Chapter 8

VU Arithmetic Instruction Set

8.1 Arithmetic Status

The computation of each element in a scalar or vector arithmetic operation generates status information. Arithmetic status is written to the **dp_status** mode register as an 18-bit value after each individual computation. Each bit of this status word indicates a particular item of status.

The **dp_status** mode register is overwritten after each individual computation. Therefore, one cannot retrieve the status bits for each vector element in a vector operation. Instead, bits can be chosen to contribute to (OR into) a single status bit that is shifted into the Vector Mask. (See Section 9.2.2 for details on status collection.)

The table below lists the bits in the **dp_status** control register. The symbols shown are defined in **/usr/include/cm/dp.h** and are the masks for the indicated bit.

For example, binary 0000100 is the mask for bit 2. The first five status bits, marked with (*****), are the exceptions defined by the IEEE754 Floating Point Arithmetic Standard.

Note: The opcode descriptions in Section 8.2 include a list of status bits for each opcode, indicating which of the status bits *may* be set to 1 by the opcode. Any status bits not included in an opcode's list are always set to zero by the operation.

Bit	Mask Symbol	Status
0	**`DP_STATUS_ENABLE_MASK_INEXACT`**	Float result is inexact(*)
1	**`DP_STATUS_ENABLE_MASK_DIVIDE_BY_ZERO`**	Division by zero(*)
2	**`DP_STATUS_ENABLE_MASK_UNDERFLOW`**	Float underflow(*)
3	**`DP_STATUS_ENABLE_MASK_OVERFLOW`**	Float overflow(*)
4	**`DP_STATUS_ENABLE_MASK_INVALID_OPERATION`**	Invalid operation(*)
5	**`DP_STATUS_ENABLE_MASK_INT_OVERFLOW`**	Integer overflow
6	**`DP_STATUS_ENABLE_MASK_NEGATIVE_UNSIGNED`**	Negative integer result
7	**`DP_STATUS_ENABLE_MASK_DENORM_INPUT`**	Float input denormalized
8	**`DP_STATUS_ENABLE_MASK_ZERO`**	Float/integer result of zero
9	**`DP_STATUS_ENABLE_MASK_POSITIVE`**	Float/integer result positive
10	**`DP_STATUS_ENABLE_MASK_NEGATIVE`**	Float/integer result negative
11	**`DP_STATUS_ENABLE_MASK_INTEGER_CARRY`**	Integer carry
12	**`DP_STATUS_ENABLE_MASK_INFINITY`**	Float result is +/– infinity
13	**`DP_STATUS_ENABLE_MASK_NAN`**	Float result is a NaN
14	**`DP_STATUS_ENABLE_MASK_DENORM`**	Float result is denormal
15	**`DP_STATUS_ENABLE_MASK_UNORDERED`**	(Internal, do not use)
16	**`DP_STATUS_ENABLE_MASK_UNDER`**	(Internal, do not use)
17	**`DP_STATUS_ENABLE_MASK_DENO`**	(Internal, do not use)

The status bits are defined as follows:

- **`inexact`** is asserted when the delivered result after rounding differs from what would have been computed were both the exponent range and precision unbounded. **`inexact`** is never asserted when **`invalid`** is active.

- **`divide_by_zero`** is active when a division by zero is attempted, and the operands are not invalid.

- **`underflow`** is active when an IEEE floating point underflow is detected after rounding. This flag is also active when an IEEE denormalized number is clipped to zero in Fast mode. **`underflow`** is never active when an integer result is produced.

  Implementation Note: Currently, the underflow signal is generated by the logical OR of the **`under`** status with the logical AND of the **`deno`** and **`zero`** status bits.

- **`overflow`** is active when a floating point result exceeds in magnitude, after rounding, the largest finite number in the destination format, were the exponent range unbounded. It is never active when the result of a computation is an integer.

- **`invalid_operation`** becomes active when any of the following occur:

    1. A floating point operation that generates status has a signaling NaN as an operand.
    2. Two infinities with opposite signs are added.
    3. An infinite floating point number is an operand in a floating-point to integer conversion. In this case, the result is saturated to the maximum integer of the proper sign, and the **`int_overflow`** bit is set.
    4. An attempt is made to convert to an integer a floating point number that is out of the range of the integer. In this case, the result is saturated to the maximum integer of the proper sign, and the **`int_overflow`** bit is set.
    5. A NaN is an operand in a floating-point to integer conversion. The result will be a zero, and the **`zero`** flag will be raised.
    6. An attempt is made to multiply 0 times infinity.
    7. An attempt is made to divide 0 by 0 or infinity by infinity.
    8. A square root of a non-zero number less than zero is attempted.

- The **`int_overflow`** flag is raised when an integer result is to be produced from an arithmetic operation or conversion, and an overflow occurs. This occurs in the following situations:

    1. For two's complement addition, this is the XOR of the *msb* and the *msb*+1 bits.
    2. For unsigned results, this is the value of the *msb*+1 bit, if the operands are both positive.
    3. For conversions, this occurs when a floating-point number outside the range of a destination integer is converted to integer. (This is similar to the "**`V`**" overflow bit that one would see on a microprocessor.)
    4. For integer multiplication, this occurs when a 1 is found in the upper half of the result.

