Chapter 9

Accelerator Chip Notes

This chapter discusses the architecture of the accelerator chip, from a DPEAC programmer's perspective. Specifically, the areas discussed here are:

- Instruction Pipelining
- Conditionalization
- Floating–point Formats
- Fast mode vs. IEEE mode

9.1 Instruction Pipelining

9.1.1 Pipeline Hazards

The accelerator chip executes instructions in a pipelined fashion: that is, it attempts to begin the execution of each element of a vectored arithmetic or memory instruction in each of its clock cycles.

When two or more vectored instructions are executed one after the other, the accelerator chip attempts to execute them without breaking the pipeline, so that their execution can overlap. This creates so–called "pipeline hazards": potential situations in which the results of sequentially executed instructions may not yield the same results as would be gotten if the instructions were executed without overlap. For example, consider two vectored instructions of vector length 4, the first of which writes its results to **`V2`** (**`R16`**, **`R17`**, **`R18`**, and **`R19`**) and the second of which reads its operands from **`R19`**, **`R20`**, **`R21`**, and **`R22`**. When the second instruction reads its first operand from **`R19`**, the value it reads will not have been written yet by the first instruction.

To control and prevent pipeline hazards, you need to learn a set of rules for juxtaposing instructions so that you can recognize a potential hazard, and use the **`dpsynch`** instruction to guarantee non–overlapped execution. There are seven conditions under which a pipeline hazard is possible; these conditions are enumerated and described below.

Performance Note: Performing vector computations only on aligned vectors in ascending order (that is with a positive stride), and preventing vector computations from “wrapping” around the end of the register file (that is, from **`R127`** to **`R0`**) prevents hazards 1 and 4 and allows **`dpas`** to implictly avoid hazards 2, 3, and 5. Hazard 6 is a very rare situation, but should be considered when issuing instructions to a subset of the DPs.

HAZARD 1: Reading from a register written less than 4 cycles ago.

The results of an operation typically cannot be used until 4 operations later. The accelerator chip pads all operations out to an effective vector length of 4 (including scalar operations). Thus, an operation of vector length 2 is executed as: *operation0*, *operation1*, **`NOP`**, **`NOP`**.

This avoids hazards as long as vectored data is always operated on in ascending registers, since the result from the first vector element of one vectored instruction will not be needed until the beginning of the next vectored instruction, which is guaranteed (by the padding) to be 4 operations in the future. There are three exceptions to this rule, described under Hazards 2, 3, and 4 below.

HAZARD 2: Store of a register after less than 7 cycles

If an instruction stores data to memory from a vector of registers, and the data to be stored is the result of an arithmetic operation in the previous instruction, the computed data must be written to the register 7 pipeline stages before it is stored in memory (for double precision), or 5 pipeline stages (for single precision data). This implies that a vector length of 7 or more is required for the data to be stored correctly, assuming registers are computed in the same order as they are stored on the subsequent instruction (e.g. in the normal ascending order).

Performance Note: The **`dpas`** assembler, by default, avoids this hazard by inserting an **`fnops`** operation before all instructions that perform stores. Such operations are padded to 8 cycles. This is effectively the same as assembling the instruction with a pad modifier of **`pad:8`**.

HAZARD 3: Register memory indirection after less than 8 cycles

If an instruction uses indirect addressing of memory, and the vector of indirection offsets (*Ria*) was calculated with an arithmetic operation in the previous instruction, each offset must have been written to its register at last 8 pipeline stages before it is used. Thus, assuming the offsets are used in the same order in which they were computed, a vector length of 8 is required for this instruction sequence to work correctly. This does not apply to indirect regster accessing, which has the normal latency of 4 operations.

Performance Note: The **`dpas`** assembler, by default, avoids this hazard by inserting an **`fnops`** operation before each instruction that uses memory indirection, effectively padding it out to 8 cycles. (This is effectively the same as assembling the instruction with a pad modifier of **`pad:8`**.)

HAZARD 4: Loading from a register used as the second operand of an ALU (non–multipler) operation after less than 3 cycles.

A relaxation of the 4–cycle latency rule is that a number of arithmetic operations (specifically, those that use the ALU circuitry of the accelerator chip) only require their second ALU operand (the first operand for shift operations) to have been written at least 2 operations previously rather than 4. This is because the second operand is read 2 cycles later in the pipeline than operands are normally read. The operations for which this exception applies are:

- **`neg`** and **`not`**
- **`enc`**, **`clas`**, **`exp`**, and **`mant`**
- type conversion operations: **`cvtf`**, **`dftof`**, **`ftodu`**, etc.
- shift operations: **`shl`**, **`shr`**, etc. (first operand, the amount to be shifted)
- **`add`**, **`sub`**
- **`sub`**
- 2–operand bitwise logicals: **`and`**, **`nand`**, **`xor`**, etc.
- comparisons: **`cmp`**, **`le`**, **`gt`**, etc.
- triadics: **`dum`**, **`mad`**, **`msb`**, etc. (the operand that is not in the multiply)

An additional hazard occurs with this 2–cycle latency operand because it is read 2 cycles later in the pipeline. In a vector operation of these instructions, in the last two element calculations, the second operand register (as it is being strided) is vulnerable to loads (from memory, or from ACC or EPC operations) of that register done in the first two cycles of the next instruction.

