Chapter 4

DPEAC Instruction Formats

The accelerator chip supports five basic instruction formats:

- The Short format
- The Long formats:
  - Immediate format
  - Memory Stride format
  - Mode Set format
  - Register Stride format

The long formats involve double word (64-bit) instructions, and so require more cycles to issue. The Short format provides faster results. Other then speed, the formats differ principally in the types of stride and vector length control offered. In addition, the Mode Set format supports the setting of modes relating to contextualization.

Note: When an instruction combines an Arithmetic Operation and a Memory Operation, the two must be in the same format.

4.1 Elements of an Instruction

Each VU instruction consists of one to three elements, separated by semicolons:

- An optional Arithmetic operation
- An optional Memory operation
- Zero or more Modifiers

The instruction can optionally end in a semicolon.

An example of a legal VU instructions is

```
faddv V2,V3,V4;    floadv [%i1],V3;    vmmode=cond
```

The arithmentic operation in this instruction performs a vector single precision floating point addition operation (**faddv**). The number of additions performed (the vector length) is taken from the control register **dp_vector_length**. The two source operands are **V2** and **V3**, each with a register stride of one. Thus, the operation uses the consecutive registers that start at **R16** and **R24** respectively. The result of the addition operation is placed into **V4**, that is, into consecutive registers starting at **R32**.

The data for the second operand is furnished by the Memory portion of the instruction, which performs a vector load from memory (**floadv**) into register **V3**. (Loads of a register are always done before the register is read as an operand.) The memory address, located in **%i1**, both determines which VUs of the four available are to execute the instruction and specifies the offset at which the data is to be found in the memory banks of the selected VUs. Data is loaded into register **V3** with a stride of one.

The modifier **vmmode=cond** specifies that the instruction is to be performed conditionally, with each element enabled or nullified by a corresponding bit in the **dp_vector_mask**.

4.2 The Arithmetic Operation

This portion of the VU instruction specifies the arithmetic operation. It is specified by an opcode and a number of operands.

The opcodes available are discussed in the next major section, “Arithmetic Operations”. This includes the definition of the arithmetic performed and the status generated. Each opcode has a scalar form and a vector form. The scalar form, which is indicated by the letter “**s**” appended to the opcode name, does a single operation. The vector form, indicated by the letter “**v**”, operates on arrays of data whose length is taken from the Vector Length control register.

The operands to arithmetic operations are registers. Each DPEAC instruction includes a number of register specification fields. There are five kinds of fields used to specify a register; each represents a different way in which a register can be used in an operation. These register fields correspond to the register operands supplied to a DPEAC operator in source code.

In the descriptions of DPEAC operators below, the five kind of register operands are indicated by the following placeholders:

- *Rs1* — the first source operand register
- *Rs2* — the second source operand register
- *Rd* — the ALU destination register
- *Ria* — the Indirect Addressing register
- *Rls* — the Load/Store register

  (**`Note:`** An *Rls* operand typically specifies a register for a memory operation, but can also specify the third operand in 3-operand arithmetic operations)

A stride is associated with each register specifier. By default, the stride is

- One, for single-precision arithmetic operations
- Two, for double-precision arithmetic operations

For vector operations, an operand is taken from the set of registers specified by the register specifier and extending through subsequent registers as implied by the register stride.

The register stride for all register strides is, by default, one or two, depending on the precision, single or double, of the arithmetic operation. Default strides can, however, be altered for all register operands.

The stride for the first register operand (*Rs1*) is stored in the VU chip as a settable mode (**`dp_stride_rs1`**), while the Register Stride format for instructions allows arbitrary register striding of the *Rs2*, *Rd*, and *Rls* register operands.

The arithmetic operation includes an opcode and a number of operands:

- Monadic (one-operand) operations take one source operand.
- Dyadic operations take two operands.
- Triadic operations take three operands.

In all cases, the first operand is specified by the *Rs1* register, and is strided by the *Rs1* stride (**`dp_stride_rs1`**).

For the dyadic operations, the second operand is specified by the *Rs2* register, strided by one (single precision) or two (double precision).

The triadic operations take a third operand, added between the *Rs1* and *Rs2*. This register is specified using the *Rls* register specifier, normally (and possibly still) used as the register to or from which memory data is transferred.

All three types of operation write their results to the *Rd* register, strided by one or two.