- **`negative_unsigned`** is active when a negative result is generated for an unsigned integer during arithmetic and conversions. The result is forced to zero, and the **`zero`** flag is set.

- **denorm_input** is active when a denormal operand is detected. This flag can be active regardless of the underflow handling mode.

- **zero** is active when the integer or floating point result is zero. Negative floating point zero is also indicated with this flag.

- **positive** is active when the integer result is not zero or positive, or the floating point result is not zero, positive or a NaN. In the current implementation, the pos-result signal is generated by the logical NOR of the **zero**, **negative**, and **nan-result** bits (i.e., **positive** is set when these are all clear).

- **negative** is used during comparison operations, to indicate that the second operand is larger than the first operand. (For integer arithmetic operations, this flag is similar to the "**S**" or "**N**" flag found on microprocessors.) When an answer is in two's complement format, the **negative** flag will be the value of the *msb* of the result. When an unsigned result is produced, the **negative** flag will always be zero. In floating point operations that produce status other than comparisons, the **negative** flag will be equal to the sign bit in the result. During conversions, **negative** will be equal to the sign of the result.

- **integer_carry** indicates that a carry has been generated from the *msb* of the result in the ALU's adder during an integer arithmetic instruction. For shift instructions, **integer_carry** is equal to the bit to the left of the *msb* or the bit to the right of the *lsb* depending on the direction of the shift.

- **infinity** indicates the floating point result is an infinity of either sign.

- **nan** indicates that the result is a quiet NaN. It is valid for instructions that produce floating point results in the ALU, with the exception of the **enc** and **move** instructions. The **enc** instruction will not set this status bit, even if a quiet or signaling NaN is created.

- **denorm** is active when the result after rounding and after Fast mode clipping is an IEEE denormal number. It is not active when a result after rounding is an inexact zero. **denorm** indicates the class of the result and not necessarily occurrence of the IEEE *tiny* condition.

  In the current implementation, the **denorm** signal is generated by the logical AND of **deno** and the logical NOT of **zero**.

Implementation Note

The following signals are only used in the current accelerator implementation, and are included merely to facilitate testing.

- **`unordered`** is active for the comparison operation when at least one of the operands is a signaling or quiet NaN.
- **`under`** is active when an IEEE floating point underflow is detected after rounding. It is never active when an integer result is produced.
- **`deno`** is active when the result after rounding and before Fast mode clipping is an IEEE denormal number. It is not active when a result after rounding is an inexact zero. **`deno`** indicates the class of the result and not necessarily occurrence of the IEEE tiny condition.

8.2 VU Arithmetic Instructions

8.2.1 {i,di,f,df}abs

Takes the absolute value of its operand.

Possible status outputs:

`invalid`	Invalid operand; operand was a signaling NaN.
`int-overflow`	Result overflows the destination integer format.
`zero`	Result is zero.
`positive`	Result is not zero, negative or a quiet NaN.
`integer-carry`	Integer carry out was generated during negation of a two's complement integer.
`infinity`	Result is an infinity.
`nan`	Result is a quiet NaN.
`denorm`	Result is a denormal number.
`(deno)`	Result is a denormal number before Fast mode clipping.

8.2.2 {i,di,u,du,f,df}add, sub, subr

The **add** operation adds its first two operands.
The **sub** operation subtracts the second operand from the first.
The **subr** operation subtracts the first operand from the second.

Integer overflows during addition are signaled, but the result is not saturated. Negative results for unsigned integers are saturated to zero, and the negative-unsigned flag is asserted.

NOTE: The accelerator chip handles unsigned subtraction in a nonstandard way. Whenever a larger number is subtracted from a smaller number, the result is zero, rather than wrapping. This can occur in the unsigned variants of the subtract instructions and the triadics with negated operands.

Possible status outputs:

inexact	Result cannot be represented exactly.
underflow	Result underflows the destination format.
overflow	Result overflows the destination floating-point format.
invalid	Invalid operand; operand was a signaling NaN or operands are oppositely signed infinities.
int-overflow	Result overflows the destination integer format.
negative-unsigned	A negative result was generated for an unsigned integer.
zero	Result is zero.
positive	Result is not zero, negative, or a quiet NaN.
negative	Result has a negative sign.
integer-carry	Integer carry out was generated.
infinity	Result is an infinity.
nan	Result is a quiet NaN.
denorm	Result is a denormal number.
(deno)	Result is a denormal number before Fast mode clipping.

8.2.3 {i,di,u,du}addc, subc, sbrc

These three functions are similar to the **add**, **sub**, and **subr** operations, except that the **addc**, **subc**, and **sbrc** operations include a carry bit in the computation. The bits being shifted off of the Vector Mask (normally used for conditionalization) are used here as the carry input.