For example, consider the following sequence, a vector add operation of length 4 (shown unrolled) followed by a load of arbitrary length. The tildes (**~**) indicate arbitrary registers, distinct from any that are explicitly named.

```
fadd ~,R32,~  !element 1
fadd ~,R33,~  !element 2
fadd ~,R34,~  !element 3 (in peril from floadv below)
fadd ~,R35,~  !element 4 (in peril from floadv below)
floadv [~],R34     !load R34, R35, ...
```

The hazard occurs because the load of **R34** occurs before the element 3 calculation. Since all vector operations are padded to 4 elements, this cannot happen for a vector length of 1 (including scalars) or 2. At a vector length of 3, just the last element's second operand is in peril.

This hazard can occur from loading registers using the **epc** (popcount) or **count** (context count) operations. In the following example, an elemental popcount replaces the role of the **floadv** in the previous example:

```
fadd ~,R32,~  !element 1
fadd ~,R33,~  !element 2
fadd ~,R34,~  !element 3 (in peril from the epcv below)
fadd ~,R35,~  !element 4 (in peril from the epcv below)
epcv(V12)=R34:1   !Elemental Popcount to R34, R35, ...
```

Immediate operations are implemented by loading the immediate value into **R0** and forcing the *Rs2* stride to zero. As a result they can create this type of hazard:

```
fadd ~,R126,~      !element 1
fadd ~,R127,~      !element 2
fadd ~,R0,~        !element 3
fadd ~,R1,~        !element 4
fmulv ~,$nnn,~
```

The loading of *nnn* into **R0** affects the previous instruction's element 3 calculation. If the **fmulv** were a double precision **dfmulv** operation, the previous instruction's element 4 calculation would also be affected, since the immediate value for the dfmulv would be loaded into both **R0** and **R1**. For shift operations, this does not apply, since its first operand is the 2–cycle latency operand. There is no hazard caused by two ALU–only instructions in a row that both use immediate operands — the hardware delays the second instruction to keep it from overwriting **R0** too soon.

A more common case (though still on the rare side), is when the first instruction's second operand is strided by zero:

```
fadd ~,R8,~   !element 1
fadd ~,R8,~   !element 2
fadd ~,R8,~   !element 3
fadd ~,R8,~   !element 4

floadv [~],V1 !load R8, R9, ...
```

Here, **R8** (the first element of **V1**) will be overwritten by the load, corrupting the element 3 and element 4 computations. Moreover, if the second ALU operand is strided by zero and chainloaded, it will be vulnerable to loads within the same instruction:

```
fadd ~,R8,~; floadv[mem1],R8:0  <-- Rs1 is hit by ...
fadd ~,R8,~; floadv[mem2],R8:0
fadd ~,R8,~; floadv[mem3],R8:0  <--  ... this load
fadd ~,R8,~; floadv[mem4],R8:0
```

Here the *Rs1* operand of the element 1 calculation will not be the value loaded from memory location *mem1*, because that value will not be read out of **R8** until two cycles later, after being loaded with the value from location *mem3*. This hazard is especially notorious in shifts and in the inverted forms of the 2–operation instructions (e.g. **fmadiv**) where the second ALU operand is taken from the *Rs1* operand and where zero–striding is commonplace:

```
dfmadiv S2:0,V2, V4; dfloadv [%i0],S2:0
```

or

```
ushrv S2:0,4,V4; dfloadv [%i10],S2:0 ;load/shift, FAILS!
      ^
      This is ALU operand 2!
```

Usage Note: Because this kind of operation is relatively obscure, the **dpas** assembler issues a warning whenever an instruction attempts to chain–load with a stride of zero into the register that will be read as the second ALU operand.

HAZARD 5: Reading/writing a register from the IU that is in use

Another hazard occurs when the SPARC IU reads or writes registers (via the DPEAC **dprd** and **dpwrt** instruction) that are being read or written by a currently active instruction. The **dprd** and **dpwrt** operations are not synchronized to the pipelined instruction stream by the accelerator chip, and therefore can interact unpredictably. When the IU reads a register, as many as three of the previous instructions can be actively writing the register. When the IU writes a register, only the immediately preceding instruction can be active.

Performance Note: **dpas** automatically inserts synchronization code before **dprd** and **dpwrt** instructions to avoid this hazard, though this can be disabled.

HAZARD 6: Collecting status on a VU that is subsequently disabled

The current implementation of the accelerator architecture has hazard potential when collecting status. Two VUs are housed on each accelerator chip. If an instruction is issued that collects meaningful status into **vector_mask** for one VU of a pair, and the next instruction is issued only to the other VU on the same chip, the **vector_mask** register will be corrupted in the disabled VU, effectively losing the status collected by the first instruction. To correct for this, rather than disabling the VU, you can operate it with conditionalization enabled and a vector mask of 0's.

HAZARD 7: Chain load into destination register of an exchange.

The current implementation of the assembler arhitecture disallows chain–loading into a register that is also used as the destination register when the **exchange** modifier is used. In particular, the following is illegal:

```
dumovev V2,V2; duloadv [%i1],V2; exchange
```

This must be recoded to use different source and destination registers in the arithmetic operations. For example,

```
dumovev V2,V6; duloadv [%i1],V2; exchange
```

9.1.2 The [no]pad Modifier

The [**no**]**pad** modifier may be appended to any VU Instruction (i.e. arithmetic and/or memory operations), and is useful in avoiding pipeline hazards. The syntax is:

Modifier	Effect
pad:*N*	Pad to effective vector length of *N*
nopad	Do not artifically pad

The **pad** modifier causes the instruction to execute, with respect to latency, as if it had an effective vector length of *N* or greater. Instructions are assembled with **pad:4** by default. Values of *N* which are less than 4 are allowed, but are equivalent to 4.

