERRUPT_WORD_WAVE_LO_WAVE_ID_OFFSET
SQ_INTERRUPT_WORD_WAVE_LO_WAVE_ID_SIZE
: Wave identifier

SQ_INTERRUPT_WORD_WAVE_HI_CU_ID_OFFSET
SQ_INTERRUPT_WORD_WAVE_HI_CU_ID_SIZE
: Compute unit ID

SQ_INTERRUPT_WORD_WAVE_HI_ENCODING_OFFSET
SQ_INTERRUPT_WORD_WAVE_HI_ENCODING_SIZE
: Which encoding is used for this interrupt

SQ_INTERRUPT_WORD_WAVE_HI_SE_ID_OFFSET
SQ_INTERRUPT_WORD_WAVE_HI_SE_ID_SIZE
: Shader engine ID

SQ_INTERRUPT_WORD_WAVE_HI_VM_ID_OFFSET
SQ_INTERRUPT_WORD_WAVE_HI_VM_ID_SIZE
: Virtual memory ID

autoreg_fields : := (continued)

SQ_INTERRUPT_WORD_CMN_CTXID_ENCODING_OFFSET
SQ_INTERRUPT_WORD_CMN_CTXID_ENCODING_SIZE
: Which encoding is used for this interrupt

SQ_INTERRUPT_WORD_CMN_CTXID_SE_ID_OFFSET
SQ_INTERRUPT_WORD_CMN_CTXID_SE_ID_SIZE
: Shader engine ID

SQ_INTERRUPT_WORD_AUTO_CTXID_CMD_TIMESTAMP_OFFSET
SQ_INTERRUPT_WORD_AUTO_CTXID_CMD_TIMESTAMP_SIZE
: Indicates the SQ_CMD_TIMESTAMP register has been updated.

SQ_INTERRUPT_WORD_AUTO_CTXID_ENCODING_OFFSET
SQ_INTERRUPT_WORD_AUTO_CTXID_ENCODING_SIZE
: Which encoding is used for this interrupt

SQ_INTERRUPT_WORD_AUTO_CTXID_HOST_CMD_OVERFLOW_OFFSET
SQ_INTERRUPT_WORD_AUTO_CTXID_HOST_CMD_OVERFLOW_SIZE
: Host command fifo has overflowed

SQ_INTERRUPT_WORD_AUTO_CTXID_HOST_REG_OVERFLOW_OFFSET
SQ_INTERRUPT_WORD_AUTO_CTXID_HOST_REG_OVERFLOW_SIZE
: Host register fifo has overflowed

SQ_INTERRUPT_WORD_AUTO_CTXID_IMMED_OVERFLOW_OFFSET
SQ_INTERRUPT_WORD_AUTO_CTXID_IMMED_OVERFLOW_SIZE
: Immediate register fifo has overflowed

SQ_INTERRUPT_WORD_AUTO_CTXID_REG_TIMESTAMP_OFFSET
SQ_INTERRUPT_WORD_AUTO_CTXID_REG_TIMESTAMP_SIZE
: Indicates the SQ_REG_TIMESTAMP register has been updated.

SQ_INTERRUPT_WORD_AUTO_CTXID_SE_ID_OFFSET
SQ_INTERRUPT_WORD_AUTO_CTXID_SE_ID_SIZE
: Shader engine ID

SQ_INTERRUPT_WORD_AUTO_CTXID_THREAD_TRACE_OFFSET
SQ_INTERRUPT_WORD_AUTO_CTXID_THREAD_TRACE_SIZE
: Set when thread trace generates an interrupt do to thread trace finish.

SQ_INTERRUPT_WORD_AUTO_CTXID_THREAD_TRACE_BUF_FULL_OFFSET
SQ_INTERRUPT_WORD_AUTO_CTXID_THREAD_TRACE_BUF_FULL_SIZE
: Indicates a thread trace buffer has been filled.

SQ_INTERRUPT_WORD_AUTO_CTXID_THREAD_TRACE_UTC_ERROR_OFFSET
SQ_INTERRUPT_WORD_AUTO_CTXID_THREAD_TRACE_UTC_ERROR_SIZE
: Indicates a thread trace buffer has encountered an UTC error.

SQ_INTERRUPT_WORD_AUTO_CTXID_WLT_OFFSET
SQ_INTERRUPT_WORD_AUTO_CTXID_WLT_SIZE
: Set when wave lifetime is exceeded and generates an interrupt.

SQ_INTERRUPT_WORD_WAVE_CTXID_CU_ID_OFFSET
SQ_INTERRUPT_WORD_WAVE_CTXID_CU_ID_SIZE
: Compute unit ID

SQ_INTERRUPT_WORD_WAVE_CTXID_DATA_OFFSET
SQ_INTERRUPT_WORD_WAVE_CTXID_DATA_SIZE
: User-supplied data from M0[11:0]

SQ_INTERRUPT_WORD_WAVE_CTXID_ENCODING_OFFSET
SQ_INTERRUPT_WORD_WAVE_CTXID_ENCODING_SIZE
: Which encoding is used for this interrupt

autoreg_fields : := (continued)

SQ_INTERRUPT_WORD_WAVE_CTXID_PRIV_OFFSET
SQ_INTERRUPT_WORD_WAVE_CTXID_PRIV_SIZE
: Set if interrupt was generated by the trap handler.

SQ_INTERRUPT_WORD_WAVE_CTXID_SE_ID_OFFSET
SQ_INTERRUPT_WORD_WAVE_CTXID_SE_ID_SIZE
: Shader engine ID

SQ_INTERRUPT_WORD_WAVE_CTXID_SH_ID_OFFSET
SQ_INTERRUPT_WORD_WAVE_CTXID_SH_ID_SIZE
: Shader array ID

SQ_INTERRUPT_WORD_WAVE_CTXID_SIMD_ID_OFFSET
SQ_INTERRUPT_WORD_WAVE_CTXID_SIMD_ID_SIZE
: SIMD number

SQ_INTERRUPT_WORD_WAVE_CTXID_WAVE_ID_OFFSET
SQ_INTERRUPT_WORD_WAVE_CTXID_WAVE_ID_SIZE
: Wave identifier

SQ_THREAD_TRACE_STATUS_BUSY_OFFSET
SQ_THREAD_TRACE_STATUS_BUSY_SIZE
: Flag indicating that events have been logged and are being written to memory. The BASE and SIZE registers should not be changed while this bit is set (or while there are any events in the pipe that could be logged).

SQ_THREAD_TRACE_STATUS_FINISH_DONE_OFFSET
SQ_THREAD_TRACE_STATUS_FINISH_DONE_SIZE
: Set when a THREAD_TRACE_FINISH event has completed and all data has been written to the buffer. One bit per pipe. Cleared when SQ_THREAD_TRACE_CTRL.RESET_BUFFER is set or a new THREAD_TRACE_FINISH event is received. LSB = ME1, pipe 0. MSB = GFX(ME0), max pipe.

SQ_THREAD_TRACE_STATUS_FINISH_PENDING_OFFSET
SQ_THREAD_TRACE_STATUS_FINISH_PENDING_SIZE
: Set when a THREAD_TRACE_FINISH event has been received. One bit per pipe. Cleared when SQ_THREAD_TRACE_CTRL.RESET_BUFFER is set or THREAD_TRACE_FINISH event completes the pipeline. LSB = ME1, pipe 0. MSB = GFX(ME0), max pipe.

autoreg_fields : := (continued)

SQ_THREAD_TRACE_STATUS_FULL_OFFSET
SQ_THREAD_TRACE_STATUS_FULL_SIZE
: Flag indicating a complete buffer has been written. In wrap mode, this means the pointer has wrapped since SQ_THREAD_TRACE_BUFFER_BASE was last written. Writing SQ_THREAD_TRACE_BUFFER_BASE or writing SQ_THREAD_TRACE_CTRL.RESET_BUFFER will clear this bit.

SQ_THREAD_TRACE_STATUS_NEW_BUF_OFFSET
SQ_THREAD_TRACE_STATUS_NEW_BUF_SIZE
: New buffer has been provided to be used when current buffer fills.

SQ_THREAD_TRACE_STATUS_UTC_ERROR_OFFSET
SQ_THREAD_TRACE_STATUS_UTC_ERROR_SIZE
: UTC error detected. Can be written to reset the error.

SQ_DSM_CNTL_LDS_ENABLE_SINGLE_WRITE01_OFFSET
SQ_DSM_CNTL_LDS_ENABLE_SINGLE_WRITE01_SIZE
: 0: If irritation enabled, corrupt all entries when writing, while irritator data (0+1) is Hi; 1: If irritation enabled, corrupt only the very next write after every Lo to Hi transition of irritator data (0+1)

SQ_DSM_CNTL_LDS_ENABLE_SINGLE_WRITE23_OFFSET
SQ_DSM_CNTL_LDS_ENABLE_SINGLE_WRITE23_SIZE
: 0: If irritation enabled, corrupt all entries when writing, while irritator data (0+1) is Hi; 1: If irritation enabled, corrupt only the very next write after every Lo to Hi transition of irritator data (0+1)

SQ_DSM_CNTL_SEL_DSM_LDS_IRRITATOR_DATA0_OFFSET
SQ_DSM_CNTL_SEL_DSM_LDS_IRRITATOR_DATA0_SIZE
: Select irritator bit 0, for mem irritation

SQ_DSM_CNTL_SEL_DSM_LDS_IRRITATOR_DATA1_OFFSET
SQ_DSM_CNTL_SEL_DSM_LDS_IRRITATOR_DATA1_SIZE
: Select irritator bit 1, for mem irritation

SQ_DSM_CNTL_SEL_DSM_LDS_IRRITATOR_DATA2_OFFSET
SQ_DSM_CNTL_SEL_DSM_LDS_IRRITATOR_DATA2_SIZE
: Select irritator bit 2, for sp input buffer irritation

SQ_DSM_CNTL_SEL_DSM_LDS_IRRITATOR_DATA3_OFFSET
SQ_DSM_CNTL_SEL_DSM_LDS_IRRITATOR_DATA3_SIZE
: Select irritator bit 3, for sp input buffer irritation

SQ_DSM_CNTL_SEL_DSM_SGPR_IRRITATOR_DATA0_OFFSET
SQ_DSM_CNTL_SEL_DSM_SGPR_IRRITATOR_DATA0_SIZE
: Select irritator bit 0, for irritation

SQ_DSM_CNTL_SEL_DSM_SGPR_IRRITATOR_DATA1_OFFSET
SQ_DSM_CNTL_SEL_DSM_SGPR_IRRITATOR_DATA1_SIZE
: Select irritator bit 1, for irritation

SQ_DSM_CNTL_SEL_DSM_SP_IRRITATOR_DATA0_OFFSET
SQ_DSM_CNTL_SEL_DSM_SP_IRRITATOR_DATA0_SIZE
: Select irritator bit 0, for irritation

autoreg_fields ::= (continued)

SQ_DSM_CNTL_SEL_DSM_SP_IRRITATOR_DATA1_OFFSET
SQ_DSM_CNTL_SEL_DSM_SP_IRRITATOR_DATA1_SIZE
: Select irritator bit 1, for irritation

SQ_DSM_CNTL_SGPR_ENABLE_SINGLE_WRITE_OFFSET
SQ_DSM_CNTL_SGPR_ENABLE_SINGLE_WRITE_SIZE
: 0: If irritation enabled, corrupt all entries when writing, while irritator data (0+1) is Hi; 1: If irritation enabled, corrupt only the very next write after every Lo to Hi transition of irritator data (0+1)

SQ_DSM_CNTL_SPI_BACKPRESSURE_0_OFFSET
SQ_DSM_CNTL_SPI_BACKPRESSURE_0_SIZE
: Signals backpressure to SPI when irritator bit 0 is 1.

SQ_DSM_CNTL_SPI_BACKPRESSURE_1_OFFSET
SQ_DSM_CNTL_SPI_BACKPRESSURE_1_SIZE
: Signals backpressure to SPI when irritator bit 1 is 1.

SQ_DSM_CNTL_SP_ENABLE_SINGLE_WRITE_OFFSET
SQ_DSM_CNTL_SP_ENABLE_SINGLE_WRITE_SIZE
: 0: If irritation enabled, corrupt all entries when writing, while irritator data (0+1) is Hi; 1: If irritation enabled, corrupt only the very next write after every Lo to Hi transition of irritator data (0+1)

SQ_DSM_CNTL_WAVEFRONT_STALL_0_OFFSET
SQ_DSM_CNTL_WAVEFRONT_STALL_0_SIZE
: Halts all wavefronts when irritator bit 0 is 1.

SQ_DSM_CNTL_WAVEFRONT_STALL_1_OFFSET
SQ_DSM_CNTL_WAVEFRONT_STALL_1_SIZE
: Halts all wavefronts when irritator bit 1 is 1.

SQ_DSM_CNTL2_LDS_D_ENABLE_ERROR_INJECT_OFFSET
SQ_DSM_CNTL2_LDS_D_ENABLE_ERROR_INJECT_SIZE
: Enable error inject

SQ_DSM_CNTL2_LDS_D_SELECT_INJECT_DELAY_OFFSET
SQ_DSM_CNTL2_LDS_D_SELECT_INJECT_DELAY_SIZE
: Select inject delay

SQ_DSM_CNTL2_LDS_INJECT_DELAY_OFFSET
SQ_DSM_CNTL2_LDS_INJECT_DELAY_SIZE
: Number of events + 1 to delay error injection.

SQ_DSM_CNTL2_LDS_I_ENABLE_ERROR_INJECT_OFFSET
SQ_DSM_CNTL2_LDS_I_ENABLE_ERROR_INJECT_SIZE
: Enable error inject

SQ_DSM_CNTL2_LDS_I_SELECT_INJECT_DELAY_OFFSET
SQ_DSM_CNTL2_LDS_I_SELECT_INJECT_DELAY_SIZE
: Select inject delay

SQ_DSM_CNTL2_SGPR_ENABLE_ERROR_INJECT_OFFSET
SQ_DSM_CNTL2_SGPR_ENABLE_ERROR_INJECT_SIZE
: Enable error inject

SQ_DSM_CNTL2_SGPR_SELECT_INJECT_DELAY_OFFSET
SQ_DSM_CNTL2_SGPR_SELECT_INJECT_DELAY_SIZE
: Select inject delay

SQ_DSM_CNTL2_SP_ENABLE_ERROR_INJECT_OFFSET
SQ_DSM_CNTL2_SP_ENABLE_ERROR_INJECT_SIZE
: Enable error inject

SQ_DSM_CNTL2_SP_INJECT_DELAY_OFFSET
SQ_DSM_CNTL2_SP_INJECT_DELAY_SIZE
: Number of events + 1 to delay error injection.

autoreg_fields : := (continued)

SQ_DSM_CNTL2_SP_SELECT_INJECT_DELAY_OFFSET
SQ_DSM_CNTL2_SP_SELECT_INJECT_DELAY_SIZE
Select inject delay
SQ_DSM_CNTL2_SQ_INJECT_DELAY_OFFSET
SQ_DSM_CNTL2_SQ_INJECT_DELAY_SIZE
Number of events + 1 to delay error injection.
SQC_DSM_CNTLA_DATA_BANK_RAM_DSM_IRRITATOR_DATA_OFFSET
SQC_DSM_CNTLA_DATA_BANK_RAM_DSM_IRRITATOR_DATA_SIZE
SQC_DSM_CNTLA_DATA_BANK_RAM_ENABLE_SINGLE_WRITE_OFFSET
SQC_DSM_CNTLA_DATA_BANK_RAM_ENABLE_SINGLE_WRITE_SIZE
SQC_DSM_CNTLA_DATA_DIRTY_BIT_RAM_DSM_IRRITATOR_DATA_OFFSET
SQC_DSM_CNTLA_DATA_DIRTY_BIT_RAM_DSM_IRRITATOR_DATA_SIZE
SQC_DSM_CNTLA_DATA_DIRTY_BIT_RAM_ENABLE_SINGLE_WRITE_OFFSET
SQC_DSM_CNTLA_DATA_DIRTY_BIT_RAM_ENABLE_SINGLE_WRITE_SIZE
SQC_DSM_CNTLA_DATA_HIT_FIFO_DSM_IRRITATOR_DATA_OFFSET
SQC_DSM_CNTLA_DATA_HIT_FIFO_DSM_IRRITATOR_DATA_SIZE
SQC_DSM_CNTLA_DATA_HIT_FIFO_ENABLE_SINGLE_WRITE_OFFSET
SQC_DSM_CNTLA_DATA_HIT_FIFO_ENABLE_SINGLE_WRITE_SIZE
SQC_DSM_CNTLA_DATA_MISS_FIFO_DSM_IRRITATOR_DATA_OFFSET
SQC_DSM_CNTLA_DATA_MISS_FIFO_DSM_IRRITATOR_DATA_SIZE
SQC_DSM_CNTLA_DATA_MISS_FIFO_ENABLE_SINGLE_WRITE_OFFSET
SQC_DSM_CNTLA_DATA_MISS_FIFO_ENABLE_SINGLE_WRITE_SIZE
SQC_DSM_CNTLA_DATA_TAG_RAM_DSM_IRRITATOR_DATA_OFFSET
SQC_DSM_CNTLA_DATA_TAG_RAM_DSM_IRRITATOR_DATA_SIZE
SQC_DSM_CNTLA_DATA_TAG_RAM_ENABLE_SINGLE_WRITE_OFFSET
SQC_DSM_CNTLA_DATA_TAG_RAM_ENABLE_SINGLE_WRITE_SIZE
SQC_DSM_CNTLA_INST_BANK_RAM_DSM_IRRITATOR_DATA_OFFSET
SQC_DSM_CNTLA_INST_BANK_RAM_DSM_IRRITATOR_DATA_SIZE
SQC_DSM_CNTLA_INST_BANK_RAM_ENABLE_SINGLE_WRITE_OFFSET
SQC_DSM_CNTLA_INST_BANK_RAM_ENABLE_SINGLE_WRITE_SIZE
SQC_DSM_CNTLA_INST_MISS_FIFO_DSM_IRRITATOR_DATA_OFFSET
SQC_DSM_CNTLA_INST_MISS_FIFO_DSM_IRRITATOR_DATA_SIZE
SQC_DSM_CNTLA_INST_MISS_FIFO_ENABLE_SINGLE_WRITE_OFFSET
SQC_DSM_CNTLA_INST_MISS_FIFO_ENABLE_SINGLE_WRITE_SIZE
SQC_DSM_CNTLA_INST_TAG_RAM_DSM_IRRITATOR_DATA_OFFSET
SQC_DSM_CNTLA_INST_TAG_RAM_DSM_IRRITATOR_DATA_SIZE
SQC_DSM_CNTLA_INST_TAG_RAM_ENABLE_SINGLE_WRITE_OFFSET
SQC_DSM_CNTLA_INST_TAG_RAM_ENABLE_SINGLE_WRITE_SIZE
SQC_DSM_CNTLA_INST_UTCL1_MISS_FIFO_DSM_IRRITATOR_DATA_OFFSET
SQC_DSM_CNTLA_INST_UTCL1_MISS_FIFO_DSM_IRRITATOR_DATA_SIZE
SQC_DSM_CNTLA_INST_UTCL1_MISS_FIFO_ENABLE_SINGLE_WRITE_OFFSET
SQC_DSM_CNTLA_INST_UTCL1_MISS_FIFO_ENABLE_SINGLE_WRITE_SIZE
SQC_DSM_CNTL2A_DATA_BANK_RAM_ENABLE_ERROR_INJECT_OFFSET
SQC_DSM_CNTL2A_DATA_BANK_RAM_ENABLE_ERROR_INJECT_SIZE
SQC_DSM_CNTL2A_DATA_BANK_RAM_SELECT_INJECT_DELAY_OFFSET
SQC_DSM_CNTL2A_DATA_BANK_RAM_SELECT_INJECT_DELAY_SIZE
SQC_DSM_CNTL2A_DATA_DIRTY_BIT_RAM_ENABLE_ERROR_INJECT_OFFSET
SQC_DSM_CNTL2A_DATA_DIRTY_BIT_RAM_ENABLE_ERROR_INJECT_SIZE
SQC_DSM_CNTL2A_DATA_DIRTY_BIT_RAM_SELECT_INJECT_DELAY_OFFSET
SQC_DSM_CNTL2A_DATA_DIRTY_BIT_RAM_SELECT_INJECT_DELAY_SIZE

autoreg_fields ::= (continued)

SQC_DSM_CNTL2A_DATA_HIT_FIFO_ENABLE_ERROR_INJECT_OFFSET
SQC_DSM_CNTL2A_DATA_HIT_FIFO_ENABLE_ERROR_INJECT_SIZE
SQC_DSM_CNTL2A_DATA_HIT_FIFO_SELECT_INJECT_DELAY_OFFSET
SQC_DSM_CNTL2A_DATA_HIT_FIFO_SELECT_INJECT_DELAY_SIZE
SQC_DSM_CNTL2A_DATA_MISS_FIFO_ENABLE_ERROR_INJECT_OFFSET
SQC_DSM_CNTL2A_DATA_MISS_FIFO_ENABLE_ERROR_INJECT_SIZE
SQC_DSM_CNTL2A_DATA_MISS_FIFO_SELECT_INJECT_DELAY_OFFSET
SQC_DSM_CNTL2A_DATA_MISS_FIFO_SELECT_INJECT_DELAY_SIZE
SQC_DSM_CNTL2A_DATA_TAG_RAM_ENABLE_ERROR_INJECT_OFFSET
SQC_DSM_CNTL2A_DATA_TAG_RAM_ENABLE_ERROR_INJECT_SIZE
SQC_DSM_CNTL2A_DATA_TAG_RAM_SELECT_INJECT_DELAY_OFFSET
SQC_DSM_CNTL2A_DATA_TAG_RAM_SELECT_INJECT_DELAY_SIZE
SQC_DSM_CNTL2A_INST_BANK_RAM_ENABLE_ERROR_INJECT_OFFSET
SQC_DSM_CNTL2A_INST_BANK_RAM_ENABLE_ERROR_INJECT_SIZE
SQC_DSM_CNTL2A_INST_BANK_RAM_SELECT_INJECT_DELAY_OFFSET
SQC_DSM_CNTL2A_INST_BANK_RAM_SELECT_INJECT_DELAY_SIZE
SQC_DSM_CNTL2A_INST_MISS_FIFO_ENABLE_ERROR_INJECT_OFFSET
SQC_DSM_CNTL2A_INST_MISS_FIFO_ENABLE_ERROR_INJECT_SIZE
SQC_DSM_CNTL2A_INST_MISS_FIFO_SELECT_INJECT_DELAY_OFFSET
SQC_DSM_CNTL2A_INST_MISS_FIFO_SELECT_INJECT_DELAY_SIZE
SQC_DSM_CNTL2A_INST_TAG_RAM_ENABLE_ERROR_INJECT_OFFSET
SQC_DSM_CNTL2A_INST_TAG_RAM_ENABLE_ERROR_INJECT_SIZE
SQC_DSM_CNTL2A_INST_TAG_RAM_SELECT_INJECT_DELAY_OFFSET
SQC_DSM_CNTL2A_INST_TAG_RAM_SELECT_INJECT_DELAY_SIZE
SQC_DSM_CNTL2A_INST_UTCL1_MISS_FIFO_ENABLE_ERROR_INJECT_OFFSET
SQC_DSM_CNTL2A_INST_UTCL1_MISS_FIFO_ENABLE_ERROR_INJECT_SIZE
SQC_DSM_CNTL2A_INST_UTCL1_MISS_FIFO_SELECT_INJECT_DELAY_OFFSET
SQC_DSM_CNTL2A_INST_UTCL1_MISS_FIFO_SELECT_INJECT_DELAY_SIZE
SQC_DSM_CNTLB_DATA_BANK_RAM_DSM_IRRITATOR_DATA_OFFSET
SQC_DSM_CNTLB_DATA_BANK_RAM_DSM_IRRITATOR_DATA_SIZE
SQC_DSM_CNTLB_DATA_BANK_RAM_ENABLE_SINGLE_WRITE_OFFSET
SQC_DSM_CNTLB_DATA_BANK_RAM_ENABLE_SINGLE_WRITE_SIZE
SQC_DSM_CNTLB_DATA_DIRTY_BIT_RAM_DSM_IRRITATOR_DATA_OFFSET
SQC_DSM_CNTLB_DATA_DIRTY_BIT_RAM_DSM_IRRITATOR_DATA_SIZE
SQC_DSM_CNTLB_DATA_DIRTY_BIT_RAM_ENABLE_SINGLE_WRITE_OFFSET
SQC_DSM_CNTLB_DATA_DIRTY_BIT_RAM_ENABLE_SINGLE_WRITE_SIZE
SQC_DSM_CNTLB_DATA_HIT_FIFO_DSM_IRRITATOR_DATA_OFFSET
SQC_DSM_CNTLB_DATA_HIT_FIFO_DSM_IRRITATOR_DATA_SIZE
SQC_DSM_CNTLB_DATA_HIT_FIFO_ENABLE_SINGLE_WRITE_OFFSET
SQC_DSM_CNTLB_DATA_HIT_FIFO_ENABLE_SINGLE_WRITE_SIZE
SQC_DSM_CNTLB_DATA_MISS_FIFO_DSM_IRRITATOR_DATA_OFFSET
SQC_DSM_CNTLB_DATA_MISS_FIFO_DSM_IRRITATOR_DATA_SIZE
SQC_DSM_CNTLB_DATA_MISS_FIFO_ENABLE_SINGLE_WRITE_OFFSET
SQC_DSM_CNTLB_DATA_MISS_FIFO_ENABLE_SINGLE_WRITE_SIZE
SQC_DSM_CNTLB_DATA_TAG_RAM_DSM_IRRITATOR_DATA_OFFSET
SQC_DSM_CNTLB_DATA_TAG_RAM_DSM_IRRITATOR_DATA_SIZE
SQC_DSM_CNTLB_DATA_TAG_RAM_ENABLE_SINGLE_WRITE_OFFSET
SQC_DSM_CNTLB_DATA_TAG_RAM_ENABLE_SINGLE_WRITE_SIZE

autoreg_fields : := (continued)

SQC_DSM_CNTLB_INST_BANK_RAM_DSM_IRRITATOR_DATA_OFFSET
SQC_DSM_CNTLB_INST_BANK_RAM_DSM_IRRITATOR_DATA_SIZE
SQC_DSM_CNTLB_INST_BANK_RAM_ENABLE_SINGLE_WRITE_OFFSET
SQC_DSM_CNTLB_INST_BANK_RAM_ENABLE_SINGLE_WRITE_SIZE
SQC_DSM_CNTLB_INST_MISS_FIFO_DSM_IRRITATOR_DATA_OFFSET
SQC_DSM_CNTLB_INST_MISS_FIFO_DSM_IRRITATOR_DATA_SIZE
SQC_DSM_CNTLB_INST_MISS_FIFO_ENABLE_SINGLE_WRITE_OFFSET
SQC_DSM_CNTLB_INST_MISS_FIFO_ENABLE_SINGLE_WRITE_SIZE
SQC_DSM_CNTLB_INST_TAG_RAM_DSM_IRRITATOR_DATA_OFFSET
SQC_DSM_CNTLB_INST_TAG_RAM_DSM_IRRITATOR_DATA_SIZE
SQC_DSM_CNTLB_INST_TAG_RAM_ENABLE_SINGLE_WRITE_OFFSET
SQC_DSM_CNTLB_INST_TAG_RAM_ENABLE_SINGLE_WRITE_SIZE
SQC_DSM_CNTLB_INST_UTCL1_MISS_FIFO_DSM_IRRITATOR_DATA_OFFSET
SQC_DSM_CNTLB_INST_UTCL1_MISS_FIFO_DSM_IRRITATOR_DATA_SIZE
SQC_DSM_CNTLB_INST_UTCL1_MISS_FIFO_ENABLE_SINGLE_WRITE_OFFSET
SQC_DSM_CNTLB_INST_UTCL1_MISS_FIFO_ENABLE_SINGLE_WRITE_SIZE
SQC_DSM_CNTL2B_DATA_BANK_RAM_ENABLE_ERROR_INJECT_OFFSET
SQC_DSM_CNTL2B_DATA_BANK_RAM_ENABLE_ERROR_INJECT_SIZE
SQC_DSM_CNTL2B_DATA_BANK_RAM_SELECT_INJECT_DELAY_OFFSET
SQC_DSM_CNTL2B_DATA_BANK_RAM_SELECT_INJECT_DELAY_SIZE
SQC_DSM_CNTL2B_DATA_DIRTY_BIT_RAM_ENABLE_ERROR_INJECT_OFFSET
SQC_DSM_CNTL2B_DATA_DIRTY_BIT_RAM_ENABLE_ERROR_INJECT_SIZE
SQC_DSM_CNTL2B_DATA_DIRTY_BIT_RAM_SELECT_INJECT_DELAY_OFFSET
SQC_DSM_CNTL2B_DATA_DIRTY_BIT_RAM_SELECT_INJECT_DELAY_SIZE
SQC_DSM_CNTL2B_DATA_HIT_FIFO_ENABLE_ERROR_INJECT_OFFSET
SQC_DSM_CNTL2B_DATA_HIT_FIFO_ENABLE_ERROR_INJECT_SIZE
SQC_DSM_CNTL2B_DATA_HIT_FIFO_SELECT_INJECT_DELAY_OFFSET
SQC_DSM_CNTL2B_DATA_HIT_FIFO_SELECT_INJECT_DELAY_SIZE
SQC_DSM_CNTL2B_DATA_MISS_FIFO_ENABLE_ERROR_INJECT_OFFSET
SQC_DSM_CNTL2B_DATA_MISS_FIFO_ENABLE_ERROR_INJECT_SIZE
SQC_DSM_CNTL2B_DATA_MISS_FIFO_SELECT_INJECT_DELAY_OFFSET
SQC_DSM_CNTL2B_DATA_MISS_FIFO_SELECT_INJECT_DELAY_SIZE
SQC_DSM_CNTL2B_DATA_TAG_RAM_ENABLE_ERROR_INJECT_OFFSET
SQC_DSM_CNTL2B_DATA_TAG_RAM_ENABLE_ERROR_INJECT_SIZE
SQC_DSM_CNTL2B_DATA_TAG_RAM_SELECT_INJECT_DELAY_OFFSET
SQC_DSM_CNTL2B_DATA_TAG_RAM_SELECT_INJECT_DELAY_SIZE
SQC_DSM_CNTL2B_INST_BANK_RAM_ENABLE_ERROR_INJECT_OFFSET
SQC_DSM_CNTL2B_INST_BANK_RAM_ENABLE_ERROR_INJECT_SIZE
SQC_DSM_CNTL2B_INST_BANK_RAM_SELECT_INJECT_DELAY_OFFSET
SQC_DSM_CNTL2B_INST_BANK_RAM_SELECT_INJECT_DELAY_SIZE
SQC_DSM_CNTL2B_INST_MISS_FIFO_ENABLE_ERROR_INJECT_OFFSET
SQC_DSM_CNTL2B_INST_MISS_FIFO_ENABLE_ERROR_INJECT_SIZE
SQC_DSM_CNTL2B_INST_MISS_FIFO_SELECT_INJECT_DELAY_OFFSET
SQC_DSM_CNTL2B_INST_MISS_FIFO_SELECT_INJECT_DELAY_SIZE
SQC_DSM_CNTL2B_INST_TAG_RAM_ENABLE_ERROR_INJECT_OFFSET
SQC_DSM_CNTL2B_INST_TAG_RAM_ENABLE_ERROR_INJECT_SIZE
SQC_DSM_CNTL2B_INST_TAG_RAM_SELECT_INJECT_DELAY_OFFSET
SQC_DSM_CNTL2B_INST_TAG_RAM_SELECT_INJECT_DELAY_SIZE

autoreg_fields : := (continued)

SQC_DSM_CNTL2B_INST_UTCL1_MISS_FIFO_ENABLE_ERROR_INJECT_OFFSET
SQC_DSM_CNTL2B_INST_UTCL1_MISS_FIFO_ENABLE_ERROR_INJECT_SIZE
SQC_DSM_CNTL2B_INST_UTCL1_MISS_FIFO_SELECT_INJECT_DELAY_OFFSET
SQC_DSM_CNTL2B_INST_UTCL1_MISS_FIFO_SELECT_INJECT_DELAY_SIZE
SQ_UTCL1_CNTL1_CLIENTID_OFFSET
SQ_UTCL1_CNTL1_CLIENTID_SIZE
: clientID of UTCL1.

SQ_UTCL1_CNTL1_ENABLE_LFIFO_PRI_ARB_OFFSET
SQ_UTCL1_CNTL1_ENABLE_LFIFO_PRI_ARB_SIZE
: Enable priority arbitration to latency (miss) FIFO instead of default round-robin. Not Applicable.

SQ_UTCL1_CNTL1_ENABLE_PUSH_LFIFO_OFFSET
SQ_UTCL1_CNTL1_ENABLE_PUSH_LFIFO_SIZE
: Enable hit requests that have outstanding requests to the same cacheline in the latency (miss) FIFO to be pushed into latency (miss) FIFO instead of hit fifo. Not Applicable.

SQ_UTCL1_CNTL1_FORCE_4K_L2_RESP_OFFSET
SQ_UTCL1_CNTL1_FORCE_4K_L2_RESP_SIZE
: When set to 1 forces the page size on the L2 response to be 4k. Note that if this is set, then the GPUVM_64K_DEFAULT cannot be set.

SQ_UTCL1_CNTL1_FORCE_IN_ORDER_OFFSET
SQ_UTCL1_CNTL1_FORCE_IN_ORDER_SIZE
: Force requests to return in order. Not Applicable.

SQ_UTCL1_CNTL1_FORCE_MISS_OFFSET
SQ_UTCL1_CNTL1_FORCE_MISS_SIZE
: Force all requests to miss in UTCL1 L1 and make requests to L2.

SQ_UTCL1_CNTL1_GPUVM_64K_DEF_OFFSET
SQ_UTCL1_CNTL1_GPUVM_64K_DEF_SIZE
: When set to 1 causes the UTCL1 to allocate gpuvm cachelines on a 64k boundary instead of 4k. Useful fo rclients that will always be working out of FB.

SQ_UTCL1_CNTL1_GPUVM_PERM_MODE_OFFSET
SQ_UTCL1_CNTL1_GPUVM_PERM_MODE_SIZE
: Control how gpuvm permission checks should be handled in the cache. 0: permissions are handled similarly to the ATC. 1: permissions are not checked to look for cache hit, only after translation returns;

SQ_UTCL1_CNTL1_REDUCE_CACHE_SIZE_BY_2_OFFSET
SQ_UTCL1_CNTL1_REDUCE_CACHE_SIZE_BY_2_SIZE
: Reduce cache line size by half.

SQ_UTCL1_CNTL1_REDUCE_FIFO_DEPTH_BY_2_OFFSET
SQ_UTCL1_CNTL1_REDUCE_FIFO_DEPTH_BY_2_SIZE
: Reduce latency FIFO depth by half.

autoreg_fields ::= (continued)

SQ_UTCL1_CNTL1_REG_INVALIDATE_ALL_OFFSET
SQ_UTCL1_CNTL1_REG_INVALIDATE_ALL_SIZE
: Invalidate all UTCL1 cache lines.

SQ_UTCL1_CNTL1_REG_INVALIDATE_ALL_VMID_OFFSET
SQ_UTCL1_CNTL1_REG_INVALIDATE_ALL_VMID_SIZE
: Use with REG_INV_TOGGLE 0: invalidate UTCL1 cache lines only matched with vmid; 1: invalidate all UTCL1 cache lines.

SQ_UTCL1_CNTL1_REG_INVALIDATE_TOGGLE_OFFSET
SQ_UTCL1_CNTL1_REG_INVALIDATE_TOGGLE_SIZE
: Register bit toggle from 0 to 1 to trigger invalidation to all UTCL1 cache lines or per vmid.

SQ_UTCL1_CNTL1_REG_INVALIDATE_VMID_OFFSET
SQ_UTCL1_CNTL1_REG_INVALIDATE_VMID_SIZE
: VMID for invalidation with REG_INV_TOGGLE.

SQ_UTCL1_CNTL1_RESP_FAULT_MODE_OFFSET
SQ_UTCL1_CNTL1_RESP_FAULT_MODE_SIZE
: Overrides error response types. 0: Error, 1: PRT, 2: Success, 3: Xnack

SQ_UTCL1_CNTL1_RESP_MODE_OFFSET
SQ_UTCL1_CNTL1_RESP_MODE_SIZE
: Overrides xnack(retry) response types. 0: Xnack, 1: PRT, 2: Error, 3: Success

SQ_UTCL1_CNTL1_USERVM_DIS_OFFSET
SQ_UTCL1_CNTL1_USERVM_DIS_SIZE
: 0: Enable uservm resource; 1: Disable uservm resource.

SQ_UTCL1_CNTL2_DIS_EDC_OFFSET
SQ_UTCL1_CNTL2_DIS_EDC_SIZE
: 0: Enable parity error check in cache line; 1: disable parity error check in cache line.

SQ_UTCL1_CNTL2_FORCE_FRAG_2M_TO_64K_OFFSET
SQ_UTCL1_CNTL2_FORCE_FRAG_2M_TO_64K_SIZE
SQ_UTCL1_CNTL2_FORCE_GPUVM_INV_ACK_OFFSET
SQ_UTCL1_CNTL2_FORCE_GPUVM_INV_ACK_SIZE
SQ_UTCL1_CNTL2_FORCE_SNOOP_OFFSET
SQ_UTCL1_CNTL2_FORCE_SNOOP_SIZE
SQ_UTCL1_CNTL2_GPUVM_INV_MODE_OFFSET
SQ_UTCL1_CNTL2_GPUVM_INV_MODE_SIZE
: 0: only invalidate the cacheline(s) if there are no outstanding requests; 1: always invalidate the cacheline(s).

SQ_UTCL1_CNTL2_LFIFO_SCAN_DISABLE_OFFSET
SQ_UTCL1_CNTL2_LFIFO_SCAN_DISABLE_SIZE
: Disable LFifo scan.

SQ_UTCL1_CNTL2_LINE_VALID_OFFSET
SQ_UTCL1_CNTL2_LINE_VALID_SIZE
: UTCL1 cache line valid status. 0: no cache line is valid in UTCL1; 1: at least 1 cache line is valid in UTCL1.

SQ_UTCL1_CNTL2_MTYPE_OVRD_DIS_OFFSET
SQ_UTCL1_CNTL2_MTYPE_OVRD_DIS_SIZE
: Disable override of original MTYPE.

SQ_UTCL1_CNTL2_PREFETCH_PAGE_OFFSET
SQ_UTCL1_CNTL2_PREFETCH_PAGE_SIZE
: Number of pages to prefetch for translation.

autoreg_fields : := (continued)

SQ_UTCL1_CNTL2_RETRY_TIMER_OFFSET
SQ_UTCL1_CNTL2_RETRY_TIMER_SIZE
: Number of 64-cycles to wait before retrying an xnacked request.

SQ_UTCL1_CNTL2_SHOOTDOWN_OPT_OFFSET
SQ_UTCL1_CNTL2_SHOOTDOWN_OPT_SIZE
SQ_UTCL1_CNTL2_SPARE_OFFSET
SQ_UTCL1_CNTL2_SPARE_SIZE
: .

