Chapter 3

The DPEAC Instruction Set

The DPEAC instruction set is an extension of SPARC assembly code, providing extra instructions that are used to manipulate the vector units. When a routine containing DPEAC code is assembled, each DPEAC instruction is translated into one or more SPARC memory operations that send appropriately assembled instruction word(s) to the VU hardware.

3.1 DPEAC Code

A DPEAC routine consists of a series of *statements*. Each statement is either a SPARC instruction, a *DPEAC statement*, or a *DPEAC accessor instruction*. A DPEAC statement can occupy either a single text line or several text lines, with a "****" character immediately preceding each linebreak but the last.

A *DPEAC statement* consists of one or more *DPEAC instructions*, separated by semicolons. (An optional extra semicolon can follow the last instruction.) DPEAC instructions are grouped in three categories:

- *arithmetic instructions*, which cause the VUs to perform register arithmetic

```
dfaddv   V0,V2,V4
```

- *memory instructions*, which move data between VU registers and memory

```
floadv   [%i0]:4,V0
fstorev  [%i1]:8,R16
```

- *modifiers*, which alter the assembly or execution of the DPEAC statement

```
vmcurrent ; noalign ; vmnew
```

A single DPEAC statement represents a single VU operation, and thus can contain both a memory instruction and an arithmetic instruction (but no more than one of each type). At least one memory or arithmetic instruction must be present. When a DPEAC statement includes both memory and arithmetic instructions, the memory instruction executes first, and any value it obtains from memory can be used by the arithmetic instruction.

A DPEAC statement may also include any number (including zero) of modifiers, as permitted by the statement's format.

The components of a DPEAC statement may be arranged in any order, but for readability you should adopt a consistent form. A good "canonical" DPEAC statement order, used by many DPEAC programmers, is:

arithmetic-op `;` *memory-op* `;` *modifier-1* `;` *modifier-2 ...*

This order is recommended because although the memory operation and modifiers are usually executed and/or applied before the arithmetic operation, it is the arithmetic part of the instruction that is typically of the greatest interest.

3.1.1 Chain Loading

When a DPEAC statement refers to the same register in both the memory and arithmetic operations, and when the memory operation is a **`load`**, the loaded value from the memory operation is used in the arithmetic operation. This is called *chain loading*. In a vector operation, this can happen for each step in the vector operation.

There are some modifier operations (such as population counting), that can also chain load, and some modifier operations that *cannot* chain load. Section 4.3 lists the DPEAC modifiers and indicates any that can or cannot chain load.

3.1.2 DPEAC Accessor Instructions

A DPEAC accessor instruction is a DPEAC instruction that doesn't correspond to a VU arithmetic/memory operation. DPEAC accessor instructions are typically utility operations such as reading and writing VU registers from the SPARC, directly reading and writing parallel memory locations, etc. Accessor instructions can be recognized by their "**`dp`**" prefix: **`dpset`**, **`dpget`**, etc.

3.2 DPEAC Syntax

3.2.1 General Syntax

Numbers are 64-bit constants, parsed as in C. Numbers starting with 0 are octal by default, and numbers starting with a non-zero digit are decimal. The `0x` (hex), `0b` (binary), `0o` (octal), and `0n` (decimal) integer forms are provided, as well as `0f` (32-bit), `0r` (32-bit) and `0d` (64-bit) IEEE float forms.

ASCII constants appear in single quotes ('ABC'), and represent the integer obtained by concatenating the character bytes (the first byte is most significant). Comments are denoted by a "`!`", and extend to the end of the line (as in `as`). C-like `/*`*comment*`*/` and `#`*comment* forms are provided by `dpas` itself.

Expressions in DPEAC evaluate to a constant when assembled. There are three classes: *constant-expressions*, *as-expressions*, and *general-expressions*.