As a result, these operations always operate unconditionally, regardless of the setting of the **vector_mask_mode** control register and ignoring any use of the modifiers affectiong conditionalization (**always**, **condmem**, and **condalu**). If the **vminvert** modifier is used, the bit shifted from the Vector Mask is complemented before being used in the computation. If a memory operation accompanies these arithmetic operations, it is conditionalized normally by the Vector Mask bits.

NOTE: The accelerator chip handles unsigned subtraction in a nonstandard way. Whenever a larger number is subtracted from a smaller number, the result is zero, rather than wrapping. This can occur in the unsigned variants of the subtract instructions and the triadics with negated operands.

Possible status outputs:

int-overflow	Result overflows the destination integer format.
negative-unsigned	A negative result was generated for an unsigned integer.
zero	Result is zero.
positive	Result is not zero or negative.
negative	Result has a negative sign.
integer-carry	Integer carry out was generated.

NOTE: In the current implementation, for **subc** and **sbrc**, if the operands and carry are zero, then **integer-carry** is set to 1.

8.2.4 {u,du}and, andc, nand, or, nor, xor

These operations perform, respectively, a bitwise logical AND, a bitwise AND with the first operand complemented, a bitwise NAND, a bitwise OR, a bitwise NOR, and a bitwise XOR of the two operands.

Possible status outputs:

`zero`	Result is zero.
`positive`	Result is not zero.

8.2.5 {f,df}clas

Floating point number class function. Produces an integer indicating the number class of the operand. (The size of the integer will be the size of the operand.) Does not flag signaling NaN's as an exception. Encodes the integer result as:

Integer Result	Interpretation
Hex 0001	Signaling NaN
Hex 0002	Quiet NaN
Hex 0004	Negative Infinity
Hex 0008	Negative Normalized Finite Non–Zero
Hex 0010	Negative Denormalized
Hex 0020	Negative Zero
Hex 0040	Positive Zero
Hex 0080	Positive Denormalized
Hex 0100	Positive Normalized Finite Non–Zero
Hex 0200	Positive Infinity

Possible status outputs:

`positive`	Always asserted.

8.2.6 {i,di,u,du,f,df}cmp

The generic **`cmp`** operation is supported only for completeness. More specific compares (such as **`fgtv`**) are preferred. For **`cmp`**, a third argument is given that specifies the compare code (0–7):

rD[3:1]	Opcode	Function
0	gt	greater than, >
1	eq	equal, =
2	lt	less than, <
3	ge	greater than or equal, >=
4	un	unordered (one or both args are NaNs), ?
5	lg	less than or greater than, <>
6	ne	not equal, ?<>
7	le	less than or equal, <=

Possible status outputs:

`invalid`	Invalid operand; at least one operand is NaN.
`zero`	Operands are equal.
`positive`	Result is not zero, negative or a quiet NaN.
`negative`	Second operand is greater than the first.
`unordered`	At least one operand is a NaN.

8.2.7 cvtf, cvtfi, cvti, cvtir

The general conversion opcodes (**`cvtf`**, **`cvtfi`**, **`cvti`**, and **`cvtir`**) are supported only for completeness. The specific conversion opcodes, such as **`ftoiv`**, are preferred. The specific opcodes expect two operands, a source and a destination. the general opcodes take three operands: a source, a convert code, and a destination. The covert code specifies the conversion done. The table below lists the convert codes that apply to each of the general conversion opcodes.

Opcode	Code	Conversion
`cvtf`{**v**,**s**}	**`CVTFCD_F_DF`**(3)	Single precision float to double precision
`cvtf`{**v**,**s**}	**`CVTFCD_DF_F`**(9)	Double precision float to single precision
`cvtfi`{**v**,**s**}	**`CVTFICD_I_F`**(1)	Single signed integer to single float
`cvtfi`{**v**,**s**}	**`CVTFICD_I_DF`**(3)	Single signed integer to double float
`cvtfi`{**v**,**s**}	**`CVTFICD_U_F`**(5)	Single unsigned integer to single float
`cvtfi`{**v**,**s**}	**`CVTFICD_U_DF`**(7)	Single unsigned integer to double float

Opcode	Code	Conversion
cvtfi{**v**,**s**}	**CVTFICD_DI_F**(9)	Double signed integer to single float
cvtfi{**v**,**s**}	**CVTFICD_DI_DF**(11)	Double signed integer to double float
cvtfi{**v**,**s**}	**CVTFICD_DU_F**(13)	Double unsigned integer to single float
cvtfi{**v**,**s**}	**CVTFICD_DU_DF**(15)	Double unsigned integer to double float
cvti[**r**]{**v**,**s**}	**CVTICD_F_I**(4)	Single float to single signed integer
cvti[**r**]{**v**,**s**}	**CVTICD_F_U**(5)	Single float to single unsigned integer
cvti[**r**]{**v**,**s**}	**CVTICD_F_DI**(6)	Single float to double signed integer
cvti[**r**]{**v**,**s**}	**CVTICD_F_DU**(7)	Single float to double unsigned integer
cvti[**r**]{**v**,**s**}	**CVTICD_DF_I**(12)	Double float to single signed integer
cvti[**r**]{**v**,**s**}	**CVTICD_DF_U**(13)	Double float to single unsigned integer
cvti[**r**]{**v**,**s**}	**CVTICD_DF_DI**(14)	Double float to double signed integer
cvti[**r**]{**v**,**s**}	**CVTICD_DF_DU**(14)	Double float to double unsigned integer

The **cvtir** opcode performs the same float–to–integer conversions as the **cvti** opcode, except that **cvti** truncates its result and **cvtir** rounds the result to the nearest integer.