SQ_UTCL1_STATUS_FAULT_DETECTED_OFFSET
SQ_UTCL1_STATUS_FAULT_DETECTED_SIZE
: Fault error detected. Write 1 to clear.

SQ_UTCL1_STATUS_PRT_DETECTED_OFFSET
SQ_UTCL1_STATUS_PRT_DETECTED_SIZE
: PRT fault detected. Write 1 to clear.

SQ_UTCL1_STATUS_RESERVED_OFFSET
SQ_UTCL1_STATUS_RESERVED_SIZE
: Reserved.

SQ_UTCL1_STATUS_RETRY_DETECTED_OFFSET
SQ_UTCL1_STATUS_RETRY_DETECTED_SIZE
: Xnack retry detected. Write 1 to clear.

SQ_UTCL1_STATUS_UNUSED_OFFSET
SQ_UTCL1_STATUS_UNUSED_SIZE
: Unused.

SQC_ICACHE_UTCL1_CNTL1_CLIENTID_OFFSET
SQC_ICACHE_UTCL1_CNTL1_CLIENTID_SIZE
SQC_ICACHE_UTCL1_CNTL1_CLIENT_INVALIDATE_ALL_VMID_OFFSET
SQC_ICACHE_UTCL1_CNTL1_CLIENT_INVALIDATE_ALL_VMID_SIZE
SQC_ICACHE_UTCL1_CNTL1_ENABLE_LFIFO_PRI_ARB_OFFSET
SQC_ICACHE_UTCL1_CNTL1_ENABLE_LFIFO_PRI_ARB_SIZE
SQC_ICACHE_UTCL1_CNTL1_ENABLE_PUSH_LFIFO_OFFSET
SQC_ICACHE_UTCL1_CNTL1_ENABLE_PUSH_LFIFO_SIZE
SQC_ICACHE_UTCL1_CNTL1_FORCE_4K_L2_RESP_OFFSET
SQC_ICACHE_UTCL1_CNTL1_FORCE_4K_L2_RESP_SIZE
SQC_ICACHE_UTCL1_CNTL1_FORCE_IN_ORDER_OFFSET
SQC_ICACHE_UTCL1_CNTL1_FORCE_IN_ORDER_SIZE
SQC_ICACHE_UTCL1_CNTL1_FORCE_MISS_OFFSET
SQC_ICACHE_UTCL1_CNTL1_FORCE_MISS_SIZE
SQC_ICACHE_UTCL1_CNTL1_GPUVM_64K_DEF_OFFSET
SQC_ICACHE_UTCL1_CNTL1_GPUVM_64K_DEF_SIZE
SQC_ICACHE_UTCL1_CNTL1_GPUVM_PERM_MODE_OFFSET
SQC_ICACHE_UTCL1_CNTL1_GPUVM_PERM_MODE_SIZE
SQC_ICACHE_UTCL1_CNTL1_REDUCE_CACHE_SIZE_BY_2_OFFSET
SQC_ICACHE_UTCL1_CNTL1_REDUCE_CACHE_SIZE_BY_2_SIZE
SQC_ICACHE_UTCL1_CNTL1_REDUCE_FIFO_DEPTH_BY_2_OFFSET
SQC_ICACHE_UTCL1_CNTL1_REDUCE_FIFO_DEPTH_BY_2_SIZE
SQC_ICACHE_UTCL1_CNTL1_REG_INVALIDATE_ALL_VMID_OFFSET
SQC_ICACHE_UTCL1_CNTL1_REG_INVALIDATE_ALL_VMID_SIZE

autoreg_fields : := (continued)

SQC_ICACHE_UTCL1_CNTL1_REG_INVALIDATE_TOGGLE_OFFSET
SQC_ICACHE_UTCL1_CNTL1_REG_INVALIDATE_TOGGLE_SIZE
SQC_ICACHE_UTCL1_CNTL1_REG_INVALIDATE_VMID_OFFSET
SQC_ICACHE_UTCL1_CNTL1_REG_INVALIDATE_VMID_SIZE
SQC_ICACHE_UTCL1_CNTL1_RESP_FAULT_MODE_OFFSET
SQC_ICACHE_UTCL1_CNTL1_RESP_FAULT_MODE_SIZE
SQC_ICACHE_UTCL1_CNTL1_RESP_MODE_OFFSET
SQC_ICACHE_UTCL1_CNTL1_RESP_MODE_SIZE
SQC_ICACHE_UTCL1_CNTL2_ARB_BURST_MODE_OFFSET
SQC_ICACHE_UTCL1_CNTL2_ARB_BURST_MODE_SIZE
: lfifo/hit_fifo arbiter burst mode.
SQC_ICACHE_UTCL1_CNTL2_DIS_EDC_OFFSET
SQC_ICACHE_UTCL1_CNTL2_DIS_EDC_SIZE
SQC_ICACHE_UTCL1_CNTL2_ENABLE_PERF_EVENT_RD_WR_OFFSET
SQC_ICACHE_UTCL1_CNTL2_ENABLE_PERF_EVENT_RD_WR_SIZE
: Enable read/write performance events filtering.
SQC_ICACHE_UTCL1_CNTL2_ENABLE_PERF_EVENT_VMID_OFFSET
SQC_ICACHE_UTCL1_CNTL2_ENABLE_PERF_EVENT_VMID_SIZE
: Enable VMID performance events filtering.
SQC_ICACHE_UTCL1_CNTL2_FORCE_FRAG_2M_TO_64K_OFFSET
SQC_ICACHE_UTCL1_CNTL2_FORCE_FRAG_2M_TO_64K_SIZE
: Force UTCL1 to cache 64K page size when UTCL2 return fragment size is 2M.
SQC_ICACHE_UTCL1_CNTL2_FORCE_GPUVM_INV_ACK_OFFSET
SQC_ICACHE_UTCL1_CNTL2_FORCE_GPUVM_INV_ACK_SIZE
SQC_ICACHE_UTCL1_CNTL2_FORCE_SNOOP_OFFSET
SQC_ICACHE_UTCL1_CNTL2_FORCE_SNOOP_SIZE
SQC_ICACHE_UTCL1_CNTL2_GPUVM_INV_MODE_OFFSET
SQC_ICACHE_UTCL1_CNTL2_GPUVM_INV_MODE_SIZE
SQC_ICACHE_UTCL1_CNTL2_LFIFO_SCAN_DISABLE_OFFSET
SQC_ICACHE_UTCL1_CNTL2_LFIFO_SCAN_DISABLE_SIZE
SQC_ICACHE_UTCL1_CNTL2_LINE_VALID_OFFSET
SQC_ICACHE_UTCL1_CNTL2_LINE_VALID_SIZE
SQC_ICACHE_UTCL1_CNTL2_MTYPE_OVRD_DIS_OFFSET
SQC_ICACHE_UTCL1_CNTL2_MTYPE_OVRD_DIS_SIZE
SQC_ICACHE_UTCL1_CNTL2_PERF_EVENT_RD_WR_OFFSET
SQC_ICACHE_UTCL1_CNTL2_PERF_EVENT_RD_WR_SIZE
: Read(0) or write(1) performance events filtering.
SQC_ICACHE_UTCL1_CNTL2_PERF_EVENT_VMID_OFFSET
SQC_ICACHE_UTCL1_CNTL2_PERF_EVENT_VMID_SIZE
: VMID for performance events filtering.
SQC_ICACHE_UTCL1_CNTL2_SHOOTDOWN_OPT_OFFSET
SQC_ICACHE_UTCL1_CNTL2_SHOOTDOWN_OPT_SIZE
SQC_ICACHE_UTCL1_CNTL2_SPARE_OFFSET
SQC_ICACHE_UTCL1_CNTL2_SPARE_SIZE
SQC_ICACHE_UTCL1_STATUS_FAULT_DETECTED_OFFSET
SQC_ICACHE_UTCL1_STATUS_FAULT_DETECTED_SIZE
: Fault error detected. Write 1 to clear.

autoreg_fields : := (continued)

SQC_ICACHE_UTCL1_STATUS_PRT_DETECTED_OFFSET
SQC_ICACHE_UTCL1_STATUS_PRT_DETECTED_SIZE
PRT fault detected. Write 1 to clear.
SQC_ICACHE_UTCL1_STATUS_RETRY_DETECTED_OFFSET
SQC_ICACHE_UTCL1_STATUS_RETRY_DETECTED_SIZE
Xnack retry detected. Write 1 to clear.
SQC_DCACHE_UTCL1_CNTL1_CLIENTID_OFFSET
SQC_DCACHE_UTCL1_CNTL1_CLIENTID_SIZE
SQC_DCACHE_UTCL1_CNTL1_CLIENT_INVALIDATE_ALL_VMID_OFFSET
SQC_DCACHE_UTCL1_CNTL1_CLIENT_INVALIDATE_ALL_VMID_SIZE
SQC_DCACHE_UTCL1_CNTL1_ENABLE_LFIFO_PRI_ARB_OFFSET
SQC_DCACHE_UTCL1_CNTL1_ENABLE_LFIFO_PRI_ARB_SIZE
SQC_DCACHE_UTCL1_CNTL1_ENABLE_PUSH_LFIFO_OFFSET
SQC_DCACHE_UTCL1_CNTL1_ENABLE_PUSH_LFIFO_SIZE
SQC_DCACHE_UTCL1_CNTL1_FORCE_4K_L2_RESP_OFFSET
SQC_DCACHE_UTCL1_CNTL1_FORCE_4K_L2_RESP_SIZE
SQC_DCACHE_UTCL1_CNTL1_FORCE_IN_ORDER_OFFSET
SQC_DCACHE_UTCL1_CNTL1_FORCE_IN_ORDER_SIZE
SQC_DCACHE_UTCL1_CNTL1_FORCE_MISS_OFFSET
SQC_DCACHE_UTCL1_CNTL1_FORCE_MISS_SIZE
SQC_DCACHE_UTCL1_CNTL1_GPUVM_64K_DEF_OFFSET
SQC_DCACHE_UTCL1_CNTL1_GPUVM_64K_DEF_SIZE
SQC_DCACHE_UTCL1_CNTL1_GPUVM_PERM_MODE_OFFSET
SQC_DCACHE_UTCL1_CNTL1_GPUVM_PERM_MODE_SIZE
SQC_DCACHE_UTCL1_CNTL1_REDUCE_CACHE_SIZE_BY_2_OFFSET
SQC_DCACHE_UTCL1_CNTL1_REDUCE_CACHE_SIZE_BY_2_SIZE
SQC_DCACHE_UTCL1_CNTL1_REDUCE_FIFO_DEPTH_BY_2_OFFSET
SQC_DCACHE_UTCL1_CNTL1_REDUCE_FIFO_DEPTH_BY_2_SIZE
SQC_DCACHE_UTCL1_CNTL1_REG_INVALIDATE_ALL_VMID_OFFSET
SQC_DCACHE_UTCL1_CNTL1_REG_INVALIDATE_ALL_VMID_SIZE
SQC_DCACHE_UTCL1_CNTL1_REG_INVALIDATE_TOGGLE_OFFSET
SQC_DCACHE_UTCL1_CNTL1_REG_INVALIDATE_TOGGLE_SIZE
SQC_DCACHE_UTCL1_CNTL1_REG_INVALIDATE_VMID_OFFSET
SQC_DCACHE_UTCL1_CNTL1_REG_INVALIDATE_VMID_SIZE
SQC_DCACHE_UTCL1_CNTL1_RESP_FAULT_MODE_OFFSET
SQC_DCACHE_UTCL1_CNTL1_RESP_FAULT_MODE_SIZE
SQC_DCACHE_UTCL1_CNTL1_RESP_MODE_OFFSET
SQC_DCACHE_UTCL1_CNTL1_RESP_MODE_SIZE
SQC_DCACHE_UTCL1_CNTL2_ARB_BURST_MODE_OFFSET
SQC_DCACHE_UTCL1_CNTL2_ARB_BURST_MODE_SIZE
lfifo/hit_fifo arbiter burst mode.
SQC_DCACHE_UTCL1_CNTL2_DIS_EDC_OFFSET
SQC_DCACHE_UTCL1_CNTL2_DIS_EDC_SIZE

autoreg_fields ::= (continued)

SQC_DCACHE_UTCL1_CNTL2_ENABLE_PERF_EVENT_RD_WR_OFFSET
SQC_DCACHE_UTCL1_CNTL2_ENABLE_PERF_EVENT_RD_WR_SIZE
: Enable read/write performance events filtering.

SQC_DCACHE_UTCL1_CNTL2_ENABLE_PERF_EVENT_VMID_OFFSET
SQC_DCACHE_UTCL1_CNTL2_ENABLE_PERF_EVENT_VMID_SIZE
: Enable VMID performance events filtering.

SQC_DCACHE_UTCL1_CNTL2_FORCE_FRAG_2M_TO_64K_OFFSET
SQC_DCACHE_UTCL1_CNTL2_FORCE_FRAG_2M_TO_64K_SIZE
: Force UTCL1 to cache 64K page size when UTCL2 return fragment size is 2M.

SQC_DCACHE_UTCL1_CNTL2_FORCE_GPUVM_INV_ACK_OFFSET
SQC_DCACHE_UTCL1_CNTL2_FORCE_GPUVM_INV_ACK_SIZE
SQC_DCACHE_UTCL1_CNTL2_FORCE_SNOOP_OFFSET
SQC_DCACHE_UTCL1_CNTL2_FORCE_SNOOP_SIZE
SQC_DCACHE_UTCL1_CNTL2_GPUVM_INV_MODE_OFFSET
SQC_DCACHE_UTCL1_CNTL2_GPUVM_INV_MODE_SIZE
SQC_DCACHE_UTCL1_CNTL2_LFIFO_SCAN_DISABLE_OFFSET
SQC_DCACHE_UTCL1_CNTL2_LFIFO_SCAN_DISABLE_SIZE
SQC_DCACHE_UTCL1_CNTL2_LINE_VALID_OFFSET
SQC_DCACHE_UTCL1_CNTL2_LINE_VALID_SIZE
SQC_DCACHE_UTCL1_CNTL2_MTYPE_OVRD_DIS_OFFSET
SQC_DCACHE_UTCL1_CNTL2_MTYPE_OVRD_DIS_SIZE
SQC_DCACHE_UTCL1_CNTL2_PERF_EVENT_RD_WR_OFFSET
SQC_DCACHE_UTCL1_CNTL2_PERF_EVENT_RD_WR_SIZE
: Read(0) or write(1) performance events filtering.

SQC_DCACHE_UTCL1_CNTL2_PERF_EVENT_VMID_OFFSET
SQC_DCACHE_UTCL1_CNTL2_PERF_EVENT_VMID_SIZE
: VMID for performance events filtering.

SQC_DCACHE_UTCL1_CNTL2_SHOOTDOWN_OPT_OFFSET
SQC_DCACHE_UTCL1_CNTL2_SHOOTDOWN_OPT_SIZE
SQC_DCACHE_UTCL1_CNTL2_SPARE_OFFSET
SQC_DCACHE_UTCL1_CNTL2_SPARE_SIZE
SQC_DCACHE_UTCL1_STATUS_FAULT_DETECTED_OFFSET
SQC_DCACHE_UTCL1_STATUS_FAULT_DETECTED_SIZE
: Fault error detected. Write 1 to clear.

SQC_DCACHE_UTCL1_STATUS_PRT_DETECTED_OFFSET
SQC_DCACHE_UTCL1_STATUS_PRT_DETECTED_SIZE
: PRT fault detected. Write 1 to clear.

SQC_DCACHE_UTCL1_STATUS_RETRY_DETECTED_OFFSET
SQC_DCACHE_UTCL1_STATUS_RETRY_DETECTED_SIZE
: Xnack retry detected. Write 1 to clear.

F Flags

The following flags are used to indicate special properties of certain instructions.

SEN_ATCPROBE

ATC probe instruction. This instruction is used to prefetch an ATC translation and does not directly perform a memory operation.

`s_atc_probe`, `s_atc_probe_buffer`

SEN_FLAT

FLAT instruction.

`flat_atomic_add`, `flat_atomic_add_x2`, `flat_atomic_and`,
`flat_atomic_and_x2`, `flat_atomic_cmpswap`, `flat_atomic_cmpswap_x2`,
`flat_atomic_dec`, `flat_atomic_dec_x2`, `flat_atomic_inc`,
`flat_atomic_inc_x2`, `flat_atomic_or`, `flat_atomic_or_x2`,
`flat_atomic_smax`, `flat_atomic_smax_x2`, `flat_atomic_smin`,
`flat_atomic_smin_x2`, `flat_atomic_sub`, `flat_atomic_sub_x2`,
`flat_atomic_swap`, `flat_atomic_swap_x2`, `flat_atomic_umax`,
`flat_atomic_umax_x2`, `flat_atomic_umin`, `flat_atomic_umin_x2`,
`flat_atomic_xor`, `flat_atomic_xor_x2`, `flat_load_dword`,
`flat_load_dwordx2`, `flat_load_dwordx3`, `flat_load_dwordx4`,
`flat_load_sbyte`, `flat_load_sbyte_d16`, `flat_load_sbyte_d16_hi`,
`flat_load_short_d16`, `flat_load_short_d16_hi`, `flat_load_sshort`,
`flat_load_ubyte`, `flat_load_ubyte_d16`, `flat_load_ubyte_d16_hi`,
`flat_load_ushort`, `flat_store_byte`, `flat_store_byte_d16_hi`,
`flat_store_dword`, `flat_store_dwordx2`, `flat_store_dwordx3`,
`flat_store_dwordx4`, `flat_store_short`, `flat_store_short_d16_hi`

SEN_GLOBAL

GLOBAL instruction.

`global_atomic_add`, `global_atomic_add_x2`, `global_atomic_and`,
`global_atomic_and_x2`, `global_atomic_cmpswap`, `global_atomic_cmpswap_x2`,
`global_atomic_dec`, `global_atomic_dec_x2`, `global_atomic_inc`,
`global_atomic_inc_x2`, `global_atomic_or`, `global_atomic_or_x2`,
`global_atomic_smax`, `global_atomic_smax_x2`, `global_atomic_smin`,
`global_atomic_smin_x2`, `global_atomic_sub`, `global_atomic_sub_x2`,
`global_atomic_swap`, `global_atomic_swap_x2`, `global_atomic_umax`,
`global_atomic_umax_x2`, `global_atomic_umin`, `global_atomic_umin_x2`,
`global_atomic_xor`, `global_atomic_xor_x2`, `global_load_dword`,
`global_load_dwordx2`, `global_load_dwordx3`, `global_load_dwordx4`,
`global_load_sbyte`, `global_load_sbyte_d16`, `global_load_sbyte_d16_hi`,
`global_load_short_d16`, `global_load_short_d16_hi`, `global_load_sshort`,
`global_load_ubyte`, `global_load_ubyte_d16`, `global_load_ubyte_d16_hi`,
`global_load_ushort`, `global_store_byte`, `global_store_byte_d16_hi`,
`global_store_dword`, `global_store_dwordx2`, `global_store_dwordx3`,

global_store_dwordx4, global_store_short, global_store_short_d16_hi

SEN_NODST

Instruction provides no destination operands.

s_cbranch_join, s_rfe_b64, s_rfe_restore_b64,
s_set_gpr_idx_idx, s_setpc_b64

SEN_NOOPR

Instruction has no operands at all.

buffer_wbinvl1, buffer_wbinvl1_vol, s_dcache_inv,
s_dcache_inv_vol, s_dcache_wb, s_dcache_wb_vol,
v_clrexcp, v_nop

SEN_NOSRC

Instruction takes no source operands.

s_barrier, s_endpgm, s_endpgm_ordered_ps_done,
s_endpgm_saved, s_getpc_b64, s_icache_inv,
s_set_gpr_idx_off, s_ttracedata, s_wakeup

SEN_SCRATCH

SCRATCH instruction.

scratch_load_dword, scratch_load_dwordx2, scratch_load_dwordx3,
scratch_load_dwordx4, scratch_load_sbyte, scratch_load_sbyte_d16,
scratch_load_sbyte_d16_hi, scratch_load_short_d16, scratch_load_short_d16_hi,
scratch_load_sshort, scratch_load_ubyte, scratch_load_ubyte_d16,
scratch_load_ubyte_d16_hi, scratch_load_ushort, scratch_store_byte,
scratch_store_byte_d16_hi, scratch_store_dword, scratch_store_dwordx2,
scratch_store_dwordx3, scratch_store_dwordx4, scratch_store_short,
scratch_store_short_d16_hi

SEN_VOP2

This instruction can only be used in the VOP3 encoding, but it does not require all 3 source VGPR operands.

v_add_f64, v_add_i16, v_add_i32,
v_ashrrev_i64, v_bcnt_u32_b32, v_bfm_b32,
v_cvt_pk_i16_i32, v_cvt_pk_u16_u32, v_cvt_pkaccum_u8_f32,
v_cvt_pknorm_i16_f16, v_cvt_pknorm_i16_f32, v_cvt_pknorm_u16_f16,
v_cvt_pknorm_u16_f32, v_cvt_pkrtz_f16_f32, v_ldexp_f16,
v_ldexp_f32, v_ldexp_f64, v_lshlrev_b64,
v_lshrrev_b64, v_max_f64, v_mbcnt_hi_u32_b32,

```
v_mbcnt_lo_u32_b32,  v_min_f64,           v_mul_f64,
v_mul_hi_i32,        v_mul_hi_u32,        v_mul_lo_u32,
v_pack_b32_f16,      v_pk_add_f16,        v_pk_add_i16,
v_pk_add_u16,        v_pk_ashrrev_i16,    v_pk_lshlrev_b16,
v_pk_lshrrev_b16,    v_pk_max_f16,        v_pk_max_i16,
v_pk_max_u16,        v_pk_min_f16,        v_pk_min_i16,
v_pk_min_u16,        v_pk_mul_f16,        v_pk_mul_lo_u16,
v_pk_sub_i16,        v_pk_sub_u16,        v_sub_i16,
v_sub_i32,           v_trig_preop_f64
```

ASIC_32BANK_LDS

This instruction is only available if ASIC has 32 bank lds.

`v_interp_p1ll_f16`

ASIC_DEEP_LEARNING

This instruction is only available if ASIC support deep learning primitiaves.

```
v_dot2_f32_f16,  v_dot2_i32_i16,    v_dot2_i32_i16_i8,
v_dot2_u32_u16,  v_dot2_u32_u16_u8, v_dot2c_f32_f16,
v_dot2c_i32_i16, v_dot4_i32_i8,     v_dot4_u32_u8,
v_dot4c_i32_i8,  v_dot8_i32_i4,     v_dot8_u32_u4,
v_dot8c_i32_i4,  v_fmac_f32,        v_pk_fmac_f16,
v_xnor_b32
```

ASIC_FED_INSTRUCTIONS

This instruction is only available if ASIC supports EDC and has MOV_FED instructions.

`s_mov_fed_b32`, `v_mov_fed_b32`

ASIC_LARGE_DS_READ

This instruction is only available if ASIC supports 96b and 128b DS_READ opcodes.

`ds_read_b128`, `ds_read_b96`

ASIC_LEGACY_LOG

This instruction is only available if ASIC has the EXP_LEGACY and LOG_LEGACY opcodes.

`v_exp_legacy_f32`, `v_log_legacy_f32`

OPF_ACNT

Image sample instruction uses a MIPID, LOD or CLAMP in the image address. Internally, ACNT needs to be incremented by 1 for this operation.

```
image_gather4_b_cl,     image_gather4_b_cl_a,     image_gather4_b_cl_o,
image_gather4_b_cl_o_a, image_gather4_c_b_cl,     image_gather4_c_b_cl_a,
image_gather4_c_b_cl_o, image_gather4_c_b_cl_o_a, image_gather4_c_cl,
image_gather4_c_cl_a,   image_gather4_c_cl_o,     image_gather4_c_cl_o_a,
image_gather4_c_l,      image_gather4_c_l_o,      image_gather4_cl,
image_gather4_cl_a,     image_gather4_cl_o,       image_gather4_cl_o_a,
image_gather4_l,        image_gather4_l_o,        image_get_resinfo,
image_load_mip,         image_load_mip_pck,       image_load_mip_pck_sgn,
image_sample_b_cl,      image_sample_b_cl_a,      image_sample_b_cl_o,
image_sample_b_cl_o_a,  image_sample_c_b_cl,      image_sample_c_b_cl_a,
image_sample_c_b_cl_o,  image_sample_c_b_cl_o_a,  image_sample_c_cd_cl,
image_sample_c_cd_cl_o, image_sample_c_cl,        image_sample_c_cl_a,
image_sample_c_cl_o,    image_sample_c_cl_o_a,    image_sample_c_d_cl,
image_sample_c_d_cl_o,  image_sample_c_l,         image_sample_c_l_o,
image_sample_cd_cl,     image_sample_cd_cl_o,     image_sample_cl,
image_sample_cl_a,      image_sample_cl_o,        image_sample_cl_o_a,
image_sample_d_cl,      image_sample_d_cl_o,      image_sample_l,
image_sample_l_o,       image_store_mip,          image_store_mip_pck
```

OPF_ALLOW_RTN_TO_LDS

This vector memory instruction is allowed to return data to LDS memory (instead of to VGPRs) by setting lds=1 in the microcode.

```
buffer_load_dword,  buffer_load_format_x, buffer_load_sbyte,
buffer_load_sshort, buffer_load_ubyte,    buffer_load_ushort
```

OPF_ATOMIC_CMPSWAP

Atomic instructions with CMPSWAP; marked because these need a different dmask value.

```
buffer_atomic_cmpswap,   buffer_atomic_cmpswap_x2,   flat_atomic_cmpswap,
flat_atomic_cmpswap_x2,  global_atomic_cmpswap,      global_atomic_cmpswap_x2,
image_atomic_cmpswap,    s_atomic_cmpswap,           s_atomic_cmpswap_x2,
s_buffer_atomic_cmpswap, s_buffer_atomic_cmpswap_x2
```

OPF_BIAS

Image sample instruction includes bias in the image address.

```
image_gather4_b,        image_gather4_b_a,        image_gather4_b_cl,
image_gather4_b_cl_a,   image_gather4_b_cl_o,     image_gather4_b_cl_o_a,
image_gather4_b_o,      image_gather4_b_o_a,      image_gather4_c_b,
image_gather4_c_b_a,    image_gather4_c_b_cl,     image_gather4_c_b_cl_a,
image_gather4_c_b_cl_o, image_gather4_c_b_cl_o_a, image_gather4_c_b_o,
image_gather4_c_b_o_a,  image_sample_b,           image_sample_b_a,
image_sample_b_cl,      image_sample_b_cl_a,      image_sample_b_cl_o,
image_sample_b_cl_o_a,  image_sample_b_o,         image_sample_b_o_a,
```

image_sample_c_b, image_sample_c_b_a, image_sample_c_b_cl,
image_sample_c_b_cl_a, image_sample_c_b_cl_o, image_sample_c_b_cl_o_a,
image_sample_c_b_o, image_sample_c_b_o_a

OPF_BREAK_ISTREAM

Signals an unconditional break in the instruction stream after this instruction. The shader either stops at this instruction or jumps elsewhere; it will NOT (in general) continue on to execute the next instruction in memory. Conditional branches must NOT have this flag set.

s_branch, s_call_b64, s_endpgm,
s_endpgm_ordered_ps_done, s_endpgm_saved, s_rfe_b64,
s_rfe_restore_b64, s_setpc_b64, s_swappc_b64

OPF_BUFCONST

Scalar memory operation uses a 128-bit resource buffer constant instead of a 64-bit address constant.

s_atc_probe_buffer, s_buffer_atomic_add, s_buffer_atomic_add_x2,
s_buffer_atomic_and, s_buffer_atomic_and_x2, s_buffer_atomic_cmpswap,
s_buffer_atomic_cmpswap_x2, s_buffer_atomic_dec, s_buffer_atomic_dec_x2,
s_buffer_atomic_inc, s_buffer_atomic_inc_x2, s_buffer_atomic_or,
s_buffer_atomic_or_x2, s_buffer_atomic_smax, s_buffer_atomic_smax_x2,
s_buffer_atomic_smin, s_buffer_atomic_smin_x2, s_buffer_atomic_sub,
s_buffer_atomic_sub_x2, s_buffer_atomic_swap, s_buffer_atomic_swap_x2,
s_buffer_atomic_umax, s_buffer_atomic_umax_x2, s_buffer_atomic_umin,
s_buffer_atomic_umin_x2, s_buffer_atomic_xor, s_buffer_atomic_xor_x2,
s_buffer_load_dword, s_buffer_load_dwordx16, s_buffer_load_dwordx2,
s_buffer_load_dwordx4, s_buffer_load_dwordx8, s_buffer_store_dword,
s_buffer_store_dwordx2, s_buffer_store_dwordx4

OPF_CACGRP0

Opcodes for CAC_VALU_GROUP0.

v_div_fmas_f64, v_fma_f64, v_mul_f64,
v_pk_fma_f16

OPF_CACGRP1

Opcodes for CAC_VALU_GROUP1.

v_add3_u32, v_add_f32, v_add_lshl_u32,
v_alignbit_b32, v_alignbyte_b32, v_div_fmas_f32,
v_fma_f32, v_interp_p1_f32, v_interp_p1ll_f16,
v_interp_p1lv_f16, v_interp_p2_f16, v_interp_p2_f32,
v_interp_p2_legacy_f16, v_lshl_add_u32, v_mac_f32,
v_mad_f32, v_mad_i32_i24, v_mad_i64_i32,
v_mad_legacy_f32, v_mad_mix_f32, v_mad_mixhi_f16,
v_mad_mixlo_f16, v_mad_u32_u24, v_mad_u64_u32,

v_madak_f32, v_madmk_f32, v_max3_f32,
v_max3_i32, v_max3_u32, v_med3_f32,
v_med3_i32, v_med3_u32, v_min3_f32,
v_min3_i32, v_min3_u32, v_mqsad_pk_u16_u8,
v_mqsad_u32_u8, v_msad_u8, v_mul_f32,
v_mul_hi_i32_i24, v_mul_hi_u32_u24, v_mul_i32_i24,
v_mul_legacy_f32, v_mul_u32_u24, v_pk_add_f16,
v_pk_mad_i16, v_pk_mad_u16, v_pk_mul_f16,
v_pk_mul_lo_u16, v_qsad_pk_u16_u8, v_sad_hi_u8,
v_sad_u16, v_sad_u32, v_sad_u8,
v_sub_f32, v_subrev_f32

OPF_CACGRP2

Opcodes for CAC_VALU_GROUP2.

v_add_co_u32, v_add_f64, v_add_i32,
v_add_u32, v_addc_co_u32, v_and_b32,
v_and_or_b32, v_ashrrev_i32, v_ashrrev_i64,
v_bcnt_u32_b32, v_bfe_i32, v_bfe_u32,
v_bfi_b32, v_bfm_b32, v_bfrev_b32,
v_ceil_f32, v_cndmask_b32, v_cos_f16,
v_cubeid_f32, v_cubema_f32, v_cubesc_f32,
v_cubetc_f32, v_cvt_f32_f16, v_cvt_f32_i32,
v_cvt_f32_u32, v_cvt_f32_ubyte0, v_cvt_f32_ubyte1,
v_cvt_f32_ubyte2, v_cvt_f32_ubyte3, v_cvt_flr_i32_f32,
v_cvt_i32_f32, v_cvt_off_f32_i4, v_cvt_pk_i16_i32,
v_cvt_pk_u16_u32, v_cvt_pk_u8_f32, v_cvt_pkaccum_u8_f32,
v_cvt_pknorm_i16_f16, v_cvt_pknorm_i16_f32, v_cvt_pknorm_u16_f16,
v_cvt_pknorm_u16_f32, v_cvt_pkrtz_f16_f32, v_cvt_rpi_i32_f32,
v_cvt_u32_f32, v_div_fixup_f32, v_div_fixup_f64,
v_div_scale_f32, v_div_scale_f64, v_exp_f16,
v_exp_f32, v_exp_legacy_f32, v_ffbh_i32,
v_ffbh_u32, v_ffbl_b32, v_floor_f32,
v_fma_f16, v_fma_legacy_f16, v_fract_f32,
v_fract_f64, v_ldexp_f32, v_ldexp_f64,
v_lerp_u8, v_log_f16, v_log_f32,
v_log_legacy_f32, v_lshl_or_b32, v_lshlrev_b32,
v_lshlrev_b64, v_lshrrev_b32, v_lshrrev_b64,
v_mac_f16, v_mad_f16, v_mad_i16,
v_mad_i32_i16, v_mad_legacy_f16, v_mad_legacy_i16,
v_mad_legacy_u16, v_mad_u16, v_mad_u32_u16,
v_madak_f16, v_madmk_f16, v_max3_f16,
v_max3_i16, v_max3_u16, v_max_f32,
v_max_f64, v_max_i32, v_max_u32,
v_mbcnt_hi_u32_b32, v_mbcnt_lo_u32_b32, v_med3_f16,
v_med3_i16, v_med3_u16, v_min3_f16,
v_min3_i16, v_min3_u16, v_min_f32,
v_min_f64, v_min_i32, v_min_u32,
v_mov_b32, v_mov_fed_b32, v_mul_hi_i32,
v_mul_hi_u32, v_mul_lo_u32, v_not_b32,
v_or3_b32, v_or_b32, v_pack_b32_f16,
v_perm_b32, v_pk_add_i16, v_pk_add_u16,

v_pk_ashrrev_i16, v_pk_lshlrev_b16, v_pk_lshrrev_b16,
v_pk_max_f16, v_pk_max_i16, v_pk_max_u16,
v_pk_min_f16, v_pk_min_i16, v_pk_min_u16,
v_pk_sub_i16, v_pk_sub_u16, v_rcp_f16,
v_rcp_f32, v_rcp_f64, v_rcp_iflag_f32,
v_rndne_f32, v_rsq_f16, v_rsq_f32,
v_rsq_f64, v_sin_f16, v_sin_f32,
v_sqrt_f16, v_sqrt_f32, v_sqrt_f64,
v_sub_co_u32, v_sub_i32, v_sub_u32,
v_subb_co_u32, v_subbrev_co_u32, v_subrev_co_u32,
v_subrev_u32, v_swap_b32, v_trunc_f32,
v_writelane_b32, v_xad_u32, v_xor_b32

OPF_CLAMP

Image sample instruction includes clamp data in the image address.

image_gather4_b_cl, image_gather4_b_cl_a, image_gather4_b_cl_o,
image_gather4_b_cl_o_a, image_gather4_c_b_cl, image_gather4_c_b_cl_a,
image_gather4_c_b_cl_o, image_gather4_c_b_cl_o_a, image_gather4_c_cl,
image_gather4_c_cl_a, image_gather4_c_cl_o, image_gather4_c_cl_o_a,
image_gather4_cl, image_gather4_cl_a, image_gather4_cl_o,
image_gather4_cl_o_a, image_sample_b_cl, image_sample_b_cl_a,
image_sample_b_cl_o, image_sample_b_cl_o_a, image_sample_c_b_cl,
image_sample_c_b_cl_a, image_sample_c_b_cl_o, image_sample_c_b_cl_o_a,
image_sample_c_cd_cl, image_sample_c_cd_cl_o, image_sample_c_cl,
image_sample_c_cl_a, image_sample_c_cl_o, image_sample_c_cl_o_a,
image_sample_c_d_cl, image_sample_c_d_cl_o, image_sample_cd_cl,
image_sample_cd_cl_o, image_sample_cl, image_sample_cl_a,
image_sample_cl_o, image_sample_cl_o_a, image_sample_d_cl,
image_sample_d_cl_o

OPF_COMP

Image sample instruction includes depth compare data in the image address.

image_gather4_c, image_gather4_c_a, image_gather4_c_b,
image_gather4_c_b_a, image_gather4_c_b_cl, image_gather4_c_b_cl_a,
image_gather4_c_b_cl_o, image_gather4_c_b_cl_o_a, image_gather4_c_b_o,
image_gather4_c_b_o_a, image_gather4_c_cl, image_gather4_c_cl_a,
image_gather4_c_cl_o, image_gather4_c_cl_o_a, image_gather4_c_l,
image_gather4_c_l_o, image_gather4_c_lz, image_gather4_c_lz_o,
image_gather4_c_o, image_gather4_c_o_a, image_sample_c,
image_sample_c_a, image_sample_c_b, image_sample_c_b_a,
image_sample_c_b_cl, image_sample_c_b_cl_a, image_sample_c_b_cl_o,
image_sample_c_b_cl_o_a, image_sample_c_b_o, image_sample_c_b_o_a,
image_sample_c_cd, image_sample_c_cd_cl, image_sample_c_cd_cl_o,
image_sample_c_cd_o, image_sample_c_cl, image_sample_c_cl_a,
image_sample_c_cl_o, image_sample_c_cl_o_a, image_sample_c_d,
image_sample_c_d_cl, image_sample_c_d_cl_o, image_sample_c_d_o,
image_sample_c_l, image_sample_c_l_o, image_sample_c_lz,
image_sample_c_lz_o, image_sample_c_o, image_sample_c_o_a

OPF_D16

Opcode is an MBUF formatted or a FLAT/SCRATCH/GLOBAL operation that asserts D16. The texture pipeline should return packed FP16 data instead of FP32 data.