A *constant-expression* is evaluated at `dpas` assembly time, using 64-bit integer arithmetic (signed for products/divisions, else unsigned). The following operators are supported, and are evaluated in the order shown (first across, then down):

+	Unary plus (no-op)	–	Negate (2's complement)
!	Logical not	~	Invert (1's complement)
%lo	Low 10 bits	%hi	High 22 bits
&	Bitwise AND	|	Bitwise OR
^	Bitwise XOR		
*	Signed multiply	/	Signed divide
<<	Logical Left shift	>>	Logical Right shift
+	Addition	–	Subtraction
<	Less than	<=	Less than or equal
==	Equal to	!=	Not equal to (<> also allowed)
>	Greater than	>=	Greater than or equal
&&	Logical AND	||	Logical OR

A constant-expression can include symbols only if they can be translated into constants by the `dpas` preprocessor. Floating-point constants are allowed, but are "cast" as integers. (Note that floating-point constants and the operators `<=` and `&&` are `dpas` extensions to `as` expression syntax.)

An *as-expression* is passed directly to the `as` assembler, and must follow `as` syntax. It is evaluated as a 32-bit integer. It can contain any symbols processed by `as`, but cannot contain either float constants or the operators `<=` and `&&`.

A *general-expression* can be either a constant-expression or an as-expression. If `dpas` cannot parse a general-expression as a constant-expression, it assumes it is an as-expression, and passes it directly to `as`.

3.2.2 SPARC CPU Registers

DPEAC memory instructions refer to SPARC registers by these symbols:

`%r0` – `%r31`	IU registers	`%0` – `%31`	IU registers (alternate form)
`%i0` – `%i7`	in registers	`%o0` – `%o7`	out registers
`%f0` – `%f31`	FP registers	`%g0` – `%g7`	global registers
`%psr`	Processor state register	`%fsr`	FP state register
`%wim`	Window invalid register	`%tbr`	Trap base register
`%y`	Multiply-step (Y) register	`%fq`	FP queue register

(**Note:** Later descriptions denote an arbitrary SPARC register as *as-register.*)

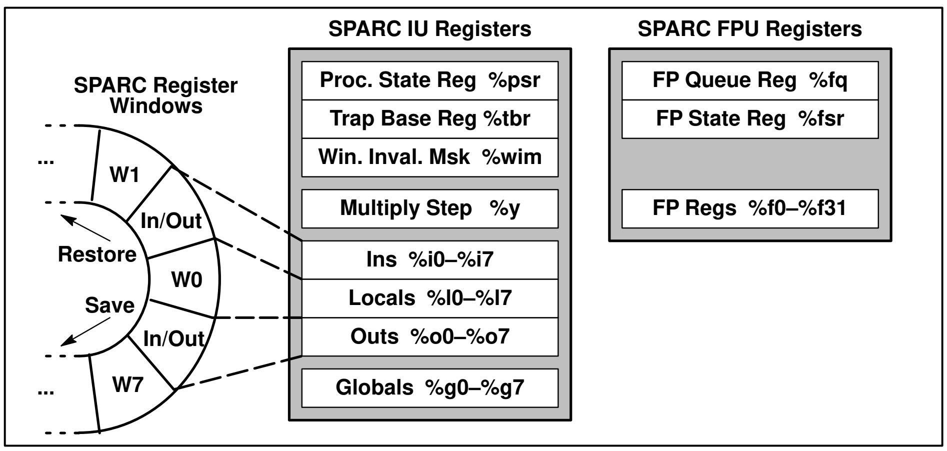

Figure 12. SPARC CPU registers accessible from DPEAC.

Register Restrictions: The following SPARC registers are used by DPEAC operations to store default values, so these registers should be avoided:

Register	Usage	
`%l6`	(Reserved)	Default memory operand (Selects all VU's)
`%l7`	(Reserved)	`dp_instruction_ext` register pointer
`%g2`, `%g3`	(Temporaries)	Used as temporaries in VU instructions

`%l6` and `%l7` are initialized by `dpas`, which expects them to be preserved. `dpas` assumes that these registers are no longer correct if a `.seg` directive or a `dpunset` accessor instruction is included. `%g2` and `%g3` are overwritten by `dpas` code execution, but are not expected to be preserved. Thus, these registers can be used as temporaries in code that has no VU instructions.