The **nopad** modifier is equivalent to **pad:0** and guarantees that no artificial padding is performed.

Note that the padding is not exact, and must never be relied upon to produce exact inter–instruction execution latencies.

Performance Note: Instructions that perform double precision store operations or that use memory indirect addressing are assembled, by default, with **pad:8** to correct pipeline hazards 2 or 3. The **dpas** assembler takes this conservative action because it has no way of knowing if the stored data (or indirect offsets) were computed in the previous instruction. If they were not, you can add the **nopad** modifier to prevent the (slower) padded execution.

9.1.3 Correcting Pipeline Hazards

There are two methods for correcting pipeline hazards. The safest method for separating two operations is to insert a **dpsync** instruction between them. This is the only method for separating a vector operation and a **dprd** or **dpwrt** instruction, and **dpas** does this automatically before any **dprd** or **dpwrt** unless the **nosync** operand is specified.

Appending the **pad:8** modifier has a similar effect, and will generally save a significant number of cycles. Pipeline hazards 1, 2, 3, and 4 can always be solved by this. However, padding does not correct hazard 5 (the **dprd** and **dpwrt** hazard). You must use the **dpsync** instruction to correct that.

In short, to minimize the possibility of pipeline hazards, follow these rules:

1. Always use aligned vectors: start operations on vector boundaries and process vectors in ascending order. (This eliminates hazard 1, and avoids 4.)
2. Do not use scalar registers (S0 through S31) as vectors if you use them as scalars. (Together with the above rule, this eliminates hazard 4.)
3. Don't override the **`sync`** default in **`dprd`** and **`dpwrt`** instructions or the **`pad`** defaults for DP Instructions. (This eliminates hazards 2, 3, and 5.)
4. Be careful when collecting status not to issue a subsequent instruction directed only at individual VUs. (This avoids hazard 6.)
5. Don't chain load into *Rd* in an exchange operation. (This avoids hazard 7.)

9.1.4 The Stages of the Pipeline

Figure 5 shows the stages of the pipeline through which each element of each scalar or vector instruction must pass. Each stage of the pipeline is 2 SPARC (system clock) cycles long because the 2 VUs on the accelerator chip must execute the stage in a serialized fashion.

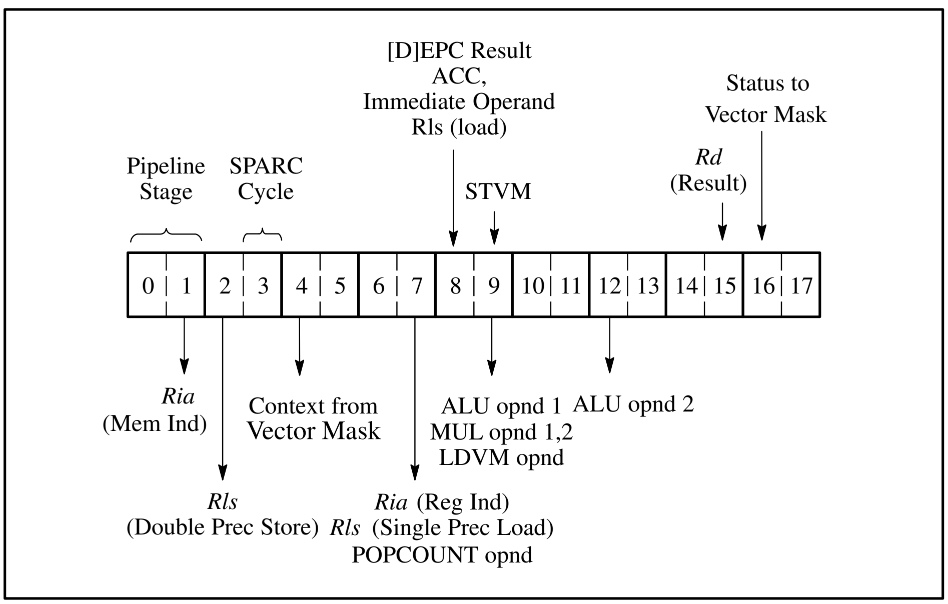

Figure 5. Datapath pipeline stages.

A new element is started at each pipeline stage. The resulting pattern for an operation having a vector length of four is shown in Figure 6. Also shown is the first element of a subsequent vector operation, which generally follows in the next pipe stage. (Note that this could just as easily be the fifth element of the first vector instruction.)