buffer_load_format_d16_hi_x, buffer_load_format_d16_x, buffer_load_format_d16_xy,
buffer_load_format_d16_xyz, buffer_load_format_d16_xyzw, buffer_load_sbyte_d16,
buffer_load_sbyte_d16_hi, buffer_load_short_d16, buffer_load_short_d16_hi,
buffer_load_ubyte_d16, buffer_load_ubyte_d16_hi, buffer_store_byte_d16_hi,
buffer_store_format_d16_hi_x, buffer_store_format_d16_x, buffer_store_format_d16_xy,
buffer_store_format_d16_xyz, buffer_store_format_d16_xyzw, buffer_store_short_d16_hi,
flat_load_sbyte_d16, flat_load_sbyte_d16_hi, flat_load_short_d16,
flat_load_short_d16_hi, flat_load_ubyte_d16, flat_load_ubyte_d16_hi,
flat_store_byte_d16_hi, flat_store_short_d16_hi, global_load_sbyte_d16,
global_load_sbyte_d16_hi, global_load_short_d16, global_load_short_d16_hi,
global_load_ubyte_d16, global_load_ubyte_d16_hi, global_store_byte_d16_hi,
global_store_short_d16_hi, image_gather4, image_gather4_a,
image_gather4_b, image_gather4_b_a, image_gather4_b_cl,
image_gather4_b_cl_a, image_gather4_b_cl_o, image_gather4_b_cl_o_a,
image_gather4_b_o, image_gather4_b_o_a, image_gather4_c,
image_gather4_c_a, image_gather4_c_b, image_gather4_c_b_a,
image_gather4_c_b_cl, image_gather4_c_b_cl_a, image_gather4_c_b_cl_o,
image_gather4_c_b_cl_o_a, image_gather4_c_b_o, image_gather4_c_b_o_a,
image_gather4_c_cl, image_gather4_c_cl_a, image_gather4_c_cl_o,
image_gather4_c_cl_o_a, image_gather4_c_l, image_gather4_c_l_o,
image_gather4_c_lz, image_gather4_c_lz_o, image_gather4_c_o,
image_gather4_c_o_a, image_gather4_cl, image_gather4_cl_a,
image_gather4_cl_o, image_gather4_cl_o_a, image_gather4_l,
image_gather4_l_o, image_gather4_lz, image_gather4_lz_o,
image_gather4_o, image_gather4_o_a, image_gather4h,
image_load, image_load_mip, image_sample,
image_sample_a, image_sample_b, image_sample_b_a,
image_sample_b_cl, image_sample_b_cl_a, image_sample_b_cl_o,
image_sample_b_cl_o_a, image_sample_b_o, image_sample_b_o_a,
image_sample_c, image_sample_c_a, image_sample_c_b,
image_sample_c_b_a, image_sample_c_b_cl, image_sample_c_b_cl_a,
image_sample_c_b_cl_o, image_sample_c_b_cl_o_a, image_sample_c_b_o,
image_sample_c_b_o_a, image_sample_c_cd, image_sample_c_cd_cl,
image_sample_c_cd_cl_o, image_sample_c_cd_o, image_sample_c_cl,
image_sample_c_cl_a, image_sample_c_cl_o, image_sample_c_cl_o_a,
image_sample_c_d, image_sample_c_d_cl, image_sample_c_d_cl_o,
image_sample_c_d_o, image_sample_c_l, image_sample_c_l_o,
image_sample_c_lz, image_sample_c_lz_o, image_sample_c_o,
image_sample_c_o_a, image_sample_cd, image_sample_cd_cl,
image_sample_cd_cl_o, image_sample_cd_o, image_sample_cl,
image_sample_cl_a, image_sample_cl_o, image_sample_cl_o_a,
image_sample_d, image_sample_d_cl, image_sample_d_cl_o,
image_sample_d_o, image_sample_l, image_sample_l_o,
image_sample_lz, image_sample_lz_o, image_sample_o,
image_sample_o_a, image_store, image_store_mip,
scratch_load_sbyte_d16, scratch_load_sbyte_d16_hi, scratch_load_short_d16,
scratch_load_short_d16_hi, scratch_load_ubyte_d16, scratch_load_ubyte_d16_hi,
scratch_store_byte_d16_hi, scratch_store_short_d16_hi, tbuffer_load_format_d16_x,
tbuffer_load_format_d16_xy, tbuffer_load_format_d16_xyz, tbuffer_load_format_d16_xyzw,
tbuffer_store_format_d16_x, tbuffer_store_format_d16_xy, tbuffer_store_format_d16_xyz,

tbuffer_store_format_d16_xyzw

OPF_D16_CH_1

Opcode is an MBUF D16 formatted operation with one channel, i.e. _X.

buffer_load_format_d16_x, buffer_store_format_d16_x, tbuffer_load_format_d16_x, tbuffer_store_format_d16_x

OPF_D16_CH_3

Opcode is an MBUF D16 formatted operation with three channels, i.e. _XYZ.

buffer_load_format_d16_xyz, buffer_store_format_d16_xyz, tbuffer_load_format_d16_xyz, tbuffer_store_format_d16_xyz

OPF_DACCUM

All destinations are 'accumulate'-type operands – they contribute to the input of the instruction as well. Note that SDWA with DST_PRESERVE can also create this effect on instructions that do not normally have 'accumulate'-style destinations.

s_addk_i32, s_bitset0_b32, s_bitset0_b64,
s_bitset1_b32, s_bitset1_b64, s_cmov_b32,
s_cmov_b64, s_cmovk_i32, s_mulk_i32,
v_cvt_pkaccum_u8_f32, v_dot2c_f32_f16, v_dot2c_i32_i16,
v_dot4c_i32_i8, v_dot8c_i32_i4, v_fmac_f32,
v_interp_p2_f32, v_mac_f16, v_mac_f32,
v_pk_fmac_f16

OPF_DERIV

Image sample instruction includes derivative data in the image address.

image_sample_c_cd, image_sample_c_cd_cl, image_sample_c_cd_cl_o,
image_sample_c_cd_o, image_sample_c_d, image_sample_c_d_cl,
image_sample_c_d_cl_o, image_sample_c_d_o, image_sample_cd,
image_sample_cd_cl, image_sample_cd_cl_o, image_sample_cd_o,
image_sample_d, image_sample_d_cl, image_sample_d_cl_o,
image_sample_d_o

OPF_DPFP

DPFP (double precision floating point) instruction that should be disabled when dis_dpfp is set.

v_add_f64, v_div_fixup_f64, v_div_fmas_f64,
v_div_scale_f64, v_fma_f64, v_mul_f64,
v_rcp_f64, v_rsq_f64, v_sqrt_f64

OPF_DS0A

DS instruction takes no LDS address.

```
ds_gws_barrier, ds_gws_init,             ds_gws_sema_br,
ds_gws_sema_p,  ds_gws_sema_release_all, ds_gws_sema_v,
ds_nop,         ds_read_addtid_b32,      ds_write_addtid_b32
```

OPF_DS128BIT

DS instruction operates on 128 bits of data.

```
ds_read_b128, ds_write_b128
```

OPF_DS16BIT

DS instruction operates on 16 bits of data.

```
ds_read_i16,         ds_read_u16,  ds_read_u16_d16,
ds_read_u16_d16_hi, ds_write_b16, ds_write_b16_d16_hi
```

OPF_DS1A

DS instruction takes one LDS address operand and operates on one location in LDS memory.

```
ds_add_f32,             ds_add_rtn_f32,      ds_add_rtn_u32,
ds_add_rtn_u64,         ds_add_src2_f32,     ds_add_src2_u32,
ds_add_src2_u64,        ds_add_u32,          ds_add_u64,
ds_and_b32,             ds_and_b64,          ds_and_rtn_b32,
ds_and_rtn_b64,         ds_and_src2_b32,     ds_and_src2_b64,
ds_bpermute_b32,        ds_cmpst_b32,        ds_cmpst_b64,
ds_cmpst_f32,           ds_cmpst_f64,        ds_cmpst_rtn_b32,
ds_cmpst_rtn_b64,       ds_cmpst_rtn_f32,    ds_cmpst_rtn_f64,
ds_condxchg32_rtn_b64, ds_dec_rtn_u32,       ds_dec_rtn_u64,
ds_dec_src2_u32,        ds_dec_src2_u64,     ds_dec_u32,
ds_dec_u64,             ds_inc_rtn_u32,      ds_inc_rtn_u64,
ds_inc_src2_u32,        ds_inc_src2_u64,     ds_inc_u32,
ds_inc_u64,             ds_max_f32,          ds_max_f64,
ds_max_i32,             ds_max_i64,          ds_max_rtn_f32,
ds_max_rtn_f64,         ds_max_rtn_i32,      ds_max_rtn_i64,
ds_max_rtn_u32,         ds_max_rtn_u64,      ds_max_src2_f32,
ds_max_src2_f64,        ds_max_src2_i32,     ds_max_src2_i64,
ds_max_src2_u32,        ds_max_src2_u64,     ds_max_u32,
ds_max_u64,             ds_min_f32,          ds_min_f64,
ds_min_i32,             ds_min_i64,          ds_min_rtn_f32,
ds_min_rtn_f64,         ds_min_rtn_i32,      ds_min_rtn_i64,
ds_min_rtn_u32,         ds_min_rtn_u64,      ds_min_src2_f32,
ds_min_src2_f64,        ds_min_src2_i32,     ds_min_src2_i64,
ds_min_src2_u32,        ds_min_src2_u64,     ds_min_u32,
ds_min_u64,             ds_mskor_b32,        ds_mskor_b64,
ds_mskor_rtn_b32,       ds_mskor_rtn_b64,    ds_or_b32,
ds_or_b64,              ds_or_rtn_b32,       ds_or_rtn_b64,
```

```
ds_or_src2_b32,        ds_or_src2_b64,      ds_ordered_count,
ds_permute_b32,        ds_read_b128,        ds_read_b32,
ds_read_b64,           ds_read_b96,         ds_read_i16,
ds_read_i8,            ds_read_i8_d16,      ds_read_i8_d16_hi,
ds_read_u16,           ds_read_u16_d16,     ds_read_u16_d16_hi,
ds_read_u8,            ds_read_u8_d16,      ds_read_u8_d16_hi,
ds_rsub_rtn_u32,       ds_rsub_rtn_u64,     ds_rsub_src2_u32,
ds_rsub_src2_u64,      ds_rsub_u32,         ds_rsub_u64,
ds_sub_rtn_u32,        ds_sub_rtn_u64,      ds_sub_src2_u32,
ds_sub_src2_u64,       ds_sub_u32,          ds_sub_u64,
ds_swizzle_b32,        ds_wrap_rtn_b32,     ds_write_b128,
ds_write_b16,          ds_write_b16_d16_hi, ds_write_b32,
ds_write_b64,          ds_write_b8,         ds_write_b8_d16_hi,
ds_write_b96,          ds_write_src2_b32,   ds_write_src2_b64,
ds_wrxchg_rtn_b32,     ds_wrxchg_rtn_b64,   ds_xor_b32,
ds_xor_b64,            ds_xor_rtn_b32,      ds_xor_rtn_b64,
ds_xor_src2_b32,       ds_xor_src2_b64
```

OPF_DS1D

DS instruction takes one data value from a VGPR.

```
ds_add_f32,            ds_add_rtn_f32,      ds_add_rtn_u32,
ds_add_rtn_u64,        ds_add_u32,          ds_add_u64,
ds_and_b32,            ds_and_b64,          ds_and_rtn_b32,
ds_and_rtn_b64,        ds_bpermute_b32,     ds_condxchg32_rtn_b64,
ds_dec_rtn_u32,        ds_dec_rtn_u64,      ds_dec_u32,
ds_dec_u64,            ds_gws_barrier,      ds_gws_init,
ds_gws_sema_br,        ds_inc_rtn_u32,      ds_inc_rtn_u64,
ds_inc_u32,            ds_inc_u64,          ds_max_f32,
ds_max_f64,            ds_max_i32,          ds_max_i64,
ds_max_rtn_f32,        ds_max_rtn_f64,      ds_max_rtn_i32,
ds_max_rtn_i64,        ds_max_rtn_u32,      ds_max_rtn_u64,
ds_max_u32,            ds_max_u64,          ds_min_f32,
ds_min_f64,            ds_min_i32,          ds_min_i64,
ds_min_rtn_f32,        ds_min_rtn_f64,      ds_min_rtn_i32,
ds_min_rtn_i64,        ds_min_rtn_u32,      ds_min_rtn_u64,
ds_min_u32,            ds_min_u64,          ds_or_b32,
ds_or_b64,             ds_or_rtn_b32,       ds_or_rtn_b64,
ds_permute_b32,        ds_rsub_rtn_u32,     ds_rsub_rtn_u64,
ds_rsub_u32,           ds_rsub_u64,         ds_sub_rtn_u32,
ds_sub_rtn_u64,        ds_sub_u32,          ds_sub_u64,
ds_write_addtid_b32,   ds_write_b128,       ds_write_b16,
ds_write_b16_d16_hi,   ds_write_b32,        ds_write_b64,
ds_write_b8,           ds_write_b8_d16_hi,  ds_write_b96,
ds_wrxchg_rtn_b32,     ds_wrxchg_rtn_b64,   ds_xor_b32,
ds_xor_b64,            ds_xor_rtn_b32,      ds_xor_rtn_b64
```

OPF_DS2A

DS instruction takes one LDS address operand and generates a second LDS address value internally.

ds_read2_b32, ds_read2_b64, ds_read2st64_b32,
ds_read2st64_b64, ds_write2_b32, ds_write2_b64,
ds_write2st64_b32, ds_write2st64_b64, ds_wrxchg2_rtn_b32,
ds_wrxchg2_rtn_b64, ds_wrxchg2st64_rtn_b32, ds_wrxchg2st64_rtn_b64

OPF_DS2D

DS instruction takes two data values from VGPRs.

ds_cmpst_b32, ds_cmpst_b64, ds_cmpst_f32,
ds_cmpst_f64, ds_cmpst_rtn_b32, ds_cmpst_rtn_b64,
ds_cmpst_rtn_f32, ds_cmpst_rtn_f64, ds_mskor_b32,
ds_mskor_b64, ds_mskor_rtn_b32, ds_mskor_rtn_b64,
ds_wrap_rtn_b32, ds_write2_b32, ds_write2_b64,
ds_write2st64_b32, ds_write2st64_b64, ds_wrxchg2_rtn_b32,
ds_wrxchg2_rtn_b64, ds_wrxchg2st64_rtn_b32, ds_wrxchg2st64_rtn_b64

OPF_DS32FLT

DS instruction operates on 32 bits of floating-point data.

ds_add_f32, ds_add_rtn_f32, ds_add_src2_f32,
ds_cmpst_f32, ds_cmpst_rtn_f32, ds_max_f32,
ds_max_rtn_f32, ds_max_src2_f32, ds_min_f32,
ds_min_rtn_f32, ds_min_src2_f32

OPF_DS64BIT

DS instruction operates on 64 bits of data.

ds_add_rtn_u64, ds_add_src2_u64, ds_add_u64,
ds_and_b64, ds_and_rtn_b64, ds_and_src2_b64,
ds_cmpst_b64, ds_cmpst_rtn_b64, ds_condxchg32_rtn_b64,
ds_dec_rtn_u64, ds_dec_src2_u64, ds_dec_u64,
ds_inc_rtn_u64, ds_inc_src2_u64, ds_inc_u64,
ds_max_i64, ds_max_rtn_i64, ds_max_rtn_u64,
ds_max_src2_i64, ds_max_src2_u64, ds_max_u64,
ds_min_i64, ds_min_rtn_i64, ds_min_rtn_u64,
ds_min_src2_i64, ds_min_src2_u64, ds_min_u64,
ds_mskor_b64, ds_mskor_rtn_b64, ds_or_b64,
ds_or_rtn_b64, ds_or_src2_b64, ds_read2_b64,
ds_read2st64_b64, ds_read_b64, ds_rsub_rtn_u64,
ds_rsub_src2_u64, ds_rsub_u64, ds_sub_rtn_u64,
ds_sub_src2_u64, ds_sub_u64, ds_write2_b64,
ds_write2st64_b64, ds_write_b64, ds_write_src2_b64,
ds_wrxchg2_rtn_b64, ds_wrxchg2st64_rtn_b64, ds_wrxchg_rtn_b64,
ds_xor_b64, ds_xor_rtn_b64, ds_xor_src2_b64

OPF_DS64FLT

DS instruction operates on 64 bits of floating-point data.

ds_cmpst_f64, ds_cmpst_rtn_f64, ds_max_f64, ds_max_rtn_f64, ds_max_src2_f64, ds_min_f64, ds_min_rtn_f64, ds_min_src2_f64

OPF_DS8BIT

DS instruction operates on 8 bits of data.

ds_read_i8_d16, ds_read_i8_d16_hi, ds_read_u8, ds_read_u8_d16, ds_read_u8_d16_hi, ds_write_b8, ds_write_b8_d16_hi

OPF_DS96BIT

DS instruction operates on 96 bits of data.

ds_read_b96, ds_write_b96

OPF_DSRTN

DS instruction returns a value to save to a VGPR.

ds_add_rtn_f32, ds_add_rtn_u32, ds_add_rtn_u64, ds_and_rtn_b32, ds_and_rtn_b64, ds_append, ds_bpermute_b32, ds_cmpst_rtn_b32, ds_cmpst_rtn_b64, ds_cmpst_rtn_f32, ds_cmpst_rtn_f64, ds_condxchg32_rtn_b64, ds_consume, ds_dec_rtn_u32, ds_dec_rtn_u64, ds_inc_rtn_u32, ds_inc_rtn_u64, ds_max_rtn_f32, ds_max_rtn_f64, ds_max_rtn_i32, ds_max_rtn_i64, ds_max_rtn_u32, ds_max_rtn_u64, ds_min_rtn_f32, ds_min_rtn_f64, ds_min_rtn_i32, ds_min_rtn_i64, ds_min_rtn_u32, ds_min_rtn_u64, ds_mskor_rtn_b32, ds_mskor_rtn_b64, ds_or_rtn_b32, ds_or_rtn_b64, ds_ordered_count, ds_permute_b32, ds_read2_b32, ds_read2_b64, ds_read2st64_b32, ds_read2st64_b64, ds_read_addtid_b32, ds_read_b128, ds_read_b32, ds_read_b64, ds_read_b96, ds_read_i16, ds_read_i8, ds_read_i8_d16, ds_read_i8_d16_hi, ds_read_u16, ds_read_u16_d16, ds_read_u16_d16_hi, ds_read_u8, ds_read_u8_d16, ds_read_u8_d16_hi, ds_rsub_rtn_u32, ds_rsub_rtn_u64, ds_sub_rtn_u32, ds_sub_rtn_u64, ds_swizzle_b32, ds_wrap_rtn_b32, ds_wrxchg2_rtn_b32, ds_wrxchg2_rtn_b64, ds_wrxchg2st64_rtn_b32, ds_wrxchg2st64_rtn_b64, ds_wrxchg_rtn_b32, ds_wrxchg_rtn_b64, ds_xor_rtn_b32, ds_xor_rtn_b64

OPF_DS_ADDTID

DS opcode is an ADDTID opcode; these opcodes have additional constraints, notably wait state hazards.

`ds_read_addtid_b32`, `ds_write_addtid_b32`

OPF_FMAS

Opcode is an FMAS opcode (part of the division macro).

`v_div_fmas_f32`, `v_div_fmas_f64`

OPF_GATHER4

Image sample instruction is a GATHER4 operation with additional restrictions on how it may be used.

```
image_gather4,          image_gather4_a,        image_gather4_b,
image_gather4_b_a,      image_gather4_b_cl,     image_gather4_b_cl_a,
image_gather4_b_cl_o,   image_gather4_b_cl_o_a, image_gather4_b_o,
image_gather4_b_o_a,    image_gather4_c,        image_gather4_c_a,
image_gather4_c_b,      image_gather4_c_b_a,    image_gather4_c_b_cl,
image_gather4_c_b_cl_a, image_gather4_c_b_cl_o, image_gather4_c_b_cl_o_a,
image_gather4_c_b_o,    image_gather4_c_b_o_a,  image_gather4_c_cl,
image_gather4_c_cl_a,   image_gather4_c_cl_o,   image_gather4_c_cl_o_a,
image_gather4_c_l,      image_gather4_c_l_o,    image_gather4_c_lz,
image_gather4_c_lz_o,   image_gather4_c_o,      image_gather4_c_o_a,
image_gather4_cl,       image_gather4_cl_a,     image_gather4_cl_o,
image_gather4_cl_o_a,   image_gather4_l,        image_gather4_l_o,
image_gather4_lz,       image_gather4_lz_o,     image_gather4_o,
image_gather4_o_a,      image_gather4h,         image_gather4h_pck
```

OPF_GATHER8

Image sample instruction is a GATHER8 operation with additional restrictions on how it may be used.

`image_gather8h_pck`

OPF_GATHERH

Image sample instruction is a GATHER*H operation (gather-horizontal) with additional restrictions on how it may be used.

`image_gather4h`, `image_gather4h_pck`, `image_gather8h_pck`

OPF_GDSONLY

DS instruction is only valid for GDS unit; the instruction is not valid for LDS.

```
ds_gws_barrier,  ds_gws_init,             ds_gws_sema_br,
ds_gws_sema_p,   ds_gws_sema_release_all, ds_gws_sema_v,
ds_ordered_count
```

OPF_GRADADJUST

image_gather4_a, image_gather4_b_a, image_gather4_b_cl_a,
image_gather4_b_cl_o_a, image_gather4_b_o_a, image_gather4_c_a,
image_gather4_c_b_a, image_gather4_c_b_cl_a, image_gather4_c_b_cl_o_a,
image_gather4_c_b_o_a, image_gather4_c_cl_a, image_gather4_c_cl_o_a,
image_gather4_c_o_a, image_gather4_cl_a, image_gather4_cl_o_a,
image_gather4_o_a, image_sample_a, image_sample_b_a,
image_sample_b_cl_a, image_sample_b_cl_o_a, image_sample_b_o_a,
image_sample_c_a, image_sample_c_b_a, image_sample_c_b_cl_a,
image_sample_c_b_cl_o_a, image_sample_c_b_o_a, image_sample_c_cl_a,
image_sample_c_cl_o_a, image_sample_c_o_a, image_sample_cl_a,
image_sample_cl_o_a, image_sample_o_a

OPF_IMPLIED_LITERAL

Vector opcode has an implicit literal value and thus uses the SRC_LITERAL slot, though no operand position explicitly uses SRC_LITERAL.

v_madak_f16, v_madak_f32, v_madmk_f16,
v_madmk_f32

OPF_INTERNAL

Instruction is for internal use only, and should never appear in shader text.

OPF_INTERP

Instruction is an interpolation opcode; when used in VOP3 encoding, src0 and src1 must be swapped.

v_interp_mov_f32, v_interp_p1_f32, v_interp_p1ll_f16,
v_interp_p1lv_f16, v_interp_p2_f16, v_interp_p2_f32,
v_interp_p2_legacy_f16

OPF_LABEL

Instruction can branch to a program label verifiable at compile time (compare OPF_WRPC).

s_cbranch_i_fork

OPF_LOD

Image sample instruction includes LOD data in the image address.

image_gather4_c_l, image_gather4_c_l_o, image_gather4_l,
image_gather4_l_o, image_sample_c_l, image_sample_c_l_o,
image_sample_l, image_sample_l_o

OPF_LZ

Image sample instruction uses MIP level zero.

```
image_gather4_c_lz, image_gather4_c_lz_o, image_gather4_lz,
image_gather4_lz_o, image_sample_c_lz,    image_sample_c_lz_o,
image_sample_lz,    image_sample_lz_o
```

OPF_MEMFMT

M*BUF instruction uses a data format (for MTBUF, we use the dfmt field in the instruction; for MUBUF, we use the dfmt field in the resource).

```
buffer_load_format_d16_hi_x,  buffer_load_format_d16_x,     buffer_load_format_d16_xy,
buffer_load_format_d16_xyz,   buffer_load_format_d16_xyzw,  buffer_load_format_x,
buffer_load_format_xy,        buffer_load_format_xyz,       buffer_load_format_xyzw,
buffer_store_format_d16_hi_x, buffer_store_format_d16_x,    buffer_store_format_d16_xy,
buffer_store_format_d16_xyz,  buffer_store_format_d16_xyzw, buffer_store_format_x,
buffer_store_format_xy,       buffer_store_format_xyz,      buffer_store_format_xyzw,
tbuffer_load_format_d16_x,    tbuffer_load_format_d16_xy,   tbuffer_load_format_d16_xyz,
tbuffer_load_format_d16_xyzw, tbuffer_load_format_x,        tbuffer_load_format_xy,
tbuffer_load_format_xyz,      tbuffer_load_format_xyzw,     tbuffer_store_format_d16_x,
tbuffer_store_format_d16_xy,  tbuffer_store_format_d16_xyz, tbuffer_store_format_d16_xyzw,
tbuffer_store_format_x,       tbuffer_store_format_xy,      tbuffer_store_format_xyz,
tbuffer_store_format_xyzw
```

OPF_MEM_ATOMIC

Instruction performs an atomic operation on memory. This flag is mutually exclusive with OPF_MEM_STORE.

```
buffer_atomic_add,        buffer_atomic_add_x2,    buffer_atomic_and,
buffer_atomic_and_x2,     buffer_atomic_cmpswap,   buffer_atomic_cmpswap_x2,
buffer_atomic_dec,        buffer_atomic_dec_x2,    buffer_atomic_inc,
buffer_atomic_inc_x2,     buffer_atomic_or,        buffer_atomic_or_x2,
buffer_atomic_smax,       buffer_atomic_smax_x2,   buffer_atomic_smin,
buffer_atomic_smin_x2,    buffer_atomic_sub,       buffer_atomic_sub_x2,
buffer_atomic_swap,       buffer_atomic_swap_x2,   buffer_atomic_umax,
buffer_atomic_umax_x2,    buffer_atomic_umin,      buffer_atomic_umin_x2,
buffer_atomic_xor,        buffer_atomic_xor_x2,    ds_add_f32,
ds_add_rtn_f32,           ds_add_rtn_u32,          ds_add_rtn_u64,
ds_add_src2_f32,          ds_add_src2_u32,         ds_add_src2_u64,
ds_add_u32,               ds_add_u64,              ds_and_b32,
ds_and_b64,               ds_and_rtn_b32,          ds_and_rtn_b64,
ds_and_src2_b32,          ds_and_src2_b64,         ds_append,
ds_bpermute_b32,          ds_cmpst_b32,            ds_cmpst_b64,
ds_cmpst_f32,             ds_cmpst_f64,            ds_cmpst_rtn_b32,
ds_cmpst_rtn_b64,         ds_cmpst_rtn_f32,        ds_cmpst_rtn_f64,
ds_condxchg32_rtn_b64,    ds_consume,              ds_dec_rtn_u32,
ds_dec_rtn_u64,           ds_dec_src2_u32,         ds_dec_src2_u64,
ds_dec_u32,               ds_dec_u64,              ds_inc_rtn_u32,
ds_inc_rtn_u64,           ds_inc_src2_u32,         ds_inc_src2_u64,
ds_inc_u32,               ds_inc_u64,              ds_max_f32,
ds_max_f64,               ds_max_i32,              ds_max_i64,
```

ds_max_rtn_f32, ds_max_rtn_f64, ds_max_rtn_i32,
ds_max_rtn_i64, ds_max_rtn_u32, ds_max_rtn_u64,
ds_max_src2_f32, ds_max_src2_f64, ds_max_src2_i32,
ds_max_src2_i64, ds_max_src2_u32, ds_max_src2_u64,
ds_max_u32, ds_max_u64, ds_min_f32,
ds_min_f64, ds_min_i32, ds_min_i64,
ds_min_rtn_f32, ds_min_rtn_f64, ds_min_rtn_i32,
ds_min_rtn_i64, ds_min_rtn_u32, ds_min_rtn_u64,
ds_min_src2_f32, ds_min_src2_f64, ds_min_src2_i32,
ds_min_src2_i64, ds_min_src2_u32, ds_min_src2_u64,
ds_min_u32, ds_min_u64, ds_mskor_b32,
ds_mskor_b64, ds_mskor_rtn_b32, ds_mskor_rtn_b64,
ds_or_b32, ds_or_b64, ds_or_rtn_b32,
ds_or_rtn_b64, ds_or_src2_b32, ds_or_src2_b64,
ds_permute_b32, ds_rsub_rtn_u32, ds_rsub_rtn_u64,
ds_rsub_src2_u32, ds_rsub_src2_u64, ds_rsub_u32,
ds_rsub_u64, ds_sub_rtn_u32, ds_sub_rtn_u64,
ds_sub_src2_u32, ds_sub_src2_u64, ds_sub_u32,
ds_sub_u64, ds_swizzle_b32, ds_wrap_rtn_b32,
ds_write_src2_b32, ds_write_src2_b64, ds_wrxchg2_rtn_b32,
ds_wrxchg2_rtn_b64, ds_wrxchg2st64_rtn_b32, ds_wrxchg2st64_rtn_b64,
ds_wrxchg_rtn_b32, ds_wrxchg_rtn_b64, ds_xor_b32,
ds_xor_b64, ds_xor_rtn_b32, ds_xor_rtn_b64,
ds_xor_src2_b32, ds_xor_src2_b64, flat_atomic_add,
flat_atomic_add_x2, flat_atomic_and, flat_atomic_and_x2,
flat_atomic_cmpswap, flat_atomic_cmpswap_x2, flat_atomic_dec,
flat_atomic_dec_x2, flat_atomic_inc, flat_atomic_inc_x2,
flat_atomic_or, flat_atomic_or_x2, flat_atomic_smax,
flat_atomic_smax_x2, flat_atomic_smin, flat_atomic_smin_x2,
flat_atomic_sub, flat_atomic_sub_x2, flat_atomic_swap,
flat_atomic_swap_x2, flat_atomic_umax, flat_atomic_umax_x2,
flat_atomic_umin, flat_atomic_umin_x2, flat_atomic_xor,
flat_atomic_xor_x2, global_atomic_add, global_atomic_add_x2,
global_atomic_and, global_atomic_and_x2, global_atomic_cmpswap,
global_atomic_cmpswap_x2, global_atomic_dec, global_atomic_dec_x2,
global_atomic_inc, global_atomic_inc_x2, global_atomic_or,
global_atomic_or_x2, global_atomic_smax, global_atomic_smax_x2,
global_atomic_smin, global_atomic_smin_x2, global_atomic_sub,
global_atomic_sub_x2, global_atomic_swap, global_atomic_swap_x2,
global_atomic_umax, global_atomic_umax_x2, global_atomic_umin,
global_atomic_umin_x2, global_atomic_xor, global_atomic_xor_x2,
image_atomic_add, image_atomic_and, image_atomic_cmpswap,
image_atomic_dec, image_atomic_inc, image_atomic_or,
image_atomic_smax, image_atomic_smin, image_atomic_sub,
image_atomic_swap, image_atomic_umax, image_atomic_umin,
image_atomic_xor, s_atomic_add, s_atomic_add_x2,
s_atomic_and, s_atomic_and_x2, s_atomic_cmpswap,
s_atomic_cmpswap_x2, s_atomic_dec, s_atomic_dec_x2,
s_atomic_inc, s_atomic_inc_x2, s_atomic_or,
s_atomic_or_x2, s_atomic_smax, s_atomic_smax_x2,
s_atomic_smin, s_atomic_smin_x2, s_atomic_sub,
s_atomic_sub_x2, s_atomic_swap, s_atomic_swap_x2,
s_atomic_umax, s_atomic_umax_x2, s_atomic_umin,
s_atomic_umin_x2, s_atomic_xor, s_atomic_xor_x2,

```
s_buffer_atomic_add,      s_buffer_atomic_add_x2,  s_buffer_atomic_and,
s_buffer_atomic_and_x2,   s_buffer_atomic_cmpswap, s_buffer_atomic_cmpswap_x2,
s_buffer_atomic_dec,      s_buffer_atomic_dec_x2,  s_buffer_atomic_inc,
s_buffer_atomic_inc_x2,   s_buffer_atomic_or,      s_buffer_atomic_or_x2,
s_buffer_atomic_smax,     s_buffer_atomic_smax_x2, s_buffer_atomic_smin,
s_buffer_atomic_smin_x2,  s_buffer_atomic_sub,     s_buffer_atomic_sub_x2,
s_buffer_atomic_swap,     s_buffer_atomic_swap_x2, s_buffer_atomic_umax,
s_buffer_atomic_umax_x2,  s_buffer_atomic_umin,    s_buffer_atomic_umin_x2,
s_buffer_atomic_xor,      s_buffer_atomic_xor_x2
```

OPF_MEM_STORE

Instruction stores data into memory. This flag is mutually exclusive with OPF_MEM_ATOMIC.

```
buffer_store_byte,            buffer_store_byte_d16_hi,      buffer_store_dword,
buffer_store_dwordx2,         buffer_store_dwordx3,          buffer_store_dwordx4,
buffer_store_format_d16_hi_x, buffer_store_format_d16_x,     buffer_store_format_d16_xy,
buffer_store_format_d16_xyz,  buffer_store_format_d16_xyzw,  buffer_store_format_x,
buffer_store_format_xy,       buffer_store_format_xyz,       buffer_store_format_xyzw,
buffer_store_short,           buffer_store_short_d16_hi,     ds_write2_b32,
ds_write2_b64,                ds_write2st64_b32,             ds_write2st64_b64,
ds_write_addtid_b32,          ds_write_b128,                 ds_write_b16,
ds_write_b16_d16_hi,          ds_write_b32,                  ds_write_b64,
ds_write_b8,                  ds_write_b8_d16_hi,            ds_write_b96,
flat_store_byte,              flat_store_byte_d16_hi,        flat_store_dword,
flat_store_dwordx2,           flat_store_dwordx3,            flat_store_dwordx4,
flat_store_short,             flat_store_short_d16_hi,       global_store_byte,
global_store_byte_d16_hi,     global_store_dword,            global_store_dwordx2,
global_store_dwordx3,         global_store_dwordx4,          global_store_short,
global_store_short_d16_hi,    image_store,                   image_store_mip,
image_store_mip_pck,          image_store_pck,               s_buffer_store_dword,
s_buffer_store_dwordx2,       s_buffer_store_dwordx4,        s_scratch_store_dword,
s_scratch_store_dwordx2,      s_scratch_store_dwordx4,       s_store_dword,
s_store_dwordx2,              s_store_dwordx4,               scratch_store_byte,
scratch_store_byte_d16_hi,    scratch_store_dword,           scratch_store_dwordx2,
scratch_store_dwordx3,        scratch_store_dwordx4,         scratch_store_short,
scratch_store_short_d16_hi,   tbuffer_store_format_d16_x,    tbuffer_store_format_d16_xy,
tbuffer_store_format_d16_xyz, tbuffer_store_format_d16_xyzw, tbuffer_store_format_x,
tbuffer_store_format_xy,      tbuffer_store_format_xyz,      tbuffer_store_format_xyzw
```

OPF_MEM_STORE_LDS

Instruction stores data from LDS into memory. This flag is mutually exclusive with OPF_MEM_ATOMIC and OPF_MEM_STORE.

```
buffer_store_lds_dword
```

OPF_MIPID

Image sample instruction takes a user-supplied miplevel ID instead of computing it based on LOD.

image_get_resinfo, image_load_mip, image_load_mip_pck,
image_load_mip_pck_sgn, image_store_mip, image_store_mip_pck

OPF_MOV

Instruction unconditionally moves data from a source to a destination. Useful for dataflow analysis.

s_mov_b32, s_mov_b64, s_mov_fed_b32,
v_mov_b32, v_mov_fed_b32, v_swap_b32

OPF_MOVRELD

Instruction uses a relative GPR destination.

s_movreld_b32, s_movreld_b64

OPF_MOVRELS

Instruction uses a relative GPR source.

s_movrels_b32, s_movrels_b64

OPF_NODPP

Vector instruction cannot use DPP mode.

64-bit operations (I64, U64, DF64) cannot use DPP because of bank conflicts.

Operations already using a 64-bit instruction (e.g. MADAK, MADMK) can never use DPP.

READLANE variants are not implemented with DPP.

v_ceil_f64, v_clrexcp, v_cmp_class_f64,
v_cmp_eq_f64, v_cmp_eq_i64, v_cmp_eq_u64,
v_cmp_f_f64, v_cmp_f_i64, v_cmp_f_u64,
v_cmp_ge_f64, v_cmp_ge_i64, v_cmp_ge_u64,
v_cmp_gt_f64, v_cmp_gt_i64, v_cmp_gt_u64,
v_cmp_le_f64, v_cmp_le_i64, v_cmp_le_u64,
v_cmp_lg_f64, v_cmp_lt_f64, v_cmp_lt_i64,
v_cmp_lt_u64, v_cmp_ne_i64, v_cmp_ne_u64,
v_cmp_neq_f64, v_cmp_nge_f64, v_cmp_ngt_f64,
v_cmp_nle_f64, v_cmp_nlg_f64, v_cmp_nlt_f64,
v_cmp_o_f64, v_cmp_t_i64, v_cmp_t_u64,
v_cmp_tru_f64, v_cmp_u_f64, v_cmpx_class_f64,
v_cmpx_eq_f64, v_cmpx_eq_i64, v_cmpx_eq_u64,
v_cmpx_f_f64, v_cmpx_f_i64, v_cmpx_f_u64,
v_cmpx_ge_f64, v_cmpx_ge_i64, v_cmpx_ge_u64,
v_cmpx_gt_f64, v_cmpx_gt_i64, v_cmpx_gt_u64,
v_cmpx_le_f64, v_cmpx_le_i64, v_cmpx_le_u64,
v_cmpx_lg_f64, v_cmpx_lt_f64, v_cmpx_lt_i64,

```
v_cmpx_lt_u64,  v_cmpx_ne_i64,       v_cmpx_ne_u64,
v_cmpx_neq_f64, v_cmpx_nge_f64,      v_cmpx_ngt_f64,
v_cmpx_nle_f64, v_cmpx_nlg_f64,      v_cmpx_nlt_f64,
v_cmpx_o_f64,   v_cmpx_t_i64,        v_cmpx_t_u64,
v_cmpx_tru_f64, v_cmpx_u_f64,        v_cvt_f32_f64,
v_cvt_f64_f32,  v_cvt_f64_i32,       v_cvt_f64_u32,
v_cvt_i32_f64,  v_cvt_u32_f64,       v_floor_f64,
v_fract_f64,    v_frexp_exp_i32_f64, v_frexp_mant_f64,
v_madak_f16,    v_madak_f32,         v_madmk_f16,
v_madmk_f32,    v_rcp_f64,           v_readfirstlane_b32,
v_rndne_f64,    v_rsq_f64,           v_sqrt_f64,
v_swap_b32,     v_trunc_f64
```

OPF_NOSDWA

Vector instruction cannot use SDWA mode.

Operations already using a 64-bit instruction (e.g. MADAK, MADMK) can never use DPP.