The basic syntax of arithmetic operations can thus be summarized as follows:

Monadic Operation:	*opcode Rs1, Rd*
Dyadic Operation:	*opcode Rs1, Rs2, Rd*
Triadic Operation:	*opcode Rs1, Rls, Rs2, Rd*

NOTE: Compare instructions modify the Arithmetic Operation syntax slightly in that they do not specify an *Rd* register (since there is no computed result).

4.3 The Memory Operation

The memory operations cause vectors of data (words or doublewords), to be loaded from or stored to memory.

The following memory operations are supported. Data types are as in the arithmetic operations, i.e., one of **`f`** (single float), **`df`** (double float), **`i`** (signed 32-bit integer), **`di`** (signed 64-bit integer), or **`du`** (unsigned 64-bit integer).

Opcode	Function
`{f,dr,i,di,u,du}load{v,s}`	Load Vls from memory
`{f,dr,i,di,u,du}store{v,s}`	Store Vls to memory
`memnop`	No memory operation

When a memory operation is executed in the same instruction with an arithmetic operation, the two are done is synchrony, with the load or store preceding the arithmetic operation.

Note how this affects the use of the data:

- If data is being loaded into the same vector of registers from which one of the arithmetic source operands is being taken, the loaded data are used as the operands in an element-wise fashion. This is called "chain loading".
- Data simultanously stored from a register being used as a source or destination register is stored before the computation. Thus, it is not possible to store results as they are computed.

Like arithmetic opcodes, memory operation opcodes have a scalar and a vector form, indicated by the letters "**`s`**" or "**`v`**", respectively, appended to the operand. Within a single instruction, both operations must be either vector or scalar. Vector arithmetic operations and scalar memory operations cannot be mixed.

The basic syntax of a memory operation is

mem–opcode *as–mem–operand, Rls*

The *as–mem–operand* is nothing more than a SPARC address specifier, for example "**`[%i3+8]`**". The implied virtual memory address must be in one of the Parallel Instruction Space regions (see the "Accelerator Chip Memory Mapping" section at the beginning of this document).

As such, the address implies (1) which VUs will participate in the instruction, and (2) the offset into the associated Parallel Stack or Parallel Heap regions at which the memory data will be accessed. Even in instructions that do not perform a memory operation, it may still be desirable to specify a memory address. (An instruction modifier, the **`maddr`** modifier, provides this capability.)

For vector operations, subsequent memory data will be accessed at locations determined by adding a memory stride to this base location. This stride is always given in bytes.

By default, the memory stride is taken from the **`dp_stride_memory`** control register, but all instruction formats offer ways of overriding this default, including, in one format, changing the default value.

Note

Storing single word data is three times slower than storing doubleword data, due to the fact that the hardware must perform a read/modify/write sequence on a 64-bit wide memory data bus. As a result, you should avoid single-word stores when possible in favor of storing aligned pairs of words. See the section “Single Precision Memory Data Issues” for more on this.

As an example of a memory operation, here is a double-word load instruction:

```
dfloadv  [%i3+%i4]:=%i5,V5
```

This instruction computes the base memory address by adding the contents of the SPARC registers **`%i3`** and **`%i4`**. This example uses the instruction format that allows the default memory stride to be set. In this case, the stride is set by the contents of **`%i5`**. This stride will also be used as the memory stride in this instruction, thus defining the memory vector. The memory opcode, “**`dfloadv`**” specifies a doubleword load, and the destination vector is **`V5`**, i.e. the subsequent registers starting at **`R40`**.

4.4 The Short Instruction Format

The Short instruction format is the basic instruction format for the accelerator. Instructions should be coded in this format whenever possible, for such instructions occupy only a single word and therefore are more readily dispatched to the accelerator chips by the SPARC. The long formats augment the capabilities of the short format, at the cost of slightly slower instruction dispatch performance.

The short format instruction has the following syntax:

*opcode***`v`** {*Rs1*[**`:mode`**]`,`*Ss1***`:0`**}`,`*Vs2*`,`*Vd*`;` \
        mem–opcode as–mem–operand[`:`*tempstride*]`,`*Vls*`;`
    `pad:`*N*`;`
    `maddr=`*as–mem–operand*`;`
    {**`vmcurrent`**, **`vmrotate`**}`;`

Either the memory or the arithmetic operation can be omitted, in which case they are not performed (**NOP** operations are assembled).