3.2.3 Vector Unit Data Registers

DPEAC arithmetic and memory instructions refer to the 128 VU *data registers* by the following names:

`R0` – **`R127`**	All 128 registers in sequential order
`V0` – **`V15`**	Vector regs (first in each vector, same as **`R0`**, **`R8`** ... **`R120`**)
`S0` – **`S15`**	Scalar regs (single precision), same as **`R0`** – **`R15`**
`S0` – **`S30`** (even)	Scalar regs (double precision), same as **`R0`** – **`R30`** (even)

Restrictions: DPEAC statements in immediate format use the **`R0`** and **`R1`** registers to store immediate operands, so these registers should be avoided.

The VU data registers are grouped in banks of 8, called *vector* registers. The special register names **`V0`** – **`V15`** are used to refer to the first data register in each vector. When a vector instruction requires an "aligned vector" operand, the operand must be one of the **`V`***nn* registers (or the equivalent **`R`***nn*).

V0	V1	V2	V3	V4	V5	V6	V7		V14	V15
R0	R8	R16	R24	R32	R40	R48	R56	_ _ _ _ _ _	R112	R120
R1	R9	R17	R25	R33	R41	R49	R57	_ _ _ _ _ _	R113	R121
R2	R10	R18	R26	R34	R42	R50	R58	_ _ _ _ _ _	R114	R122
R3	R11	R19	R27	R35	R43	R51	R59	_ _ _ _ _ _	R115	R123
R4	R12	R20	R28	R36	R44	R52	R60	_ _ _ _ _ _	R116	R124
R5	R13	R21	R29	R37	R45	R53	R61	_ _ _ _ _ _	R117	R125
R6	R14	R22	R30	R38	R46	R54	R62	_ _ _ _ _ _	R118	R126
R7	R15	R23	R31	R39	R47	R55	R63	_ _ _ _ _ _	R119	R127

Figure 13. VU data registers: 16 vectors of 8 registers.

A subset of these registers is designated as the *scalar registers*. These are **`S0`** – **`S15`** (singleword), or the even registers from **`S0`** – **`S30`** (doubleword). (The **`S`***nn* names are equivalent to **`R`***nn*, and explicitly show use of scalar registers.) Scalar operations that use scalar registers assemble into efficient instructions.

You can apply a *register offset* to a data register to access one of the registers succeeding it in **`R`***nn* order (this is mainly useful for accessing the elements of **`V`***nn* vectors):

Regn`[`*k*`]`	Refers to register *Regn* + *k*. (Ex: **`V2[5]`** = **`R16`**+**`5`** = **`R21`**)

3.2.4 Vector Unit Control Registers

There are symbols for all the VU control registers, as described in Section 2.5. However, these symbols are typically only used for accessor instructions such as **`dpset`** and **`dpget`**. DPEAC statement formats allow you to implicitly use and/or set the value of one or more control registers while executing a VU operation. See the mode set format in particular (Section 3.9) for examples.

3.2.5 VU Register and Memory Stride Markers

Some VU arithmetic and memory operations can stride through a group of registers or memory addresses. The stride length is indicated by a *stride marker* attached to the appropriate register or memory operand. The generic syntax of these markers is shown below.

Important: The stride markers shown here are not valid for all statement formats; most statement formats restrict the types of stride markers that are allowed.

Register Stride Markers

The general syntax of register stride markers is shown below, where *register* is any valid VU register, and *stride* and *set-stride* are constant expressions in the range –128 to +128. Register striding is always in terms of the **`R`***nn* ordering, even when a **`V`***nn* register name is specified. A stride of zero causes the same register to be used at each step.

Syntax	Effect
register	Use unit stride (1 for words, 2 for doublewords).
register`:`*stride*	Temporarily use specified *stride*.
register`:`**`mode`**	Use stride value stored in **`dp_stride_rs1`**.
register`:=`*stride*	Set **`dp_stride_rs1`** to *stride* and use it.
register`:`*stride*`=`*set-stride*	Set **`dp_stride_rs1`** to *set-stride*, but use *stride*.
register`=`*stride*	Set **`dp_stride_rs1`** to *stride* (scalar ops only).