READLANE variants are not implemented with DPP.

```
v_clrexcp,   v_mac_f16,           v_mac_f32,
v_madak_f16, v_madak_f32,         v_madmk_f16,
v_madmk_f32, v_readfirstlane_b32, v_swap_b32
```

OPF_NOVOP3

Vector instruction cannot be promoted to the VOP3 encoding; it must use the VOP2/VOP1/VOPC encoding. As a consequence these instructions may not use the input modifiers -x, abs(x), nor may they use the output modifiers mul, div, clamp.

```
v_madak_f16, v_madak_f32,         v_madmk_f16,
v_madmk_f32, v_readfirstlane_b32, v_swap_b32
```

OPF_OFFSET

Image sample instruction includes offset data in the image address.

```
image_gather4_b_cl_o,   image_gather4_b_cl_o_a,  image_gather4_b_o,
image_gather4_b_o_a,    image_gather4_c_b_cl_o,  image_gather4_c_b_cl_o_a,
image_gather4_c_b_o,    image_gather4_c_b_o_a,   image_gather4_c_cl_o,
image_gather4_c_cl_o_a, image_gather4_c_l_o,     image_gather4_c_lz_o,
image_gather4_c_o,      image_gather4_c_o_a,     image_gather4_cl_o,
image_gather4_cl_o_a,   image_gather4_l_o,       image_gather4_lz_o,
image_gather4_o,        image_gather4_o_a,       image_sample_b_cl_o,
image_sample_b_cl_o_a,  image_sample_b_o,        image_sample_b_o_a,
image_sample_c_b_cl_o,  image_sample_c_b_cl_o_a, image_sample_c_b_o,
image_sample_c_b_o_a,   image_sample_c_cd_cl_o,  image_sample_c_cd_o,
image_sample_c_cl_o,    image_sample_c_cl_o_a,   image_sample_c_d_cl_o,
image_sample_c_d_o,     image_sample_c_l_o,      image_sample_c_lz_o,
image_sample_c_o,       image_sample_c_o_a,      image_sample_cd_cl_o,
```

```
image_sample_cd_o,      image_sample_cl_o,      image_sample_cl_o_a,
image_sample_d_cl_o,    image_sample_d_o,       image_sample_l_o,
image_sample_lz_o,      image_sample_o,         image_sample_o_a
```

OPF_OPSEL

Vector instruction supports the setting of some or all of the OP_SEL field.

```
v_add_i16,              v_alignbit_b32,         v_alignbyte_b32,
v_cvt_pknorm_i16_f16,   v_cvt_pknorm_u16_f16,   v_div_fixup_f16,
v_div_fixup_legacy_f16, v_fma_f16,              v_fma_legacy_f16,
v_interp_p2_f16,        v_interp_p2_legacy_f16, v_mad_f16,
v_mad_i16,              v_mad_i32_i16,          v_mad_legacy_f16,
v_mad_legacy_i16,       v_mad_legacy_u16,       v_mad_u16,
v_mad_u32_u16,          v_max3_f16,             v_max3_i16,
v_max3_u16,             v_med3_f16,             v_med3_i16,
v_med3_u16,             v_min3_f16,             v_min3_i16,
v_min3_u16,             v_pack_b32_f16,         v_sub_i16
```

OPF_OPSEL_VOP3P

Vector instruction supports the setting of some or all of the OP_SEL field for VOP3P.

```
v_mad_mix_f32,    v_mad_mixhi_f16, v_mad_mixlo_f16,
v_pk_add_f16,     v_pk_add_i16,    v_pk_add_u16,
v_pk_ashrrev_i16, v_pk_fma_f16,    v_pk_lshlrev_b16,
v_pk_lshrrev_b16, v_pk_mad_i16,    v_pk_mad_u16,
v_pk_max_f16,     v_pk_max_i16,    v_pk_max_u16,
v_pk_min_f16,     v_pk_min_i16,    v_pk_min_u16,
v_pk_mul_f16,     v_pk_mul_lo_u16, v_pk_sub_i16,
v_pk_sub_u16
```

OPF_RDEX

Instruction implicitly reads EXEC (this does not include VOPs where EXEC is used solely to determine which threads are executing).

```
s_and_saveexec_b64,    s_andn1_saveexec_b64, s_andn1_wrexec_b64,
s_andn2_saveexec_b64, s_andn2_wrexec_b64,    s_cbranch_execnz,
s_cbranch_execz,       s_nand_saveexec_b64,  s_nor_saveexec_b64,
s_or_saveexec_b64,     s_orn1_saveexec_b64,  s_orn2_saveexec_b64,
s_xnor_saveexec_b64,   s_xor_saveexec_b64
```

OPF_RDFLAT

Instruction implicitly reads the FLAT_SCRATCH SGPR pair.

```
flat_atomic_add,            flat_atomic_add_x2,     flat_atomic_and,
flat_atomic_and_x2,         flat_atomic_cmpswap,    flat_atomic_cmpswap_x2,
flat_atomic_dec,            flat_atomic_dec_x2,     flat_atomic_inc,
```

flat_atomic_inc_x2, flat_atomic_or, flat_atomic_or_x2,
flat_atomic_smax, flat_atomic_smax_x2, flat_atomic_smin,
flat_atomic_smin_x2, flat_atomic_sub, flat_atomic_sub_x2,
flat_atomic_swap, flat_atomic_swap_x2, flat_atomic_umax,
flat_atomic_umax_x2, flat_atomic_umin, flat_atomic_umin_x2,
flat_atomic_xor, flat_atomic_xor_x2, flat_load_dword,
flat_load_dwordx2, flat_load_dwordx3, flat_load_dwordx4,
flat_load_sbyte, flat_load_sbyte_d16, flat_load_sbyte_d16_hi,
flat_load_short_d16, flat_load_short_d16_hi, flat_load_sshort,
flat_load_ubyte, flat_load_ubyte_d16, flat_load_ubyte_d16_hi,
flat_load_ushort, flat_store_byte, flat_store_byte_d16_hi,
flat_store_dword, flat_store_dwordx2, flat_store_dwordx3,
flat_store_dwordx4, flat_store_short, flat_store_short_d16_hi,
scratch_load_dword, scratch_load_dwordx2, scratch_load_dwordx3,
scratch_load_dwordx4, scratch_load_sbyte, scratch_load_sbyte_d16,
scratch_load_sbyte_d16_hi, scratch_load_short_d16, scratch_load_short_d16_hi,
scratch_load_sshort, scratch_load_ubyte, scratch_load_ubyte_d16,
scratch_load_ubyte_d16_hi, scratch_load_ushort, scratch_store_byte,
scratch_store_byte_d16_hi, scratch_store_dword, scratch_store_dwordx2,
scratch_store_dwordx3, scratch_store_dwordx4, scratch_store_short,
scratch_store_short_d16_hi

OPF_RDM0

Instruction implicitly reads M0.

ds_append, ds_consume, ds_gws_barrier,
ds_gws_init, ds_gws_sema_br, ds_gws_sema_p,
ds_gws_sema_release_all, ds_gws_sema_v, ds_ordered_count,
ds_read_addtid_b32, ds_write_addtid_b32, flat_atomic_add,
flat_atomic_add_x2, flat_atomic_and, flat_atomic_and_x2,
flat_atomic_cmpswap, flat_atomic_cmpswap_x2, flat_atomic_dec,
flat_atomic_dec_x2, flat_atomic_inc, flat_atomic_inc_x2,
flat_atomic_or, flat_atomic_or_x2, flat_atomic_smax,
flat_atomic_smax_x2, flat_atomic_smin, flat_atomic_smin_x2,
flat_atomic_sub, flat_atomic_sub_x2, flat_atomic_swap,
flat_atomic_swap_x2, flat_atomic_umax, flat_atomic_umax_x2,
flat_atomic_umin, flat_atomic_umin_x2, flat_atomic_xor,
flat_atomic_xor_x2, flat_load_dword, flat_load_dwordx2,
flat_load_dwordx3, flat_load_dwordx4, flat_load_sbyte,
flat_load_sbyte_d16, flat_load_sbyte_d16_hi, flat_load_short_d16,
flat_load_short_d16_hi, flat_load_sshort, flat_load_ubyte,
flat_load_ubyte_d16, flat_load_ubyte_d16_hi, flat_load_ushort,
flat_store_byte, flat_store_byte_d16_hi, flat_store_dword,
flat_store_dwordx2, flat_store_dwordx3, flat_store_dwordx4,
flat_store_short, flat_store_short_d16_hi, global_atomic_add,
global_atomic_add_x2, global_atomic_and, global_atomic_and_x2,
global_atomic_cmpswap, global_atomic_cmpswap_x2, global_atomic_dec,
global_atomic_dec_x2, global_atomic_inc, global_atomic_inc_x2,
global_atomic_or, global_atomic_or_x2, global_atomic_smax,
global_atomic_smax_x2, global_atomic_smin, global_atomic_smin_x2,
global_atomic_sub, global_atomic_sub_x2, global_atomic_swap,
global_atomic_swap_x2, global_atomic_umax, global_atomic_umax_x2,

```
global_atomic_umin,          global_atomic_umin_x2,     global_atomic_xor,
global_atomic_xor_x2,        global_load_dword,         global_load_dwordx2,
global_load_dwordx3,         global_load_dwordx4,       global_load_sbyte,
global_load_sbyte_d16,       global_load_sbyte_d16_hi,  global_load_short_d16,
global_load_short_d16_hi,    global_load_sshort,        global_load_ubyte,
global_load_ubyte_d16,       global_load_ubyte_d16_hi,  global_load_ushort,
global_store_byte,           global_store_byte_d16_hi,  global_store_dword,
global_store_dwordx2,        global_store_dwordx3,      global_store_dwordx4,
global_store_short,          global_store_short_d16_hi, s_movreld_b32,
s_movreld_b64,               s_movrels_b32,             s_movrels_b64,
s_sendmsg,                   s_sendmsghalt,             s_set_gpr_idx_idx,
s_set_gpr_idx_mode,          s_set_gpr_idx_on,          s_ttracedata,
scratch_load_dword,          scratch_load_dwordx2,      scratch_load_dwordx3,
scratch_load_dwordx4,        scratch_load_sbyte,        scratch_load_sbyte_d16,
scratch_load_sbyte_d16_hi,   scratch_load_short_d16,    scratch_load_short_d16_hi,
scratch_load_sshort,         scratch_load_ubyte,        scratch_load_ubyte_d16,
scratch_load_ubyte_d16_hi,   scratch_load_ushort,       scratch_store_byte,
scratch_store_byte_d16_hi,   scratch_store_dword,       scratch_store_dwordx2,
scratch_store_dwordx3,       scratch_store_dwordx4,     scratch_store_short,
scratch_store_short_d16_hi, v_interp_mov_f32,          v_interp_p1_f32,
v_interp_p2_f32
```

OPF_RDPC

Instruction implicitly reads PC (not including for the fetch of this instruction).

`s_call_b64`, `s_getpc_b64`, `s_swappc_b64`

OPF_RDSCC

Instruction implicitly reads SCC.

```
s_addc_u32,     s_cbranch_scc0, s_cbranch_scc1,
s_cmov_b32,     s_cmov_b64,     s_cmovk_i32,
s_cselect_b32, s_cselect_b64,  s_subb_u32
```

OPF_RDVCC

Instruction implicitly reads VCC. The instruction always reads VCC; this flag should not be used for carry-in cases where a different SGPR pair may be named depending on instruction encoding. Compare OPF_VCCS.

`s_cbranch_vccnz`, `s_cbranch_vccz`, `v_div_fmas_f32`,
`v_div_fmas_f64`

OPF_S0_I4

`v_dot8_i32_i4`

OPF_S0_I8

v_dot2_i32_i16_i8, v_dot4_i32_i8, v_dot4c_i32_i8, v_dot8c_i32_i4

OPF_S0_U4

v_dot8_u32_u4

OPF_S0_U8

v_dot2_u32_u16_u8, v_dot4_u32_u8

OPF_S1_I4

v_dot8_i32_i4

OPF_S1_I8

v_dot4_i32_i8, v_dot4c_i32_i8, v_dot8c_i32_i4

OPF_S1_U4

v_dot8_u32_u4

OPF_S1_U8

v_dot4_u32_u8

OPF_SAMPLE

Image instruction uses a sampler.

image_gather4,	image_gather4_a,	image_gather4_b,
image_gather4_b_a,	image_gather4_b_cl,	image_gather4_b_cl_a,
image_gather4_b_cl_o,	image_gather4_b_cl_o_a,	image_gather4_b_o,
image_gather4_b_o_a,	image_gather4_c,	image_gather4_c_a,
image_gather4_c_b,	image_gather4_c_b_a,	image_gather4_c_b_cl,
image_gather4_c_b_cl_a,	image_gather4_c_b_cl_o,	image_gather4_c_b_cl_o_a,
image_gather4_c_b_o,	image_gather4_c_b_o_a,	image_gather4_c_cl,
image_gather4_c_cl_a,	image_gather4_c_cl_o,	image_gather4_c_cl_o_a,
image_gather4_c_l,	image_gather4_c_l_o,	image_gather4_c_lz,
image_gather4_c_lz_o,	image_gather4_c_o,	image_gather4_c_o_a,
image_gather4_cl,	image_gather4_cl_a,	image_gather4_cl_o,
image_gather4_cl_o_a,	image_gather4_l,	image_gather4_l_o,
image_gather4_lz,	image_gather4_lz_o,	image_gather4_o,

```
image_gather4_o_a,      image_gather4h,          image_gather4h_pck,
image_gather8h_pck,     image_get_lod,           image_sample,
image_sample_a,         image_sample_b,          image_sample_b_a,
image_sample_b_cl,      image_sample_b_cl_a,     image_sample_b_cl_o,
image_sample_b_cl_o_a,  image_sample_b_o,        image_sample_b_o_a,
image_sample_c,         image_sample_c_a,        image_sample_c_b,
image_sample_c_b_a,     image_sample_c_b_cl,     image_sample_c_b_cl_a,
image_sample_c_b_cl_o,  image_sample_c_b_cl_o_a, image_sample_c_b_o,
image_sample_c_b_o_a,   image_sample_c_cd,       image_sample_c_cd_cl,
image_sample_c_cd_cl_o, image_sample_c_cd_o,     image_sample_c_cl,
image_sample_c_cl_a,    image_sample_c_cl_o,     image_sample_c_cl_o_a,
image_sample_c_d,       image_sample_c_d_cl,     image_sample_c_d_cl_o,
image_sample_c_d_o,     image_sample_c_l,        image_sample_c_l_o,
image_sample_c_lz,      image_sample_c_lz_o,     image_sample_c_o,
image_sample_c_o_a,     image_sample_cd,         image_sample_cd_cl,
image_sample_cd_cl_o,   image_sample_cd_o,       image_sample_cl,
image_sample_cl_a,      image_sample_cl_o,       image_sample_cl_o_a,
image_sample_d,         image_sample_d_cl,       image_sample_d_cl_o,
image_sample_d_o,       image_sample_l,          image_sample_l_o,
image_sample_lz,        image_sample_lz_o,       image_sample_o,
image_sample_o_a
```

OPF_SDST

```
v_cmpx_class_f16, v_cmpx_class_f32, v_cmpx_class_f64,
v_cmpx_eq_f16,    v_cmpx_eq_f32,    v_cmpx_eq_f64,
v_cmpx_eq_i16,    v_cmpx_eq_i32,    v_cmpx_eq_i64,
v_cmpx_eq_u16,    v_cmpx_eq_u32,    v_cmpx_eq_u64,
v_cmpx_f_f16,     v_cmpx_f_f32,     v_cmpx_f_f64,
v_cmpx_f_i16,     v_cmpx_f_i32,     v_cmpx_f_i64,
v_cmpx_f_u16,     v_cmpx_f_u32,     v_cmpx_f_u64,
v_cmpx_ge_f16,    v_cmpx_ge_f32,    v_cmpx_ge_f64,
v_cmpx_ge_i16,    v_cmpx_ge_i32,    v_cmpx_ge_i64,
v_cmpx_ge_u16,    v_cmpx_ge_u32,    v_cmpx_ge_u64,
v_cmpx_gt_f16,    v_cmpx_gt_f32,    v_cmpx_gt_f64,
v_cmpx_gt_i16,    v_cmpx_gt_i32,    v_cmpx_gt_i64,
v_cmpx_gt_u16,    v_cmpx_gt_u32,    v_cmpx_gt_u64,
v_cmpx_le_f16,    v_cmpx_le_f32,    v_cmpx_le_f64,
v_cmpx_le_i16,    v_cmpx_le_i32,    v_cmpx_le_i64,
v_cmpx_le_u16,    v_cmpx_le_u32,    v_cmpx_le_u64,
v_cmpx_lg_f16,    v_cmpx_lg_f32,    v_cmpx_lg_f64,
v_cmpx_lt_f16,    v_cmpx_lt_f32,    v_cmpx_lt_f64,
v_cmpx_lt_i16,    v_cmpx_lt_i32,    v_cmpx_lt_i64,
v_cmpx_lt_u16,    v_cmpx_lt_u32,    v_cmpx_lt_u64,
v_cmpx_ne_i16,    v_cmpx_ne_i32,    v_cmpx_ne_i64,
v_cmpx_ne_u16,    v_cmpx_ne_u32,    v_cmpx_ne_u64,
v_cmpx_neq_f16,   v_cmpx_neq_f32,   v_cmpx_neq_f64,
v_cmpx_nge_f16,   v_cmpx_nge_f32,   v_cmpx_nge_f64,
v_cmpx_ngt_f16,   v_cmpx_ngt_f32,   v_cmpx_ngt_f64,
v_cmpx_nle_f16,   v_cmpx_nle_f32,   v_cmpx_nle_f64,
v_cmpx_nlg_f16,   v_cmpx_nlg_f32,   v_cmpx_nlg_f64,
v_cmpx_nlt_f16,   v_cmpx_nlt_f32,   v_cmpx_nlt_f64,
v_cmpx_o_f16,     v_cmpx_o_f32,     v_cmpx_o_f64,
```

```
v_cmpx_t_i16,     v_cmpx_t_i32,     v_cmpx_t_i64,
v_cmpx_t_u16,     v_cmpx_t_u32,     v_cmpx_t_u64,
v_cmpx_tru_f16,   v_cmpx_tru_f32,   v_cmpx_tru_f64,
v_cmpx_u_f16,     v_cmpx_u_f32,     v_cmpx_u_f64
```

OPF_SMEM_SCRATCH

Scalar memory instruction uses scratch-style addressing.

```
s_scratch_load_dword,  s_scratch_load_dwordx2,  s_scratch_load_dwordx4,
s_scratch_store_dword, s_scratch_store_dwordx2, s_scratch_store_dwordx4
```

OPF_SQXLATE

SQ performs a translation on this vector opcode before sending it to SP.

```
v_ashrrev_i16,        v_ashrrev_i32,    v_ashrrev_i64,
v_cvt_pkaccum_u8_f32, v_lshlrev_b16,    v_lshlrev_b32,
v_lshlrev_b64,        v_lshrrev_b16,    v_lshrrev_b32,
v_lshrrev_b64,        v_mac_f16,        v_mac_f32,
v_madak_f16,          v_madak_f32,      v_madmk_f16,
v_madmk_f32,          v_pk_ashrrev_i16, v_pk_lshlrev_b16,
v_pk_lshrrev_b16,     v_sub_f16,        v_sub_f32,
v_subbrev_co_u32,     v_subrev_co_u32,  v_subrev_f16,
v_subrev_f32,         v_subrev_u16,     v_subrev_u32,
v_writelane_b32
```

OPF_TRANS

Vector instruction implemented via a lookup table and polynomial expansion.

```
v_cos_f16,  v_cos_f32,        v_exp_f16,
v_exp_f32,  v_exp_legacy_f32, v_log_f16,
v_log_f32,  v_log_legacy_f32, v_rcp_f16,
v_rcp_f32,  v_rcp_f64,        v_rcp_iflag_f32,
v_rsq_f16,  v_rsq_f32,        v_rsq_f64,
v_sin_f16,  v_sin_f32,        v_sqrt_f16,
v_sqrt_f32, v_sqrt_f64
```

OPF_VCCD

Vector instruction has a scalar carry-out in addition to its normal vector output. Depending on the encoding, this carry-out may be implicitly written to VCC or may be explicitly written to an SGPR pair. Compare OPF_VCCS.

```
v_add_co_u32,     v_addc_co_u32, v_div_scale_f32,
v_div_scale_f64, v_mad_i64_i32, v_mad_u64_u32,
v_sub_co_u32,     v_subb_co_u32, v_subbrev_co_u32,
v_subrev_co_u32
```

OPF_VCCS

Vector instruction has a scalar carry-in. Depending on the encoding, this carry-in may be implicitly read from VCC or may be explicitly read from an SGPR pair. Compare OPF_VCCD and OPF_RDVCC.

v_addc_co_u32, v_cndmask_b32, v_subb_co_u32,
v_subbrev_co_u32

OPF_WREX

Instruction implicitly writes EXEC.

s_and_saveexec_b64, s_andn1_saveexec_b64, s_andn1_wrexec_b64,
s_andn2_saveexec_b64, s_andn2_wrexec_b64, s_cbranch_join,
s_nand_saveexec_b64, s_nor_saveexec_b64, s_or_saveexec_b64,
s_orn1_saveexec_b64, s_orn2_saveexec_b64, s_xnor_saveexec_b64,
s_xor_saveexec_b64, v_cmpx_class_f16, v_cmpx_class_f32,
v_cmpx_class_f64, v_cmpx_eq_f16, v_cmpx_eq_f32,
v_cmpx_eq_f64, v_cmpx_eq_i16, v_cmpx_eq_i32,
v_cmpx_eq_i64, v_cmpx_eq_u16, v_cmpx_eq_u32,
v_cmpx_eq_u64, v_cmpx_f_f16, v_cmpx_f_f32,
v_cmpx_f_f64, v_cmpx_f_i16, v_cmpx_f_i32,
v_cmpx_f_i64, v_cmpx_f_u16, v_cmpx_f_u32,
v_cmpx_f_u64, v_cmpx_ge_f16, v_cmpx_ge_f32,
v_cmpx_ge_f64, v_cmpx_ge_i16, v_cmpx_ge_i32,
v_cmpx_ge_i64, v_cmpx_ge_u16, v_cmpx_ge_u32,
v_cmpx_ge_u64, v_cmpx_gt_f16, v_cmpx_gt_f32,
v_cmpx_gt_f64, v_cmpx_gt_i16, v_cmpx_gt_i32,
v_cmpx_gt_i64, v_cmpx_gt_u16, v_cmpx_gt_u32,
v_cmpx_gt_u64, v_cmpx_le_f16, v_cmpx_le_f32,
v_cmpx_le_f64, v_cmpx_le_i16, v_cmpx_le_i32,
v_cmpx_le_i64, v_cmpx_le_u16, v_cmpx_le_u32,
v_cmpx_le_u64, v_cmpx_lg_f16, v_cmpx_lg_f32,
v_cmpx_lg_f64, v_cmpx_lt_f16, v_cmpx_lt_f32,
v_cmpx_lt_f64, v_cmpx_lt_i16, v_cmpx_lt_i32,
v_cmpx_lt_i64, v_cmpx_lt_u16, v_cmpx_lt_u32,
v_cmpx_lt_u64, v_cmpx_ne_i16, v_cmpx_ne_i32,
v_cmpx_ne_i64, v_cmpx_ne_u16, v_cmpx_ne_u32,
v_cmpx_ne_u64, v_cmpx_neq_f16, v_cmpx_neq_f32,
v_cmpx_neq_f64, v_cmpx_nge_f16, v_cmpx_nge_f32,
v_cmpx_nge_f64, v_cmpx_ngt_f16, v_cmpx_ngt_f32,
v_cmpx_ngt_f64, v_cmpx_nle_f16, v_cmpx_nle_f32,
v_cmpx_nle_f64, v_cmpx_nlg_f16, v_cmpx_nlg_f32,
v_cmpx_nlg_f64, v_cmpx_nlt_f16, v_cmpx_nlt_f32,
v_cmpx_nlt_f64, v_cmpx_o_f16, v_cmpx_o_f32,
v_cmpx_o_f64, v_cmpx_t_i16, v_cmpx_t_i32,
v_cmpx_t_i64, v_cmpx_t_u16, v_cmpx_t_u32,
v_cmpx_t_u64, v_cmpx_tru_f16, v_cmpx_tru_f32,
v_cmpx_tru_f64, v_cmpx_u_f16, v_cmpx_u_f32,
v_cmpx_u_f64

OPF_WRM0

Instruction implicitly writes M0.

s_set_gpr_idx_idx, s_set_gpr_idx_mode, s_set_gpr_idx_on

OPF_WRPC

Instruction implicitly writes PC to an arbitrary register-based value, often an indirect jump (compare SEN_LABEL).

s_call_b64, s_cbranch_g_fork, s_cbranch_join,
s_rfe_b64, s_rfe_restore_b64, s_setpc_b64,
s_swappc_b64

OPF_WRSCC

Instruction implicitly writes SCC.

s_abs_i32, s_absdiff_i32, s_add_i32,
s_add_u32, s_addc_u32, s_addk_i32,
s_and_b32, s_and_b64, s_and_saveexec_b64,
s_andn1_saveexec_b64, s_andn1_wrexec_b64, s_andn2_b32,
s_andn2_b64, s_andn2_saveexec_b64, s_andn2_wrexec_b64,
s_ashr_i32, s_ashr_i64, s_bcnt0_i32_b32,
s_bcnt0_i32_b64, s_bcnt1_i32_b32, s_bcnt1_i32_b64,
s_bfe_i32, s_bfe_i64, s_bfe_u32,
s_bfe_u64, s_bitcmp0_b32, s_bitcmp0_b64,
s_bitcmp1_b32, s_bitcmp1_b64, s_cmp_eq_i32,
s_cmp_eq_u32, s_cmp_eq_u64, s_cmp_ge_i32,
s_cmp_ge_u32, s_cmp_gt_i32, s_cmp_gt_u32,
s_cmp_le_i32, s_cmp_le_u32, s_cmp_lg_i32,
s_cmp_lg_u32, s_cmp_lg_u64, s_cmp_lt_i32,
s_cmp_lt_u32, s_cmpk_eq_i32, s_cmpk_eq_u32,
s_cmpk_ge_i32, s_cmpk_ge_u32, s_cmpk_gt_i32,
s_cmpk_gt_u32, s_cmpk_le_i32, s_cmpk_le_u32,
s_cmpk_lg_i32, s_cmpk_lg_u32, s_cmpk_lt_i32,
s_cmpk_lt_u32, s_lshl1_add_u32, s_lshl2_add_u32,
s_lshl3_add_u32, s_lshl4_add_u32, s_lshl_b32,
s_lshl_b64, s_lshr_b32, s_lshr_b64,
s_max_i32, s_max_u32, s_min_i32,
s_min_u32, s_nand_b32, s_nand_b64,
s_nand_saveexec_b64, s_nor_b32, s_nor_b64,
s_nor_saveexec_b64, s_not_b32, s_not_b64,
s_or_b32, s_or_b64, s_or_saveexec_b64,
s_orn1_saveexec_b64, s_orn2_b32, s_orn2_b64,
s_orn2_saveexec_b64, s_quadmask_b32, s_quadmask_b64,
s_sub_i32, s_sub_u32, s_subb_u32,
s_wqm_b32, s_wqm_b64, s_xnor_b32,
s_xnor_b64, s_xnor_saveexec_b64, s_xor_b32,
s_xor_b64, s_xor_saveexec_b64

G Opcode Values

The following are all of the opcode values for each instruction encoding, sorted by opcode value.

Opcode Values

Enc	Opcode	Base (dec)	Base (hex)	VOP3 (dec)	VOP3 (hex)
SOP1	s_mov_b32	0	0x000		
SOP1	s_mov_b64	1	0x001		
SOP1	s_cmov_b32	2	0x002		
SOP1	s_cmov_b64	3	0x003		
SOP1	s_not_b32	4	0x004		
SOP1	s_not_b64	5	0x005		
SOP1	s_wqm_b32	6	0x006		
SOP1	s_wqm_b64	7	0x007		
SOP1	s_brev_b32	8	0x008		
SOP1	s_brev_b64	9	0x009		
SOP1	s_bcnt0_i32_b32	10	0x00a		
SOP1	s_bcnt0_i32_b64	11	0x00b		
SOP1	s_bcnt1_i32_b32	12	0x00c		
SOP1	s_bcnt1_i32_b64	13	0x00d		
SOP1	s_ff0_i32_b32	14	0x00e		
SOP1	s_ff0_i32_b64	15	0x00f		
SOP1	s_ff1_i32_b32	16	0x010		
SOP1	s_ff1_i32_b64	17	0x011		
SOP1	s_flbit_i32_b32	18	0x012		
SOP1	s_flbit_i32_b64	19	0x013		
SOP1	s_flbit_i32	20	0x014		
SOP1	s_flbit_i32_i64	21	0x015		
SOP1	s_sext_i32_i8	22	0x016		
SOP1	s_sext_i32_i16	23	0x017		
SOP1	s_bitset0_b32	24	0x018		
SOP1	s_bitset0_b64	25	0x019		
SOP1	s_bitset1_b32	26	0x01a		
SOP1	s_bitset1_b64	27	0x01b		
SOP1	s_getpc_b64	28	0x01c		
SOP1	s_setpc_b64	29	0x01d		
SOP1	s_swappc_b64	30	0x01e		
SOP1	s_rfe_b64	31	0x01f		
SOP1	s_and_saveexec_b64	32	0x020		
SOP1	s_or_saveexec_b64	33	0x021		
SOP1	s_xor_saveexec_b64	34	0x022		
SOP1	s_andn2_saveexec_b64	35	0x023		
SOP1	s_orn2_saveexec_b64	36	0x024		
SOP1	s_nand_saveexec_b64	37	0x025		
SOP1	s_nor_saveexec_b64	38	0x026		
SOP1	s_xnor_saveexec_b64	39	0x027		
SOP1	s_quadmask_b32	40	0x028		
SOP1	s_quadmask_b64	41	0x029		
SOP1	s_movrels_b32	42	0x02a		
SOP1	s_movrels_b64	43	0x02b		
SOP1	s_movreld_b32	44	0x02c		
SOP1	s_movreld_b64	45	0x02d		
SOP1	s_cbranch_join	46	0x02e		

Opcode Values (continued)

Enc	Opcode	Base (dec)	Base (hex)	VOP3 (dec)	VOP3 (hex)
SOP1	s_abs_i32	48	0x030		
SOP1	s_mov_fed_b32	49	0x031		
SOP1	s_set_gpr_idx_idx	50	0x032		
SOP1	s_andn1_saveexec_b64	51	0x033		
SOP1	s_orn1_saveexec_b64	52	0x034		
SOP1	s_andn1_wrexec_b64	53	0x035		
SOP1	s_andn2_wrexec_b64	54	0x036		
SOP1	s_bitreplicate_b64_b32	55	0x037		
SOPC	s_cmp_eq_i32	0	0x000		
SOPC	s_cmp_lg_i32	1	0x001		
SOPC	s_cmp_gt_i32	2	0x002		
SOPC	s_cmp_ge_i32	3	0x003		
SOPC	s_cmp_lt_i32	4	0x004		
SOPC	s_cmp_le_i32	5	0x005		
SOPC	s_cmp_eq_u32	6	0x006		
SOPC	s_cmp_lg_u32	7	0x007		
SOPC	s_cmp_gt_u32	8	0x008		
SOPC	s_cmp_ge_u32	9	0x009		
SOPC	s_cmp_lt_u32	10	0x00a		
SOPC	s_cmp_le_u32	11	0x00b		
SOPC	s_bitcmp0_b32	12	0x00c		
SOPC	s_bitcmp1_b32	13	0x00d		
SOPC	s_bitcmp0_b64	14	0x00e		
SOPC	s_bitcmp1_b64	15	0x00f		
SOPC	s_setvskip	16	0x010		
SOPC	s_set_gpr_idx_on	17	0x011		
SOPC	s_cmp_eq_u64	18	0x012		
SOPC	s_cmp_lg_u64	19	0x013		
SOPP	s_nop	0	0x000		
SOPP	s_endpgm	1	0x001		
SOPP	s_branch	2	0x002		
SOPP	s_wakeup	3	0x003		
SOPP	s_cbranch_scc0	4	0x004		
SOPP	s_cbranch_scc1	5	0x005		
SOPP	s_cbranch_vccz	6	0x006		
SOPP	s_cbranch_vccnz	7	0x007		
SOPP	s_cbranch_execz	8	0x008		
SOPP	s_cbranch_execnz	9	0x009		
SOPP	s_barrier	10	0x00a		
SOPP	s_setkill	11	0x00b		
SOPP	s_waitcnt	12	0x00c		
SOPP	s_sethalt	13	0x00d		
SOPP	s_sleep	14	0x00e		
SOPP	s_setprio	15	0x00f		
SOPP	s_sendmsg	16	0x010		
SOPP	s_sendmsghalt	17	0x011		
SOPP	s_trap	18	0x012		
SOPP	s_icache_inv	19	0x013		
SOPP	s_incperflevel	20	0x014		
SOPP	s_decperflevel	21	0x015		
SOPP	s_ttracedata	22	0x016		
SOPP	s_cbranch_cdbgsys	23	0x017		

Opcode Values (continued)

Enc	Opcode	Base (dec)	Base (hex)	VOP3 (dec)	VOP3 (hex)
SOPP	s_cbranch_cdbguser	24	0x018		
SOPP	s_cbranch_cdbgsys_or_user	25	0x019		
SOPP	s_cbranch_cdbgsys_and_user	26	0x01a		
SOPP	s_endpgm_saved	27	0x01b		
SOPP	s_set_gpr_idx_off	28	0x01c		
SOPP	s_set_gpr_idx_mode	29	0x01d		
SOPP	s_endpgm_ordered_ps_done	30	0x01e		
SOPK	s_movk_i32	0	0x000		
SOPK	s_cmovk_i32	1	0x001		
SOPK	s_cmpk_eq_i32	2	0x002		
SOPK	s_cmpk_lg_i32	3	0x003		
SOPK	s_cmpk_gt_i32	4	0x004		
SOPK	s_cmpk_ge_i32	5	0x005		
SOPK	s_cmpk_lt_i32	6	0x006		
SOPK	s_cmpk_le_i32	7	0x007		
SOPK	s_cmpk_eq_u32	8	0x008		
SOPK	s_cmpk_lg_u32	9	0x009		
SOPK	s_cmpk_gt_u32	10	0x00a		
SOPK	s_cmpk_ge_u32	11	0x00b		
SOPK	s_cmpk_lt_u32	12	0x00c		
SOPK	s_cmpk_le_u32	13	0x00d		
SOPK	s_addk_i32	14	0x00e		
SOPK	s_mulk_i32	15	0x00f		
SOPK	s_cbranch_i_fork	16	0x010		
SOPK	s_getreg_b32	17	0x011		
SOPK	s_setreg_b32	18	0x012		
SOPK	s_setreg_imm32_b32	20	0x014		
SOPK	s_call_b64	21	0x015		
SOP2	s_add_u32	0	0x000		
SOP2	s_sub_u32	1	0x001		
SOP2	s_add_i32	2	0x002		
SOP2	s_sub_i32	3	0x003		
SOP2	s_addc_u32	4	0x004		
SOP2	s_subb_u32	5	0x005		
SOP2	s_min_i32	6	0x006		
SOP2	s_min_u32	7	0x007		
SOP2	s_max_i32	8	0x008		
SOP2	s_max_u32	9	0x009		
SOP2	s_cselect_b32	10	0x00a		
SOP2	s_cselect_b64	11	0x00b		
SOP2	s_and_b32	12	0x00c		
SOP2	s_and_b64	13	0x00d		
SOP2	s_or_b32	14	0x00e		
SOP2	s_or_b64	15	0x00f		
SOP2	s_xor_b32	16	0x010		
SOP2	s_xor_b64	17	0x011		
SOP2	s_andn2_b32	18	0x012		
SOP2	s_andn2_b64	19	0x013		
SOP2	s_orn2_b32	20	0x014		
SOP2	s_orn2_b64	21	0x015		
SOP2	s_nand_b32	22	0x016		
SOP2	s_nand_b64	23	0x017		

Opcode Values (continued)

Enc	Opcode	Base (dec)	Base (hex)	VOP3 (dec)	VOP3 (hex)
SOP2	s_nor_b32	24	0x018		
SOP2	s_nor_b64	25	0x019		
SOP2	s_xnor_b32	26	0x01a		
SOP2	s_xnor_b64	27	0x01b		
SOP2	s_lshl_b32	28	0x01c		
SOP2	s_lshl_b64	29	0x01d		
SOP2	s_lshr_b32	30	0x01e		
SOP2	s_lshr_b64	31	0x01f		
SOP2	s_ashr_i32	32	0x020		
SOP2	s_ashr_i64	33	0x021		
SOP2	s_bfm_b32	34	0x022		
SOP2	s_bfm_b64	35	0x023		
SOP2	s_mul_i32	36	0x024		
SOP2	s_bfe_u32	37	0x025		
SOP2	s_bfe_i32	38	0x026		
SOP2	s_bfe_u64	39	0x027		
SOP2	s_bfe_i64	40	0x028		
SOP2	s_cbranch_g_fork	41	0x029		
SOP2	s_absdiff_i32	42	0x02a		
SOP2	s_rfe_restore_b64	43	0x02b		
SOP2	s_mul_hi_u32	44	0x02c		
SOP2	s_mul_hi_i32	45	0x02d		
SOP2	s_lshl1_add_u32	46	0x02e		
SOP2	s_lshl2_add_u32	47	0x02f		
SOP2	s_lshl3_add_u32	48	0x030		
SOP2	s_lshl4_add_u32	49	0x031		
SOP2	s_pack_ll_b32_b16	50	0x032		
SOP2	s_pack_lh_b32_b16	51	0x033		
SOP2	s_pack_hh_b32_b16	52	0x034		
SMEM	s_load_dword	0	0x000		
SMEM	s_load_dwordx2	1	0x001		
SMEM	s_load_dwordx4	2	0x002		
SMEM	s_load_dwordx8	3	0x003		
SMEM	s_load_dwordx16	4	0x004		
SMEM	s_scratch_load_dword	5	0x005		
SMEM	s_scratch_load_dwordx2	6	0x006		
SMEM	s_scratch_load_dwordx4	7	0x007		
SMEM	s_buffer_load_dword	8	0x008		
SMEM	s_buffer_load_dwordx2	9	0x009		
SMEM	s_buffer_load_dwordx4	10	0x00a		
SMEM	s_buffer_load_dwordx8	11	0x00b		
SMEM	s_buffer_load_dwordx16	12	0x00c		
SMEM	s_store_dword	16	0x010		
SMEM	s_store_dwordx2	17	0x011		
SMEM	s_store_dwordx4	18	0x012		
SMEM	s_scratch_store_dword	21	0x015		
SMEM	s_scratch_store_dwordx2	22	0x016		
SMEM	s_scratch_store_dwordx4	23	0x017		
SMEM	s_buffer_store_dword	24	0x018		
SMEM	s_buffer_store_dwordx2	25	0x019		
SMEM	s_buffer_store_dwordx4	26	0x01a		
SMEM	s_dcache_inv	32	0x020		

Opcode Values (continued)

Enc	Opcode	Base (dec)	Base (hex)	VOP3 (dec)	VOP3 (hex)
SMEM	s_dcache_wb	33	0x021		
SMEM	s_dcache_inv_vol	34	0x022		
SMEM	s_dcache_wb_vol	35	0x023		
SMEM	s_memtime	36	0x024		
SMEM	s_memrealtime	37	0x025		
SMEM	s_atc_probe	38	0x026		
SMEM	s_atc_probe_buffer	39	0x027		
SMEM	s_dcache_discard	40	0x028		
SMEM	s_dcache_discard_x2	41	0x029		
SMEM	s_buffer_atomic_swap	64	0x040		
SMEM	s_buffer_atomic_cmpswap	65	0x041		
SMEM	s_buffer_atomic_add	66	0x042		
SMEM	s_buffer_atomic_sub	67	0x043		
SMEM	s_buffer_atomic_smin	68	0x044		
SMEM	s_buffer_atomic_umin	69	0x045		
SMEM	s_buffer_atomic_smax	70	0x046		
SMEM	s_buffer_atomic_umax	71	0x047		
SMEM	s_buffer_atomic_and	72	0x048		
SMEM	s_buffer_atomic_or	73	0x049		
SMEM	s_buffer_atomic_xor	74	0x04a		
SMEM	s_buffer_atomic_inc	75	0x04b		
SMEM	s_buffer_atomic_dec	76	0x04c		
SMEM	s_buffer_atomic_swap_x2	96	0x060		
SMEM	s_buffer_atomic_cmpswap_x2	97	0x061		
SMEM	s_buffer_atomic_add_x2	98	0x062		
SMEM	s_buffer_atomic_sub_x2	99	0x063		
SMEM	s_buffer_atomic_smin_x2	100	0x064		
SMEM	s_buffer_atomic_umin_x2	101	0x065		
SMEM	s_buffer_atomic_smax_x2	102	0x066		
SMEM	s_buffer_atomic_umax_x2	103	0x067		
SMEM	s_buffer_atomic_and_x2	104	0x068		
SMEM	s_buffer_atomic_or_x2	105	0x069		
SMEM	s_buffer_atomic_xor_x2	106	0x06a		
SMEM	s_buffer_atomic_inc_x2	107	0x06b		
SMEM	s_buffer_atomic_dec_x2	108	0x06c		
SMEM	s_atomic_swap	128	0x080		
SMEM	s_atomic_cmpswap	129	0x081		
SMEM	s_atomic_add	130	0x082		
SMEM	s_atomic_sub	131	0x083		
SMEM	s_atomic_smin	132	0x084		
SMEM	s_atomic_umin	133	0x085		
SMEM	s_atomic_smax	134	0x086		
SMEM	s_atomic_umax	135	0x087		
SMEM	s_atomic_and	136	0x088		
SMEM	s_atomic_or	137	0x089		
SMEM	s_atomic_xor	138	0x08a		
SMEM	s_atomic_inc	139	0x08b		
SMEM	s_atomic_dec	140	0x08c		
SMEM	s_atomic_swap_x2	160	0x0a0		
SMEM	s_atomic_cmpswap_x2	161	0x0a1		
SMEM	s_atomic_add_x2	162	0x0a2		
SMEM	s_atomic_sub_x2	163	0x0a3		