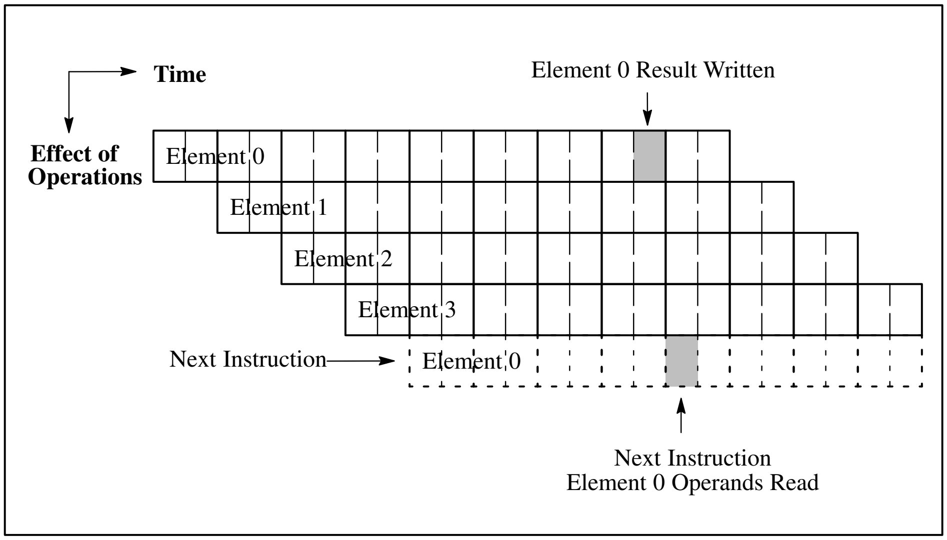

Figure 6. Datapath Pipeline

Generally, a register written as a result is readable four operations later. As an example, the upper marked box shows a register written in the second half of the eighth stage of the element 0 pipe. The lower marked box shows the register being read as an operand in the first half of the fourth stage of the element zero pipe of the next vector operation. Note that this cycle is after the cycle in which the register was last written. Therefore, the value read will be the value written above. Any attempt to read that register earlier would have resulted in reading the prior contents of the register.

9.2 How The Accelerator Does Conditionalization

The accelerator architecture offers

- The ability to execute vector operations conditionally, i.e. to be able to nullify the computation on selected elements,
- The ability to collect status bits from vector operations and to pack them into words.

9.2.1 Control Registers

The control registers that affect conditionalization are the **vector_mask** and the **vector_mask_buffer**. These can be read and written from the SPARC using the DPEAC instructions **dpset** and **dpget**. Each VU has its own set of these control registers. This makes them unlike other control registers, which are shared by the 2 VUs on each accelerator chip.

The Vector Mask can be transferred to and from a VU register using the **ldvm** and **stvm** instructions. **ldvm** loads the **vector_mask** from any single–precision register, and **stvm** stores **vector_mask** to any single–precision register.

The current version of the accelerator chip supports vector lengths up to and including 16. Therefore, these registers are 16 bits in length. If the contents of these registers are to be transferred to/from 32–bit registers, they are read to or written from the low 16 bits of the 32–bit register.

In general, at the beginning of every vector or scalar VU instruction, the **vector_mask_buffer** is copied to the **vector_mask**. The **vector_mask** is then used and/or modified by the VU instruction. This is the ”old” vector mask mode, and is the default mode for all VU instructions.

The Mode Set instruction format lets you suppress this copy, allowing the use of the resultant **vector_mask** from the previous instruction in the current one. This is the “nop” mode, and is specified by the DPEAC modifier **vmnop**.

The Mode Set format also offers the ability to reverse the copy, i.e. to copy the **vector_mask** from the previous instruction to the **vector_mask_buffer** instead. This allows that **vector_mask** value to be reused in the current and subsequent instructions. This is the “new” mode and is specified by the DPEAC modifier **vmnew**. Figure 7 shows the two registers and the copying options.

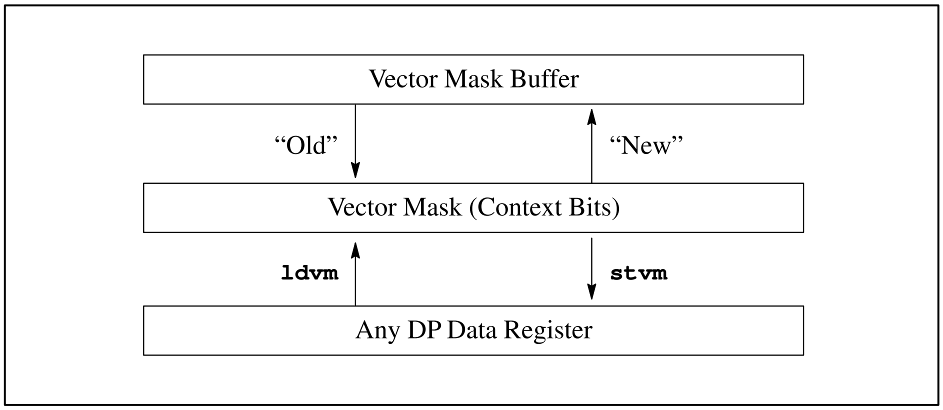

Figure 7. "New" vs. "old" mode.

9.2.2 Conditionalization and Status Collection

All VU instructions use the conditionalization mechanism except the following:

- **ldvm**
- **stvm**
- **trap**
- **etrap**

Note that even the **fnop** operation uses conditionalization.

Scalar arithmetic and memory operations are not conditionalized; they execute as if in *always* mode. Nonetheless, scalar operations perform all of the side effects of conditionalization: they move and shift the **vector_mask_buffer** and they generate status as would a vector operation of length 1.