Possible status outputs:

inexact	Result cannot be represented exactly.
underflow	Result underflows the destination format.
overflow	Result overflows the destination format.
invalid	Invalid operand; operand could be a NaN, infinite, or outside the range of the destination integer.
int-overflow	Integer overflow; operand is outside the range of the destination integer.
negative-unsigned	Attempt to convert negative value to unsigned.
zero	Result is zero.
positive	Result is not zero, negative or a quiet NaN.
negative	Result has a negative sign.
infinity	Result is an infinity.
nan	Result is a quiet NaN.
denorm	Result is a denormal number.
(deno)	Result is a denormal number before Fast mode clipping.

8.2.8 {f,df}div

This instruction divides the first operand by the second operand. When an operand is NaN, infinity, or zero, the division timing will be the same as for normalized operands and results.

Note: In the current accelerator chip, the **`div`** function cannot be used in conjuunction with a memory operation.

These operations do not execute one–per–cycle as do the other operations. Specifically, for a vector length of *N*, these instructions will take 2*N***k* Mbus (SPARC) cycles to execute, rather than the usual 2*N*, where *k* is taken from the following table:

Operation	*k* Value
`fdiv`{**v**,**s**}	4
`dfdiv`{**v**,**s**}	5

Possible status outputs:

`inexact`	Result cannot be represented exactly.
`divide-by-zero`	Division of zero into a non–zero finite number.
`overflow`	Result too large to be represented in destination format.
`underflow`	Result too small to be represented in destination format.
`invalid`	Result cannot be represented exactly.; may be caused by a signaling NaN, 0/0 or infinity / infinity.
`denorm-input`	Denormal input.
`zero`	Result is zero.
`positive`	Result is not zero, negative or a quiet NaN.
`negative`	Result has a negative sign.
`infinity`	Result is an infinity.
`nan`	Result is a quiet NaN.
`(deno)`	Result is a denormal number before Fast mode clipping.

8.2.9 dum[h]{s,m,o,x}{a,i,t}

These multiply–logical operations combine a 64 bit multiply and a 64–bit bitwise logical operation. These can be used to implement ”shift–and–mask” type constructions. The use of multiply rather than a true shift allows more complicated shifting patterns. Either the least significant or the most significant 64 bits of the multiply result can be used.

The high half can simulate a right shift. For example, multiplying by 2^{64–*N*} and using the high half of the result is effectively a right shift by *N*.

These operations all work on 64–bit unsigned values (type **du**). The logical function is based on the ”**s**,**m**,**o**,**x**” choice:

s (select)	gives the AND function
m (mask)	gives the AND–NOT function
o	gives the OR function
x	gives the XOR function

The optional “**h**” causes the most significant 64 bits of the multiply result to be used as the first operand to the logical operation, rather than the least significant 64 bits.

The last letter of the opcode indicates the triadic form:

a	accumulative
i	inverted
t	triadic

producing the possible computation patterns shown below:

Opcode	Function
dum[**h**]**sa**{**v**,**s**}	*Rd* = (*Rs1* * *Rs2*) AND *Rd*
dum[**h**]**ma**{**v**,**s**}	*Rd* = (*Rs1* * *Rs2*) AND NOT *Rd*
dum[**h**]**oa**{**v**,**s**}	*Rd* = (*Rs1* * *Rs2*) OR *Rd*
dum[**h**]**xa**{**v**,**s**}	*Rd* = (*Rs1* * *Rs2*) XOR *Rd*
dum[**h**]**si**{**v**,**s**}	*Rd* = (*Rs2* * *Rd*) AND *Rs1*
dum[**h**]**mi**{**v**,**s**}	*Rd* = (*Rs2* * *Rd*) AND NOT *Rs1*
dum[**h**]**oi**{**v**,**s**}	*Rd* = (*Rs2* * *Rd*) OR *Rs1*
dum[**h**]**xi**{**v**,**s**}	*Rd* = (*Rs2* * *Rd*) XOR *Rs1*
dum[**h**]**st**{**v**,**s**}	*Rd* = (*Rs1* * *Rls*) AND *Rs2*
dum[**h**]**mt**{**v**,**s**}	*Rd* = (*Rs1* * *Rls*) AND NOT *Rs2*
dum[**h**]**ot**{**v**,**s**}	*Rd* = (*Rs1* * *Rls*) OR *Rs2*
dum[**h**]**xt**{**v**,**s**}	*Rd* = (*Rs1* * *Rls*) XOR *Rs2*

The result status is reported for the ALU boolean only. The other status is derived entirely from the multiplication.