Opcode Values (continued)

Enc	Opcode	Base (dec)	Base (hex)	VOP3 (dec)	VOP3 (hex)
SMEM	s_atomic_smin_x2	164	0x0a4		
SMEM	s_atomic_umin_x2	165	0x0a5		
SMEM	s_atomic_smax_x2	166	0x0a6		
SMEM	s_atomic_umax_x2	167	0x0a7		
SMEM	s_atomic_and_x2	168	0x0a8		
SMEM	s_atomic_or_x2	169	0x0a9		
SMEM	s_atomic_xor_x2	170	0x0aa		
SMEM	s_atomic_inc_x2	171	0x0ab		
SMEM	s_atomic_dec_x2	172	0x0ac		
VOP1	v_nop	0	0x000	320	0x140
VOP1	v_mov_b32	1	0x001	321	0x141
VOP1	v_readfirstlane_b32	2	0x002	322	0x142
VOP1	v_cvt_i32_f64	3	0x003	323	0x143
VOP1	v_cvt_f64_i32	4	0x004	324	0x144
VOP1	v_cvt_f32_i32	5	0x005	325	0x145
VOP1	v_cvt_f32_u32	6	0x006	326	0x146
VOP1	v_cvt_u32_f32	7	0x007	327	0x147
VOP1	v_cvt_i32_f32	8	0x008	328	0x148
VOP1	v_mov_fed_b32	9	0x009	329	0x149
VOP1	v_cvt_f16_f32	10	0x00a	330	0x14a
VOP1	v_cvt_f32_f16	11	0x00b	331	0x14b
VOP1	v_cvt_rpi_i32_f32	12	0x00c	332	0x14c
VOP1	v_cvt_flr_i32_f32	13	0x00d	333	0x14d
VOP1	v_cvt_off_f32_i4	14	0x00e	334	0x14e
VOP1	v_cvt_f32_f64	15	0x00f	335	0x14f
VOP1	v_cvt_f64_f32	16	0x010	336	0x150
VOP1	v_cvt_f32_ubyte0	17	0x011	337	0x151
VOP1	v_cvt_f32_ubyte1	18	0x012	338	0x152
VOP1	v_cvt_f32_ubyte2	19	0x013	339	0x153
VOP1	v_cvt_f32_ubyte3	20	0x014	340	0x154
VOP1	v_cvt_u32_f64	21	0x015	341	0x155
VOP1	v_cvt_f64_u32	22	0x016	342	0x156
VOP1	v_trunc_f64	23	0x017	343	0x157
VOP1	v_ceil_f64	24	0x018	344	0x158
VOP1	v_rndne_f64	25	0x019	345	0x159
VOP1	v_floor_f64	26	0x01a	346	0x15a
VOP1	v_fract_f32	27	0x01b	347	0x15b
VOP1	v_trunc_f32	28	0x01c	348	0x15c
VOP1	v_ceil_f32	29	0x01d	349	0x15d
VOP1	v_rndne_f32	30	0x01e	350	0x15e
VOP1	v_floor_f32	31	0x01f	351	0x15f
VOP1	v_exp_f32	32	0x020	352	0x160
VOP1	v_log_f32	33	0x021	353	0x161
VOP1	v_rcp_f32	34	0x022	354	0x162
VOP1	v_rcp_iflag_f32	35	0x023	355	0x163
VOP1	v_rsq_f32	36	0x024	356	0x164
VOP1	v_rcp_f64	37	0x025	357	0x165
VOP1	v_rsq_f64	38	0x026	358	0x166
VOP1	v_sqrt_f32	39	0x027	359	0x167
VOP1	v_sqrt_f64	40	0x028	360	0x168
VOP1	v_sin_f32	41	0x029	361	0x169
VOP1	v_cos_f32	42	0x02a	362	0x16a

Opcode Values (continued)

Enc	Opcode	Base (dec)	Base (hex)	VOP3 (dec)	VOP3 (hex)
VOP1	v_not_b32	43	0x02b	363	0x16b
VOP1	v_bfrev_b32	44	0x02c	364	0x16c
VOP1	v_ffbh_u32	45	0x02d	365	0x16d
VOP1	v_ffbl_b32	46	0x02e	366	0x16e
VOP1	v_ffbh_i32	47	0x02f	367	0x16f
VOP1	v_frexp_exp_i32_f64	48	0x030	368	0x170
VOP1	v_frexp_mant_f64	49	0x031	369	0x171
VOP1	v_fract_f64	50	0x032	370	0x172
VOP1	v_frexp_exp_i32_f32	51	0x033	371	0x173
VOP1	v_frexp_mant_f32	52	0x034	372	0x174
VOP1	v_clrexcp	53	0x035	373	0x175
VOP1	v_screen_partition_4se_b32	55	0x037	375	0x177
VOP1	v_cvt_f16_u16	57	0x039	377	0x179
VOP1	v_cvt_f16_i16	58	0x03a	378	0x17a
VOP1	v_cvt_u16_f16	59	0x03b	379	0x17b
VOP1	v_cvt_i16_f16	60	0x03c	380	0x17c
VOP1	v_rcp_f16	61	0x03d	381	0x17d
VOP1	v_sqrt_f16	62	0x03e	382	0x17e
VOP1	v_rsq_f16	63	0x03f	383	0x17f
VOP1	v_log_f16	64	0x040	384	0x180
VOP1	v_exp_f16	65	0x041	385	0x181
VOP1	v_frexp_mant_f16	66	0x042	386	0x182
VOP1	v_frexp_exp_i16_f16	67	0x043	387	0x183
VOP1	v_floor_f16	68	0x044	388	0x184
VOP1	v_ceil_f16	69	0x045	389	0x185
VOP1	v_trunc_f16	70	0x046	390	0x186
VOP1	v_rndne_f16	71	0x047	391	0x187
VOP1	v_fract_f16	72	0x048	392	0x188
VOP1	v_sin_f16	73	0x049	393	0x189
VOP1	v_cos_f16	74	0x04a	394	0x18a
VOP1	v_exp_legacy_f32	75	0x04b	395	0x18b
VOP1	v_log_legacy_f32	76	0x04c	396	0x18c
VOP1	v_cvt_norm_i16_f16	77	0x04d	397	0x18d
VOP1	v_cvt_norm_u16_f16	78	0x04e	398	0x18e
VOP1	v_sat_pk_u8_i16	79	0x04f	399	0x18f
VOP1	v_swap_b32	81	0x051	401	0x191
VOPC	v_cmp_class_f32	16	0x010	16	0x010
VOPC	v_cmpx_class_f32	17	0x011	17	0x011
VOPC	v_cmp_class_f64	18	0x012	18	0x012
VOPC	v_cmpx_class_f64	19	0x013	19	0x013
VOPC	v_cmp_class_f16	20	0x014	20	0x014
VOPC	v_cmpx_class_f16	21	0x015	21	0x015
VOPC	v_cmp_f_f16	32	0x020	32	0x020
VOPC	v_cmp_lt_f16	33	0x021	33	0x021
VOPC	v_cmp_eq_f16	34	0x022	34	0x022
VOPC	v_cmp_le_f16	35	0x023	35	0x023
VOPC	v_cmp_gt_f16	36	0x024	36	0x024
VOPC	v_cmp_lg_f16	37	0x025	37	0x025
VOPC	v_cmp_ge_f16	38	0x026	38	0x026
VOPC	v_cmp_o_f16	39	0x027	39	0x027
VOPC	v_cmp_u_f16	40	0x028	40	0x028
VOPC	v_cmp_nge_f16	41	0x029	41	0x029

Opcode Values (continued)

Enc	Opcode	Base (dec)	Base (hex)	VOP3 (dec)	VOP3 (hex)
VOPC	v_cmp_nlg_f16	42	0x02a	42	0x02a
VOPC	v_cmp_ngt_f16	43	0x02b	43	0x02b
VOPC	v_cmp_nle_f16	44	0x02c	44	0x02c
VOPC	v_cmp_neq_f16	45	0x02d	45	0x02d
VOPC	v_cmp_nlt_f16	46	0x02e	46	0x02e
VOPC	v_cmp_tru_f16	47	0x02f	47	0x02f
VOPC	v_cmpx_f_f16	48	0x030	48	0x030
VOPC	v_cmpx_lt_f16	49	0x031	49	0x031
VOPC	v_cmpx_eq_f16	50	0x032	50	0x032
VOPC	v_cmpx_le_f16	51	0x033	51	0x033
VOPC	v_cmpx_gt_f16	52	0x034	52	0x034
VOPC	v_cmpx_lg_f16	53	0x035	53	0x035
VOPC	v_cmpx_ge_f16	54	0x036	54	0x036
VOPC	v_cmpx_o_f16	55	0x037	55	0x037
VOPC	v_cmpx_u_f16	56	0x038	56	0x038
VOPC	v_cmpx_nge_f16	57	0x039	57	0x039
VOPC	v_cmpx_nlg_f16	58	0x03a	58	0x03a
VOPC	v_cmpx_ngt_f16	59	0x03b	59	0x03b
VOPC	v_cmpx_nle_f16	60	0x03c	60	0x03c
VOPC	v_cmpx_neq_f16	61	0x03d	61	0x03d
VOPC	v_cmpx_nlt_f16	62	0x03e	62	0x03e
VOPC	v_cmpx_tru_f16	63	0x03f	63	0x03f
VOPC	v_cmp_f_f32	64	0x040	64	0x040
VOPC	v_cmp_lt_f32	65	0x041	65	0x041
VOPC	v_cmp_eq_f32	66	0x042	66	0x042
VOPC	v_cmp_le_f32	67	0x043	67	0x043
VOPC	v_cmp_gt_f32	68	0x044	68	0x044
VOPC	v_cmp_lg_f32	69	0x045	69	0x045
VOPC	v_cmp_ge_f32	70	0x046	70	0x046
VOPC	v_cmp_o_f32	71	0x047	71	0x047
VOPC	v_cmp_u_f32	72	0x048	72	0x048
VOPC	v_cmp_nge_f32	73	0x049	73	0x049
VOPC	v_cmp_nlg_f32	74	0x04a	74	0x04a
VOPC	v_cmp_ngt_f32	75	0x04b	75	0x04b
VOPC	v_cmp_nle_f32	76	0x04c	76	0x04c
VOPC	v_cmp_neq_f32	77	0x04d	77	0x04d
VOPC	v_cmp_nlt_f32	78	0x04e	78	0x04e
VOPC	v_cmp_tru_f32	79	0x04f	79	0x04f
VOPC	v_cmpx_f_f32	80	0x050	80	0x050
VOPC	v_cmpx_lt_f32	81	0x051	81	0x051
VOPC	v_cmpx_eq_f32	82	0x052	82	0x052
VOPC	v_cmpx_le_f32	83	0x053	83	0x053
VOPC	v_cmpx_gt_f32	84	0x054	84	0x054
VOPC	v_cmpx_lg_f32	85	0x055	85	0x055
VOPC	v_cmpx_ge_f32	86	0x056	86	0x056
VOPC	v_cmpx_o_f32	87	0x057	87	0x057
VOPC	v_cmpx_u_f32	88	0x058	88	0x058
VOPC	v_cmpx_nge_f32	89	0x059	89	0x059
VOPC	v_cmpx_nlg_f32	90	0x05a	90	0x05a
VOPC	v_cmpx_ngt_f32	91	0x05b	91	0x05b
VOPC	v_cmpx_nle_f32	92	0x05c	92	0x05c
VOPC	v_cmpx_neq_f32	93	0x05d	93	0x05d

Opcode Values (continued)

Enc	Opcode	Base (dec)	Base (hex)	VOP3 (dec)	VOP3 (hex)
VOPC	v_cmpx_nlt_f32	94	0x05e	94	0x05e
VOPC	v_cmpx_tru_f32	95	0x05f	95	0x05f
VOPC	v_cmp_f_f64	96	0x060	96	0x060
VOPC	v_cmp_lt_f64	97	0x061	97	0x061
VOPC	v_cmp_eq_f64	98	0x062	98	0x062
VOPC	v_cmp_le_f64	99	0x063	99	0x063
VOPC	v_cmp_gt_f64	100	0x064	100	0x064
VOPC	v_cmp_lg_f64	101	0x065	101	0x065
VOPC	v_cmp_ge_f64	102	0x066	102	0x066
VOPC	v_cmp_o_f64	103	0x067	103	0x067
VOPC	v_cmp_u_f64	104	0x068	104	0x068
VOPC	v_cmp_nge_f64	105	0x069	105	0x069
VOPC	v_cmp_nlg_f64	106	0x06a	106	0x06a
VOPC	v_cmp_ngt_f64	107	0x06b	107	0x06b
VOPC	v_cmp_nle_f64	108	0x06c	108	0x06c
VOPC	v_cmp_neq_f64	109	0x06d	109	0x06d
VOPC	v_cmp_nlt_f64	110	0x06e	110	0x06e
VOPC	v_cmp_tru_f64	111	0x06f	111	0x06f
VOPC	v_cmpx_f_f64	112	0x070	112	0x070
VOPC	v_cmpx_lt_f64	113	0x071	113	0x071
VOPC	v_cmpx_eq_f64	114	0x072	114	0x072
VOPC	v_cmpx_le_f64	115	0x073	115	0x073
VOPC	v_cmpx_gt_f64	116	0x074	116	0x074
VOPC	v_cmpx_lg_f64	117	0x075	117	0x075
VOPC	v_cmpx_ge_f64	118	0x076	118	0x076
VOPC	v_cmpx_o_f64	119	0x077	119	0x077
VOPC	v_cmpx_u_f64	120	0x078	120	0x078
VOPC	v_cmpx_nge_f64	121	0x079	121	0x079
VOPC	v_cmpx_nlg_f64	122	0x07a	122	0x07a
VOPC	v_cmpx_ngt_f64	123	0x07b	123	0x07b
VOPC	v_cmpx_nle_f64	124	0x07c	124	0x07c
VOPC	v_cmpx_neq_f64	125	0x07d	125	0x07d
VOPC	v_cmpx_nlt_f64	126	0x07e	126	0x07e
VOPC	v_cmpx_tru_f64	127	0x07f	127	0x07f
VOPC	v_cmp_f_i16	160	0x0a0	160	0x0a0
VOPC	v_cmp_lt_i16	161	0x0a1	161	0x0a1
VOPC	v_cmp_eq_i16	162	0x0a2	162	0x0a2
VOPC	v_cmp_le_i16	163	0x0a3	163	0x0a3
VOPC	v_cmp_gt_i16	164	0x0a4	164	0x0a4
VOPC	v_cmp_ne_i16	165	0x0a5	165	0x0a5
VOPC	v_cmp_ge_i16	166	0x0a6	166	0x0a6
VOPC	v_cmp_t_i16	167	0x0a7	167	0x0a7
VOPC	v_cmp_f_u16	168	0x0a8	168	0x0a8
VOPC	v_cmp_lt_u16	169	0x0a9	169	0x0a9
VOPC	v_cmp_eq_u16	170	0x0aa	170	0x0aa
VOPC	v_cmp_le_u16	171	0x0ab	171	0x0ab
VOPC	v_cmp_gt_u16	172	0x0ac	172	0x0ac
VOPC	v_cmp_ne_u16	173	0x0ad	173	0x0ad
VOPC	v_cmp_ge_u16	174	0x0ae	174	0x0ae
VOPC	v_cmp_t_u16	175	0x0af	175	0x0af
VOPC	v_cmpx_f_i16	176	0x0b0	176	0x0b0
VOPC	v_cmpx_lt_i16	177	0x0b1	177	0x0b1

Opcode Values (continued)

Enc	Opcode	Base (dec)	Base (hex)	VOP3 (dec)	VOP3 (hex)
VOPC	v_cmpx_eq_i16	178	0x0b2	178	0x0b2
VOPC	v_cmpx_le_i16	179	0x0b3	179	0x0b3
VOPC	v_cmpx_gt_i16	180	0x0b4	180	0x0b4
VOPC	v_cmpx_ne_i16	181	0x0b5	181	0x0b5
VOPC	v_cmpx_ge_i16	182	0x0b6	182	0x0b6
VOPC	v_cmpx_t_i16	183	0x0b7	183	0x0b7
VOPC	v_cmpx_f_u16	184	0x0b8	184	0x0b8
VOPC	v_cmpx_lt_u16	185	0x0b9	185	0x0b9
VOPC	v_cmpx_eq_u16	186	0x0ba	186	0x0ba
VOPC	v_cmpx_le_u16	187	0x0bb	187	0x0bb
VOPC	v_cmpx_gt_u16	188	0x0bc	188	0x0bc
VOPC	v_cmpx_ne_u16	189	0x0bd	189	0x0bd
VOPC	v_cmpx_ge_u16	190	0x0be	190	0x0be
VOPC	v_cmpx_t_u16	191	0x0bf	191	0x0bf
VOPC	v_cmp_f_i32	192	0x0c0	192	0x0c0
VOPC	v_cmp_lt_i32	193	0x0c1	193	0x0c1
VOPC	v_cmp_eq_i32	194	0x0c2	194	0x0c2
VOPC	v_cmp_le_i32	195	0x0c3	195	0x0c3
VOPC	v_cmp_gt_i32	196	0x0c4	196	0x0c4
VOPC	v_cmp_ne_i32	197	0x0c5	197	0x0c5
VOPC	v_cmp_ge_i32	198	0x0c6	198	0x0c6
VOPC	v_cmp_t_i32	199	0x0c7	199	0x0c7
VOPC	v_cmp_f_u32	200	0x0c8	200	0x0c8
VOPC	v_cmp_lt_u32	201	0x0c9	201	0x0c9
VOPC	v_cmp_eq_u32	202	0x0ca	202	0x0ca
VOPC	v_cmp_le_u32	203	0x0cb	203	0x0cb
VOPC	v_cmp_gt_u32	204	0x0cc	204	0x0cc
VOPC	v_cmp_ne_u32	205	0x0cd	205	0x0cd
VOPC	v_cmp_ge_u32	206	0x0ce	206	0x0ce
VOPC	v_cmp_t_u32	207	0x0cf	207	0x0cf
VOPC	v_cmpx_f_i32	208	0x0d0	208	0x0d0
VOPC	v_cmpx_lt_i32	209	0x0d1	209	0x0d1
VOPC	v_cmpx_eq_i32	210	0x0d2	210	0x0d2
VOPC	v_cmpx_le_i32	211	0x0d3	211	0x0d3
VOPC	v_cmpx_gt_i32	212	0x0d4	212	0x0d4
VOPC	v_cmpx_ne_i32	213	0x0d5	213	0x0d5
VOPC	v_cmpx_ge_i32	214	0x0d6	214	0x0d6
VOPC	v_cmpx_t_i32	215	0x0d7	215	0x0d7
VOPC	v_cmpx_f_u32	216	0x0d8	216	0x0d8
VOPC	v_cmpx_lt_u32	217	0x0d9	217	0x0d9
VOPC	v_cmpx_eq_u32	218	0x0da	218	0x0da
VOPC	v_cmpx_le_u32	219	0x0db	219	0x0db
VOPC	v_cmpx_gt_u32	220	0x0dc	220	0x0dc
VOPC	v_cmpx_ne_u32	221	0x0dd	221	0x0dd
VOPC	v_cmpx_ge_u32	222	0x0de	222	0x0de
VOPC	v_cmpx_t_u32	223	0x0df	223	0x0df
VOPC	v_cmp_f_i64	224	0x0e0	224	0x0e0
VOPC	v_cmp_lt_i64	225	0x0e1	225	0x0e1
VOPC	v_cmp_eq_i64	226	0x0e2	226	0x0e2
VOPC	v_cmp_le_i64	227	0x0e3	227	0x0e3
VOPC	v_cmp_gt_i64	228	0x0e4	228	0x0e4
VOPC	v_cmp_ne_i64	229	0x0e5	229	0x0e5

Opcode Values (continued)

Enc	Opcode	Base (dec)	Base (hex)	VOP3 (dec)	VOP3 (hex)
VOPC	v_cmp_ge_i64	230	0x0e6	230	0x0e6
VOPC	v_cmp_t_i64	231	0x0e7	231	0x0e7
VOPC	v_cmp_f_u64	232	0x0e8	232	0x0e8
VOPC	v_cmp_lt_u64	233	0x0e9	233	0x0e9
VOPC	v_cmp_eq_u64	234	0x0ea	234	0x0ea
VOPC	v_cmp_le_u64	235	0x0eb	235	0x0eb
VOPC	v_cmp_gt_u64	236	0x0ec	236	0x0ec
VOPC	v_cmp_ne_u64	237	0x0ed	237	0x0ed
VOPC	v_cmp_ge_u64	238	0x0ee	238	0x0ee
VOPC	v_cmp_t_u64	239	0x0ef	239	0x0ef
VOPC	v_cmpx_f_i64	240	0x0f0	240	0x0f0
VOPC	v_cmpx_lt_i64	241	0x0f1	241	0x0f1
VOPC	v_cmpx_eq_i64	242	0x0f2	242	0x0f2
VOPC	v_cmpx_le_i64	243	0x0f3	243	0x0f3
VOPC	v_cmpx_gt_i64	244	0x0f4	244	0x0f4
VOPC	v_cmpx_ne_i64	245	0x0f5	245	0x0f5
VOPC	v_cmpx_ge_i64	246	0x0f6	246	0x0f6
VOPC	v_cmpx_t_i64	247	0x0f7	247	0x0f7
VOPC	v_cmpx_f_u64	248	0x0f8	248	0x0f8
VOPC	v_cmpx_lt_u64	249	0x0f9	249	0x0f9
VOPC	v_cmpx_eq_u64	250	0x0fa	250	0x0fa
VOPC	v_cmpx_le_u64	251	0x0fb	251	0x0fb
VOPC	v_cmpx_gt_u64	252	0x0fc	252	0x0fc
VOPC	v_cmpx_ne_u64	253	0x0fd	253	0x0fd
VOPC	v_cmpx_ge_u64	254	0x0fe	254	0x0fe
VOPC	v_cmpx_t_u64	255	0x0ff	255	0x0ff
VOP2	v_cndmask_b32	0	0x000	256	0x100
VOP2	v_add_f32	1	0x001	257	0x101
VOP2	v_sub_f32	2	0x002	258	0x102
VOP2	v_subrev_f32	3	0x003	259	0x103
VOP2	v_mul_legacy_f32	4	0x004	260	0x104
VOP2	v_mul_f32	5	0x005	261	0x105
VOP2	v_mul_i32_i24	6	0x006	262	0x106
VOP2	v_mul_hi_i32_i24	7	0x007	263	0x107
VOP2	v_mul_u32_u24	8	0x008	264	0x108
VOP2	v_mul_hi_u32_u24	9	0x009	265	0x109
VOP2	v_min_f32	10	0x00a	266	0x10a
VOP2	v_max_f32	11	0x00b	267	0x10b
VOP2	v_min_i32	12	0x00c	268	0x10c
VOP2	v_max_i32	13	0x00d	269	0x10d
VOP2	v_min_u32	14	0x00e	270	0x10e
VOP2	v_max_u32	15	0x00f	271	0x10f
VOP2	v_lshrrev_b32	16	0x010	272	0x110
VOP2	v_ashrrev_i32	17	0x011	273	0x111
VOP2	v_lshlrev_b32	18	0x012	274	0x112
VOP2	v_and_b32	19	0x013	275	0x113
VOP2	v_or_b32	20	0x014	276	0x114
VOP2	v_xor_b32	21	0x015	277	0x115
VOP2	v_mac_f32	22	0x016	278	0x116
VOP2	v_madmk_f32	23	0x017	279	0x117
VOP2	v_madak_f32	24	0x018	280	0x118
VOP2	v_add_co_u32	25	0x019	281	0x119

Opcode Values (continued)

Enc	Opcode	Base (dec)	Base (hex)	VOP3 (dec)	VOP3 (hex)
VOP2	v_sub_co_u32	26	0x01a	282	0x11a
VOP2	v_subrev_co_u32	27	0x01b	283	0x11b
VOP2	v_addc_co_u32	28	0x01c	284	0x11c
VOP2	v_subb_co_u32	29	0x01d	285	0x11d
VOP2	v_subbrev_co_u32	30	0x01e	286	0x11e
VOP2	v_add_f16	31	0x01f	287	0x11f
VOP2	v_sub_f16	32	0x020	288	0x120
VOP2	v_subrev_f16	33	0x021	289	0x121
VOP2	v_mul_f16	34	0x022	290	0x122
VOP2	v_mac_f16	35	0x023	291	0x123
VOP2	v_madmk_f16	36	0x024	292	0x124
VOP2	v_madak_f16	37	0x025	293	0x125
VOP2	v_add_u16	38	0x026	294	0x126
VOP2	v_sub_u16	39	0x027	295	0x127
VOP2	v_subrev_u16	40	0x028	296	0x128
VOP2	v_mul_lo_u16	41	0x029	297	0x129
VOP2	v_lshlrev_b16	42	0x02a	298	0x12a
VOP2	v_lshrrev_b16	43	0x02b	299	0x12b
VOP2	v_ashrrev_i16	44	0x02c	300	0x12c
VOP2	v_max_f16	45	0x02d	301	0x12d
VOP2	v_min_f16	46	0x02e	302	0x12e
VOP2	v_max_u16	47	0x02f	303	0x12f
VOP2	v_max_i16	48	0x030	304	0x130
VOP2	v_min_u16	49	0x031	305	0x131
VOP2	v_min_i16	50	0x032	306	0x132
VOP2	v_ldexp_f16	51	0x033	307	0x133
VOP2	v_add_u32	52	0x034	308	0x134
VOP2	v_sub_u32	53	0x035	309	0x135
VOP2	v_subrev_u32	54	0x036	310	0x136
VOP2	v_dot2c_f32_f16	55	0x037	311	0x137
VOP2	v_dot2c_i32_i16	56	0x038	312	0x138
VOP2	v_dot4c_i32_i8	57	0x039	313	0x139
VOP2	v_dot8c_i32_i4	58	0x03a	314	0x13a
VOP2	v_fmac_f32	59	0x03b	315	0x13b
VOP2	v_pk_fmac_f16	60	0x03c	316	0x13c
VOP2	v_xnor_b32	61	0x03d	317	0x13d
VINTRP	v_interp_p1_f32	0	0x000	624	0x270
VINTRP	v_interp_p2_f32	1	0x001	625	0x271
VINTRP	v_interp_mov_f32	2	0x002	626	0x272
VOP3P	v_pk_mad_i16	0	0x000		
VOP3P	v_pk_mul_lo_u16	1	0x001		
VOP3P	v_pk_add_i16	2	0x002		
VOP3P	v_pk_sub_i16	3	0x003		
VOP3P	v_pk_lshlrev_b16	4	0x004		
VOP3P	v_pk_lshrrev_b16	5	0x005		
VOP3P	v_pk_ashrrev_i16	6	0x006		
VOP3P	v_pk_max_i16	7	0x007		
VOP3P	v_pk_min_i16	8	0x008		
VOP3P	v_pk_mad_u16	9	0x009		
VOP3P	v_pk_add_u16	10	0x00a		
VOP3P	v_pk_sub_u16	11	0x00b		
VOP3P	v_pk_max_u16	12	0x00c		

Opcode Values (continued)

Enc	Opcode	Base (dec)	Base (hex)	VOP3 (dec)	VOP3 (hex)
VOP3P	v_pk_min_u16	13	0x00d		
VOP3P	v_pk_fma_f16	14	0x00e		
VOP3P	v_pk_add_f16	15	0x00f		
VOP3P	v_pk_mul_f16	16	0x010		
VOP3P	v_pk_min_f16	17	0x011		
VOP3P	v_pk_max_f16	18	0x012		
VOP3P	v_mad_mix_f32	32	0x020		
VOP3P	v_mad_mixlo_f16	33	0x021		
VOP3P	v_mad_mixhi_f16	34	0x022		
VOP3P	v_dot2_f32_f16	35	0x023		
VOP3P	v_dot2_i32_i16_i8	36	0x024		
VOP3P	v_dot2_u32_u16_u8	37	0x025		
VOP3P	v_dot2_i32_i16	38	0x026		
VOP3P	v_dot2_u32_u16	39	0x027		
VOP3P	v_dot4_i32_i8	40	0x028		
VOP3P	v_dot4_u32_u8	41	0x029		
VOP3P	v_dot8_i32_i4	42	0x02a		
VOP3P	v_dot8_u32_u4	43	0x02b		
VOP3	v_mad_legacy_f32	448	0x1c0	448	0x1c0
VOP3	v_mad_f32	449	0x1c1	449	0x1c1
VOP3	v_mad_i32_i24	450	0x1c2	450	0x1c2
VOP3	v_mad_u32_u24	451	0x1c3	451	0x1c3
VOP3	v_cubeid_f32	452	0x1c4	452	0x1c4
VOP3	v_cubesc_f32	453	0x1c5	453	0x1c5
VOP3	v_cubetc_f32	454	0x1c6	454	0x1c6
VOP3	v_cubema_f32	455	0x1c7	455	0x1c7
VOP3	v_bfe_u32	456	0x1c8	456	0x1c8
VOP3	v_bfe_i32	457	0x1c9	457	0x1c9
VOP3	v_bfi_b32	458	0x1ca	458	0x1ca
VOP3	v_fma_f32	459	0x1cb	459	0x1cb
VOP3	v_fma_f64	460	0x1cc	460	0x1cc
VOP3	v_lerp_u8	461	0x1cd	461	0x1cd
VOP3	v_alignbit_b32	462	0x1ce	462	0x1ce
VOP3	v_alignbyte_b32	463	0x1cf	463	0x1cf
VOP3	v_min3_f32	464	0x1d0	464	0x1d0
VOP3	v_min3_i32	465	0x1d1	465	0x1d1
VOP3	v_min3_u32	466	0x1d2	466	0x1d2
VOP3	v_max3_f32	467	0x1d3	467	0x1d3
VOP3	v_max3_i32	468	0x1d4	468	0x1d4
VOP3	v_max3_u32	469	0x1d5	469	0x1d5
VOP3	v_med3_f32	470	0x1d6	470	0x1d6
VOP3	v_med3_i32	471	0x1d7	471	0x1d7
VOP3	v_med3_u32	472	0x1d8	472	0x1d8
VOP3	v_sad_u8	473	0x1d9	473	0x1d9
VOP3	v_sad_hi_u8	474	0x1da	474	0x1da
VOP3	v_sad_u16	475	0x1db	475	0x1db
VOP3	v_sad_u32	476	0x1dc	476	0x1dc
VOP3	v_cvt_pk_u8_f32	477	0x1dd	477	0x1dd
VOP3	v_div_fixup_f32	478	0x1de	478	0x1de
VOP3	v_div_fixup_f64	479	0x1df	479	0x1df
VOP3	v_div_scale_f32	480	0x1e0	480	0x1e0
VOP3	v_div_scale_f64	481	0x1e1	481	0x1e1

Opcode Values (continued)

Enc	Opcode	Base (dec)	Base (hex)	VOP3 (dec)	VOP3 (hex)
VOP3	v_div_fmas_f32	482	0x1e2	482	0x1e2
VOP3	v_div_fmas_f64	483	0x1e3	483	0x1e3
VOP3	v_msad_u8	484	0x1e4	484	0x1e4
VOP3	v_qsad_pk_u16_u8	485	0x1e5	485	0x1e5
VOP3	v_mqsad_pk_u16_u8	486	0x1e6	486	0x1e6
VOP3	v_mqsad_u32_u8	487	0x1e7	487	0x1e7
VOP3	v_mad_u64_u32	488	0x1e8	488	0x1e8
VOP3	v_mad_i64_i32	489	0x1e9	489	0x1e9
VOP3	v_mad_legacy_f16	490	0x1ea	490	0x1ea
VOP3	v_mad_legacy_u16	491	0x1eb	491	0x1eb
VOP3	v_mad_legacy_i16	492	0x1ec	492	0x1ec
VOP3	v_perm_b32	493	0x1ed	493	0x1ed
VOP3	v_fma_legacy_f16	494	0x1ee	494	0x1ee
VOP3	v_div_fixup_legacy_f16	495	0x1ef	495	0x1ef
VOP3	v_cvt_pkaccum_u8_f32	496	0x1f0	496	0x1f0
VOP3	v_mad_u32_u16	497	0x1f1	497	0x1f1
VOP3	v_mad_i32_i16	498	0x1f2	498	0x1f2
VOP3	v_xad_u32	499	0x1f3	499	0x1f3
VOP3	v_min3_f16	500	0x1f4	500	0x1f4
VOP3	v_min3_i16	501	0x1f5	501	0x1f5
VOP3	v_min3_u16	502	0x1f6	502	0x1f6
VOP3	v_max3_f16	503	0x1f7	503	0x1f7
VOP3	v_max3_i16	504	0x1f8	504	0x1f8
VOP3	v_max3_u16	505	0x1f9	505	0x1f9
VOP3	v_med3_f16	506	0x1fa	506	0x1fa
VOP3	v_med3_i16	507	0x1fb	507	0x1fb
VOP3	v_med3_u16	508	0x1fc	508	0x1fc
VOP3	v_lshl_add_u32	509	0x1fd	509	0x1fd
VOP3	v_add_lshl_u32	510	0x1fe	510	0x1fe
VOP3	v_add3_u32	511	0x1ff	511	0x1ff
VOP3	v_lshl_or_b32	512	0x200	512	0x200
VOP3	v_and_or_b32	513	0x201	513	0x201
VOP3	v_or3_b32	514	0x202	514	0x202
VOP3	v_mad_f16	515	0x203	515	0x203
VOP3	v_mad_u16	516	0x204	516	0x204
VOP3	v_mad_i16	517	0x205	517	0x205
VOP3	v_fma_f16	518	0x206	518	0x206
VOP3	v_div_fixup_f16	519	0x207	519	0x207
VOP3	v_interp_p1ll_f16	628	0x274	628	0x274
VOP3	v_interp_p1lv_f16	629	0x275	629	0x275
VOP3	v_interp_p2_legacy_f16	630	0x276	630	0x276
VOP3	v_interp_p2_f16	631	0x277	631	0x277
VOP3	v_add_f64	640	0x280	640	0x280
VOP3	v_mul_f64	641	0x281	641	0x281
VOP3	v_min_f64	642	0x282	642	0x282
VOP3	v_max_f64	643	0x283	643	0x283
VOP3	v_ldexp_f64	644	0x284	644	0x284
VOP3	v_mul_lo_u32	645	0x285	645	0x285
VOP3	v_mul_hi_u32	646	0x286	646	0x286
VOP3	v_mul_hi_i32	647	0x287	647	0x287
VOP3	v_ldexp_f32	648	0x288	648	0x288
VOP3	v_readlane_b32	649	0x289	649	0x289

Opcode Values (continued)

Enc	Opcode	Base (dec)	Base (hex)	VOP3 (dec)	VOP3 (hex)
VOP3	v_writelane_b32	650	0x28a	650	0x28a
VOP3	v_bcnt_u32_b32	651	0x28b	651	0x28b
VOP3	v_mbcnt_lo_u32_b32	652	0x28c	652	0x28c
VOP3	v_mbcnt_hi_u32_b32	653	0x28d	653	0x28d
VOP3	v_lshlrev_b64	655	0x28f	655	0x28f
VOP3	v_lshrrev_b64	656	0x290	656	0x290
VOP3	v_ashrrev_i64	657	0x291	657	0x291
VOP3	v_trig_preop_f64	658	0x292	658	0x292
VOP3	v_bfm_b32	659	0x293	659	0x293
VOP3	v_cvt_pknorm_i16_f32	660	0x294	660	0x294
VOP3	v_cvt_pknorm_u16_f32	661	0x295	661	0x295
VOP3	v_cvt_pkrtz_f16_f32	662	0x296	662	0x296
VOP3	v_cvt_pk_u16_u32	663	0x297	663	0x297
VOP3	v_cvt_pk_i16_i32	664	0x298	664	0x298
VOP3	v_cvt_pknorm_i16_f16	665	0x299	665	0x299
VOP3	v_cvt_pknorm_u16_f16	666	0x29a	666	0x29a
VOP3	v_add_i32	668	0x29c	668	0x29c
VOP3	v_sub_i32	669	0x29d	669	0x29d
VOP3	v_add_i16	670	0x29e	670	0x29e
VOP3	v_sub_i16	671	0x29f	671	0x29f
VOP3	v_pack_b32_f16	672	0x2a0	672	0x2a0
DS	ds_add_u32	0	0x000		
DS	ds_sub_u32	1	0x001		
DS	ds_rsub_u32	2	0x002		
DS	ds_inc_u32	3	0x003		
DS	ds_dec_u32	4	0x004		
DS	ds_min_i32	5	0x005		
DS	ds_max_i32	6	0x006		
DS	ds_min_u32	7	0x007		
DS	ds_max_u32	8	0x008		
DS	ds_and_b32	9	0x009		
DS	ds_or_b32	10	0x00a		
DS	ds_xor_b32	11	0x00b		
DS	ds_mskor_b32	12	0x00c		
DS	ds_write_b32	13	0x00d		
DS	ds_write2_b32	14	0x00e		
DS	ds_write2st64_b32	15	0x00f		
DS	ds_cmpst_b32	16	0x010		
DS	ds_cmpst_f32	17	0x011		
DS	ds_min_f32	18	0x012		
DS	ds_max_f32	19	0x013		
DS	ds_nop	20	0x014		
DS	ds_add_f32	21	0x015		
DS	ds_write_addtid_b32	29	0x01d		
DS	ds_write_b8	30	0x01e		
DS	ds_write_b16	31	0x01f		
DS	ds_add_rtn_u32	32	0x020		
DS	ds_sub_rtn_u32	33	0x021		
DS	ds_rsub_rtn_u32	34	0x022		
DS	ds_inc_rtn_u32	35	0x023		
DS	ds_dec_rtn_u32	36	0x024		
DS	ds_min_rtn_i32	37	0x025		

Opcode Values (continued)