As each operation proceeds, the **vector_mask** is shifted by one prior to the computation for each element. A 1–bit control register, **vector_mask_direction**, can be set to select the shift direction, either to the right (the normal case), or to the left. The bit shifted off the end is used to conditionalize the operation on that element. When shifting to the right, the bit shifted out of the *lsb* (bit 0) is the bit used.

The bits shifted off the **vector_mask** are called the context bits for the vector operation.

At the end of the vector operation, the **`vector_mask`** is left shifted N positions where N is the vector length. Figure 8 shows the shifting in both directions. The bit marked 0 is the context bit used for the first element of the vector, 1 is the second, and so on, up through the last bit 3 (assuming a vector length of 4).

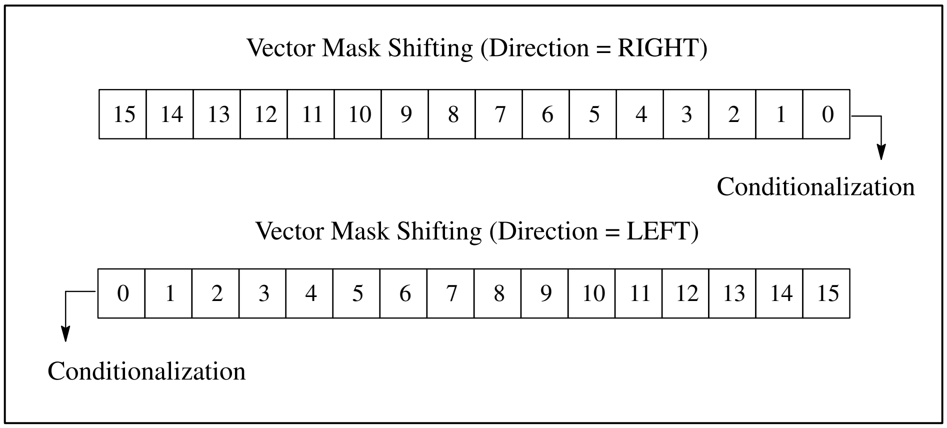

Figure 8. Shifting directions.

The conditionalization of the arithmetic operation (e.g. **`faddv`**) and the memory operation (e.g. **`floadv`**) can be individually controlled. The **`vector_mask_mode`** register selects which are conditionalized. The **`dpset`** instruction in DPEAC is used to set this register. Additionally, the Mode Set format VU instructions lets the vector mask mode be set temporarily for the current instruction. The following DPEAC modifiers are used to exert this control:

Modifier	What is Conditionalized
`cond`	Both Arithmetic and Memory operations
`condalu`	Arithmetic operations only
`condio`	Memory operations only
`always`	No conditionalization (vector mask still shifted)

Note: In the accelerator chip, conditionalizing memory operations does not suppress page fault detection for deselected elements. Similarly, conditionalizing arithmetic operations has no effect on pipeline latency of deselected operations. As a result, conditionalization does not improve performance, and you may find it conceptually simpler to use conditionalization only to affect store operations, or to suppress arithmetic status (see ”Status Collection”, below). In the Mode Set instruction format, the sense of the bit used for conditionalization can be inverted by using the **`vminvert`** keyword. The corresponding **`vmtrue`** keyword can be used to get the default, non–inverted, operation.

9.2.3 Status Collection

Every VU arithmetic computation yields a single status bit. For the **cmp** operation, the status bit is the result of the comparison. For all other operations, the bit is 1 if the operation resulted in any of the conditions masked by the **status_enable** control register. For example, bit 3 of the **status_enable** register corresponds to the overflow condition. If this bit is set, any operation that results in an overflow will cause the status for that operation to be set. If more than one bit in **status_enable** is set, then the occurrence of any of the selected conditions will cause the status to be set. For arithmetic operations on deselected vector elements (those for which the conditionalization bit is zero), the status output is zero. Memory operations do not affect the status bit output.

The following table lists the bits in the **status_enable** and **status_control** registers. 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.

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_enable** can be written using the **dpset** instruction. The following instruction shows the setting of two status bits, **negative** and **zero**.

```
dpset *,(DP_STATUS_ENABLE_MASK_NEGATIVE   \
         | DP_STATUS_ENABLE_MASK_ZERO), dp_status_enable
```

Status bits are shifted into the **vector_mask** as it is shifted. By default, bits are shifted into the end opposite that from which bits are being shifted off for conditionalization. This default is the **rotate** mode. Figure 9 shows the results of an operation having vector length 4 when the rotate mode is used in either direction.