The *Rs1* operand can be any vector in the register file, and several striding options are available for it. When striding by zero, *Rs1* must be one of the scalar registers, **S0** through **S15** (**S30** for double precision). Use of the **:mode** stride syntax indicates that the *Rs1* register stride is to be taken from the control register **dp_stride_rs1**. If omitted, the unit stride is used.

The *Vs2*, *Vd*, and *Vls* operands must be aligned vectors, and each must be strided by the unit stride (1 for single precision, 2 for double). The unit stride can be explicitly coded (e.g. "**Vn:1**") without effect.

The syntax for specifying the *as–mem–operand* is described in Section 3.10. The specified memory address must be in the Parallel Instruction Space. (See Chapter 2.)

The vector length used in a short format instruction is taken from the **dp_vector_length** control register, which holds the desired vector length minus one.

The Short format offers a small amount of control over the memory stride, in that it can be overridden, for this instruction only, to 4 or 8 for single precision memory operations, or to 8 or 16 for double precision memory operations. Overriding the stride, however, imposes additional restrictions on the specification of *Rs1*: it can no longer be any arbitrary register in the register file; instead, it, like *Rs2* and the others, must be vector aligned, or—if strided by zero—must be one of the scalar (*Snn*) registers.

Scalar Short Format

The scalar form of the short format is similar:

*opcode***s** {*Rs1*[**:mode**],*Ss1***:0**},*Ss2*,*Sd*; \
mem–opcode as–mem–operand[:*tempstride*],*Sls*;
pad:*N*;
maddr=*as–mem–operand*;
{**vmcurrent**, **vmrotate**};

The *Ss2*, *Sd*, and *Sls* operands must each be one of the scalar registers **S0** through **S15** (or **S30** for double precision).

It is not legal to mix a vector arithmetic operation (*opcode* ending with a **v**), with a scalar memory operation (*opcode* ending with an **s**).

Modifiers

Conditionalization of short format instructions is controlled by the current setting of various control registers. This format allows no override of these settings.

The **pad** modifier is used to pad the Datapath chip pipeline to avoid pipeline hazards. This is described in more detail in Section 9.1.2.

The **maddr** modifier is seldom used. It is used to specify a memory address when no memory operation has been specified. Recall that the selection of which VUs are to execute the instruction comes from the memory address. Thus, the **maddr** modifier allows instructions that do not reference memory to be directed at specific VU's. (See the discussion under the **dpchgbk** instruction for details on the use of this modifier.)

The **vmrotate** and **vmcurrent** modifiers are used only for "comparison" arithmetic operations. Such operations compare two values for some numerical relationship, and write a one or a zero into the **dp_vector_mask** as their status result:

Modifier	Effect
vmrotate	shift status into top of rotating **dp_vector_mask**
vmcurrent	write status to incrementing bit numbers in **dp_vector_mask**

The default is the rotate mode, which can be given explicitly with the **vmrotate** keyword. In this case, as bits are being rotated off one end of the Vector Mask for use in conditionalization, the comparison status bits are being rotated into the opposite end. Specifying the **vmcurrent** modifier causes the status bits to be placed into incrementing bit positions starting at the *lsb* (bit 0) and proceding toward the *msb*. (The remainder of the Vector Mask is still rotated).

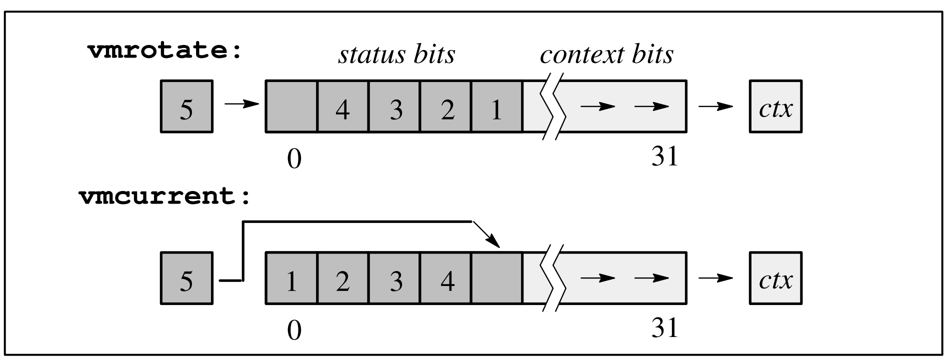

Figure 4. The two modes for movement of bits in dp_vector_mask

Some examples of instructions in the short format are

```
fmulv V4,V2,V7                        ! V4 <- V2 * V7
faddv V0,V1,V2; floadv [%i2],V1       ! V2 <- V0 + mem[%i2];
                                      ! V1 <- mem[%i2]
fstorev [%i4],V2                      ! mem[%i4] <- V2
```

Unless otherwise noted, all features and modifiers in the Short Format can be used in any of the extended formats described below.