Enc	Opcode	Base (dec)	Base (hex)	VOP3 (dec)	VOP3 (hex)
DS	ds_max_rtn_i32	38	0x026		
DS	ds_min_rtn_u32	39	0x027		
DS	ds_max_rtn_u32	40	0x028		
DS	ds_and_rtn_b32	41	0x029		
DS	ds_or_rtn_b32	42	0x02a		
DS	ds_xor_rtn_b32	43	0x02b		
DS	ds_mskor_rtn_b32	44	0x02c		
DS	ds_wrxchg_rtn_b32	45	0x02d		
DS	ds_wrxchg2_rtn_b32	46	0x02e		
DS	ds_wrxchg2st64_rtn_b32	47	0x02f		
DS	ds_cmpst_rtn_b32	48	0x030		
DS	ds_cmpst_rtn_f32	49	0x031		
DS	ds_min_rtn_f32	50	0x032		
DS	ds_max_rtn_f32	51	0x033		
DS	ds_wrap_rtn_b32	52	0x034		
DS	ds_add_rtn_f32	53	0x035		
DS	ds_read_b32	54	0x036		
DS	ds_read2_b32	55	0x037		
DS	ds_read2st64_b32	56	0x038		
DS	ds_read_i8	57	0x039		
DS	ds_read_u8	58	0x03a		
DS	ds_read_i16	59	0x03b		
DS	ds_read_u16	60	0x03c		
DS	ds_swizzle_b32	61	0x03d		
DS	ds_permute_b32	62	0x03e		
DS	ds_bpermute_b32	63	0x03f		
DS	ds_add_u64	64	0x040		
DS	ds_sub_u64	65	0x041		
DS	ds_rsub_u64	66	0x042		
DS	ds_inc_u64	67	0x043		
DS	ds_dec_u64	68	0x044		
DS	ds_min_i64	69	0x045		
DS	ds_max_i64	70	0x046		
DS	ds_min_u64	71	0x047		
DS	ds_max_u64	72	0x048		
DS	ds_and_b64	73	0x049		
DS	ds_or_b64	74	0x04a		
DS	ds_xor_b64	75	0x04b		
DS	ds_mskor_b64	76	0x04c		
DS	ds_write_b64	77	0x04d		
DS	ds_write2_b64	78	0x04e		
DS	ds_write2st64_b64	79	0x04f		
DS	ds_cmpst_b64	80	0x050		
DS	ds_cmpst_f64	81	0x051		
DS	ds_min_f64	82	0x052		
DS	ds_max_f64	83	0x053		
DS	ds_write_b8_d16_hi	84	0x054		
DS	ds_write_b16_d16_hi	85	0x055		
DS	ds_read_u8_d16	86	0x056		
DS	ds_read_u8_d16_hi	87	0x057		
DS	ds_read_i8_d16	88	0x058		
DS	ds_read_i8_d16_hi	89	0x059		

Opcode Values (continued)

Enc	Opcode	Base (dec)	Base (hex)	VOP3 (dec)	VOP3 (hex)
DS	ds_read_u16_d16	90	0x05a		
DS	ds_read_u16_d16_hi	91	0x05b		
DS	ds_add_rtn_u64	96	0x060		
DS	ds_sub_rtn_u64	97	0x061		
DS	ds_rsub_rtn_u64	98	0x062		
DS	ds_inc_rtn_u64	99	0x063		
DS	ds_dec_rtn_u64	100	0x064		
DS	ds_min_rtn_i64	101	0x065		
DS	ds_max_rtn_i64	102	0x066		
DS	ds_min_rtn_u64	103	0x067		
DS	ds_max_rtn_u64	104	0x068		
DS	ds_and_rtn_b64	105	0x069		
DS	ds_or_rtn_b64	106	0x06a		
DS	ds_xor_rtn_b64	107	0x06b		
DS	ds_mskor_rtn_b64	108	0x06c		
DS	ds_wrxchg_rtn_b64	109	0x06d		
DS	ds_wrxchg2_rtn_b64	110	0x06e		
DS	ds_wrxchg2st64_rtn_b64	111	0x06f		
DS	ds_cmpst_rtn_b64	112	0x070		
DS	ds_cmpst_rtn_f64	113	0x071		
DS	ds_min_rtn_f64	114	0x072		
DS	ds_max_rtn_f64	115	0x073		
DS	ds_read_b64	118	0x076		
DS	ds_read2_b64	119	0x077		
DS	ds_read2st64_b64	120	0x078		
DS	ds_condxchg32_rtn_b64	126	0x07e		
DS	ds_add_src2_u32	128	0x080		
DS	ds_sub_src2_u32	129	0x081		
DS	ds_rsub_src2_u32	130	0x082		
DS	ds_inc_src2_u32	131	0x083		
DS	ds_dec_src2_u32	132	0x084		
DS	ds_min_src2_i32	133	0x085		
DS	ds_max_src2_i32	134	0x086		
DS	ds_min_src2_u32	135	0x087		
DS	ds_max_src2_u32	136	0x088		
DS	ds_and_src2_b32	137	0x089		
DS	ds_or_src2_b32	138	0x08a		
DS	ds_xor_src2_b32	139	0x08b		
DS	ds_write_src2_b32	141	0x08d		
DS	ds_min_src2_f32	146	0x092		
DS	ds_max_src2_f32	147	0x093		
DS	ds_add_src2_f32	149	0x095		
DS	ds_gws_sema_release_all	152	0x098		
DS	ds_gws_init	153	0x099		
DS	ds_gws_sema_v	154	0x09a		
DS	ds_gws_sema_br	155	0x09b		
DS	ds_gws_sema_p	156	0x09c		
DS	ds_gws_barrier	157	0x09d		
DS	ds_read_addtid_b32	182	0x0b6		
DS	ds_consume	189	0x0bd		
DS	ds_append	190	0x0be		
DS	ds_ordered_count	191	0x0bf		

Opcode Values (continued)

Enc	Opcode	Base (dec)	Base (hex)	VOP3 (dec)	VOP3 (hex)
DS	ds_add_src2_u64	192	0x0c0		
DS	ds_sub_src2_u64	193	0x0c1		
DS	ds_rsub_src2_u64	194	0x0c2		
DS	ds_inc_src2_u64	195	0x0c3		
DS	ds_dec_src2_u64	196	0x0c4		
DS	ds_min_src2_i64	197	0x0c5		
DS	ds_max_src2_i64	198	0x0c6		
DS	ds_min_src2_u64	199	0x0c7		
DS	ds_max_src2_u64	200	0x0c8		
DS	ds_and_src2_b64	201	0x0c9		
DS	ds_or_src2_b64	202	0x0ca		
DS	ds_xor_src2_b64	203	0x0cb		
DS	ds_write_src2_b64	205	0x0cd		
DS	ds_min_src2_f64	210	0x0d2		
DS	ds_max_src2_f64	211	0x0d3		
DS	ds_write_b96	222	0x0de		
DS	ds_write_b128	223	0x0df		
DS	ds_read_b96	254	0x0fe		
DS	ds_read_b128	255	0x0ff		
MUBUF	buffer_load_format_x	0	0x000		
MUBUF	buffer_load_format_xy	1	0x001		
MUBUF	buffer_load_format_xyz	2	0x002		
MUBUF	buffer_load_format_xyzw	3	0x003		
MUBUF	buffer_store_format_x	4	0x004		
MUBUF	buffer_store_format_xy	5	0x005		
MUBUF	buffer_store_format_xyz	6	0x006		
MUBUF	buffer_store_format_xyzw	7	0x007		
MUBUF	buffer_load_format_d16_x	8	0x008		
MUBUF	buffer_load_format_d16_xy	9	0x009		
MUBUF	buffer_load_format_d16_xyz	10	0x00a		
MUBUF	buffer_load_format_d16_xyzw	11	0x00b		
MUBUF	buffer_store_format_d16_x	12	0x00c		
MUBUF	buffer_store_format_d16_xy	13	0x00d		
MUBUF	buffer_store_format_d16_xyz	14	0x00e		
MUBUF	buffer_store_format_d16_xyzw	15	0x00f		
MUBUF	buffer_load_ubyte	16	0x010		
MUBUF	buffer_load_sbyte	17	0x011		
MUBUF	buffer_load_ushort	18	0x012		
MUBUF	buffer_load_sshort	19	0x013		
MUBUF	buffer_load_dword	20	0x014		
MUBUF	buffer_load_dwordx2	21	0x015		
MUBUF	buffer_load_dwordx3	22	0x016		
MUBUF	buffer_load_dwordx4	23	0x017		
MUBUF	buffer_store_byte	24	0x018		
MUBUF	buffer_store_byte_d16_hi	25	0x019		
MUBUF	buffer_store_short	26	0x01a		
MUBUF	buffer_store_short_d16_hi	27	0x01b		
MUBUF	buffer_store_dword	28	0x01c		
MUBUF	buffer_store_dwordx2	29	0x01d		
MUBUF	buffer_store_dwordx3	30	0x01e		
MUBUF	buffer_store_dwordx4	31	0x01f		
MUBUF	buffer_load_ubyte_d16	32	0x020		

Opcode Values (continued)

Enc	Opcode	Base (dec)	Base (hex)	VOP3 (dec)	VOP3 (hex)
MUBUF	buffer_load_ubyte_d16_hi	33	0x021		
MUBUF	buffer_load_sbyte_d16	34	0x022		
MUBUF	buffer_load_sbyte_d16_hi	35	0x023		
MUBUF	buffer_load_short_d16	36	0x024		
MUBUF	buffer_load_short_d16_hi	37	0x025		
MUBUF	buffer_load_format_d16_hi_x	38	0x026		
MUBUF	buffer_store_format_d16_hi_x	39	0x027		
MUBUF	buffer_store_lds_dword	61	0x03d		
MUBUF	buffer_wbinvl1	62	0x03e		
MUBUF	buffer_wbinvl1_vol	63	0x03f		
MUBUF	buffer_atomic_swap	64	0x040		
MUBUF	buffer_atomic_cmpswap	65	0x041		
MUBUF	buffer_atomic_add	66	0x042		
MUBUF	buffer_atomic_sub	67	0x043		
MUBUF	buffer_atomic_smin	68	0x044		
MUBUF	buffer_atomic_umin	69	0x045		
MUBUF	buffer_atomic_smax	70	0x046		
MUBUF	buffer_atomic_umax	71	0x047		
MUBUF	buffer_atomic_and	72	0x048		
MUBUF	buffer_atomic_or	73	0x049		
MUBUF	buffer_atomic_xor	74	0x04a		
MUBUF	buffer_atomic_inc	75	0x04b		
MUBUF	buffer_atomic_dec	76	0x04c		
MUBUF	buffer_atomic_swap_x2	96	0x060		
MUBUF	buffer_atomic_cmpswap_x2	97	0x061		
MUBUF	buffer_atomic_add_x2	98	0x062		
MUBUF	buffer_atomic_sub_x2	99	0x063		
MUBUF	buffer_atomic_smin_x2	100	0x064		
MUBUF	buffer_atomic_umin_x2	101	0x065		
MUBUF	buffer_atomic_smax_x2	102	0x066		
MUBUF	buffer_atomic_umax_x2	103	0x067		
MUBUF	buffer_atomic_and_x2	104	0x068		
MUBUF	buffer_atomic_or_x2	105	0x069		
MUBUF	buffer_atomic_xor_x2	106	0x06a		
MUBUF	buffer_atomic_inc_x2	107	0x06b		
MUBUF	buffer_atomic_dec_x2	108	0x06c		
MTBUF	tbuffer_load_format_x	0	0x000		
MTBUF	tbuffer_load_format_xy	1	0x001		
MTBUF	tbuffer_load_format_xyz	2	0x002		
MTBUF	tbuffer_load_format_xyzw	3	0x003		
MTBUF	tbuffer_store_format_x	4	0x004		
MTBUF	tbuffer_store_format_xy	5	0x005		
MTBUF	tbuffer_store_format_xyz	6	0x006		
MTBUF	tbuffer_store_format_xyzw	7	0x007		
MTBUF	tbuffer_load_format_d16_x	8	0x008		
MTBUF	tbuffer_load_format_d16_xy	9	0x009		
MTBUF	tbuffer_load_format_d16_xyz	10	0x00a		
MTBUF	tbuffer_load_format_d16_xyzw	11	0x00b		
MTBUF	tbuffer_store_format_d16_x	12	0x00c		
MTBUF	tbuffer_store_format_d16_xy	13	0x00d		
MTBUF	tbuffer_store_format_d16_xyz	14	0x00e		

Opcode Values (continued)

Enc	Opcode	Base (dec)	Base (hex)	VOP3 (dec)	VOP3 (hex)
MTBUF	tbuffer_store_format_d16_xyzw	15	0x00f		
MIMG	image_load	0	0x000		
MIMG	image_load_mip	1	0x001		
MIMG	image_load_pck	2	0x002		
MIMG	image_load_pck_sgn	3	0x003		
MIMG	image_load_mip_pck	4	0x004		
MIMG	image_load_mip_pck_sgn	5	0x005		
MIMG	image_store	8	0x008		
MIMG	image_store_mip	9	0x009		
MIMG	image_store_pck	10	0x00a		
MIMG	image_store_mip_pck	11	0x00b		
MIMG	image_get_resinfo	14	0x00e		
MIMG	image_atomic_swap	16	0x010		
MIMG	image_atomic_cmpswap	17	0x011		
MIMG	image_atomic_add	18	0x012		
MIMG	image_atomic_sub	19	0x013		
MIMG	image_atomic_smin	20	0x014		
MIMG	image_atomic_umin	21	0x015		
MIMG	image_atomic_smax	22	0x016		
MIMG	image_atomic_umax	23	0x017		
MIMG	image_atomic_and	24	0x018		
MIMG	image_atomic_or	25	0x019		
MIMG	image_atomic_xor	26	0x01a		
MIMG	image_atomic_inc	27	0x01b		
MIMG	image_atomic_dec	28	0x01c		
MIMG	image_sample	32	0x020		
MIMG	image_sample_cl	33	0x021		
MIMG	image_sample_d	34	0x022		
MIMG	image_sample_d_cl	35	0x023		
MIMG	image_sample_l	36	0x024		
MIMG	image_sample_b	37	0x025		
MIMG	image_sample_b_cl	38	0x026		
MIMG	image_sample_lz	39	0x027		
MIMG	image_sample_c	40	0x028		
MIMG	image_sample_c_cl	41	0x029		
MIMG	image_sample_c_d	42	0x02a		
MIMG	image_sample_c_d_cl	43	0x02b		
MIMG	image_sample_c_l	44	0x02c		
MIMG	image_sample_c_b	45	0x02d		
MIMG	image_sample_c_b_cl	46	0x02e		
MIMG	image_sample_c_lz	47	0x02f		
MIMG	image_sample_o	48	0x030		
MIMG	image_sample_cl_o	49	0x031		
MIMG	image_sample_d_o	50	0x032		
MIMG	image_sample_d_cl_o	51	0x033		
MIMG	image_sample_l_o	52	0x034		
MIMG	image_sample_b_o	53	0x035		
MIMG	image_sample_b_cl_o	54	0x036		
MIMG	image_sample_lz_o	55	0x037		
MIMG	image_sample_c_o	56	0x038		
MIMG	image_sample_c_cl_o	57	0x039		
MIMG	image_sample_c_d_o	58	0x03a		

Opcode Values (continued)

Enc	Opcode	Base (dec)	Base (hex)	VOP3 (dec)	VOP3 (hex)
MIMG	image_sample_c_d_cl_o	59	0x03b		
MIMG	image_sample_c_l_o	60	0x03c		
MIMG	image_sample_c_b_o	61	0x03d		
MIMG	image_sample_c_b_cl_o	62	0x03e		
MIMG	image_sample_c_lz_o	63	0x03f		
MIMG	image_gather4	64	0x040		
MIMG	image_gather4_cl	65	0x041		
MIMG	image_gather4h	66	0x042		
MIMG	image_gather4_l	68	0x044		
MIMG	image_gather4_b	69	0x045		
MIMG	image_gather4_b_cl	70	0x046		
MIMG	image_gather4_lz	71	0x047		
MIMG	image_gather4_c	72	0x048		
MIMG	image_gather4_c_cl	73	0x049		
MIMG	image_gather4h_pck	74	0x04a		
MIMG	image_gather8h_pck	75	0x04b		
MIMG	image_gather4_c_l	76	0x04c		
MIMG	image_gather4_c_b	77	0x04d		
MIMG	image_gather4_c_b_cl	78	0x04e		
MIMG	image_gather4_c_lz	79	0x04f		
MIMG	image_gather4_o	80	0x050		
MIMG	image_gather4_cl_o	81	0x051		
MIMG	image_gather4_l_o	84	0x054		
MIMG	image_gather4_b_o	85	0x055		
MIMG	image_gather4_b_cl_o	86	0x056		
MIMG	image_gather4_lz_o	87	0x057		
MIMG	image_gather4_c_o	88	0x058		
MIMG	image_gather4_c_cl_o	89	0x059		
MIMG	image_gather4_c_l_o	92	0x05c		
MIMG	image_gather4_c_b_o	93	0x05d		
MIMG	image_gather4_c_b_cl_o	94	0x05e		
MIMG	image_gather4_c_lz_o	95	0x05f		
MIMG	image_get_lod	96	0x060		
MIMG	image_sample_cd	104	0x068		
MIMG	image_sample_cd_cl	105	0x069		
MIMG	image_sample_c_cd	106	0x06a		
MIMG	image_sample_c_cd_cl	107	0x06b		
MIMG	image_sample_cd_o	108	0x06c		
MIMG	image_sample_cd_cl_o	109	0x06d		
MIMG	image_sample_c_cd_o	110	0x06e		
MIMG	image_sample_c_cd_cl_o	111	0x06f		
MIMG	image_sample_a	160	0x0a0		
MIMG	image_sample_cl_a	161	0x0a1		
MIMG	image_sample_b_a	165	0x0a5		
MIMG	image_sample_b_cl_a	166	0x0a6		
MIMG	image_sample_c_a	168	0x0a8		
MIMG	image_sample_c_cl_a	169	0x0a9		
MIMG	image_sample_c_b_a	173	0x0ad		
MIMG	image_sample_c_b_cl_a	174	0x0ae		
MIMG	image_sample_o_a	176	0x0b0		
MIMG	image_sample_cl_o_a	177	0x0b1		
MIMG	image_sample_b_o_a	181	0x0b5		

Opcode Values (continued)

Enc	Opcode	Base (dec)	Base (hex)	VOP3 (dec)	VOP3 (hex)
MIMG	image_sample_b_cl_o_a	182	0x0b6		
MIMG	image_sample_c_o_a	184	0x0b8		
MIMG	image_sample_c_cl_o_a	185	0x0b9		
MIMG	image_sample_c_b_o_a	189	0x0bd		
MIMG	image_sample_c_b_cl_o_a	190	0x0be		
MIMG	image_gather4_a	192	0x0c0		
MIMG	image_gather4_cl_a	193	0x0c1		
MIMG	image_gather4_b_a	197	0x0c5		
MIMG	image_gather4_b_cl_a	198	0x0c6		
MIMG	image_gather4_c_a	200	0x0c8		
MIMG	image_gather4_c_cl_a	201	0x0c9		
MIMG	image_gather4_c_b_a	205	0x0cd		
MIMG	image_gather4_c_b_cl_a	206	0x0ce		
MIMG	image_gather4_o_a	208	0x0d0		
MIMG	image_gather4_cl_o_a	209	0x0d1		
MIMG	image_gather4_b_o_a	213	0x0d5		
MIMG	image_gather4_b_cl_o_a	214	0x0d6		
MIMG	image_gather4_c_o_a	216	0x0d8		
MIMG	image_gather4_c_cl_o_a	217	0x0d9		
MIMG	image_gather4_c_b_o_a	221	0x0dd		
MIMG	image_gather4_c_b_cl_o_a	222	0x0de		
EXP	exp	0	0x000		
FLAT	flat_load_ubyte	16	0x010		
FLAT	global_load_ubyte	16	0x010		
FLAT	scratch_load_ubyte	16	0x010		
FLAT	flat_load_sbyte	17	0x011		
FLAT	global_load_sbyte	17	0x011		
FLAT	scratch_load_sbyte	17	0x011		
FLAT	flat_load_ushort	18	0x012		
FLAT	global_load_ushort	18	0x012		
FLAT	scratch_load_ushort	18	0x012		
FLAT	flat_load_sshort	19	0x013		
FLAT	global_load_sshort	19	0x013		
FLAT	scratch_load_sshort	19	0x013		
FLAT	flat_load_dword	20	0x014		
FLAT	global_load_dword	20	0x014		
FLAT	scratch_load_dword	20	0x014		
FLAT	flat_load_dwordx2	21	0x015		
FLAT	global_load_dwordx2	21	0x015		
FLAT	scratch_load_dwordx2	21	0x015		
FLAT	flat_load_dwordx3	22	0x016		
FLAT	global_load_dwordx3	22	0x016		
FLAT	scratch_load_dwordx3	22	0x016		
FLAT	flat_load_dwordx4	23	0x017		
FLAT	global_load_dwordx4	23	0x017		
FLAT	scratch_load_dwordx4	23	0x017		
FLAT	flat_store_byte	24	0x018		
FLAT	global_store_byte	24	0x018		
FLAT	scratch_store_byte	24	0x018		
FLAT	flat_store_byte_d16_hi	25	0x019		
FLAT	global_store_byte_d16_hi	25	0x019		
FLAT	scratch_store_byte_d16_hi	25	0x019		

Opcode Values (continued)

Enc	Opcode	Base (dec)	Base (hex)	VOP3 (dec)	VOP3 (hex)
FLAT	flat_store_short	26	0x01a		
FLAT	global_store_short	26	0x01a		
FLAT	scratch_store_short	26	0x01a		
FLAT	flat_store_short_d16_hi	27	0x01b		
FLAT	global_store_short_d16_hi	27	0x01b		
FLAT	scratch_store_short_d16_hi	27	0x01b		
FLAT	flat_store_dword	28	0x01c		
FLAT	global_store_dword	28	0x01c		
FLAT	scratch_store_dword	28	0x01c		
FLAT	flat_store_dwordx2	29	0x01d		
FLAT	global_store_dwordx2	29	0x01d		
FLAT	scratch_store_dwordx2	29	0x01d		
FLAT	flat_store_dwordx3	30	0x01e		
FLAT	global_store_dwordx3	30	0x01e		
FLAT	scratch_store_dwordx3	30	0x01e		
FLAT	flat_store_dwordx4	31	0x01f		
FLAT	global_store_dwordx4	31	0x01f		
FLAT	scratch_store_dwordx4	31	0x01f		
FLAT	flat_load_ubyte_d16	32	0x020		
FLAT	global_load_ubyte_d16	32	0x020		
FLAT	scratch_load_ubyte_d16	32	0x020		
FLAT	flat_load_ubyte_d16_hi	33	0x021		
FLAT	global_load_ubyte_d16_hi	33	0x021		
FLAT	scratch_load_ubyte_d16_hi	33	0x021		
FLAT	flat_load_sbyte_d16	34	0x022		
FLAT	global_load_sbyte_d16	34	0x022		
FLAT	scratch_load_sbyte_d16	34	0x022		
FLAT	flat_load_sbyte_d16_hi	35	0x023		
FLAT	global_load_sbyte_d16_hi	35	0x023		
FLAT	scratch_load_sbyte_d16_hi	35	0x023		
FLAT	flat_load_short_d16	36	0x024		
FLAT	global_load_short_d16	36	0x024		
FLAT	scratch_load_short_d16	36	0x024		
FLAT	flat_load_short_d16_hi	37	0x025		
FLAT	global_load_short_d16_hi	37	0x025		
FLAT	scratch_load_short_d16_hi	37	0x025		
FLAT	flat_atomic_swap	64	0x040		
FLAT	global_atomic_swap	64	0x040		
FLAT	flat_atomic_cmpswap	65	0x041		
FLAT	global_atomic_cmpswap	65	0x041		
FLAT	flat_atomic_add	66	0x042		
FLAT	global_atomic_add	66	0x042		
FLAT	flat_atomic_sub	67	0x043		
FLAT	global_atomic_sub	67	0x043		
FLAT	flat_atomic_smin	68	0x044		
FLAT	global_atomic_smin	68	0x044		
FLAT	flat_atomic_umin	69	0x045		
FLAT	global_atomic_umin	69	0x045		
FLAT	flat_atomic_smax	70	0x046		
FLAT	global_atomic_smax	70	0x046		
FLAT	flat_atomic_umax	71	0x047		
FLAT	global_atomic_umax	71	0x047		

Opcode Values (continued)

Enc	Opcode	Base (dec)	Base (hex)	VOP3 (dec)	VOP3 (hex)
FLAT	flat_atomic_and	72	0x048		
FLAT	global_atomic_and	72	0x048		
FLAT	flat_atomic_or	73	0x049		
FLAT	global_atomic_or	73	0x049		
FLAT	flat_atomic_xor	74	0x04a		
FLAT	global_atomic_xor	74	0x04a		
FLAT	flat_atomic_inc	75	0x04b		
FLAT	global_atomic_inc	75	0x04b		
FLAT	flat_atomic_dec	76	0x04c		
FLAT	global_atomic_dec	76	0x04c		
FLAT	flat_atomic_swap_x2	96	0x060		
FLAT	global_atomic_swap_x2	96	0x060		
FLAT	flat_atomic_cmpswap_x2	97	0x061		
FLAT	global_atomic_cmpswap_x2	97	0x061		
FLAT	flat_atomic_add_x2	98	0x062		
FLAT	global_atomic_add_x2	98	0x062		
FLAT	flat_atomic_sub_x2	99	0x063		
FLAT	global_atomic_sub_x2	99	0x063		
FLAT	flat_atomic_smin_x2	100	0x064		
FLAT	global_atomic_smin_x2	100	0x064		
FLAT	flat_atomic_umin_x2	101	0x065		
FLAT	global_atomic_umin_x2	101	0x065		
FLAT	flat_atomic_smax_x2	102	0x066		
FLAT	global_atomic_smax_x2	102	0x066		
FLAT	flat_atomic_umax_x2	103	0x067		
FLAT	global_atomic_umax_x2	103	0x067		
FLAT	flat_atomic_and_x2	104	0x068		
FLAT	global_atomic_and_x2	104	0x068		
FLAT	flat_atomic_or_x2	105	0x069		
FLAT	global_atomic_or_x2	105	0x069		
FLAT	flat_atomic_xor_x2	106	0x06a		
FLAT	global_atomic_xor_x2	106	0x06a		
FLAT	flat_atomic_inc_x2	107	0x06b		
FLAT	global_atomic_inc_x2	107	0x06b		
FLAT	flat_atomic_dec_x2	108	0x06c		
FLAT	global_atomic_dec_x2	108	0x06c		

H Illegal Opcode Patterns

This table shows all 32-bit patterns that will result in an illegal opcode if encountered as the first DWORD of an instruction. MSB is shown first and x indicates a particular bit position is “don’t-care”.

Illegal Opcode Patterns

Rank	Pattern			
0	01111100_000xxxxx_xxxxxxxx_xxxxxxxx	01111100_001011xx_xxxxxxxx_xxxxxxxx	01111100_0011xxxx_xxxxxxxx_xxxxxxxx	01111101_00xxxxxx_xxxxxxxx_xxxxxxxx
4	0111111x_xxxxxxx0_0111000x_xxxxxxxx	0111111x_xxxxxxx0_101001xx_xxxxxxxx	0111111x_xxxxxxx0_10101xxx_xxxxxxxx	0111111x_xxxxxxx0_1011xxxx_xxxxxxxx
8	0111111x_xxxxxxx0_11xxxxxx_xxxxxxxx	0111111x_xxxxxxx1_xxxxxxxx_xxxxxxxx	10011010_1xxxxxxx_xxxxxxxx_xxxxxxxx	10011011_xxxxxxxx_xxxxxxxx_xxxxxxxx
12	100111xx_xxxxxxxx_xxxxxxxx_xxxxxxxx	1010xxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx	10111011_xxxxxxxx_xxxxxxxx_xxxxxxxx	1011110x_xxxxxxxx_xxxxxxxx_xxxxxxxx
16	10111110_0xxxxxxx_xxxxxxxx_xxxxxxxx	10111110_1xxxxxxx_00111xxx_xxxxxxxx	10111110_1xxxxxxx_01xxxxxx_xxxxxxxx	10111110_1xxxxxxx_1xxxxxxx_xxxxxxxx
20	10111111_000101xx_xxxxxxxx_xxxxxxxx	10111111_00011xxx_xxxxxxxx_xxxxxxxx	10111111_001xxxxx_xxxxxxxx_xxxxxxxx	10111111_01xxxxxx_xxxxxxxx_xxxxxxxx
24	10111111_10011111_xxxxxxxx_xxxxxxxx	10111111_101xxxxx_xxxxxxxx_xxxxxxxx	10111111_11xxxxxx_xxxxxxxx_xxxxxxxx	11001xxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
28	110101xx_xxxxxx11_xxxxxxxx_xxxxxxxx	1110x1xx_xxxxxxxx_xxxxxxxx_xxxxxxxx	111101xx_xxxxxxxx_xxxxxxxx_xxxxxxxx	11111xxx_xxxxxxxx_xxxxxxxx_xxxxxxxx

Major Changes From gfx75 Until Current

Changes From gfx75 Until Current		
Opcode	**Current**	**gfx75**
buffer_atomic_add	MUBUF opcode 66	MUBUF opcode 50
buffer_atomic_add_x2	MUBUF opcode 98	MUBUF opcode 82
buffer_atomic_and	MUBUF opcode 72	MUBUF opcode 57
buffer_atomic_and_x2	MUBUF opcode 104	MUBUF opcode 89
buffer_atomic_cmpswap	MUBUF opcode 65	MUBUF opcode 49
buffer_atomic_cmpswap_x2	MUBUF opcode 97	MUBUF opcode 81
buffer_atomic_dec	MUBUF opcode 76	MUBUF opcode 61
buffer_atomic_dec_x2	MUBUF opcode 108	MUBUF opcode 93
buffer_atomic_fcmpswap	*deleted*	MUBUF opcode 62
buffer_atomic_fcmpswap_x2	*deleted*	MUBUF opcode 94
buffer_atomic_fmax	*deleted*	MUBUF opcode 64
buffer_atomic_fmax_x2	*deleted*	MUBUF opcode 96
buffer_atomic_fmin	*deleted*	MUBUF opcode 63
buffer_atomic_fmin_x2	*deleted*	MUBUF opcode 95
buffer_atomic_inc	MUBUF opcode 75	MUBUF opcode 60
buffer_atomic_inc_x2	MUBUF opcode 107	MUBUF opcode 92
buffer_atomic_or	MUBUF opcode 73	MUBUF opcode 58
buffer_atomic_or_x2	MUBUF opcode 105	MUBUF opcode 90
buffer_atomic_smax	MUBUF opcode 70	MUBUF opcode 55
buffer_atomic_smax_x2	MUBUF opcode 102	MUBUF opcode 87
buffer_atomic_smin	MUBUF opcode 68	MUBUF opcode 53
buffer_atomic_smin_x2	MUBUF opcode 100	MUBUF opcode 85
buffer_atomic_sub	MUBUF opcode 67	MUBUF opcode 51
buffer_atomic_sub_x2	MUBUF opcode 99	MUBUF opcode 83
buffer_atomic_swap	MUBUF opcode 64	MUBUF opcode 48
buffer_atomic_swap_x2	MUBUF opcode 96	MUBUF opcode 80
buffer_atomic_umax	MUBUF opcode 71	MUBUF opcode 56
buffer_atomic_umax_x2	MUBUF opcode 103	MUBUF opcode 88
buffer_atomic_umin	MUBUF opcode 69	MUBUF opcode 54
buffer_atomic_umin_x2	MUBUF opcode 101	MUBUF opcode 86
buffer_atomic_xor	MUBUF opcode 74	MUBUF opcode 59
buffer_atomic_xor_x2	MUBUF opcode 106	MUBUF opcode 91
buffer_load_dword	MUBUF opcode 20	MUBUF opcode 12
buffer_load_dwordx2	MUBUF opcode 21	MUBUF opcode 13
buffer_load_dwordx3	MUBUF opcode 22	MUBUF opcode 15
buffer_load_dwordx4	MUBUF opcode 23	MUBUF opcode 14
buffer_load_format_d16_hi_x	MUBUF opcode 38	*new*
buffer_load_format_d16_x	MUBUF opcode 8	MUBUF opcode 128
buffer_load_format_d16_xy	MUBUF opcode 9	MUBUF opcode 129
buffer_load_format_d16_xyz	MUBUF opcode 10	MUBUF opcode 130
buffer_load_format_d16_xyzw	MUBUF opcode 11	MUBUF opcode 131
buffer_load_sbyte	MUBUF opcode 17	MUBUF opcode 9
buffer_load_sbyte_d16	MUBUF opcode 34	*new*
buffer_load_sbyte_d16_hi	MUBUF opcode 35	*new*
buffer_load_short_d16	MUBUF opcode 36	*new*
buffer_load_short_d16_hi	MUBUF opcode 37	*new*
buffer_load_sshort	MUBUF opcode 19	MUBUF opcode 11
buffer_load_ubyte	MUBUF opcode 16	MUBUF opcode 8

Changes From gfx75 Until Current (continued)

Opcode	Current	gfx75
buffer_load_ubyte_d16	MUBUF opcode 32	*new*
buffer_load_ubyte_d16_hi	MUBUF opcode 33	*new*
buffer_load_ushort	MUBUF opcode 18	MUBUF opcode 10
buffer_store_byte_d16_hi	MUBUF opcode 25	*new*
buffer_store_dwordx3	MUBUF opcode 30	MUBUF opcode 31
buffer_store_dwordx4	MUBUF opcode 31	MUBUF opcode 30
buffer_store_format_d16_hi_x	MUBUF opcode 39	*new*
buffer_store_format_d16_x	MUBUF opcode 12	MUBUF opcode 132
buffer_store_format_d16_xy	MUBUF opcode 13	MUBUF opcode 133
buffer_store_format_d16_xyz	MUBUF opcode 14	MUBUF opcode 134
buffer_store_format_d16_xyzw	MUBUF opcode 15	MUBUF opcode 135
buffer_store_lds_dword	MUBUF opcode 61	*new*
buffer_store_short_d16_hi	MUBUF opcode 27	*new*
buffer_wbinvl1	MUBUF opcode 62	MUBUF opcode 113
buffer_wbinvl1_vol	MUBUF opcode 63	MUBUF opcode 112
ds_add_f32	DS opcode 21	*new*
ds_add_rtn_f32	DS opcode 53	*new*
ds_add_src2_f32	DS opcode 149	*new*
ds_append	DS opcode 190	DS opcode 62
ds_bpermute_b32	DS opcode 63	*new*
ds_consume	DS opcode 189	DS opcode 61
ds_gws_barrier	DS opcode 157	DS opcode 29
ds_gws_init	DS opcode 153	DS opcode 25
ds_gws_sema_br	DS opcode 155	DS opcode 27
ds_gws_sema_p	DS opcode 156	DS opcode 28
ds_gws_sema_release_all	DS opcode 152	DS opcode 24
ds_gws_sema_v	DS opcode 154	DS opcode 26
ds_ordered_count	DS opcode 191	DS opcode 63
ds_permute_b32	DS opcode 62	*new*
ds_read_addtid_b32	DS opcode 182	*new*
ds_read_b128	DS opcode 255	*new*
ds_read_b96	DS opcode 254	*new*
ds_read_i8_d16	DS opcode 88	*new*
ds_read_i8_d16_hi	DS opcode 89	*new*
ds_read_u16_d16	DS opcode 90	*new*
ds_read_u16_d16_hi	DS opcode 91	*new*
ds_read_u8_d16	DS opcode 86	*new*
ds_read_u8_d16_hi	DS opcode 87	*new*
ds_swizzle_b32	DS opcode 61	DS opcode 53
ds_write_addtid_b32	DS opcode 29	*new*
ds_write_b16_d16_hi	DS opcode 85	*new*
ds_write_b8_d16_hi	DS opcode 84	*new*
flat_atomic_add	FLAT opcode 66	FLAT opcode 50
flat_atomic_add_x2	FLAT opcode 98	FLAT opcode 82
flat_atomic_and	FLAT opcode 72	FLAT opcode 57
flat_atomic_and_x2	FLAT opcode 104	FLAT opcode 89
flat_atomic_cmpswap	FLAT opcode 65	FLAT opcode 49
flat_atomic_cmpswap_x2	FLAT opcode 97	FLAT opcode 81
flat_atomic_dec	FLAT opcode 76	FLAT opcode 61
flat_atomic_dec_x2	FLAT opcode 108	FLAT opcode 93
flat_atomic_fcmpswap	*deleted*	FLAT opcode 62
flat_atomic_fcmpswap_x2	*deleted*	FLAT opcode 94

Changes From gfx75 Until Current (continued)

Opcode	Current	gfx75
flat_atomic_fmax	*deleted*	FLAT opcode 64
flat_atomic_fmax_x2	*deleted*	FLAT opcode 96
flat_atomic_fmin	*deleted*	FLAT opcode 63
flat_atomic_fmin_x2	*deleted*	FLAT opcode 95
flat_atomic_inc	FLAT opcode 75	FLAT opcode 60
flat_atomic_inc_x2	FLAT opcode 107	FLAT opcode 92
flat_atomic_or	FLAT opcode 73	FLAT opcode 58
flat_atomic_or_x2	FLAT opcode 105	FLAT opcode 90
flat_atomic_smax	FLAT opcode 70	FLAT opcode 55
flat_atomic_smax_x2	FLAT opcode 102	FLAT opcode 87
flat_atomic_smin	FLAT opcode 68	FLAT opcode 53
flat_atomic_smin_x2	FLAT opcode 100	FLAT opcode 85
flat_atomic_sub	FLAT opcode 67	FLAT opcode 51
flat_atomic_sub_x2	FLAT opcode 99	FLAT opcode 83
flat_atomic_swap	FLAT opcode 64	FLAT opcode 48
flat_atomic_swap_x2	FLAT opcode 96	FLAT opcode 80
flat_atomic_umax	FLAT opcode 71	FLAT opcode 56
flat_atomic_umax_x2	FLAT opcode 103	FLAT opcode 88
flat_atomic_umin	FLAT opcode 69	FLAT opcode 54
flat_atomic_umin_x2	FLAT opcode 101	FLAT opcode 86
flat_atomic_xor	FLAT opcode 74	FLAT opcode 59
flat_atomic_xor_x2	FLAT opcode 106	FLAT opcode 91
flat_load_dword	FLAT opcode 20	FLAT opcode 12
flat_load_dwordx2	FLAT opcode 21	FLAT opcode 13
flat_load_dwordx3	FLAT opcode 22	FLAT opcode 15
flat_load_dwordx4	FLAT opcode 23	FLAT opcode 14
flat_load_sbyte	FLAT opcode 17	FLAT opcode 9
flat_load_sbyte_d16	FLAT opcode 34	*new*
flat_load_sbyte_d16_hi	FLAT opcode 35	*new*
flat_load_short_d16	FLAT opcode 36	*new*
flat_load_short_d16_hi	FLAT opcode 37	*new*
flat_load_sshort	FLAT opcode 19	FLAT opcode 11
flat_load_ubyte	FLAT opcode 16	FLAT opcode 8
flat_load_ubyte_d16	FLAT opcode 32	*new*
flat_load_ubyte_d16_hi	FLAT opcode 33	*new*
flat_load_ushort	FLAT opcode 18	FLAT opcode 10
flat_store_byte_d16_hi	FLAT opcode 25	*new*
flat_store_dwordx3	FLAT opcode 30	FLAT opcode 31
flat_store_dwordx4	FLAT opcode 31	FLAT opcode 30
flat_store_short_d16_hi	FLAT opcode 27	*new*
global_atomic_add	FLAT opcode 66	*new*
global_atomic_add_x2	FLAT opcode 98	*new*
global_atomic_and	FLAT opcode 72	*new*
global_atomic_and_x2	FLAT opcode 104	*new*
global_atomic_cmpswap	FLAT opcode 65	*new*
global_atomic_cmpswap_x2	FLAT opcode 97	*new*
global_atomic_dec	FLAT opcode 76	*new*
global_atomic_dec_x2	FLAT opcode 108	*new*
global_atomic_inc	FLAT opcode 75	*new*
global_atomic_inc_x2	FLAT opcode 107	*new*
global_atomic_or	FLAT opcode 73	*new*
global_atomic_or_x2	FLAT opcode 105	*new*