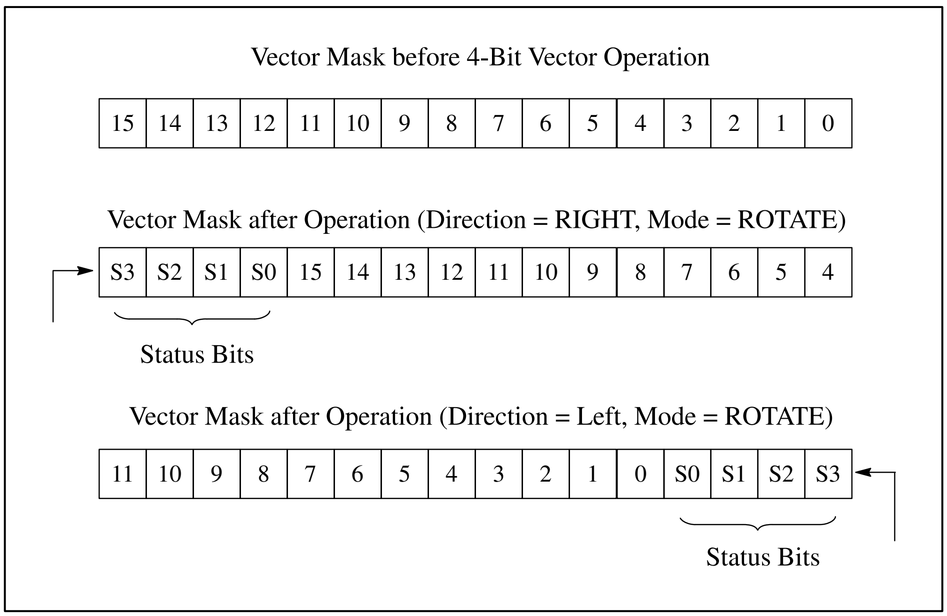

Figure 9. Status packing in the ROTATE mode.

The Mode Set instruction format provides an alternate mode, the **vmcurrent** mode. In this mode, with the direction set to **right**, the status bits are shifted in such a way so as to cause them to occupy bits 0 through *N* of the **vector_mask** at the end of the vector operation. Specifically, the status from element 0 will be left in bit 0, element 1 in bit 1, and so forth. If the direction is set to **left**, the bits will be reversed in order and will occupy the upper N bits of the vector mask.

Figure 10 shows the use of the **`current`** mode. In DPEAC, this mode is selected by the **`vmcurrent`** modifier, and the **`vmrotate`** modifier can be used to get the default (**`rotate`**) mode. The **`cmp`** operation in all instruction formats also allows the selection of this mode by appropriately setting bit 0 of the *Rd* instruction field. DPEAC allows the **`vmcurrent`** and **`vmrotate`** modifiers for the **`cmp`** operation in all formats.

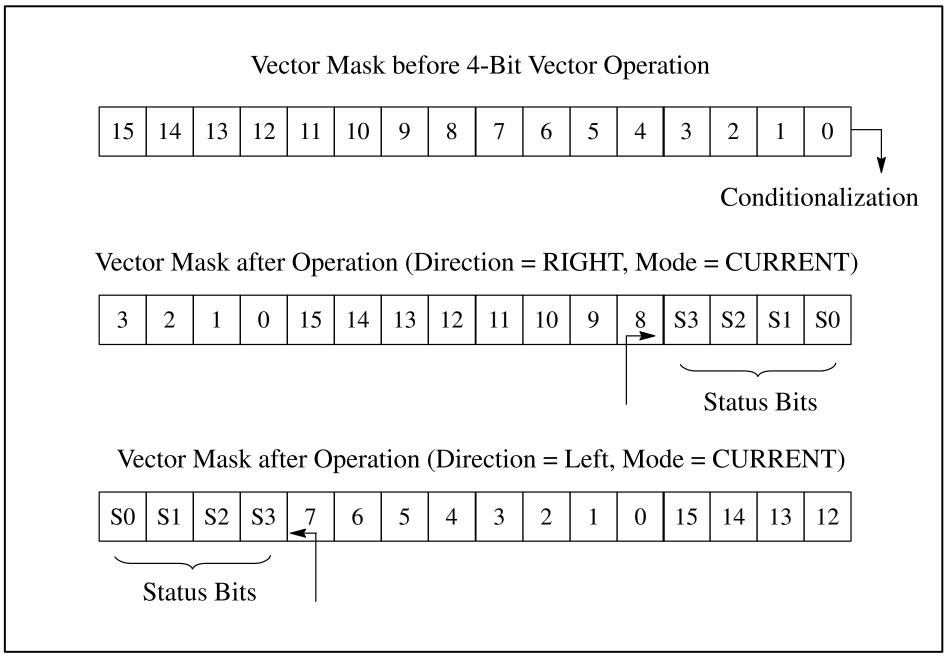

Figure 10. Status packing in the CURRENT mode.

Note: The accelerator chip contains an implementation restriction concerning the combining of single precision store memory operations and the collection of status bits to be used in the following instruction. Specifically, if an instruction contains a store–single operation of vector length less than 6 and uses the **`vmcurrent`** modifier to load its status bits into the low bits of the vector_mask, and the following instruction uses the **`vmnew`** or **`vmnop`** modifiers to immediately use the status bits generated by the first instruction, those bits may not be set up in time to be read correctly by the second instruction.

9.2.4 The merge Instruction

The **merge** instructions ({**f**,**df**,**u**,**du**,**i**,**di**}**mrg**{**v**,**s**}) act slightly differently from normal arithmetic operations. In particular, they are never conditional — all elements of each processed vector are processed. Instead, the bits being shifted off the Vector Mask are used to control the **merge** operation. If the context bits are inverted via the **vminvert** modifier, the sense of the merge is reversed. **merge** is then defined as:

```
result = if [inverted]VectorMaskBit
            then SecondOperand
            else FirstOperand
```

9.2.5 The Accumulated Context Count (ACC)

The Mode Set format VU instructions offer a feature which counts the 1's in the vector_mask during a vector operation of length *N*, and deposits the accumulating count into a separate vector of *N* registers. Each element *i* of the resulting vector will be the count of 1 bits shifted off the **vector_mask** after being shifted the first *i*+1 times. The count values are single–precision unsigned integers.