4.5 The Immediate Instruction Format

The Immediate or "LI" instruction format is identical to the short format, except that the second (*Rs2*) operand is replaced by an immediate value. The immediate value is loaded into **R0** at the start of single precision instructions, or into **R0** and **R1** for double precision instructions. (This makes general use of these registers somewhat dubious; such use, therefore, should be avoided.) **R0**, with a register stride of 0, is then used as the second operand. See Section 3.14 for a description of the syntax and semantics of immediate operands.

For 2-operand operations, the syntax for this format is:

opcode {*Rs1*[**:mode**],*Ss1*:**0**}, *immediate–operand*, *Vd*; \
    mem–opcode as–mem–operand[:*tempstride*], *Vls*;
  pad:*N*;
  maddr=*as–mem–operand*;
  [**no**]**align;**

The immediate format can also be used for 1-operand (monadic) operations, by specifying the immediate value in the normal fashion, causing it to be loaded into **R0** (or **R0**/**R1**), and specifying **R0** with a stride of zero as *Rs1*, the operand to the monadic operation.

The syntax for monadic operations is:

monadic–opcode immediate–operand, *Vd*; \
    mem–opcode as–mem–operand[:*tempstride*], *Vls*;
  pad:*N*;
  maddr=*as–mem–operand*;
  [**no**]**align;**

This allows, for example, the instruction

```
fmovev   1234,V4
```

The following equivalent form is not allowed:

```
fmovev   R0:0,1234,V4         ! NOT ALLOWED
```

The **pad** modifier is used to pad the Datapath chip pipeline to avoid pipeline hazards. This is described in more detail in Section 9.1.2.

The **maddr** modifier is used to specify a memory address when no memory operation has been specified. This allows instructions that do not reference memory to be directed at specific VUs. (See the discussion under the **dpchgbk** instruction for details on the use of this modifier.)

The **align** modifier declares that the memory address is doubleword aligned, even though the memory operation may be in single precision. This modifier allows **dpas** to generate somewhat faster code to dispatch the instruction to the accelerator chip. You are encouraged to use this in all instructions that perform single-precision loads or stores where alignment can be guaranteed.

4.6 The Memory Stride Instruction Format

The Memory Stride format is also a simple extension of the short format. The syntax is the same, except that the memory stride used in the memory operation can be explictly given. Moreover, the memory stride given is sticky. That is, it becomes the current default memory stride for all subsequent instructions of other formats. (For a discussion of memory stride syntax, see Section 3.11.)

opcode {*Rs1*[**:mode**],*Ss1*:**0**}, *Vs2*, *Vd*; \
    mem–opcode as–mem–operand[:*tempstride*],*Vls*;
  pad:*N*;
  maddr=*as–mem–operand*;
  [**no**]**align;**

Notice that the memory stride override syntax offered by the short format is still available, and overrides even the specified stride. This combination allows you to write a default stride into the control register for future operations while using one of the override choices for the current operation.

The **pad** modifier is used to pad the Datapath chip pipeline to avoid pipeline hazards. This is described in more detail in Section 9.1.2.

The **maddr** modifier specifies a memory address when no memory operation is specified. This allows instructions that do not reference memory to be directed at specific VUs. See the discussion under the dpchgbk instruction for details on the use of this modifier.

The **align** modifier declares that the memory address is doubleword aligned, even though the memory operation is in single precision. Specifying this allows **dpas** to generate somewhat faster code to dispatch the instruction to the accelerator chip. You are encouraged to use this in all instructions which perform single-precision loads or stores for which alignment can be guaranteed.

4.7 The Mode Set Instruction Format

The Mode Setting format is the most complex format. It is a long format, and uses the extra 32 bits of instruction to control a number of additional features.

In particular, the following features are available through this format:

- Register Indirection on the *Rs1* operand.
- Memory Indirection in the memory operation.
- Overide and/or setting of the default vector length.
- Override of the default Conditionalization modes.
- Calculation of Population Counts.
- Accumulated count of the conditionalization bits (ACC).
- Exchange of data between the two VUs on each accelerator chip.