Possible status outputs:

int-overflow	A 1 was found in the upper half of the result (MULT)
zero	Result is zero (ALU).
positive	Result is not zero (ALU)

8.2.10 {u,du}enc

This operation generates a floating point number by placing the first operand in the exponent field and the second operand in the mantissa field. For both operands, the values are given as unsigned integers in least significant bits. The exponent is given in biased form. The output is a floating–point value.

For floating–point values in the normalized range, the mantissa operand must contain the hidden bit. Denormal numbers are constructed from an exponent equal to 1 and a mantissa with the hidden bit cleared. The resulting denormal value will have a zero in its exponent field. No checking is performed on the resulting floating point number.

Possible status outputs:

`positive`	Always asserted.

8.2.11 etrap

This operation is not a vector operation. It forces a trap to occur if **`dp_status`** ANDed with the bits in the **`dp_interrupt_enable_green`** is non-zero. Since the **`dp_status`** register contains the status from the last element computed, this is really only useful for diagnostic purposes. This operation sets no status flags.

8.2.12 {f,df}exp

This operation extracts the biased exponent of its operand, a floating point value. The *lsb* of the exponent becomes the *lsb* of the resulting integer. The precision of the floating point operand becomes the precision of the resulting integer. The format of the result is the same as that required for the **`enc`** instruction.

The result of NaN and infinity operands is a left justified string of ones equal to the width of the exponent field of the operand. Denormal numbers produce an integer with a value of 1.

Possible status outputs:

`invalid`	Invalid operand; first operand was a signaling NaN.
`positive`	Always asserted.

8.2.13 {u,du}ffb

The **ffb** (Find First Bit) instruction returns the number of leading (most significant) zeroes above the most significant 1 bit in the operand. For example, **duffb** of binary 0001... returns 3. If no ones are present in the operand, a zero is returned. The single precision version, **duffb**, views its operand as a 64–bit unsigned number constructed by padding the 32–bit argument with zeroes. As a result, **uffb**(0xFFFFFFFF) gives 32. The result of the **ffb** instruction on an operand equal to zero is zero.

Possible status outputs:

zero	Result is zero.
positive	Result is not zero.

8.2.14 ftodf, dftof

These operations convert a floating point operand to a floating point result of another precision.

Possible status outputs:

inexact	Result cannot be represented exactly.
underflow	Result underflows the destination format.
overflow	Result overflows the destination floating–point format.
invalid	Invalid operand; operand was a signaling NaN.
zero	Result is zero.
positive	Result is not zero, negative or a quiet NaN.
negative	Result has a negative sign.
infinity	Result is an infinity.
nan	Result is a quiet NaN.
denorm	Result is a denormal number.
(under)	Result underflows dest format before Fast md. clipping.
(deno)	Result is a denormal number before Fast mode clipping.

8.2.15 {f,df}inv

This operation takes the reciprocal of its operand. When the operand is NaN, infinity, or zero, the reciprocal timing will be the same as for normalized operands and results.

NOTE: In the current accelerator chip, the **`inv`** function cannot be used in conjunction with a memory operation.

These operations do not execute one–per–cycle as do the other operations. Specifically, for a vector length of N, these instructions will take $2N*k$ Mbus (SPARC) cycles to execute, rather than the usual $2N$, where k is taken from the following table:

Operation	*k* Value
`fdiv`{**`v`**,**`s`**}	4
`dfdiv`{**`v`**,**`s`**}	5

Possible status outputs:

`inexact`	Result is inexact.
`divide-by-zero`	1/0 was attempted.
`overflow`	Result too large to be represented in destination format.
`underflow`	Result too small to be represented in destination format.
`invalid`	Invalid operand; operand is a signaling NaN.
`denorm-input`	Denormal input.
`zero`	Result is zero.
`positive`	Result is not zero, negative or a quiet NaN.
`negative`	Result has a negative sign.
`infinity`	Result is an infinity.
`nan`	Result is a quiet NaN.
`(deno)`	Result is a denormal number before Fast mode clipping.

8.2.16 {f,df}isqt

This operation divides the first operand by the square root of the second operand. No status is generated by this instruction.

This operation does not obey the IEEE standard with respect to rounding or exception detection. First, **isqt** always rounds the result toward zero. The error in the result will be at most one *lsb* in the mantissa when compared to an infinitely precise answer, and the result will be equal to or smaller than the infinitely precise answer. Second, the only exception detected is when the operand is negative and non–zero or a NaN, in which cases a NaN result is generated and the **dp_status_nan_result** bit in the **dp_status** register is set. Notably, no divide–by–zero detection is done for the case when the argument is zero, nor is underflow signalled when the result is too small. The positive status bit is always set.

Note: In the current accelerator chip, the **isqt** function cannot be used in conjunction with a memory operation.