DPEAC uses the **vmcount** modifier to specify the ACC feature:

```
MCOUNT={dp-register}:{stride}
```

For example:

```
VMCOUNT=V4:1
```

For example, assume the Vector Mask contains the binary value 0000000011001011. Then assume a vector operation of length 8 is performed using the **right** shift direction. The context bits shifted, in time order, are 1, 1, 0, 1, 0, 0, 1, and 1. If the ACC feature is specified using the DPEAC modifier "**vmcount=v4:1**", then V4 wil contain the vector 1, 2, 2, 3, 3, 3, 4, and 5 at the end of the vector operation.

9.2.6 DPEAC Conditionalization Modifier Summary

The following table summarizes the DPEAC modifier keywords thate affect conditionalization. Those marked with (d) indicate the default settings for any instruction not overriding that default.

Modifier	Format	Effect
vmmode:vmmode (d)	Mode Set	Use **vector_mask_mode**
vmmode:always	Mode Set	Do not use conditionalization
vmmode:=always	Mode Set	Set **vector_mask_mode** as well
vmmode:condmem	Mode Set	Conditionalize load/stores in this instruction
vmmode:=condmem	Mode Set	Set **vector_mask_mode** as well
vmmode:condalu	Mode Set	Conditionalize arithmetic in this instruction
vmmode:=condalu	Mode Set	Set **vector_mask_mode** as well
vmmode:=cond	Mode Set	Set **mask_mode** for full conditionalization
vmold (d)	Any	Copy **vector_mask_buffer** to **vector_mask**
vmnew	Mode Set	Copy **vector_mask** to **vector_mask_buffer**
vmnop	Mode Set	No copy
vminvert	Mode Set	Invert mask bits for conditionalization
vmtrue (d)	Any	Don't invert mask bits
vmcurrent (d)	Mode Set	Write status to incrementing mask bits
vmrotate	Any	Write status to rotating **dp_vector_mask**
vmcount=reg:stride	Mode Set	Enable Vector Mask Count

9.3 Floating–point Formats

The format of single–precision floating–point numbers is as follows:

S	exponent	fraction
1	8	23

Double precision floating point values are in the following format:

S	exponent	fraction
1	11	52

The sign bit indicates the sign. Zero indicates positive, 1 indicates negative.

For single precision numbers, the mantissa is a 24–bit value consisting of the 23 specified bits and an addition "hidden" one bit in the most significant position. An implied binary point is assumed between the hidden bit and the lower 23 bits, forming a mantissa in the range 1.0 to almost 2.0. The exponent is 7 bits and is biased upward by 127.

For double–precision numbers, the mantissa is a 53–bit value consisting of the 52 specified bits and an addition "hidden" one bit in the most significant position. An implied binary point is assumed between the hidden bit and the lower 52 bits. The exponent is 11 bits and is biased upward by 1023 (hex 1FF).

Denormalized numbers are represented by an exponent of 0, and this exponent value is reserved only for this purpose. In such cases, the actual exponent is assumed to be –128 and the hidden bit is assumed to be zero.

Infinity is indicated by an exponent of all–ones (7F hex) and a mantissa of zero. A number having an all-ones exponent and a non-zero mantissa is reserved for out-of-range values, called NaN (Not a Number) values. A NaN with the *msb* of its mantissa clear (not the hidden bit) is referred to as a "Signalling NaN", while a NaN with this bit set is a "Quiet NaN." DPEAC arithmetic always produces quiet NaN's, i.e., with the *msb* of their mantissas set, and all other mantissa bits set to zeroes. Quiet and signalling NaN's are supported as operands, per the IEEE floating point standard.

9.4 Fast Mode Vx. IEEE Mode

The accelerator chip supports arithmetic in both the "fast mode" and the "IEEE mode." The difference is related to the handling of denormalized numbers.

When the multiplier unit or ALU is in Fast mode, denormal operands and results are converted to a zero. Conversions of denormal operands takes place as the first task in the operand processors. A zero produced in Fast mode from an operand will carry the same sign as the denormal number that caused the coversion to zero. For this reason, in Fast mode, denormal operands are not processed differently from signed zeros beyond the operand processor. A ramification of this property is that the square root of a negative denormal number in Fast mode will be negative zero, and invalid will not be signaled.

When a result underflows, the decision to convert the denormal number to zero is made in the circuitry immediately prior to the rounding stge. This means that if a denormal number is produced prior to rounding, and during rounding a normalized number is generated, the Fast mode logic will not force that result to zero. When a denormal result is converted to zero, that zero will retain the sign of the denormal number.

The current implementation of the accelerator always operates the multiplier in Fast mode. Thus, denormal numbers are not fully handled or generated in the multiply, divide, and square root functions.

The ALU, which performs all other operations, can operate in either mode; the mode is controllable by a bit in the **`dp_alu_mode`** mode register. In the ALU, with the exception of the zero result flag, there is no differences in status for operations on normalized operands between Fast and IEEE mode. For instance, if a result of an instruction from the ALU in IEEE mode was denormal and negative, when that same instruction is run in Fast mode, the status out will indicate denormal, negative and zero.