Changes From gfx75 Until Current (continued)

Opcode	Current	gfx75
global_atomic_smax	FLAT opcode 70	*new*
global_atomic_smax_x2	FLAT opcode 102	*new*
global_atomic_smin	FLAT opcode 68	*new*
global_atomic_smin_x2	FLAT opcode 100	*new*
global_atomic_sub	FLAT opcode 67	*new*
global_atomic_sub_x2	FLAT opcode 99	*new*
global_atomic_swap	FLAT opcode 64	*new*
global_atomic_swap_x2	FLAT opcode 96	*new*
global_atomic_umax	FLAT opcode 71	*new*
global_atomic_umax_x2	FLAT opcode 103	*new*
global_atomic_umin	FLAT opcode 69	*new*
global_atomic_umin_x2	FLAT opcode 101	*new*
global_atomic_xor	FLAT opcode 74	*new*
global_atomic_xor_x2	FLAT opcode 106	*new*
global_load_dword	FLAT opcode 20	*new*
global_load_dwordx2	FLAT opcode 21	*new*
global_load_dwordx3	FLAT opcode 22	*new*
global_load_dwordx4	FLAT opcode 23	*new*
global_load_sbyte	FLAT opcode 17	*new*
global_load_sbyte_d16	FLAT opcode 34	*new*
global_load_sbyte_d16_hi	FLAT opcode 35	*new*
global_load_short_d16	FLAT opcode 36	*new*
global_load_short_d16_hi	FLAT opcode 37	*new*
global_load_sshort	FLAT opcode 19	*new*
global_load_ubyte	FLAT opcode 16	*new*
global_load_ubyte_d16	FLAT opcode 32	*new*
global_load_ubyte_d16_hi	FLAT opcode 33	*new*
global_load_ushort	FLAT opcode 18	*new*
global_store_byte	FLAT opcode 24	*new*
global_store_byte_d16_hi	FLAT opcode 25	*new*
global_store_dword	FLAT opcode 28	*new*
global_store_dwordx2	FLAT opcode 29	*new*
global_store_dwordx3	FLAT opcode 30	*new*
global_store_dwordx4	FLAT opcode 31	*new*
global_store_short	FLAT opcode 26	*new*
global_store_short_d16_hi	FLAT opcode 27	*new*
image_atomic_add	MIMG opcode 18	MIMG opcode 17
image_atomic_cmpswap	MIMG opcode 17	MIMG opcode 16
image_atomic_fcmpswap	*deleted*	MIMG opcode 29
image_atomic_fmax	*deleted*	MIMG opcode 31
image_atomic_fmin	*deleted*	MIMG opcode 30
image_atomic_sub	MIMG opcode 19	MIMG opcode 18
image_atomic_swap	MIMG opcode 16	MIMG opcode 15
image_gather4h	MIMG opcode 66	*new*
image_gather4h_pck	MIMG opcode 74	*new*
image_gather8h_pck	MIMG opcode 75	*new*
s_abs_i32	SOP1 opcode 48	SOP1 opcode 52
s_absdiff_i32	SOP2 opcode 42	SOP2 opcode 44
s_addk_i32	SOPK opcode 14	SOPK opcode 15
s_and_b32	SOP2 opcode 12	SOP2 opcode 14
s_and_b64	SOP2 opcode 13	SOP2 opcode 15
s_and_saveexec_b64	SOP1 opcode 32	SOP1 opcode 36

Changes From gfx75 Until Current (continued)

Opcode	Current	gfx75
s_andn1_saveexec_b64	SOP1 opcode 51	*new*
s_andn1_wrexec_b64	SOP1 opcode 53	*new*
s_andn2_b32	SOP2 opcode 18	SOP2 opcode 20
s_andn2_b64	SOP2 opcode 19	SOP2 opcode 21
s_andn2_saveexec_b64	SOP1 opcode 35	SOP1 opcode 39
s_andn2_wrexec_b64	SOP1 opcode 54	*new*
s_ashr_i32	SOP2 opcode 32	SOP2 opcode 34
s_ashr_i64	SOP2 opcode 33	SOP2 opcode 35
s_atc_probe	SMEM opcode 38	*new*
s_atc_probe_buffer	SMEM opcode 39	*new*
s_atomic_add	SMEM opcode 130	*new*
s_atomic_add_x2	SMEM opcode 162	*new*
s_atomic_and	SMEM opcode 136	*new*
s_atomic_and_x2	SMEM opcode 168	*new*
s_atomic_cmpswap	SMEM opcode 129	*new*
s_atomic_cmpswap_x2	SMEM opcode 161	*new*
s_atomic_dec	SMEM opcode 140	*new*
s_atomic_dec_x2	SMEM opcode 172	*new*
s_atomic_inc	SMEM opcode 139	*new*
s_atomic_inc_x2	SMEM opcode 171	*new*
s_atomic_or	SMEM opcode 137	*new*
s_atomic_or_x2	SMEM opcode 169	*new*
s_atomic_smax	SMEM opcode 134	*new*
s_atomic_smax_x2	SMEM opcode 166	*new*
s_atomic_smin	SMEM opcode 132	*new*
s_atomic_smin_x2	SMEM opcode 164	*new*
s_atomic_sub	SMEM opcode 131	*new*
s_atomic_sub_x2	SMEM opcode 163	*new*
s_atomic_swap	SMEM opcode 128	*new*
s_atomic_swap_x2	SMEM opcode 160	*new*
s_atomic_umax	SMEM opcode 135	*new*
s_atomic_umax_x2	SMEM opcode 167	*new*
s_atomic_umin	SMEM opcode 133	*new*
s_atomic_umin_x2	SMEM opcode 165	*new*
s_atomic_xor	SMEM opcode 138	*new*
s_atomic_xor_x2	SMEM opcode 170	*new*
s_bcnt0_i32_b32	SOP1 opcode 10	SOP1 opcode 13
s_bcnt0_i32_b64	SOP1 opcode 11	SOP1 opcode 14
s_bcnt1_i32_b32	SOP1 opcode 12	SOP1 opcode 15
s_bcnt1_i32_b64	SOP1 opcode 13	SOP1 opcode 16
s_bfe_i32	SOP2 opcode 38	SOP2 opcode 40
s_bfe_i64	SOP2 opcode 40	SOP2 opcode 42
s_bfe_u32	SOP2 opcode 37	SOP2 opcode 39
s_bfe_u64	SOP2 opcode 39	SOP2 opcode 41
s_bfm_b32	SOP2 opcode 34	SOP2 opcode 36
s_bfm_b64	SOP2 opcode 35	SOP2 opcode 37
s_bitreplicate_b64_b32	SOP1 opcode 55	*new*
s_bitset0_b32	SOP1 opcode 24	SOP1 opcode 27
s_bitset0_b64	SOP1 opcode 25	SOP1 opcode 28
s_bitset1_b32	SOP1 opcode 26	SOP1 opcode 29
s_bitset1_b64	SOP1 opcode 27	SOP1 opcode 30
s_brev_b32	SOP1 opcode 8	SOP1 opcode 11

Changes From gfx75 Until Current (continued)

Opcode	Current	gfx75
s_brev_b64	SOP1 opcode 9	SOP1 opcode 12
s_buffer_atomic_add	SMEM opcode 66	*new*
s_buffer_atomic_add_x2	SMEM opcode 98	*new*
s_buffer_atomic_and	SMEM opcode 72	*new*
s_buffer_atomic_and_x2	SMEM opcode 104	*new*
s_buffer_atomic_cmpswap	SMEM opcode 65	*new*
s_buffer_atomic_cmpswap_x2	SMEM opcode 97	*new*
s_buffer_atomic_dec	SMEM opcode 76	*new*
s_buffer_atomic_dec_x2	SMEM opcode 108	*new*
s_buffer_atomic_inc	SMEM opcode 75	*new*
s_buffer_atomic_inc_x2	SMEM opcode 107	*new*
s_buffer_atomic_or	SMEM opcode 73	*new*
s_buffer_atomic_or_x2	SMEM opcode 105	*new*
s_buffer_atomic_smax	SMEM opcode 70	*new*
s_buffer_atomic_smax_x2	SMEM opcode 102	*new*
s_buffer_atomic_smin	SMEM opcode 68	*new*
s_buffer_atomic_smin_x2	SMEM opcode 100	*new*
s_buffer_atomic_sub	SMEM opcode 67	*new*
s_buffer_atomic_sub_x2	SMEM opcode 99	*new*
s_buffer_atomic_swap	SMEM opcode 64	*new*
s_buffer_atomic_swap_x2	SMEM opcode 96	*new*
s_buffer_atomic_umax	SMEM opcode 71	*new*
s_buffer_atomic_umax_x2	SMEM opcode 103	*new*
s_buffer_atomic_umin	SMEM opcode 69	*new*
s_buffer_atomic_umin_x2	SMEM opcode 101	*new*
s_buffer_atomic_xor	SMEM opcode 74	*new*
s_buffer_atomic_xor_x2	SMEM opcode 106	*new*
s_buffer_load_dword	SMEM opcode 8	SMRD opcode 8
s_buffer_load_dwordx16	SMEM opcode 12	SMRD opcode 12
s_buffer_load_dwordx2	SMEM opcode 9	SMRD opcode 9
s_buffer_load_dwordx4	SMEM opcode 10	SMRD opcode 10
s_buffer_load_dwordx8	SMEM opcode 11	SMRD opcode 11
s_buffer_store_dword	SMEM opcode 24	*new*
s_buffer_store_dwordx2	SMEM opcode 25	*new*
s_buffer_store_dwordx4	SMEM opcode 26	*new*
s_call_b64	SOPK opcode 21	*new*
s_cbranch_g_fork	SOP2 opcode 41	SOP2 opcode 43
s_cbranch_i_fork	SOPK opcode 16	SOPK opcode 17
s_cbranch_join	SOP1 opcode 46	SOP1 opcode 50
s_cmov_b32	SOP1 opcode 2	SOP1 opcode 5
s_cmov_b64	SOP1 opcode 3	SOP1 opcode 6
s_cmovk_i32	SOPK opcode 1	SOPK opcode 2
s_cmp_eq_u64	SOPC opcode 18	*new*
s_cmp_lg_u64	SOPC opcode 19	*new*
s_cmpk_eq_i32	SOPK opcode 2	SOPK opcode 3
s_cmpk_eq_u32	SOPK opcode 8	SOPK opcode 9
s_cmpk_ge_i32	SOPK opcode 5	SOPK opcode 6
s_cmpk_ge_u32	SOPK opcode 11	SOPK opcode 12
s_cmpk_gt_i32	SOPK opcode 4	SOPK opcode 5
s_cmpk_gt_u32	SOPK opcode 10	SOPK opcode 11
s_cmpk_le_i32	SOPK opcode 7	SOPK opcode 8
s_cmpk_le_u32	SOPK opcode 13	SOPK opcode 14

Changes From gfx75 Until Current (continued)

Opcode	Current	gfx75
s_cmpk_lg_i32	SOPK opcode 3	SOPK opcode 4
s_cmpk_lg_u32	SOPK opcode 9	SOPK opcode 10
s_cmpk_lt_i32	SOPK opcode 6	SOPK opcode 7
s_cmpk_lt_u32	SOPK opcode 12	SOPK opcode 13
s_dcache_discard	SMEM opcode 40	*new*
s_dcache_discard_x2	SMEM opcode 41	*new*
s_dcache_inv	SMEM opcode 32	SMRD opcode 31
s_dcache_inv_vol	SMEM opcode 34	SMRD opcode 29
s_dcache_wb	SMEM opcode 33	*new*
s_dcache_wb_vol	SMEM opcode 35	*new*
s_endpgm_ordered_ps_done	SOPP opcode 30	*new*
s_endpgm_saved	SOPP opcode 27	*new*
s_ff0_i32_b32	SOP1 opcode 14	SOP1 opcode 17
s_ff0_i32_b64	SOP1 opcode 15	SOP1 opcode 18
s_ff1_i32_b32	SOP1 opcode 16	SOP1 opcode 19
s_ff1_i32_b64	SOP1 opcode 17	SOP1 opcode 20
s_flbit_i32	SOP1 opcode 20	SOP1 opcode 23
s_flbit_i32_b32	SOP1 opcode 18	SOP1 opcode 21
s_flbit_i32_b64	SOP1 opcode 19	SOP1 opcode 22
s_flbit_i32_i64	SOP1 opcode 21	SOP1 opcode 24
s_getpc_b64	SOP1 opcode 28	SOP1 opcode 31
s_getreg_b32	SOPK opcode 17	SOPK opcode 18
s_load_dword	SMEM opcode 0	SMRD opcode 0
s_load_dwordx16	SMEM opcode 4	SMRD opcode 4
s_load_dwordx2	SMEM opcode 1	SMRD opcode 1
s_load_dwordx4	SMEM opcode 2	SMRD opcode 2
s_load_dwordx8	SMEM opcode 3	SMRD opcode 3
s_lshl1_add_u32	SOP2 opcode 46	*new*
s_lshl2_add_u32	SOP2 opcode 47	*new*
s_lshl3_add_u32	SOP2 opcode 48	*new*
s_lshl4_add_u32	SOP2 opcode 49	*new*
s_lshl_b32	SOP2 opcode 28	SOP2 opcode 30
s_lshl_b64	SOP2 opcode 29	SOP2 opcode 31
s_lshr_b32	SOP2 opcode 30	SOP2 opcode 32
s_lshr_b64	SOP2 opcode 31	SOP2 opcode 33
s_memrealtime	SMEM opcode 37	SMRD opcode 28
s_memtime	SMEM opcode 36	SMRD opcode 30
s_mov_b32	SOP1 opcode 0	SOP1 opcode 3
s_mov_b64	SOP1 opcode 1	SOP1 opcode 4
s_mov_fed_b32	SOP1 opcode 49	SOP1 opcode 53
s_movreld_b32	SOP1 opcode 44	SOP1 opcode 48
s_movreld_b64	SOP1 opcode 45	SOP1 opcode 49
s_movrels_b32	SOP1 opcode 42	SOP1 opcode 46
s_movrels_b64	SOP1 opcode 43	SOP1 opcode 47
s_mul_hi_i32	SOP2 opcode 45	*new*
s_mul_hi_u32	SOP2 opcode 44	*new*
s_mul_i32	SOP2 opcode 36	SOP2 opcode 38
s_mulk_i32	SOPK opcode 15	SOPK opcode 16
s_nand_b32	SOP2 opcode 22	SOP2 opcode 24
s_nand_b64	SOP2 opcode 23	SOP2 opcode 25
s_nand_saveexec_b64	SOP1 opcode 37	SOP1 opcode 41
s_nor_b32	SOP2 opcode 24	SOP2 opcode 26

Changes From gfx75 Until Current (continued)

Opcode	Current	gfx75
s_nor_b64	SOP2 opcode 25	SOP2 opcode 27
s_nor_saveexec_b64	SOP1 opcode 38	SOP1 opcode 42
s_not_b32	SOP1 opcode 4	SOP1 opcode 7
s_not_b64	SOP1 opcode 5	SOP1 opcode 8
s_or_b32	SOP2 opcode 14	SOP2 opcode 16
s_or_b64	SOP2 opcode 15	SOP2 opcode 17
s_or_saveexec_b64	SOP1 opcode 33	SOP1 opcode 37
s_orn1_saveexec_b64	SOP1 opcode 52	*new*
s_orn2_b32	SOP2 opcode 20	SOP2 opcode 22
s_orn2_b64	SOP2 opcode 21	SOP2 opcode 23
s_orn2_saveexec_b64	SOP1 opcode 36	SOP1 opcode 40
s_quadmask_b32	SOP1 opcode 40	SOP1 opcode 44
s_quadmask_b64	SOP1 opcode 41	SOP1 opcode 45
s_rfe_b64	SOP1 opcode 31	SOP1 opcode 34
s_rfe_restore_b64	SOP2 opcode 43	*new*
s_scratch_load_dword	SMEM opcode 5	*new*
s_scratch_load_dwordx2	SMEM opcode 6	*new*
s_scratch_load_dwordx4	SMEM opcode 7	*new*
s_scratch_store_dword	SMEM opcode 21	*new*
s_scratch_store_dwordx2	SMEM opcode 22	*new*
s_scratch_store_dwordx4	SMEM opcode 23	*new*
s_set_gpr_idx_idx	SOP1 opcode 50	*new*
s_set_gpr_idx_mode	SOPP opcode 29	*new*
s_set_gpr_idx_off	SOPP opcode 28	*new*
s_set_gpr_idx_on	SOPC opcode 17	*new*
s_setpc_b64	SOP1 opcode 29	SOP1 opcode 32
s_setreg_b32	SOPK opcode 18	SOPK opcode 19
s_setreg_imm32_b32	SOPK opcode 20	SOPK opcode 21
s_sext_i32_i16	SOP1 opcode 23	SOP1 opcode 26
s_sext_i32_i8	SOP1 opcode 22	SOP1 opcode 25
s_store_dword	SMEM opcode 16	*new*
s_store_dwordx2	SMEM opcode 17	*new*
s_store_dwordx4	SMEM opcode 18	*new*
s_swappc_b64	SOP1 opcode 30	SOP1 opcode 33
s_wakeup	SOPP opcode 3	*new*
s_wqm_b32	SOP1 opcode 6	SOP1 opcode 9
s_wqm_b64	SOP1 opcode 7	SOP1 opcode 10
s_xnor_b32	SOP2 opcode 26	SOP2 opcode 28
s_xnor_b64	SOP2 opcode 27	SOP2 opcode 29
s_xnor_saveexec_b64	SOP1 opcode 39	SOP1 opcode 43
s_xor_b32	SOP2 opcode 16	SOP2 opcode 18
s_xor_b64	SOP2 opcode 17	SOP2 opcode 19
s_xor_saveexec_b64	SOP1 opcode 34	SOP1 opcode 38
scratch_load_dword	FLAT opcode 20	*new*
scratch_load_dwordx2	FLAT opcode 21	*new*
scratch_load_dwordx3	FLAT opcode 22	*new*
scratch_load_dwordx4	FLAT opcode 23	*new*
scratch_load_sbyte	FLAT opcode 17	*new*
scratch_load_sbyte_d16	FLAT opcode 34	*new*
scratch_load_sbyte_d16_hi	FLAT opcode 35	*new*
scratch_load_short_d16	FLAT opcode 36	*new*
scratch_load_short_d16_hi	FLAT opcode 37	*new*

Changes From gfx75 Until Current (continued)

Opcode	Current	gfx75
scratch_load_sshort	FLAT opcode 19	*new*
scratch_load_ubyte	FLAT opcode 16	*new*
scratch_load_ubyte_d16	FLAT opcode 32	*new*
scratch_load_ubyte_d16_hi	FLAT opcode 33	*new*
scratch_load_ushort	FLAT opcode 18	*new*
scratch_store_byte	FLAT opcode 24	*new*
scratch_store_byte_d16_hi	FLAT opcode 25	*new*
scratch_store_dword	FLAT opcode 28	*new*
scratch_store_dwordx2	FLAT opcode 29	*new*
scratch_store_dwordx3	FLAT opcode 30	*new*
scratch_store_dwordx4	FLAT opcode 31	*new*
scratch_store_short	FLAT opcode 26	*new*
scratch_store_short_d16_hi	FLAT opcode 27	*new*
v_add3_u32	VOP3 opcode 511	VOP3 opcode 877
v_add_co_u32	VOP2 opcode 25	*new*
v_add_f16	VOP2 opcode 31	VOP2 opcode 50
v_add_f32	VOP2 opcode 1	VOP2 opcode 3
v_add_f64	VOP3 opcode 640	VOP3 opcode 356
v_add_i16	VOP3 opcode 670	VOP3 opcode 781
v_add_i32	VOP3 opcode 668	VOP2 opcode 37
v_add_lshl_u32	VOP3 opcode 510	VOP3 opcode 839
v_add_u16	VOP2 opcode 38	VOP3 opcode 771
v_add_u32	VOP2 opcode 52	VOP3 opcode 783
v_addc_co_u32	VOP2 opcode 28	*new*
v_addc_u32	*deleted*	VOP2 opcode 40
v_alignbit_b32	VOP3 opcode 462	VOP3 opcode 334
v_alignbyte_b32	VOP3 opcode 463	VOP3 opcode 335
v_and_b32	VOP2 opcode 19	VOP2 opcode 27
v_and_or_b32	VOP3 opcode 513	VOP3 opcode 881
v_ashr_i32	*deleted*	VOP2 opcode 23
v_ashr_i64	*deleted*	VOP3 opcode 355
v_ashrrev_i16	VOP2 opcode 44	VOP3 opcode 776
v_ashrrev_i32	VOP2 opcode 17	VOP2 opcode 24
v_ashrrev_i64	VOP3 opcode 657	*new*
v_bcnt_u32_b32	VOP3 opcode 651	VOP2 opcode 34
v_bfe_i32	VOP3 opcode 457	VOP3 opcode 329
v_bfe_u32	VOP3 opcode 456	VOP3 opcode 328
v_bfi_b32	VOP3 opcode 458	VOP3 opcode 330
v_bfm_b32	VOP3 opcode 659	VOP2 opcode 30
v_bfrev_b32	VOP1 opcode 44	VOP1 opcode 56
v_ceil_f16	VOP1 opcode 69	VOP1 opcode 92
v_ceil_f32	VOP1 opcode 29	VOP1 opcode 34
v_clrexcp	VOP1 opcode 53	VOP1 opcode 65
v_cmp_class_f16	VOPC opcode 20	VOPC opcode 143
v_cmp_class_f32	VOPC opcode 16	VOPC opcode 136
v_cmp_class_f64	VOPC opcode 18	VOPC opcode 168
v_cmp_eq_f16	VOPC opcode 34	VOPC opcode 202
v_cmp_eq_f32	VOPC opcode 66	VOPC opcode 2
v_cmp_eq_f64	VOPC opcode 98	VOPC opcode 34
v_cmp_eq_i16	VOPC opcode 162	VOPC opcode 138
v_cmp_eq_i32	VOPC opcode 194	VOPC opcode 130
v_cmp_eq_i64	VOPC opcode 226	VOPC opcode 162

Changes From gfx75 Until Current (continued)

Opcode	Current	gfx75
v_cmp_eq_u32	VOPC opcode 202	VOPC opcode 194
v_cmp_eq_u64	VOPC opcode 234	VOPC opcode 226
v_cmp_f_f16	VOPC opcode 32	VOPC opcode 200
v_cmp_f_f32	VOPC opcode 64	VOPC opcode 0
v_cmp_f_f64	VOPC opcode 96	VOPC opcode 32
v_cmp_f_i16	VOPC opcode 160	*new*
v_cmp_f_i32	VOPC opcode 192	VOPC opcode 128
v_cmp_f_i64	VOPC opcode 224	VOPC opcode 160
v_cmp_f_u16	VOPC opcode 168	*new*
v_cmp_f_u32	VOPC opcode 200	VOPC opcode 192
v_cmp_f_u64	VOPC opcode 232	VOPC opcode 224
v_cmp_ge_f16	VOPC opcode 38	VOPC opcode 206
v_cmp_ge_f32	VOPC opcode 70	VOPC opcode 6
v_cmp_ge_f64	VOPC opcode 102	VOPC opcode 38
v_cmp_ge_i16	VOPC opcode 166	VOPC opcode 142
v_cmp_ge_i32	VOPC opcode 198	VOPC opcode 134
v_cmp_ge_i64	VOPC opcode 230	VOPC opcode 166
v_cmp_ge_u32	VOPC opcode 206	VOPC opcode 198
v_cmp_ge_u64	VOPC opcode 238	VOPC opcode 230
v_cmp_gt_f16	VOPC opcode 36	VOPC opcode 204
v_cmp_gt_f32	VOPC opcode 68	VOPC opcode 4
v_cmp_gt_f64	VOPC opcode 100	VOPC opcode 36
v_cmp_gt_i16	VOPC opcode 164	VOPC opcode 140
v_cmp_gt_i32	VOPC opcode 196	VOPC opcode 132
v_cmp_gt_i64	VOPC opcode 228	VOPC opcode 164
v_cmp_gt_u32	VOPC opcode 204	VOPC opcode 196
v_cmp_gt_u64	VOPC opcode 236	VOPC opcode 228
v_cmp_le_f16	VOPC opcode 35	VOPC opcode 203
v_cmp_le_f32	VOPC opcode 67	VOPC opcode 3
v_cmp_le_f64	VOPC opcode 99	VOPC opcode 35
v_cmp_le_i16	VOPC opcode 163	VOPC opcode 139
v_cmp_le_i32	VOPC opcode 195	VOPC opcode 131
v_cmp_le_i64	VOPC opcode 227	VOPC opcode 163
v_cmp_le_u32	VOPC opcode 203	VOPC opcode 195
v_cmp_le_u64	VOPC opcode 235	VOPC opcode 227
v_cmp_lg_f16	VOPC opcode 37	VOPC opcode 205
v_cmp_lg_f32	VOPC opcode 69	VOPC opcode 5
v_cmp_lg_f64	VOPC opcode 101	VOPC opcode 37
v_cmp_lt_f16	VOPC opcode 33	VOPC opcode 201
v_cmp_lt_f32	VOPC opcode 65	VOPC opcode 1
v_cmp_lt_f64	VOPC opcode 97	VOPC opcode 33
v_cmp_lt_i16	VOPC opcode 161	VOPC opcode 137
v_cmp_lt_i32	VOPC opcode 193	VOPC opcode 129
v_cmp_lt_i64	VOPC opcode 225	VOPC opcode 161
v_cmp_lt_u32	VOPC opcode 201	VOPC opcode 193
v_cmp_lt_u64	VOPC opcode 233	VOPC opcode 225
v_cmp_ne_i16	VOPC opcode 165	VOPC opcode 141
v_cmp_ne_i32	VOPC opcode 197	VOPC opcode 133
v_cmp_ne_i64	VOPC opcode 229	VOPC opcode 165
v_cmp_ne_u32	VOPC opcode 205	VOPC opcode 197
v_cmp_ne_u64	VOPC opcode 237	VOPC opcode 229
v_cmp_neq_f16	VOPC opcode 45	VOPC opcode 237

Changes From gfx75 Until Current (continued)

Opcode	Current	gfx75
v_cmp_neq_f32	VOPC opcode 77	VOPC opcode 13
v_cmp_neq_f64	VOPC opcode 109	VOPC opcode 45
v_cmp_nge_f16	VOPC opcode 41	VOPC opcode 233
v_cmp_nge_f32	VOPC opcode 73	VOPC opcode 9
v_cmp_nge_f64	VOPC opcode 105	VOPC opcode 41
v_cmp_ngt_f16	VOPC opcode 43	VOPC opcode 235
v_cmp_ngt_f32	VOPC opcode 75	VOPC opcode 11
v_cmp_ngt_f64	VOPC opcode 107	VOPC opcode 43
v_cmp_nle_f16	VOPC opcode 44	VOPC opcode 236
v_cmp_nle_f32	VOPC opcode 76	VOPC opcode 12
v_cmp_nle_f64	VOPC opcode 108	VOPC opcode 44
v_cmp_nlg_f16	VOPC opcode 42	VOPC opcode 234
v_cmp_nlg_f32	VOPC opcode 74	VOPC opcode 10
v_cmp_nlg_f64	VOPC opcode 106	VOPC opcode 42
v_cmp_nlt_f16	VOPC opcode 46	VOPC opcode 238
v_cmp_nlt_f32	VOPC opcode 78	VOPC opcode 14
v_cmp_nlt_f64	VOPC opcode 110	VOPC opcode 46
v_cmp_o_f16	VOPC opcode 39	VOPC opcode 207
v_cmp_o_f32	VOPC opcode 71	VOPC opcode 7
v_cmp_o_f64	VOPC opcode 103	VOPC opcode 39
v_cmp_t_i16	VOPC opcode 167	*new*
v_cmp_t_i32	VOPC opcode 199	VOPC opcode 135
v_cmp_t_i64	VOPC opcode 231	VOPC opcode 167
v_cmp_t_u16	VOPC opcode 175	*new*
v_cmp_t_u32	VOPC opcode 207	VOPC opcode 199
v_cmp_t_u64	VOPC opcode 239	VOPC opcode 231
v_cmp_tru_f16	VOPC opcode 47	VOPC opcode 239
v_cmp_tru_f32	VOPC opcode 79	VOPC opcode 15
v_cmp_tru_f64	VOPC opcode 111	VOPC opcode 47
v_cmp_u_f16	VOPC opcode 40	VOPC opcode 232
v_cmp_u_f32	VOPC opcode 72	VOPC opcode 8
v_cmp_u_f64	VOPC opcode 104	VOPC opcode 40
v_cmps_eq_f32	*deleted*	VOPC opcode 66
v_cmps_eq_f64	*deleted*	VOPC opcode 98
v_cmps_f_f32	*deleted*	VOPC opcode 64
v_cmps_f_f64	*deleted*	VOPC opcode 96
v_cmps_ge_f32	*deleted*	VOPC opcode 70
v_cmps_ge_f64	*deleted*	VOPC opcode 102
v_cmps_gt_f32	*deleted*	VOPC opcode 68
v_cmps_gt_f64	*deleted*	VOPC opcode 100
v_cmps_le_f32	*deleted*	VOPC opcode 67
v_cmps_le_f64	*deleted*	VOPC opcode 99
v_cmps_lg_f32	*deleted*	VOPC opcode 69
v_cmps_lg_f64	*deleted*	VOPC opcode 101
v_cmps_lt_f32	*deleted*	VOPC opcode 65
v_cmps_lt_f64	*deleted*	VOPC opcode 97
v_cmps_neq_f32	*deleted*	VOPC opcode 77
v_cmps_neq_f64	*deleted*	VOPC opcode 109
v_cmps_nge_f32	*deleted*	VOPC opcode 73
v_cmps_nge_f64	*deleted*	VOPC opcode 105
v_cmps_ngt_f32	*deleted*	VOPC opcode 75
v_cmps_ngt_f64	*deleted*	VOPC opcode 107

Changes From gfx75 Until Current (continued)

Opcode	Current	gfx75
v_cmps_nle_f32	*deleted*	VOPC opcode 76
v_cmps_nle_f64	*deleted*	VOPC opcode 108
v_cmps_nlg_f32	*deleted*	VOPC opcode 74
v_cmps_nlg_f64	*deleted*	VOPC opcode 106
v_cmps_nlt_f32	*deleted*	VOPC opcode 78
v_cmps_nlt_f64	*deleted*	VOPC opcode 110
v_cmps_o_f32	*deleted*	VOPC opcode 71
v_cmps_o_f64	*deleted*	VOPC opcode 103
v_cmps_tru_f32	*deleted*	VOPC opcode 79
v_cmps_tru_f64	*deleted*	VOPC opcode 111
v_cmps_u_f32	*deleted*	VOPC opcode 72
v_cmps_u_f64	*deleted*	VOPC opcode 104
v_cmpsx_eq_f32	*deleted*	VOPC opcode 82
v_cmpsx_eq_f64	*deleted*	VOPC opcode 114
v_cmpsx_f_f32	*deleted*	VOPC opcode 80
v_cmpsx_f_f64	*deleted*	VOPC opcode 112
v_cmpsx_ge_f32	*deleted*	VOPC opcode 86
v_cmpsx_ge_f64	*deleted*	VOPC opcode 118
v_cmpsx_gt_f32	*deleted*	VOPC opcode 84
v_cmpsx_gt_f64	*deleted*	VOPC opcode 116
v_cmpsx_le_f32	*deleted*	VOPC opcode 83
v_cmpsx_le_f64	*deleted*	VOPC opcode 115
v_cmpsx_lg_f32	*deleted*	VOPC opcode 85
v_cmpsx_lg_f64	*deleted*	VOPC opcode 117
v_cmpsx_lt_f32	*deleted*	VOPC opcode 81
v_cmpsx_lt_f64	*deleted*	VOPC opcode 113
v_cmpsx_neq_f32	*deleted*	VOPC opcode 93
v_cmpsx_neq_f64	*deleted*	VOPC opcode 125
v_cmpsx_nge_f32	*deleted*	VOPC opcode 89
v_cmpsx_nge_f64	*deleted*	VOPC opcode 121
v_cmpsx_ngt_f32	*deleted*	VOPC opcode 91
v_cmpsx_ngt_f64	*deleted*	VOPC opcode 123
v_cmpsx_nle_f32	*deleted*	VOPC opcode 92
v_cmpsx_nle_f64	*deleted*	VOPC opcode 124
v_cmpsx_nlg_f32	*deleted*	VOPC opcode 90
v_cmpsx_nlg_f64	*deleted*	VOPC opcode 122
v_cmpsx_nlt_f32	*deleted*	VOPC opcode 94
v_cmpsx_nlt_f64	*deleted*	VOPC opcode 126
v_cmpsx_o_f32	*deleted*	VOPC opcode 87
v_cmpsx_o_f64	*deleted*	VOPC opcode 119
v_cmpsx_tru_f32	*deleted*	VOPC opcode 95
v_cmpsx_tru_f64	*deleted*	VOPC opcode 127
v_cmpsx_u_f32	*deleted*	VOPC opcode 88
v_cmpsx_u_f64	*deleted*	VOPC opcode 120
v_cmpx_class_f16	VOPC opcode 21	VOPC opcode 159
v_cmpx_class_f32	VOPC opcode 17	VOPC opcode 152
v_cmpx_class_f64	VOPC opcode 19	VOPC opcode 184
v_cmpx_eq_f16	VOPC opcode 50	VOPC opcode 218
v_cmpx_eq_f32	VOPC opcode 82	VOPC opcode 18
v_cmpx_eq_f64	VOPC opcode 114	VOPC opcode 50
v_cmpx_eq_i16	VOPC opcode 178	VOPC opcode 154
v_cmpx_eq_i32	VOPC opcode 210	VOPC opcode 146

Changes From gfx75 Until Current (continued)

Opcode	Current	gfx75
v_cmpx_eq_i64	VOPC opcode 242	VOPC opcode 178
v_cmpx_eq_u32	VOPC opcode 218	VOPC opcode 210
v_cmpx_eq_u64	VOPC opcode 250	VOPC opcode 242
v_cmpx_f_f16	VOPC opcode 48	VOPC opcode 216
v_cmpx_f_f32	VOPC opcode 80	VOPC opcode 16
v_cmpx_f_f64	VOPC opcode 112	VOPC opcode 48
v_cmpx_f_i16	VOPC opcode 176	*new*
v_cmpx_f_i32	VOPC opcode 208	VOPC opcode 144
v_cmpx_f_i64	VOPC opcode 240	VOPC opcode 176
v_cmpx_f_u16	VOPC opcode 184	*new*
v_cmpx_f_u32	VOPC opcode 216	VOPC opcode 208
v_cmpx_f_u64	VOPC opcode 248	VOPC opcode 240
v_cmpx_ge_f16	VOPC opcode 54	VOPC opcode 222
v_cmpx_ge_f32	VOPC opcode 86	VOPC opcode 22
v_cmpx_ge_f64	VOPC opcode 118	VOPC opcode 54
v_cmpx_ge_i16	VOPC opcode 182	VOPC opcode 158
v_cmpx_ge_i32	VOPC opcode 214	VOPC opcode 150
v_cmpx_ge_i64	VOPC opcode 246	VOPC opcode 182
v_cmpx_ge_u32	VOPC opcode 222	VOPC opcode 214
v_cmpx_ge_u64	VOPC opcode 254	VOPC opcode 246
v_cmpx_gt_f16	VOPC opcode 52	VOPC opcode 220
v_cmpx_gt_f32	VOPC opcode 84	VOPC opcode 20
v_cmpx_gt_f64	VOPC opcode 116	VOPC opcode 52
v_cmpx_gt_i16	VOPC opcode 180	VOPC opcode 156
v_cmpx_gt_i32	VOPC opcode 212	VOPC opcode 148
v_cmpx_gt_i64	VOPC opcode 244	VOPC opcode 180
v_cmpx_gt_u32	VOPC opcode 220	VOPC opcode 212
v_cmpx_gt_u64	VOPC opcode 252	VOPC opcode 244
v_cmpx_le_f16	VOPC opcode 51	VOPC opcode 219
v_cmpx_le_f32	VOPC opcode 83	VOPC opcode 19
v_cmpx_le_f64	VOPC opcode 115	VOPC opcode 51
v_cmpx_le_i16	VOPC opcode 179	VOPC opcode 155
v_cmpx_le_i32	VOPC opcode 211	VOPC opcode 147
v_cmpx_le_i64	VOPC opcode 243	VOPC opcode 179
v_cmpx_le_u32	VOPC opcode 219	VOPC opcode 211
v_cmpx_le_u64	VOPC opcode 251	VOPC opcode 243
v_cmpx_lg_f16	VOPC opcode 53	VOPC opcode 221
v_cmpx_lg_f32	VOPC opcode 85	VOPC opcode 21
v_cmpx_lg_f64	VOPC opcode 117	VOPC opcode 53
v_cmpx_lt_f16	VOPC opcode 49	VOPC opcode 217
v_cmpx_lt_f32	VOPC opcode 81	VOPC opcode 17
v_cmpx_lt_f64	VOPC opcode 113	VOPC opcode 49
v_cmpx_lt_i16	VOPC opcode 177	VOPC opcode 153
v_cmpx_lt_i32	VOPC opcode 209	VOPC opcode 145
v_cmpx_lt_i64	VOPC opcode 241	VOPC opcode 177
v_cmpx_lt_u32	VOPC opcode 217	VOPC opcode 209
v_cmpx_lt_u64	VOPC opcode 249	VOPC opcode 241
v_cmpx_ne_i16	VOPC opcode 181	VOPC opcode 157
v_cmpx_ne_i32	VOPC opcode 213	VOPC opcode 149
v_cmpx_ne_i64	VOPC opcode 245	VOPC opcode 181
v_cmpx_ne_u32	VOPC opcode 221	VOPC opcode 213
v_cmpx_ne_u64	VOPC opcode 253	VOPC opcode 245