The format is as follows:

```
opcode[*[=]vlen]  Rs1[:[=]rs1–stride,(Ria:stride)],  Vs2,  Vd; \
        mem–opcode[*[=]vlen] as–mem–operand[:tempstride] \
        [(Ria[:stride])], Vls;
    pad:N;
    maddr=as–mem–operand
    vmmode:[=]{vmmode,always,condmem,condalu,cond}
    [no]exchange;
    {vmrotate, vmcurrent};
    [d]epc{v,s}(Vls[:unit])=Ria:stride;
    vmcount[s]=reg:stride;
    {vminvert, vmtrue};
    {vmold, vmnew, vmnop};
    [no]align;
```

The optional "*****[=]*vlen*" syntax in this format provides a way of overriding the current vector length default stored in the **dp_vector_length** control register. Including the optional equals sign (**=**) changes the default to the new override value. (Since the Vector Length control register actually contains the desired vector length minus one, the specified value less one is actually written).

The vector length override syntax is allowed on either the arithmetic or the memory opcode. It may be given on both for coding clarity, if desired, as long as the two specifications agree. The vector length can even be set in a scalar instruction, by appending "=*vlen*" to the opcode. This provides a fast way to set the default vector length for subsequent non-scalar instructions; it is much faster than writing the **dp_vector_length** register in a separate **dpset** instruction.

The *Rs1* stride can be temporarily overridden or permanently set (written to the **dp_stride_rs1** mode register) using the "*Rs1*:[=]*rs1–stride*" syntax. This feature cannot be used in conjunction with register or memory indirection or with the [**no**]**exchange**, [**d**]**epc**{**v**,**s**}, or **vmcount**[**s**] modifiers.

This format also allows the use of either Register Indirection (see Section 3.9) or Memory Indirection (see Section 3.12) but not of both together.

The **pad** modifier is used to pad the Datapath chip pipeline to avoid pipeline hazards. This is described in more detail in Section 9.1.2.

The **maddr** modifier is used to specify a memory address when no memory operation has been specified. This allows instructions that do not reference memory to be directed at specific VUs. See the discussion under the **dpchgbk** instruction for details on the use of this modifier.

The **`align`** modifier declares that the memory address is doubleword aligned, even though the memory operation may be in single precision. Specifying this allows **`dpas`** to generate somewhat faster code to dispatch the instruction to the accelerator chip. You are encouraged to use this in all instructions which perform single-precision loads or stores where alignment can be guaranteed.

The [no]exchange Modifier

The [**`no`**]**`exchange`** modifier controls the exchange of data between the two VUs housed within a single accelerator chip:

Modifier	Effect
`exchange`	Exchange results with other VU on the chip
`noexchange` (d)	Do not exchange results

Note: Here, as in the descriptions below, the (d) symbol indicates the default.

When the **`exchange`** modifier is given, all results of the ALU operation on each VU of each accelerator chip are written to the designated destination register(s) of the other VU. Thus, a pairwise exchange of the results is done between the two VUs on the chip.

Note that when performing conditionalized ALU operations, elements which are deselected in the computing VU will not be written to the opposite VU. All other elements are written, even if the corresponding element in the opposite VU is deselected. The **`exchange`** modifier cannot be used in conjunction with register or memory indirection or with the [**`d`**]**`epc`**{**`v`**,**`s`**}, or **`vmcount`**[**`s`**] modifiers.

Implementation Note: This feature is specific to the current accelerator chip, and may not be available in future implementations. Also, The current implementation does not allow chain-loading into the register being used as the destination in an exchange. For example, the following statement is illegal:

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

This must be recoded to use different source and destination registers in the arithmetic operation; for example,

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

The [d]epc{v,s} Modifier

The [**d**]**epc**{**v**,**s**} modifiers enable the "Population Count" feature of the accelerator chip. When specified, a vectored Population Count operation is performed.

Specifically, the single or double precision register *Rls* (and subsequent registers at a unit stride) are read and the the number of "1" bits in each are counted. The results, each a single precision unsigned integer between 0 and either 32 (single precision source) or 64 (double precision source), are written to *Ria* and subsequent single precision registers at the specified stride.

For each element in the vector, the count result is written before the operands are read for any accompanying arithmetic operation. Thus, it is possible to "chain load" the result counts into the arithmetic operation.