Below are some examples of addition in Fast and IEEE modes:

Example	Operand A	Operand B	Mode	Flags	Result
[1]	3F800000	3F800000	IEEE	n	40000000
	3F800000	3F800000	Fast	n	40000000
[2]	00000001	00000001	IEEE	d	00000002
	00000001	00000001	Fast	0	00000000
[3]	00FFFFFF	00400000	IEEE	n	011FFFFF
	00FFFFFF	00400000	Fast	n	00FFFFFF
[4]	00800000	807FFFFE	IEEE	d	00000001
	00800000	807FFFFE	Fast	n	00800000
[5]	80800000	00800001	IEEE	ds	80000001
	80800000	00800001	Fast	ds0	80000000

Flags: **`d`** denormal result **`0`** zero result
`s` result is negative **`n`** no flags are set

In example [1], the operands and result are not denormal in the IEEE mode, so there is no difference in the result or flags in the Fast mode.

Example [2] has two denormal operands, which in the Fast mode are converted to zero, and when added produce an exact zero. In IEEE mode, the addition of the denormal operands creates a denormal result.

In example [3], the second operand is denormal. When this is converted to zero in Fast mode, the result is the first operand. In IEEE mode, the denormal number is added to the first operand, and the result is greater tham the first operand.

Example [4] is similar to example [3]. The final example is a subtraction yielding a denormal result. In Fast mode, this result is converted to a signed zero, and in IEEE mode, the result is not affected.

In IEEE and Fast mode in example [5], the sign and denormal flags are set.

Below are some examples of multiplication in IEEE and Fast mode. Again, note that the current implementation does not implement multiplication in the IEEE mode. Thus, examples of multiplication in that mode do not apply to this implementation.

Example	Operand A	Operand B	Mode	Flags	Result
[1]	3F800000	3F800000	IEEE	n	3F800000
	3F800000	3F800000	Fast	n	3F800000
[2]	00000001	00000001	IEEE	ux0dab	00000000
	00000001	00000001	Fast	0dab	00000000
[3]	7F800000	007FFFFF	IEEE	fdb	7F800000
	7F800000	007FFFFF	Fast	iqdb	7FC00000
[4]	3F000000	00800001	IEEE	gux	00400000
	3F000000	00800001	Fast	g0	00000000
[5]	3F000000	00FFFFFF	IEEE	ux	00800000
	3F000000	00FFFFFF	Fast	g0	00000000

Flags:

`f`	result is an infinity	`q`	result is a quiet NaN
`g`	result is subnormal	`x`	result is inexact
`u`	underflow	`i`	invalid
`d`	denormal operand	`0`	zero result
`s`	result is negative	`n`	no flags are set
`a`	operand A is denormal or signaling NaN		
`b`	operand B is denormal or signaling NaN		

In example [1], the operands and result are not denormal in the IEEE mode.

Example [2] has two denormal operands, which in the IEEE mode are flagged and trapped. The product of the denormal numbers is very tiny, which leads to the underflow and inexact zero. With the Fast mode, the denormal operands are flagged and converted to zero, so that the product is an exact zero.

Example [3] is the multiplication of an infinity by a denormal number. Because the IEEE defines arithmetic on infinities to be exact (when not invalid), the result in IEEE mode is an exact infinity. In Fast mode, the denormal operand will be flagged, and converted to zero, which in turn causes the invalid flag to be raised because of the infinity * 0 operation, and the result is a quiet NaN that is accompanied with the NaN result flag.

In example [4] a normalized number slightly larger than the minimum normalized number is multiplied by 1/2. In IEEE mode, the product is a denormal number, and cannot be represented exactly. This causes the denormal result, inexact, and underflow flag to be set. In Fast mode, the subnormal result from the computation is detected and converted to a zero. Because the result was exact, the Fast mode status will not indicate inexact.

In example [5], there are some important differences in the results for the subnormal modes. For the IEEE mode (in the language of the IEEE 754–1985 specification) a number between $2^{\text{smallest–exponent}}$ and $2^{(\text{smallest–exponent}+1)}$ is multiplied by $2^{(-1)}$, which will yield an exact result less than $2^{(\text{smallest–exponent}+1)}$ if the exponent range were unbounded. Therefore the product is tiny. However, the exponent is bounded, and the result is denormal with significance to the right of the *lsb* of the mantissa of the destination format.

Therefore the product is inexact. An inexact and tiny result will cause the underflow flag to be set. Due to the significance beyond the *lsb* of the mantissa, the denormal result is rounded, and in the course of incrementing, the result goes out of denormal range. Because the result after rounding is no longer denormal, the subnormal result flag is not set.

In both the ALU and multiplier, it is possible to detect when a denormal result has been converted to a zero. In IEEE mode, the zero result and subnormal result flags are mutually exclusive, because the zero result flag and the subnormal result flag represent disjoint sets of number classes (note that the subnormal result flags in the ALU and multiplier indicate a number class, and not the tiny condition). In Fast mode, sudden underflow occurs when the result after rounding is subnormal, so that an "and" function on the subnormal result and zero flag will indicate that sudden underflow has taken place.

It is possible to detect the IEEE tiny condition (for trap handlers) after rounding in a similar fashion. The underflow flag from the ALU and multiplier are active when tiny and inexact have occurred. It is easy to show that when an exact tiny result occurs, that result will be a subnormal number. Therefore, tiny may be detected by an “or” function on the subnormal result flag and the underflow flag.