These operations do not execute one–per–cycle as do the other operations. Specifically, for a vector length of *N*, these instructions will take 2*N***k* Mbus (SPARC) cycles to execute, rather than the usual 2*N*, where *k* is taken from the following table:

Operation	*k* Value
fisqt{**v**,**s**}	5
dfisqt{**v**,**s**}	7

When an operand is NaN, infinity, or zero, the **AINVSQRTB** timing will be the same as for normalized operands and results.

Possible status outputs:

positive Always asserted.

8.2.17 lvdm

Required Syntax: **`ldvm`** *VU–register*

This operation moves the low order 16 bits of a specified register into both the Vector Mask (**`dp_vector_mask`**) and the Vector Mask Buffer (**`dp_vector_mask_buffer`**). This operation has a fairly significant cost both in execution speed and in pipeline delay, and as such should be used sparingly.

This instruction may not be combined with a memory operation (**`load`**, **`store`**), and is not affected by conditionalization. Possible status outputs: none

8.2.18 {i,di,u,du,f,df}lt, le, gt, ge, eq, ne, un, lg

These operations compare the two operands. No result is written to the register file. The result of the compare (a 1 bit if true, otherwise a 0 bit) is shifted into the Vector Mask. The rotate mode (**`vmrotate`**) is used by default, but the **`vmcurrent`** modifier can be added to change to the current mode. (See Section 9.2.2.)

Additionally, the status bits are set as if the two values were subtracted (operand 1 minus operand 2), as shown below. At most one of **`zero`**, **`negative`**, and **`unordered`** will be set.

Possible status outputs:

`invalid`	At least one operand is a signaling NaN.
`zero`	Operands are equal.
`positive`	Result is not zero, negative or a quiet NaN.
`negative`	The second operand is greater than the first operand.
`unordered`	At least one operand is a NaN.

8.2.19 {i,di,u,du,f,df}mad, msb, msr, nma

These operations perform a multiply followed by some other operation (for example, an add). There are three forms, the Accumulative, the Inverted, and the Triadic. These differ in the way in which they apply the operands in the computation. The form is signaled by the suffix: **`a =`** accumulative, **`i`** = inverted, **`t =`** true triadic.

The following table lists the available opcodes.

Opcode	Function
{**i**,**di**,**u**,**du**,**f**,**d**}**mada**{**v**,**s**}	*Rd* = (*Rs1* * *Rs2*) + *Rd*
{**i**,**di**,**u**,**du**,**f**,**d**}**msba**{**v**,**s**}	*Rd* = (*Rs1* * *Rs2*) – *Rd*
{**i**,**di**,**u**,**du**,**f**,**d**}**msra**{**v**,**s**}	*Rd* = *Rd* – (*Rs1* * *Rs2*)
{**i**,**di**,**u**,**du**,**f**,**d**}**nmaa**{**v**,**s**}	*Rd* = –*Rd* – (*Rs1* * *Rs2*)
{**i**,**di**,**u**,**du**,**f**,**d**}**madi**{**v**,**s**}	*Rd* = (*Rs2* * *Rd*) + *Rs1*
{**i**,**di**,**u**,**du**,**f**,**d**}**msbi**{**v**,**s**}	*Rd* = (*Rs2* * *Rd*) – *Rs1*
{**i**,**di**,**u**,**du**,**f**,**d**}**msri**{**v**,**s**}	*Rd* = *Rs1* – (*Rs2* * *Rd*)
{**i**,**di**,**u**,**du**,**f**,**d**}**nmai**{**v**,**s**}	*Rd* = –*Rs1* – (*Rs2* * *Rd*)
{**i**,**di**,**u**,**du**,**f**,**d**}**madt**{**v**,**s**}	*Rd* = (*Rs1* * *Rls*) + *Rs2*
{**i**,**di**,**u**,**du**,**f**,**d**}**msbt**{**v**,**s**}	*Rd* = (*Rs1* * *Rls*) – *Rs2*
{**i**,**di**,**u**,**du**,**f**,**d**}**msrt**{**v**,**s**}	*Rd* = *Rs2* – (*Rs1* * *Rls*)
{**i**,**di**,**u**,**du**,**f**,**d**}**nmat**{**v**,**s**}	*Rd* = –*Rs2* – (*Rs1* * *Rls*)

Integer overflows during addition are signaled, but the result is not saturated. Negative results for unsigned integers are saturated to zero, and the **negative-unsigned** flag is asserted. The result status (**zero**, **negative-result**, etc.) reflects the final result only. Exception status (e.g., **overflow**) is derived by OR-ing the status from both operations.