Modifier	Effect
[**d**]**epc**{**v**,**s**}**(***Vls*[**:***unit*]**)=***Ria***:***stride*	Compute Population Count

where *Vls* is any vector-aligned VU register, *Ria* is a VU register and *stride* is the stride for *Ria* as a constant or *constant-expression*.

For example, consider "**epcv(V4)=V7:2**", and assume that the vector length is set to 4. Further, assume that **R32**, **R33**, **R34**, and **R35** (the first four elements of **V4**) contain the values 6, 7, 8, and 9, respectively. The results of the Population Count operation, stored into **R56**, **R58**, **R60**, and **R62** (**V7** at a stride of 2) will be 2, 3, 1, and 2, the number of 1 bits in the source values. Note that *Rls* must always be unit strided (1 for single precision, 2 for double), and that such stride can be explictly coded if desired.

Since this operation uses the *Rls* register specifier, population counting cannot be used in conjuction with memory operations. population counting also cannot be used in conjunction with register or memory indirection or the **vmcount**[**s**] or [**no**]**exchange** modifiers.

In the current implementation, the [**d**]**epc**{**v**,**s**} modifiers cannot be used in conjunction with a long-latency arithmetic operation, i.e. [**f**,**df**]**div**, [**f**,**df**]**sqrt**, [**f**,**df**]**inv**, or [**f**,**df**]**isqt**.

Conditionalization

The remaining modifiers all affect the conditionalization mechanism. This mechanism is discussed in more detail in Section 9.2.

The vmmode modifier

The **vmmode** modifier is used to override the current Vector Mask Mode, which is stored in the control register **dp_vector_mask_mode**. It affects whether Arithmetic Operations and/or Memory Operations are to be subjected to (i.e. potentially nullified by) the conditionalization mechanism. The keyword to the modifier selects the mode. Additionally, for all except the **cond** mode, the selected mode can be either overridden just for the current instruction or used and written to the **dp_vector_mask_mode** register as the new default mode. The default is **vmmode:vmmode**.

Mode	Effect
vmmode:vmmode (d)	Use Vector Mask Mode in **dp_vector_mask_mode**
vmmode:always	Do not use conditionalization in this instruction
vmmode:=always	Set **dp_vector_mask_mode** for no conditionalization
vmmode:condmem	Conditionalize loads and stores in this instruction
vmmode:=condmem	Set **dp_vector_mask_mode** for conditionalization
vmmode:condalu	Conditionalize arithmetic in this instruction
vmmode:=condalu	Set **dp_vector_mask_mode** for conditional arithmetic
vmmode:=cond	Set **dp_vector_mask_mode** for full conditionalization

It is not possible to temporarily override the **dp_vector_mask_mode** for full conditionalization. Thus, "**vmmode:cond**" is not allowed. Since scalar instructions are always executed without conditionalization, you are allowed to add "**vmmode:always**" to any scalar instruction in any format with no effect. Similarly, you are permitted to add "**vmmode:vmmode**" to any vector instuction in any format since it represents the default action taken by the hardware.

The vmold, vmnew, And vmnop Modifiers

The **vmold**, **vmnew**, and **vmnop** modifiers control the setup of the Vector Mask and Vector Mask Buffer prior to the execution of the instruction. The default is **vmold**.

Modifier	Effect
vmold (d)	Copy **dp_vector_mask_buffer** to **dp_vector_mask**
vmnew	Copy **dp_vector_mask** to **dp_vector_mask_buffer**
vmnop	No copy

The vminvert And vmtrue Modifiers

The **vminvert** and **vmtrue** modifiers control whether or not the conditionalization bits being shifted off of the **dp_vector_mask** are to be inverted or not for the purpose of conditionalization. If inverted, the sensing of these bits is reversed; i.e. zero selects the element, and one deselects it.

Modifier	Effect
vminvert	Invert sense of vector mask bits for conditionalization
vmtrue (d)	Do not invert sense of vector mask bits

vmcount[s]

The **vmcount**[**s**] modifiers enable the "Accumulated Context Count" feature of the accelerator chip. When specified, single-precision register (and subsequent single precision registers at the given stride) will be loaded with the inclusive accumulated count of "1" bits encountered in the vector mask at each step in the vector operation. By "inclusive" we mean that the count generated for each element includes the bit shifted off of the Vector Mask for that element.