Changes From gfx75 Until Current (continued)

Opcode	Current	gfx75
v_cmpx_neq_f16	VOPC opcode 61	VOPC opcode 253
v_cmpx_neq_f32	VOPC opcode 93	VOPC opcode 29
v_cmpx_neq_f64	VOPC opcode 125	VOPC opcode 61
v_cmpx_nge_f16	VOPC opcode 57	VOPC opcode 249
v_cmpx_nge_f32	VOPC opcode 89	VOPC opcode 25
v_cmpx_nge_f64	VOPC opcode 121	VOPC opcode 57
v_cmpx_ngt_f16	VOPC opcode 59	VOPC opcode 251
v_cmpx_ngt_f32	VOPC opcode 91	VOPC opcode 27
v_cmpx_ngt_f64	VOPC opcode 123	VOPC opcode 59
v_cmpx_nle_f16	VOPC opcode 60	VOPC opcode 252
v_cmpx_nle_f32	VOPC opcode 92	VOPC opcode 28
v_cmpx_nle_f64	VOPC opcode 124	VOPC opcode 60
v_cmpx_nlg_f16	VOPC opcode 58	VOPC opcode 250
v_cmpx_nlg_f32	VOPC opcode 90	VOPC opcode 26
v_cmpx_nlg_f64	VOPC opcode 122	VOPC opcode 58
v_cmpx_nlt_f16	VOPC opcode 62	VOPC opcode 254
v_cmpx_nlt_f32	VOPC opcode 94	VOPC opcode 30
v_cmpx_nlt_f64	VOPC opcode 126	VOPC opcode 62
v_cmpx_o_f16	VOPC opcode 55	VOPC opcode 223
v_cmpx_o_f32	VOPC opcode 87	VOPC opcode 23
v_cmpx_o_f64	VOPC opcode 119	VOPC opcode 55
v_cmpx_t_i16	VOPC opcode 183	*new*
v_cmpx_t_i32	VOPC opcode 215	VOPC opcode 151
v_cmpx_t_i64	VOPC opcode 247	VOPC opcode 183
v_cmpx_t_u16	VOPC opcode 191	*new*
v_cmpx_t_u32	VOPC opcode 223	VOPC opcode 215
v_cmpx_t_u64	VOPC opcode 255	VOPC opcode 247
v_cmpx_tru_f16	VOPC opcode 63	VOPC opcode 255
v_cmpx_tru_f32	VOPC opcode 95	VOPC opcode 31
v_cmpx_tru_f64	VOPC opcode 127	VOPC opcode 63
v_cmpx_u_f16	VOPC opcode 56	VOPC opcode 248
v_cmpx_u_f32	VOPC opcode 88	VOPC opcode 24
v_cmpx_u_f64	VOPC opcode 120	VOPC opcode 56
v_cos_f16	VOP1 opcode 74	VOP1 opcode 97
v_cos_f32	VOP1 opcode 42	VOP1 opcode 54
v_cubeid_f32	VOP3 opcode 452	VOP3 opcode 324
v_cubema_f32	VOP3 opcode 455	VOP3 opcode 327
v_cubesc_f32	VOP3 opcode 453	VOP3 opcode 325
v_cubetc_f32	VOP3 opcode 454	VOP3 opcode 326
v_cvt_f16_i16	VOP1 opcode 58	VOP1 opcode 81
v_cvt_f16_u16	VOP1 opcode 57	VOP1 opcode 80
v_cvt_i16_f16	VOP1 opcode 60	VOP1 opcode 83
v_cvt_norm_i16_f16	VOP1 opcode 77	VOP1 opcode 99
v_cvt_norm_u16_f16	VOP1 opcode 78	VOP1 opcode 100
v_cvt_pk_i16_i32	VOP3 opcode 664	VOP2 opcode 49
v_cvt_pk_u16_u32	VOP3 opcode 663	VOP2 opcode 48
v_cvt_pk_u8_f32	VOP3 opcode 477	VOP3 opcode 350
v_cvt_pkaccum_u8_f32	VOP3 opcode 496	VOP2 opcode 44
v_cvt_pknorm_i16_f16	VOP3 opcode 665	VOP3 opcode 786
v_cvt_pknorm_i16_f32	VOP3 opcode 660	VOP2 opcode 45
v_cvt_pknorm_u16_f16	VOP3 opcode 666	VOP3 opcode 787
v_cvt_pknorm_u16_f32	VOP3 opcode 661	VOP2 opcode 46

Changes From gfx75 Until Current (continued)

Opcode	Current	gfx75
v_cvt_pkrtz_f16_f32	VOP3 opcode 662	VOP2 opcode 47
v_cvt_u16_f16	VOP1 opcode 59	VOP1 opcode 82
v_div_fixup_f16	VOP3 opcode 519	VOP3 opcode 863
v_div_fixup_f32	VOP3 opcode 478	VOP3 opcode 351
v_div_fixup_f64	VOP3 opcode 479	VOP3 opcode 352
v_div_fixup_legacy_f16	VOP3 opcode 495	*new*
v_div_fmas_f32	VOP3 opcode 482	VOP3 opcode 367
v_div_fmas_f64	VOP3 opcode 483	VOP3 opcode 368
v_div_scale_f32	VOP3 opcode 480	VOP3 opcode 365
v_div_scale_f64	VOP3 opcode 481	VOP3 opcode 366
v_dot2_f32_f16	VOP3P opcode 35	*new*
v_dot2_i32_i16	VOP3P opcode 38	*new*
v_dot2_i32_i16_i8	VOP3P opcode 36	*new*
v_dot2_u32_u16	VOP3P opcode 39	*new*
v_dot2_u32_u16_u8	VOP3P opcode 37	*new*
v_dot2c_f32_f16	VOP2 opcode 55	*new*
v_dot2c_i32_i16	VOP2 opcode 56	*new*
v_dot4_i32_i8	VOP3P opcode 40	*new*
v_dot4_u32_u8	VOP3P opcode 41	*new*
v_dot4c_i32_i8	VOP2 opcode 57	*new*
v_dot8_i32_i4	VOP3P opcode 42	*new*
v_dot8_u32_u4	VOP3P opcode 43	*new*
v_dot8c_i32_i4	VOP2 opcode 58	*new*
v_exp_f16	VOP1 opcode 65	VOP1 opcode 88
v_exp_f32	VOP1 opcode 32	VOP1 opcode 37
v_exp_legacy_f32	VOP1 opcode 75	*new*
v_ffbh_i32	VOP1 opcode 47	VOP1 opcode 59
v_ffbh_u32	VOP1 opcode 45	VOP1 opcode 57
v_ffbl_b32	VOP1 opcode 46	VOP1 opcode 58
v_floor_f16	VOP1 opcode 68	VOP1 opcode 91
v_floor_f32	VOP1 opcode 31	VOP1 opcode 36
v_fma_f16	VOP3 opcode 518	VOP3 opcode 843
v_fma_f32	VOP3 opcode 459	VOP3 opcode 331
v_fma_f64	VOP3 opcode 460	VOP3 opcode 332
v_fma_legacy_f16	VOP3 opcode 494	*new*
v_fmac_f32	VOP2 opcode 59	*new*
v_fract_f16	VOP1 opcode 72	VOP1 opcode 95
v_fract_f32	VOP1 opcode 27	VOP1 opcode 32
v_fract_f64	VOP1 opcode 50	VOP1 opcode 62
v_frexp_exp_i16_f16	VOP1 opcode 67	VOP1 opcode 90
v_frexp_exp_i32_f32	VOP1 opcode 51	VOP1 opcode 63
v_frexp_exp_i32_f64	VOP1 opcode 48	VOP1 opcode 60
v_frexp_mant_f16	VOP1 opcode 66	VOP1 opcode 89
v_frexp_mant_f32	VOP1 opcode 52	VOP1 opcode 64
v_frexp_mant_f64	VOP1 opcode 49	VOP1 opcode 61
v_interp_p1ll_f16	VOP3 opcode 628	VOP3 opcode 834
v_interp_p1lv_f16	VOP3 opcode 629	VOP3 opcode 835
v_interp_p2_f16	VOP3 opcode 631	VOP3 opcode 858
v_interp_p2_legacy_f16	VOP3 opcode 630	*new*
v_ldexp_f16	VOP2 opcode 51	VOP2 opcode 59
v_ldexp_f32	VOP3 opcode 648	VOP2 opcode 43
v_ldexp_f64	VOP3 opcode 644	VOP3 opcode 360

Changes From gfx75 Until Current (continued)

Opcode	Current	gfx75
v_lerp_u8	VOP3 opcode 461	VOP3 opcode 333
v_log_clamp_f32	*deleted*	VOP1 opcode 38
v_log_f16	VOP1 opcode 64	VOP1 opcode 87
v_log_f32	VOP1 opcode 33	VOP1 opcode 39
v_log_legacy_f32	VOP1 opcode 76	*new*
v_lshl_add_u32	VOP3 opcode 509	VOP3 opcode 838
v_lshl_b32	*deleted*	VOP2 opcode 25
v_lshl_b64	*deleted*	VOP3 opcode 353
v_lshl_or_b32	VOP3 opcode 512	VOP3 opcode 879
v_lshlrev_b16	VOP2 opcode 42	VOP3 opcode 788
v_lshlrev_b32	VOP2 opcode 18	VOP2 opcode 26
v_lshlrev_b64	VOP3 opcode 655	*new*
v_lshr_b32	*deleted*	VOP2 opcode 21
v_lshr_b64	*deleted*	VOP3 opcode 354
v_lshrrev_b16	VOP2 opcode 43	VOP3 opcode 775
v_lshrrev_b32	VOP2 opcode 16	VOP2 opcode 22
v_lshrrev_b64	VOP3 opcode 656	*new*
v_mac_f16	VOP2 opcode 35	VOP2 opcode 54
v_mac_f32	VOP2 opcode 22	VOP2 opcode 31
v_mac_legacy_f32	*deleted*	VOP2 opcode 6
v_mad_f16	VOP3 opcode 515	VOP3 opcode 833
v_mad_f32	VOP3 opcode 449	VOP3 opcode 321
v_mad_i16	VOP3 opcode 517	VOP3 opcode 862
v_mad_i32_i16	VOP3 opcode 498	VOP3 opcode 885
v_mad_i32_i24	VOP3 opcode 450	VOP3 opcode 322
v_mad_i64_i32	VOP3 opcode 489	VOP3 opcode 375
v_mad_legacy_f16	VOP3 opcode 490	*new*
v_mad_legacy_f32	VOP3 opcode 448	VOP3 opcode 320
v_mad_legacy_i16	VOP3 opcode 492	*new*
v_mad_legacy_u16	VOP3 opcode 491	*new*
v_mad_u16	VOP3 opcode 516	VOP3 opcode 832
v_mad_u32_u16	VOP3 opcode 497	VOP3 opcode 883
v_mad_u32_u24	VOP3 opcode 451	VOP3 opcode 323
v_mad_u64_u32	VOP3 opcode 488	VOP3 opcode 374
v_madak_f16	VOP2 opcode 37	VOP2 opcode 56
v_madak_f32	VOP2 opcode 24	VOP2 opcode 33
v_madmk_f16	VOP2 opcode 36	VOP2 opcode 55
v_madmk_f32	VOP2 opcode 23	VOP2 opcode 32
v_max3_f16	VOP3 opcode 503	VOP3 opcode 852
v_max3_f32	VOP3 opcode 467	VOP3 opcode 340
v_max3_i16	VOP3 opcode 504	VOP3 opcode 853
v_max3_i32	VOP3 opcode 468	VOP3 opcode 341
v_max3_u16	VOP3 opcode 505	VOP3 opcode 854
v_max3_u32	VOP3 opcode 469	VOP3 opcode 342
v_max_f16	VOP2 opcode 45	VOP2 opcode 57
v_max_f32	VOP2 opcode 11	VOP2 opcode 16
v_max_f64	VOP3 opcode 643	VOP3 opcode 359
v_max_i16	VOP2 opcode 48	VOP3 opcode 778
v_max_i32	VOP2 opcode 13	VOP2 opcode 18
v_max_legacy_f32	*deleted*	VOP2 opcode 14
v_max_u16	VOP2 opcode 47	VOP3 opcode 777
v_max_u32	VOP2 opcode 15	VOP2 opcode 20

Changes From gfx75 Until Current (continued)

Opcode	Current	gfx75
v_mbcnt_hi_u32_b32	VOP3 opcode 653	VOP2 opcode 36
v_mbcnt_lo_u32_b32	VOP3 opcode 652	VOP2 opcode 35
v_med3_f16	VOP3 opcode 506	VOP3 opcode 855
v_med3_f32	VOP3 opcode 470	VOP3 opcode 343
v_med3_i16	VOP3 opcode 507	VOP3 opcode 856
v_med3_i32	VOP3 opcode 471	VOP3 opcode 344
v_med3_u16	VOP3 opcode 508	VOP3 opcode 857
v_med3_u32	VOP3 opcode 472	VOP3 opcode 345
v_min3_f16	VOP3 opcode 500	VOP3 opcode 849
v_min3_f32	VOP3 opcode 464	VOP3 opcode 337
v_min3_i16	VOP3 opcode 501	VOP3 opcode 850
v_min3_i32	VOP3 opcode 465	VOP3 opcode 338
v_min3_u16	VOP3 opcode 502	VOP3 opcode 851
v_min3_u32	VOP3 opcode 466	VOP3 opcode 339
v_min_f16	VOP2 opcode 46	VOP2 opcode 58
v_min_f32	VOP2 opcode 10	VOP2 opcode 15
v_min_f64	VOP3 opcode 642	VOP3 opcode 358
v_min_i16	VOP2 opcode 50	VOP3 opcode 780
v_min_i32	VOP2 opcode 12	VOP2 opcode 17
v_min_legacy_f32	*deleted*	VOP2 opcode 13
v_min_u16	VOP2 opcode 49	VOP3 opcode 779
v_min_u32	VOP2 opcode 14	VOP2 opcode 19
v_movreld_b32	*deleted*	VOP1 opcode 66
v_movrels_b32	*deleted*	VOP1 opcode 67
v_movrelsd_b32	*deleted*	VOP1 opcode 68
v_mqsad_pk_u16_u8	VOP3 opcode 486	VOP3 opcode 371
v_mqsad_u32_u8	VOP3 opcode 487	VOP3 opcode 373
v_msad_u8	VOP3 opcode 484	VOP3 opcode 369
v_mul_f16	VOP2 opcode 34	VOP2 opcode 53
v_mul_f32	VOP2 opcode 5	VOP2 opcode 8
v_mul_f64	VOP3 opcode 641	VOP3 opcode 357
v_mul_hi_i32	VOP3 opcode 647	VOP3 opcode 364
v_mul_hi_i32_i24	VOP2 opcode 7	VOP2 opcode 10
v_mul_hi_u32	VOP3 opcode 646	VOP3 opcode 362
v_mul_hi_u32_u24	VOP2 opcode 9	VOP2 opcode 12
v_mul_i32_i24	VOP2 opcode 6	VOP2 opcode 9
v_mul_legacy_f32	VOP2 opcode 4	VOP2 opcode 7
v_mul_lo_i32	*deleted*	VOP3 opcode 363
v_mul_lo_u16	VOP2 opcode 41	VOP3 opcode 773
v_mul_lo_u32	VOP3 opcode 645	VOP3 opcode 361
v_mul_u32_u24	VOP2 opcode 8	VOP2 opcode 11
v_mullit_f32	*deleted*	VOP3 opcode 336
v_not_b32	VOP1 opcode 43	VOP1 opcode 55
v_or3_b32	VOP3 opcode 514	VOP3 opcode 882
v_or_b32	VOP2 opcode 20	VOP2 opcode 28
v_pack_b32_f16	VOP3 opcode 672	VOP3 opcode 785
v_perm_b32	VOP3 opcode 493	VOP3 opcode 836
v_pk_fma_f16	VOP3P opcode 14	*new*
v_pk_fmac_f16	VOP2 opcode 60	*new*
v_pk_mad_f16	*deleted*	VOP3P opcode 14
v_qsad_pk_u16_u8	VOP3 opcode 485	VOP3 opcode 370
v_rcp_clamp_f32	*deleted*	VOP1 opcode 40

Changes From gfx75 Until Current (continued)

Opcode	Current	gfx75
v_rcp_clamp_f64	*deleted*	VOP1 opcode 48
v_rcp_f16	VOP1 opcode 61	VOP1 opcode 84
v_rcp_f32	VOP1 opcode 34	VOP1 opcode 42
v_rcp_f64	VOP1 opcode 37	VOP1 opcode 47
v_rcp_iflag_f32	VOP1 opcode 35	VOP1 opcode 43
v_rcp_legacy_f32	*deleted*	VOP1 opcode 41
v_readlane_b32	VOP3 opcode 649	VOP2 opcode 1
v_rndne_f16	VOP1 opcode 71	VOP1 opcode 94
v_rndne_f32	VOP1 opcode 30	VOP1 opcode 35
v_rsq_clamp_f32	*deleted*	VOP1 opcode 44
v_rsq_clamp_f64	*deleted*	VOP1 opcode 50
v_rsq_f16	VOP1 opcode 63	VOP1 opcode 86
v_rsq_f32	VOP1 opcode 36	VOP1 opcode 46
v_rsq_f64	VOP1 opcode 38	VOP1 opcode 49
v_rsq_legacy_f32	*deleted*	VOP1 opcode 45
v_sad_hi_u8	VOP3 opcode 474	VOP3 opcode 347
v_sad_u16	VOP3 opcode 475	VOP3 opcode 348
v_sad_u32	VOP3 opcode 476	VOP3 opcode 349
v_sad_u8	VOP3 opcode 473	VOP3 opcode 346
v_sat_pk_u8_i16	VOP1 opcode 79	VOP1 opcode 98
v_screen_partition_4se_b32	VOP1 opcode 55	*new*
v_sin_f16	VOP1 opcode 73	VOP1 opcode 96
v_sin_f32	VOP1 opcode 41	VOP1 opcode 53
v_sqrt_f16	VOP1 opcode 62	VOP1 opcode 85
v_sqrt_f32	VOP1 opcode 39	VOP1 opcode 51
v_sqrt_f64	VOP1 opcode 40	VOP1 opcode 52
v_sub_co_u32	VOP2 opcode 26	*new*
v_sub_f16	VOP2 opcode 32	VOP2 opcode 51
v_sub_f32	VOP2 opcode 2	VOP2 opcode 4
v_sub_i16	VOP3 opcode 671	VOP3 opcode 782
v_sub_i32	VOP3 opcode 669	VOP2 opcode 38
v_sub_u16	VOP2 opcode 39	VOP3 opcode 772
v_sub_u32	VOP2 opcode 53	VOP3 opcode 784
v_subb_co_u32	VOP2 opcode 29	*new*
v_subb_u32	*deleted*	VOP2 opcode 41
v_subbrev_co_u32	VOP2 opcode 30	*new*
v_subbrev_u32	*deleted*	VOP2 opcode 42
v_subrev_co_u32	VOP2 opcode 27	*new*
v_subrev_f16	VOP2 opcode 33	VOP2 opcode 52
v_subrev_f32	VOP2 opcode 3	VOP2 opcode 5
v_subrev_i32	*deleted*	VOP2 opcode 39
v_subrev_u16	VOP2 opcode 40	*new*
v_subrev_u32	VOP2 opcode 54	*new*
v_swap_b32	VOP1 opcode 81	VOP1 opcode 101
v_trig_preop_f64	VOP3 opcode 658	VOP3 opcode 372
v_trunc_f16	VOP1 opcode 70	VOP1 opcode 93
v_trunc_f32	VOP1 opcode 28	VOP1 opcode 33
v_writelane_b32	VOP3 opcode 650	VOP2 opcode 2
v_xad_u32	VOP3 opcode 499	VOP3 opcode 837
v_xnor_b32	VOP2 opcode 61	*new*
v_xor_b32	VOP2 opcode 21	VOP2 opcode 29

J Major Changes From gfx81 Until Current

Changes From gfx81 Until Current		
Opcode	**Current**	**gfx81**
buffer_load_format_d16_hi_x	MUBUF opcode 38	*new*
buffer_load_sbyte_d16	MUBUF opcode 34	*new*
buffer_load_sbyte_d16_hi	MUBUF opcode 35	*new*
buffer_load_short_d16	MUBUF opcode 36	*new*
buffer_load_short_d16_hi	MUBUF opcode 37	*new*
buffer_load_ubyte_d16	MUBUF opcode 32	*new*
buffer_load_ubyte_d16_hi	MUBUF opcode 33	*new*
buffer_store_byte_d16_hi	MUBUF opcode 25	*new*
buffer_store_format_d16_hi_x	MUBUF opcode 39	*new*
buffer_store_short_d16_hi	MUBUF opcode 27	*new*
ds_read_addtid_b32	DS opcode 182	*new*
ds_read_i8_d16	DS opcode 88	*new*
ds_read_i8_d16_hi	DS opcode 89	*new*
ds_read_u16_d16	DS opcode 90	*new*
ds_read_u16_d16_hi	DS opcode 91	*new*
ds_read_u8_d16	DS opcode 86	*new*
ds_read_u8_d16_hi	DS opcode 87	*new*
ds_write_addtid_b32	DS opcode 29	*new*
ds_write_b16_d16_hi	DS opcode 85	*new*
ds_write_b8_d16_hi	DS opcode 84	*new*
flat_load_sbyte_d16	FLAT opcode 34	*new*
flat_load_sbyte_d16_hi	FLAT opcode 35	*new*
flat_load_short_d16	FLAT opcode 36	*new*
flat_load_short_d16_hi	FLAT opcode 37	*new*
flat_load_ubyte_d16	FLAT opcode 32	*new*
flat_load_ubyte_d16_hi	FLAT opcode 33	*new*
flat_store_byte_d16_hi	FLAT opcode 25	*new*
flat_store_short_d16_hi	FLAT opcode 27	*new*
global_atomic_add	FLAT opcode 66	*new*
global_atomic_add_x2	FLAT opcode 98	*new*
global_atomic_and	FLAT opcode 72	*new*
global_atomic_and_x2	FLAT opcode 104	*new*
global_atomic_cmpswap	FLAT opcode 65	*new*
global_atomic_cmpswap_x2	FLAT opcode 97	*new*
global_atomic_dec	FLAT opcode 76	*new*
global_atomic_dec_x2	FLAT opcode 108	*new*
global_atomic_inc	FLAT opcode 75	*new*
global_atomic_inc_x2	FLAT opcode 107	*new*
global_atomic_or	FLAT opcode 73	*new*
global_atomic_or_x2	FLAT opcode 105	*new*
global_atomic_smax	FLAT opcode 70	*new*
global_atomic_smax_x2	FLAT opcode 102	*new*
global_atomic_smin	FLAT opcode 68	*new*
global_atomic_smin_x2	FLAT opcode 100	*new*
global_atomic_sub	FLAT opcode 67	*new*
global_atomic_sub_x2	FLAT opcode 99	*new*
global_atomic_swap	FLAT opcode 64	*new*
global_atomic_swap_x2	FLAT opcode 96	*new*

Changes From gfx81 Until Current (continued)

Opcode	Current	gfx81
global_atomic_umax	FLAT opcode 71	*new*
global_atomic_umax_x2	FLAT opcode 103	*new*
global_atomic_umin	FLAT opcode 69	*new*
global_atomic_umin_x2	FLAT opcode 101	*new*
global_atomic_xor	FLAT opcode 74	*new*
global_atomic_xor_x2	FLAT opcode 106	*new*
global_load_dword	FLAT opcode 20	*new*
global_load_dwordx2	FLAT opcode 21	*new*
global_load_dwordx3	FLAT opcode 22	*new*
global_load_dwordx4	FLAT opcode 23	*new*
global_load_sbyte	FLAT opcode 17	*new*
global_load_sbyte_d16	FLAT opcode 34	*new*
global_load_sbyte_d16_hi	FLAT opcode 35	*new*
global_load_short_d16	FLAT opcode 36	*new*
global_load_short_d16_hi	FLAT opcode 37	*new*
global_load_sshort	FLAT opcode 19	*new*
global_load_ubyte	FLAT opcode 16	*new*
global_load_ubyte_d16	FLAT opcode 32	*new*
global_load_ubyte_d16_hi	FLAT opcode 33	*new*
global_load_ushort	FLAT opcode 18	*new*
global_store_byte	FLAT opcode 24	*new*
global_store_byte_d16_hi	FLAT opcode 25	*new*
global_store_dword	FLAT opcode 28	*new*
global_store_dwordx2	FLAT opcode 29	*new*
global_store_dwordx3	FLAT opcode 30	*new*
global_store_dwordx4	FLAT opcode 31	*new*
global_store_short	FLAT opcode 26	*new*
global_store_short_d16_hi	FLAT opcode 27	*new*
image_gather4_a	MIMG opcode 192	*new*
image_gather4_b_a	MIMG opcode 197	*new*
image_gather4_b_cl_a	MIMG opcode 198	*new*
image_gather4_b_cl_o_a	MIMG opcode 214	*new*
image_gather4_b_o_a	MIMG opcode 213	*new*
image_gather4_c_a	MIMG opcode 200	*new*
image_gather4_c_b_a	MIMG opcode 205	*new*
image_gather4_c_b_cl_a	MIMG opcode 206	*new*
image_gather4_c_b_cl_o_a	MIMG opcode 222	*new*
image_gather4_c_b_o_a	MIMG opcode 221	*new*
image_gather4_c_cl_a	MIMG opcode 201	*new*
image_gather4_c_cl_o_a	MIMG opcode 217	*new*
image_gather4_c_o_a	MIMG opcode 216	*new*
image_gather4_cl_a	MIMG opcode 193	*new*
image_gather4_cl_o_a	MIMG opcode 209	*new*
image_gather4_o_a	MIMG opcode 208	*new*
image_gather4h	MIMG opcode 66	*new*
image_gather4h_pck	MIMG opcode 74	*new*
image_gather8h_pck	MIMG opcode 75	*new*
image_sample_a	MIMG opcode 160	*new*
image_sample_b_a	MIMG opcode 165	*new*
image_sample_b_cl_a	MIMG opcode 166	*new*
image_sample_b_cl_o_a	MIMG opcode 182	*new*
image_sample_b_o_a	MIMG opcode 181	*new*

Changes From gfx81 Until Current (continued)

Opcode	Current	gfx81
image_sample_c_a	MIMG opcode 168	*new*
image_sample_c_b_a	MIMG opcode 173	*new*
image_sample_c_b_cl_a	MIMG opcode 174	*new*
image_sample_c_b_cl_o_a	MIMG opcode 190	*new*
image_sample_c_b_o_a	MIMG opcode 189	*new*
image_sample_c_cl_a	MIMG opcode 169	*new*
image_sample_c_cl_o_a	MIMG opcode 185	*new*
image_sample_c_o_a	MIMG opcode 184	*new*
image_sample_cl_a	MIMG opcode 161	*new*
image_sample_cl_o_a	MIMG opcode 177	*new*
image_sample_o_a	MIMG opcode 176	*new*
s_andn1_saveexec_b64	SOP1 opcode 51	*new*
s_andn1_wrexec_b64	SOP1 opcode 53	*new*
s_andn2_wrexec_b64	SOP1 opcode 54	*new*
s_atomic_add	SMEM opcode 130	*new*
s_atomic_add_x2	SMEM opcode 162	*new*
s_atomic_and	SMEM opcode 136	*new*
s_atomic_and_x2	SMEM opcode 168	*new*
s_atomic_cmpswap	SMEM opcode 129	*new*
s_atomic_cmpswap_x2	SMEM opcode 161	*new*
s_atomic_dec	SMEM opcode 140	*new*
s_atomic_dec_x2	SMEM opcode 172	*new*
s_atomic_inc	SMEM opcode 139	*new*
s_atomic_inc_x2	SMEM opcode 171	*new*
s_atomic_or	SMEM opcode 137	*new*
s_atomic_or_x2	SMEM opcode 169	*new*
s_atomic_smax	SMEM opcode 134	*new*
s_atomic_smax_x2	SMEM opcode 166	*new*
s_atomic_smin	SMEM opcode 132	*new*
s_atomic_smin_x2	SMEM opcode 164	*new*
s_atomic_sub	SMEM opcode 131	*new*
s_atomic_sub_x2	SMEM opcode 163	*new*
s_atomic_swap	SMEM opcode 128	*new*
s_atomic_swap_x2	SMEM opcode 160	*new*
s_atomic_umax	SMEM opcode 135	*new*
s_atomic_umax_x2	SMEM opcode 167	*new*
s_atomic_umin	SMEM opcode 133	*new*
s_atomic_umin_x2	SMEM opcode 165	*new*
s_atomic_xor	SMEM opcode 138	*new*
s_atomic_xor_x2	SMEM opcode 170	*new*
s_bitreplicate_b64_b32	SOP1 opcode 55	*new*
s_buffer_atomic_add	SMEM opcode 66	*new*
s_buffer_atomic_add_x2	SMEM opcode 98	*new*
s_buffer_atomic_and	SMEM opcode 72	*new*
s_buffer_atomic_and_x2	SMEM opcode 104	*new*
s_buffer_atomic_cmpswap	SMEM opcode 65	*new*
s_buffer_atomic_cmpswap_x2	SMEM opcode 97	*new*
s_buffer_atomic_dec	SMEM opcode 76	*new*
s_buffer_atomic_dec_x2	SMEM opcode 108	*new*
s_buffer_atomic_inc	SMEM opcode 75	*new*
s_buffer_atomic_inc_x2	SMEM opcode 107	*new*
s_buffer_atomic_or	SMEM opcode 73	*new*

Changes From gfx81 Until Current (continued)

Opcode	Current	gfx81
s_buffer_atomic_or_x2	SMEM opcode 105	*new*
s_buffer_atomic_smax	SMEM opcode 70	*new*
s_buffer_atomic_smax_x2	SMEM opcode 102	*new*
s_buffer_atomic_smin	SMEM opcode 68	*new*
s_buffer_atomic_smin_x2	SMEM opcode 100	*new*
s_buffer_atomic_sub	SMEM opcode 67	*new*
s_buffer_atomic_sub_x2	SMEM opcode 99	*new*
s_buffer_atomic_swap	SMEM opcode 64	*new*
s_buffer_atomic_swap_x2	SMEM opcode 96	*new*
s_buffer_atomic_umax	SMEM opcode 71	*new*
s_buffer_atomic_umax_x2	SMEM opcode 103	*new*
s_buffer_atomic_umin	SMEM opcode 69	*new*
s_buffer_atomic_umin_x2	SMEM opcode 101	*new*
s_buffer_atomic_xor	SMEM opcode 74	*new*
s_buffer_atomic_xor_x2	SMEM opcode 106	*new*
s_call_b64	SOPK opcode 21	*new*
s_dcache_discard	SMEM opcode 40	*new*
s_dcache_discard_x2	SMEM opcode 41	*new*
s_endpgm_ordered_ps_done	SOPP opcode 30	*new*
s_lshl1_add_u32	SOP2 opcode 46	*new*
s_lshl2_add_u32	SOP2 opcode 47	*new*
s_lshl3_add_u32	SOP2 opcode 48	*new*
s_lshl4_add_u32	SOP2 opcode 49	*new*
s_mul_hi_i32	SOP2 opcode 45	*new*
s_mul_hi_u32	SOP2 opcode 44	*new*
s_orn1_saveexec_b64	SOP1 opcode 52	*new*
s_pack_hh_b32_b16	SOP2 opcode 52	*new*
s_pack_lh_b32_b16	SOP2 opcode 51	*new*
s_pack_ll_b32_b16	SOP2 opcode 50	*new*
s_scratch_load_dword	SMEM opcode 5	*new*
s_scratch_load_dwordx2	SMEM opcode 6	*new*
s_scratch_load_dwordx4	SMEM opcode 7	*new*
s_scratch_store_dword	SMEM opcode 21	*new*
s_scratch_store_dwordx2	SMEM opcode 22	*new*
s_scratch_store_dwordx4	SMEM opcode 23	*new*
scratch_load_dword	FLAT opcode 20	*new*
scratch_load_dwordx2	FLAT opcode 21	*new*
scratch_load_dwordx3	FLAT opcode 22	*new*
scratch_load_dwordx4	FLAT opcode 23	*new*
scratch_load_sbyte	FLAT opcode 17	*new*
scratch_load_sbyte_d16	FLAT opcode 34	*new*
scratch_load_sbyte_d16_hi	FLAT opcode 35	*new*
scratch_load_short_d16	FLAT opcode 36	*new*
scratch_load_short_d16_hi	FLAT opcode 37	*new*
scratch_load_sshort	FLAT opcode 19	*new*
scratch_load_ubyte	FLAT opcode 16	*new*
scratch_load_ubyte_d16	FLAT opcode 32	*new*
scratch_load_ubyte_d16_hi	FLAT opcode 33	*new*
scratch_load_ushort	FLAT opcode 18	*new*
scratch_store_byte	FLAT opcode 24	*new*
scratch_store_byte_d16_hi	FLAT opcode 25	*new*
scratch_store_dword	FLAT opcode 28	*new*

Changes From gfx81 Until Current (continued)

Opcode	Current	gfx81
scratch_store_dwordx2	FLAT opcode 29	*new*
scratch_store_dwordx3	FLAT opcode 30	*new*
scratch_store_dwordx4	FLAT opcode 31	*new*
scratch_store_short	FLAT opcode 26	*new*
scratch_store_short_d16_hi	FLAT opcode 27	*new*
v_add3_u32	VOP3 opcode 511	*new*
v_add_co_u32	VOP2 opcode 25	*new*
v_add_i16	VOP3 opcode 670	*new*
v_add_i32	VOP3 opcode 668	*new*
v_add_lshl_u32	VOP3 opcode 510	*new*
v_add_u32	VOP2 opcode 52	VOP2 opcode 25
v_addc_co_u32	VOP2 opcode 28	*new*
v_addc_u32	*deleted*	VOP2 opcode 28
v_and_or_b32	VOP3 opcode 513	*new*
v_div_fixup_f16	VOP3 opcode 519	VOP3 opcode 495
v_div_fixup_legacy_f16	VOP3 opcode 495	*new*
v_dot2_f32_f16	VOP3P opcode 35	*new*
v_dot2_i32_i16	VOP3P opcode 38	*new*
v_dot2_i32_i16_i8	VOP3P opcode 36	*new*
v_dot2_u32_u16	VOP3P opcode 39	*new*
v_dot2_u32_u16_u8	VOP3P opcode 37	*new*
v_dot2c_f32_f16	VOP2 opcode 55	*new*
v_dot2c_i32_i16	VOP2 opcode 56	*new*
v_dot4_i32_i8	VOP3P opcode 40	*new*
v_dot4_u32_u8	VOP3P opcode 41	*new*
v_dot4c_i32_i8	VOP2 opcode 57	*new*
v_dot8_i32_i4	VOP3P opcode 42	*new*
v_dot8_u32_u4	VOP3P opcode 43	*new*
v_dot8c_i32_i4	VOP2 opcode 58	*new*
v_fma_f16	VOP3 opcode 518	VOP3 opcode 494
v_fma_legacy_f16	VOP3 opcode 494	*new*
v_fmac_f32	VOP2 opcode 59	*new*
v_interp_p2_f16	VOP3 opcode 631	VOP3 opcode 630
v_interp_p2_legacy_f16	VOP3 opcode 630	*new*
v_lshl_add_u32	VOP3 opcode 509	*new*
v_lshl_or_b32	VOP3 opcode 512	*new*
v_mad_f16	VOP3 opcode 515	VOP3 opcode 490
v_mad_i16	VOP3 opcode 517	VOP3 opcode 492
v_mad_i32_i16	VOP3 opcode 498	*new*
v_mad_legacy_f16	VOP3 opcode 490	*new*
v_mad_legacy_i16	VOP3 opcode 492	*new*
v_mad_legacy_u16	VOP3 opcode 491	*new*
v_mad_mix_f32	VOP3P opcode 32	*new*
v_mad_mixhi_f16	VOP3P opcode 34	*new*
v_mad_mixlo_f16	VOP3P opcode 33	*new*
v_mad_u16	VOP3 opcode 516	VOP3 opcode 491
v_mad_u32_u16	VOP3 opcode 497	*new*
v_max3_f16	VOP3 opcode 503	*new*
v_max3_i16	VOP3 opcode 504	*new*
v_max3_u16	VOP3 opcode 505	*new*
v_med3_f16	VOP3 opcode 506	*new*
v_med3_i16	VOP3 opcode 507	*new*

Changes From gfx81 Until Current (continued)

Opcode	Current	gfx81
v_med3_u16	VOP3 opcode 508	*new*
v_min3_f16	VOP3 opcode 500	*new*
v_min3_i16	VOP3 opcode 501	*new*
v_min3_u16	VOP3 opcode 502	*new*
v_or3_b32	VOP3 opcode 514	*new*
v_pack_b32_f16	VOP3 opcode 672	*new*
v_pk_add_f16	VOP3P opcode 15	*new*
v_pk_add_i16	VOP3P opcode 2	*new*
v_pk_add_u16	VOP3P opcode 10	*new*
v_pk_ashrrev_i16	VOP3P opcode 6	*new*
v_pk_fma_f16	VOP3P opcode 14	*new*
v_pk_fmac_f16	VOP2 opcode 60	*new*
v_pk_lshlrev_b16	VOP3P opcode 4	*new*
v_pk_lshrrev_b16	VOP3P opcode 5	*new*
v_pk_mad_i16	VOP3P opcode 0	*new*
v_pk_mad_u16	VOP3P opcode 9	*new*
v_pk_max_f16	VOP3P opcode 18	*new*
v_pk_max_i16	VOP3P opcode 7	*new*
v_pk_max_u16	VOP3P opcode 12	*new*
v_pk_min_f16	VOP3P opcode 17	*new*
v_pk_min_i16	VOP3P opcode 8	*new*
v_pk_min_u16	VOP3P opcode 13	*new*
v_pk_mul_f16	VOP3P opcode 16	*new*
v_pk_mul_lo_u16	VOP3P opcode 1	*new*
v_pk_sub_i16	VOP3P opcode 3	*new*
v_pk_sub_u16	VOP3P opcode 11	*new*
v_sat_pk_u8_i16	VOP1 opcode 79	*new*
v_screen_partition_4se_b32	VOP1 opcode 55	*new*
v_sub_co_u32	VOP2 opcode 26	*new*
v_sub_i16	VOP3 opcode 671	*new*
v_sub_i32	VOP3 opcode 669	*new*
v_sub_u32	VOP2 opcode 53	VOP2 opcode 26
v_subb_co_u32	VOP2 opcode 29	*new*
v_subb_u32	*deleted*	VOP2 opcode 29
v_subbrev_co_u32	VOP2 opcode 30	*new*
v_subbrev_u32	*deleted*	VOP2 opcode 30
v_subrev_co_u32	VOP2 opcode 27	*new*
v_subrev_u32	VOP2 opcode 54	VOP2 opcode 27
v_swap_b32	VOP1 opcode 81	*new*
v_xad_u32	VOP3 opcode 499	*new*
v_xnor_b32	VOP2 opcode 61	*new*

Index