Possible status outputs:

inexact	Result cannot be represented exactly.
invalid	Invalid operand.
overflow	Result overflows the destination floating–point format.
underflow	Result underflows the destination format.
int-overflow	**MULT**: a ‘1’ was found in the upper half of the result. **ALU**: result overflows the destination integer format.
negative-unsigned	A negative result was generated for an unsigned integer.
denorm-input	Denormal input.
zero	Result is zero.
positive	Result is not zero, negative or a quiet NaN.
negative	Result has a negative sign.
integer-carry	Integer carry out was generated.
infinity	Result is an infinity.
nan	Result is a quiet NaN.
denorm	Result is a denormal number.
(under)	Result underflows dest format before Fast md. clipping.
(deno)	Result is a denormal number before Fast mode clipping.

8.2.20 {f,df}mant

The operation extracts the mantissa of its operand, a floating–point number. If the operand is normalized, the hidden bit is present in the result. The *lsb* of the mantissa becomes the *lsb* of the resulting integer. A double precision floating point operand produces a double precision integer result, and a single precision floating point operand produces a single precision result. The format of the result is the same as that required for the enc instruction. A NaN input returns the value of the mantissa, with the hidden bit set to one, and sets the invalid status.

Possible status outputs:

`invalid`	Invalid operand; first operand was a signaling NaN.
`positive`	Always asserted.

8.2.21 {i,di,u,du,f,df}move

The first operand is passed through the ALU. Invalid operands are not converted into a quiet NaN. Possible status outputs:

`positive`	Always asserted.

8.2.22 {u,du}mrg

The merge instruction merges two vectors under control of the Vector Mask. Bits shifted off of the Vector Mask, normally used for conditionalization, are used in a different way. Specifically, for each element, the bits control which operand is moved to the destination. If the bit shifted from the Vector Mask is a ‘1’, the result is taken from the first operand; otherwise, it is taken from the second operand. If you use the vminvert modifier, which inverts the sense of the Vector Mask, the merge is done in the opposed fashion. As a result, this operation always operates unconditionally, regardless of the setting of the vector_mask_mode control register, and it ignores the modifiers affecting conditionalization (always, condmem, and condalu). Any memory operation that accompanies these arithmetic operations is conditionalized normally by the Vector Mask bits.

Possible status outputs:

`positive`	Always asserted.

8.2.23 {i,u,f,df}mul

This operation multiplies its two operands. Multiplication with the double integer format can be accomplished with the {**di**,**du**}**mul** and {**di**,**du**}**mulh** instructions. Multiplication of 32–bit integers will produce a 64-bit result. When an operand is NaN, infinity, or zero, the multiplication timing will be the same as for normalized operands and results.

Possible status outputs:

inexact	Result cannot be represented exactly.
overflow	Result overflows the destination floating–point format.
underflow	Result underflows the destination format.
invalid	Invalid operand; caused by 0 times infinity or by a signaling NaN operand.
int-overflow	A '1' was found in the upper half of the result.
denorm-input	Denormal input.
zero	Result is zero.
positive	Result is not zero, negative or a quiet NaN.
negative	Result has a negative sign.
infinity	Result is an infinity.
nan	Result is a quiet NaN.
(under)	Result underflows dest format before Fast md. clipping.
(deno)	Result is a denormal number before Fast mode clipping.

8.2.24 {di,du}mul, mulh

These operations generate the low (**mul**) and high (**mulh**) 64 bits of the multiplication of the two integer operands.

Possible status outputs:

int-overflow	A '1' was found in the upper half of the result.
zero	Result is zero.
positive	Result is not zero, negative or a quiet NaN.
negative	Result has a negative sign.

8.2.25 {i,di,f,df}neg

This operation subtracts the first operand from zero. Integer overflows are signaled, but the result is not saturated. Negative results for unsigned integers are saturated to zero, and the negative–unsigned flag is asserted.

Possible status outputs:

invalid	Invalid operand; operand was a signaling NaN.
int-overflow	Result overflows the destination integer format.
zero	Result is zero.
positive	Result is not zero, negative or a quiet NaN.
negative	Result has a negative sign.
negative-unsigned	Attempt to convert negative value to unsigned.
integer-carry	Integer carry out was generated during negation of a two's complement integer.
infinity	Result is an infinity.
nan	Result is a quiet NaN.
denorm	Result is a denormal number.
(deno)	Result is a denormal number before Fast mode clipping.

8.2.26 fnop{v,s}

This operation is, as its name suggests, a NOP. It does nothing.

Possible status outputs: none

8.2.27 {u,du}not

This operation performs a bitwise logical NOT of its first operand.

Possible status outputs:

zero	Result is zero.
positive	Result is not zero, negative or a quiet NaN.

8.2.28 {u,du}shl, shlr

This operation performs an integer logical left shift.

For **`shl`**, the first operand is the value to be shifted, and the unsigned value of the low 6 bits of the second operand is the shift distance. The bits in the second operand above the sixth bit are ignored. The reversed shift, **`shlr`**, shifts the second operand by the value in the low 6 bits of the first operand. These alternatives are provided so that the *Rs1* operand, which has improved accessibility and striding capability, can be used as either operand.