The scalar version, **vmcounts**, is intended for use along with scalar arithmetic and memory operations. It is an error to mix a scalar **vmcount** and any arithmetic or memory vector operation, or vice versa. The loading of these counts is not affected by the conditionalization mechanism.

For example, if **dp_vector_mask** contains 01011011 before the instruction of vector length 8 begins, and assuming the shift direction is to the right, the 8 registers starting with register and at the given stride will be written with the integers 1, 2, 2, 3, 4, 4, 5, and 5 respectively.

For each element in the vector, the **vmcount** result is written before the operands are read for any accompanying arithmetic operation. Thus, one can "chain load" the **vmcount** counts into the arithmetic operation. This modifier cannot be used in conjunction with either register or memory indirection, nor with the [**d**]**epc**{**v**,**s**}, or [**no**]**exchange** modifiers.

Modifier	Effect
vmcount=*register***:***stride*	Enable Vector Mask Count

where *register* is any VU register name, and *stride* is the stride value, a constant or *constant-expression*.

The vmrotate And vmcurrent Modifiers

The **`vmrotate`** and **`vmcurrent`** modifiers control the writing of status bits generated by the instruction into the **`dp_vector_mask`** register.

Modifier	Effect
`vmrotate` (d)	Shift status into top of rotating **`dp_vector_mask`**
`vmcurrent`	Write status to incrementing bit numbers in **`dp_vector_mask`**

The default is the **`vmrotate`** mode, which can be given explicitly with the **`vmrotate`** keyword; as bits are rotated off one end of the Vector Mask for use in conditionalization, the comparison status bits are rotated into the opposite end.

The **`vmcurrent`** mode causes the status bits to be placed into incrementing bit positions starting at the *lsb* (bit 0) and proceding toward the *msb*. (The remainder of the Vector Mask is still rotated.)

The **`vmrotate`** and **`vmcurrent`** modifiers may also be specified with the compare instructions in all instruction formats. For compares, ones are written for elements in which the comparison generated the "true" sense; zeros are written for all other elements.

4.8 The Register Stride Instruction Format

opcode {*Rs1*[**`:mode`**],*Ss1*`:0`}, *Rs2*[`:`*stride*], *Rd*[`:`*stride*]; \
mem–opcode as–mem–operand[`:`*tempstride*], *Rls*[`:`*stride*]
`maddr=`*as–mem–operand*`;`
[**`no`**]**`align;`**

The Register Stride Instruction Format is an extension of the Short format, except that it allows the *Rs2*, *Rd*, and *Rls* operand registers to be specified in full. These operand registers can, therefore, be almost any of the 128 registers of the register file, without the requirement of vector alignment. In addition, the strides for these operand registers can be given as arbitrary constants.

The encoding for the use of immediate operands by the accelerator chip causes some restrictions on the use of the Register Stride format, as follows:

- In vector instructions, *Rs2* cannot be **`R0`** through **`R7`** (**`S0`** through **`S7`**).
- In scalar single precision instructions, *Rs2* cannot be a multiple of 16.
- In scalar double precision operations, *Rs2* cannot be a multiple of 32.

This is obscure, to be sure, but violations are not likely to be common, and **dpas** will detect any violations that do occur.

The **maddr** modifier is used to specify a memory address when no memory operation has been specified. This allows instructions that do not reference memory to be directed at specific VUs. See the discussion under the **dpchgbk** instruction for details on the use of this modifier.

The **align** modifier declares that the memory address is doubleword aligned, even though the memory operation is in single precision. Specifying this allows **dpas** to generate somewhat faster code to dispatch the instruction to the accelerator chip. You are encouraged to use this in all instructions which perform single-precision loads or stores where alignment can be guaranteed.

4.9 Triadic Arithmetic Operations

Some arithmetic operations require three input operands. For example, the **fmadt** operation multiplies two operands, adds the result to a third, and stores the result into a destination.

Triadic operations take their three operands from the *Rs1*, *Rs2*, and *Rls* registers. The *Rls* register is placed between the two normal source operands. Recall that normally, *Rls* specifies the register from or to which memory operations are performed. Triadic operations may still be accompanied by a *load* memory operation, but not by a *store* operation. If the instruction includes a *load*, the Rls register will be chain-loaded into the triadic operation.

In triadic operations, the default *Rls* stride is driven by the precision of the arithmetic operation, not by the memory operation, as is usually the case. If the operation is combined with a memory load, the two *Rls* specifications must agree. The following example shows a triadic operation combined with a load. It uses the fmadtv triadic, which performs *Rd* = (*Rs1* * *Rls*) + *Rs2*. The *Rls* register is chain-loaded.

```
fmadtv V0, V1, V2, V3; floadv V1;
                                  ! V3 = (V0 * (V1=mem)) + V2
```

4.10 Instruction Format Summary

The following tables summarize the instruction syntax for dyadic and triadic operations. (For monadic operations, omit the second operand.)

Arithmetic and Memory Operation Instruction Formats

Format	Scalar Opcode	Operands
Short	*opcode*	*Rs1*, [*Sls*,] *Ss2*, *Sd*
Immed.	*opcode*	*Rs1*,[*Sls*,]{[**$**]*general–expression*,*as–register*}, *Sd*
Immed.	*monadic–opcode*	{[**$**]*general–expression*,*as–register*}, *Sd*
M. Stride	*opcode*	*Rs1*, [*Sls*,] *Ss2*, *Sd*
Mode	*opcode*	*Rs1*[**=***stride*], [*Sls*,] *Vs2*, *Sd*
Mode	*opcode*[**=***vlen*]	*Rs1*(*Ria*), [*Sls*,] *Vs2*, *Sd*
Stride	*opcode*[**=***vlen*]	*Rs1*, [*Sls*,] *Rs2*, *Rd*

Format	Opcode	Operands
Short	*mem–opcode*	*as–mem–operand*[**:***tempstride*],*Vls*
Immed	*mem–opcode*	*as–mem–operand*[**:***tempstride*],*Vls*
MStride	*mem–opcode*	*as–mem–operand*[**:***tempstride*]**=***memstride*, *Vls*
Mode Set	*mem–opcode**[=]vlen	*as–mem–operand*[**:***tempstride*][(*Ria*[**:***stride*])], *Vls*
Reg Stride	*mem–opcode*	*as–mem–operand*[**:***tempstride*], *Rls*[**:***stride*]

Modifiers

Formats	Modifier
All	**maddr=***as–mem–operand*
All	[**no**]**align**
Mode Set	**vmmode:**{[**=**]**vmmode**,[**=**]**always,**[**=**]condmem, [**=**]**condalu**,**=cond**}
Mode Set	[**no**]**exchange**
Mode Set	{**vmrotate**, **vmcurrent**}
Mode Set	[**d**]**epc**{**v**,**s**}(*Vls*[**:***unit*])**=***Ria*:*stride*
Mode Set	**vmcount**[**s**]**=***register***:***stride*
Mode Set	{**vminvert**, **vmtrue**}
Mode Set	{**vmold**, **vmnew**, **vmnop**}
Mode Set	[**no**]**align**

4.11 Single Precision Memory Data Issues

The accelerator chip operates using 64-bit wide memory data busses to its memory. As a result, you should keep several important things in mind when programming using single precision (32-bit) data, i.e. data of types **i**, **u**, or **f**.

First, due to the 64-bit wide memory, store operations of single precision data, such as **fstorev**, **istores**, etc., must be done by the hardware using a read-modify-write operation. This is three times slower than the simple write used for double precision (64-bit) data. As a result, you may want to store single precision data from the registers to memory in pairs, i.e. two values at a time. Similarly, the real and imaginary halves of single precision "complex" numbers may best be handled together in aligned pairs. Finally, it may be advantageous to pad single precision data out to 64 bits, wasting memory, but allowing the numbers to be handled singly without loss of performance. Note that in the latter cases (complex and padded), the values stored in the registers will be stored at strides of 2 in the register file, which can restrict the choice of instruction formats.

Second, the **dpas** assembler must generate slightly less efficient code for long format DPEAC instructions that perform memory operations on single precision data, which may not be double-word aligned. You have two ways of preventing this. First, on such instructions, the **align** modifier may be added to indicate that the memory address is guaranteed to be doubleword-aligned. For example:

```
faddv v1,0f1.0,v3; floadv [%i3],v1; align
```

tells **dpas** that the address in **%i3** is doubleword aligned. (The use of the immediate value forces a long format (SI) instruction.)

If all such addresses will be doubleword-aligned within the assembly of a file, the **-a** switch can be given to **dpas** to cause it to assume such alignment as the default. In this case, the **noalign** modifier can be used on instructions where memory addresses may not be doubleword-aligned.