Zeros are shifted into the low end of the result. The status integer–carry will be the value of the bit to the left of the *msb*. A negative two's complement integer is sign extended beyond the *msb*, so a zero shift on a negative two's complement number will produce **`integer-carry`**. As a result of the 6–bit shift value, it is not possible to shift a double precision value by a full 64 bits. (It is, however, possible to shift a 32–bit integer by 32).

Possible status outputs:

`integer-carry`	Value of the *msb*+1 bit.
`zero`	Result is zero.
`positive`	Result is not zero, negative or a quiet NaN.

8.2.29 {u,du,i,di}shr, shrr

This operation performs an integer shift right (logical or artihmetic). For **shr**, the first operand is the value to be shifted and the unsigned value of the low 6 bits of the second operand is the shift distance. The bits in the second operand above the sixth bit are ignored. For **shrr**, the reverse shift is performed: the second operand is the value to be shifted, and the low 6 bits of the first operand provide the shift distance.

For the unsigned data types (**u** and **du**), the shift is a logical shift; thus, zeros are shifted into the high end of the result. For the signed data types (**i** and **di**), an arithmetic shift is performed: i.e. the bits shifted into the upper end of the result are a copy of the original sign bit of the operand. For example, shifting the 32–bit hexidecimal value 80000008 right one bit by an arithmetic shift yields C0000004, while a logical shift of the same value yields 40000004.

The **integer–carry** status is the value of the bit to the right of the *lsb*.

Possible status outputs:

integer–carry	Value of the LSB–1 bit.
zero	Result is zero.
positive	Result is not zero, negative or a quiet NaN.
negative	Result has a negative sign.

8.2.30 stvm

Required Syntax: **stvm** *VU–register*

This operation moves the 16–bit value in the Vector Mask (**dp_vector_mask**) into the specified *VU–register.*

This instruction may not be combined with a memory operation (**load**, **store**), and is not affected by conditionalization. This operation has a fairly significant cost both in execution speed and in pipeline delay, and as such should be used sparingly.

Possible status outputs: none

8.2.31 {d,df}sqrt

The operation takes the IEEE square root of the first operand.

Note: In the current accelerator chip, the **`sqrt`** function cannot be used in conjunction with a memory operation.

These operations do not execute one–per–cycle as do the other operations. Specifically, for a vector length of *N*, these instructions will take 2*N* K Mbus (SPARC) cycles to execute, rather than the usual 2N, where K is taken from the following table:

Operation	*k* Value
`fsqrt`{**`v`**,**`s`**}	6
`dfsqrt`{**`v`**,**`s`**}	8

When the operand is a NaN, infinity, zero, or negative, the square root timing will be the same as for normalized operands and results.

Possible status outputs:

`inexact`	Result cannot be represented exactly.
`underflow`	Result underflows the destination format.
`invalid`	Invalid operand; operand is a negative non–zero number or a signaling NaN.
`denorm-input`	Denormal input.
`zero`	Result is zero.
`positive`	Result is not zero, negative or a quiet NaN.
`negative`	Result has a negative sign.
`infinity`	Result is an infinity.
`nan`	Result is a quiet NaN.
`(under)`	Result underflows dest format before Fast md. clipping.
`(deno)`	Result is a denormal number before Fast mode clipping.

8.2.32 {i,di,u,du,f,df}test

This operation adds the first operand to zero. Unlike **`move`**, the **`test`** instructions do assert status outputs to reflect the value moved. Specifically, **`test`** is exactly like adding zero to the operand, as far as the setting of status flags is concerned.

Possible status outputs:

`invalid`	Invalid operand; operand was a signaling NaN.
`zero`	Result is zero.
`positive`	Result is not zero, negative or a quiet NaN.
`negative`	Result has a negative sign.
`infinity`	Result is an infinity.
`nan`	Result is a quiet NaN.
`denorm`	Result is a denormal number.
`(deno)`	Result is a denormal number before Fast mode clipping.

8.2.33 {f,df}to{i,di,u,du}{[r]}

These operations convert a floating point operand to integer format. Overflows are saturated to the maximum integer value in the output format, and underflows are forced to zero. Conversion of a negative floating point number to unsigned causes the **`negative-unsigned`** status and saturates the result to zero. Conversions from NaN formats to integers create a zero result, and set the **`invalid`** status. Conversions from infinities are sauturated like overflows; they, too, cause the **`invalid`** status. The "**`r`**" forms of these operations round as indicated by the current rounding mode (set by default to "nearest"); the non-"**`r`**" forms simply truncate toward zero.

Possible status outputs:

`inexact`	Result cannot be represented exactly.
`invalid`	Invalid operand; operand could be a NaN, infinity, or outside the range of the destination integer.
`int-overflow`	Result overflows the destination integer format.
`negative-unsigned`	Attempt to convert negative value to unsigned.
`zero`	Result is zero.
`positive`	Result is not zero, negative or a quiet NaN.
`negative`	Result has a negative sign.
`nan`	The first operand is a NaN.

8.2.34 trap

This operation, which is not a vector operation, forces a trap to occur. This trap will be a green interrupt identical to the trap caused by a masked status trap.

Possible status outputs: none