Note: The last four stride marker forms shown above are valid only for the *rS1* register argument of an arithmetic instruction.

rS1 Stride Restriction: When you apply a stride of 0 to the *rS1* argument of an arithmetic operation (for example, **`R0:0`**), the *rS1* register must be one of the scalar registers **`S0`** through **`S15`**, or **`S30`** for double-precision.

VU Memory Stride Markers

The general syntax of memory stride markers is shown below, where *n* and *set-n* is a general expression or *as-register* giving the stride in bytes. Note that the *stride* value is limited to a 24-bit signed integer. A stride of zero causes the same address to be used at each step.

Syntax	Effect
memory-operand	Use default stride in **`dp_stride_memory`**.
memory-operand`:`*n*	Temporarily stride by *n* bytes.
memory-operand`:=`*n*	Set **`dp_stride_memory`** to *n* and use it.
memory-operand`:`*n*`=`*set-n*	Set **`dp_stride_memory`** to *set-n*, but use *n*.
memory-operand`=`*n*	Set **`dp_stride_memory`** to *n* (scalar ops only).

In the above formats, *n* and *set-n* are either **`4`** or **`8`** for singleword data types, or **`8`** or **`16`** for doubleword data types.

When you write DPEAC code by hand, you should make sure the default memory stride register **`dp_stride_memory`** is set to the stride you require (for example, 4 bytes for single-precision or 8 bytes for double-precision). You can use the DPEAC accessor instruction **`dpset`** for this purpose; for example:

```
dpset    ALL_DPS, 8, DP_STRIDE_MEMORY
```

3.2.6 VU Selection in DPEAC Statements

The VUs that execute a DPEAC statement are selected by the memory address specified in the statement. (Deselected VUs are effectively idle.) A DPEAC statement's memory address is:

- the value of the *memory-operand* in the memory instruction
- the value specified by the **`maddr`** modifier, if any
- If neither of these is supplied, a default address that selects all the VUs. The default address used is **`DPV_STACK_INST_PORT_ALL`**.

Typically, you won't construct these memory addresses yourself; your compiler and/or the **`dpas`** assembler generate these addresses for you.

3.2.7 VU Selection in DPEAC Accessor Instructions

The VU(s) referenced by a DPEAC accessor instruction are determined by the "VU selector" argument. This argument must be a valid VU selector as described below.

A *VU selector* is an integer or symbolic constant that specifies one or more VUs to perform a given accessor instruction. The syntax is:

Syntax	Immediate Value
constant-expression	Use the specified selector constant (see table below).
as-register	Use value from a SPARC register (all bits).
as-register`<`	Use value from SPARC register (bits `12:15`).
`*`	Select all VUs.
`*`*n*	Use both VUs on chip *n* (0=VU's 0&1, 1=VU's 2&3).

The modifier "`<`" makes *as-register* references faster (fewer SPARC operations) because only 4 bits (`12` through `15`) of the register are used. The *constant-expression* form can be either an integer VU selector value, a physical VU selector (an integer preceded by a "`$`"), or one of the symbols defined in the header file `dp.h` for these values. (Use of predefined symbols is recommended.)

The legal VU selector values, and their corresponding symbols, are:

VU	VU Selector		Physical VU Selector	
Number(s)	Value	Symbol	Selector	Symbol
VU *n*	2*n*	`DP_`*n*	`$`*n*	`DP_PHYS_NUM_`*n*
ALL VUs	8	`ALL_DPS`	`$8`	`ALL_PHYS_NUM_DPS`
VUs 0 and 1	10	`DPS_0_AND_1`	`$9`	`DP_PHYS_NUM_0_AND_1`
VUs 2 and 3	12	`DPS_2_AND_3`	`$11`	`DP_PHYS_NUM_2_AND_3`

3.3 DPEAC Instructions

3.3.1 Scalar and Vector Instructions

DPEAC memory and arithmetic instructions come in two forms: *scalar* and *vector.*

Scalar instructions execute just once for the supplied operands, and are distinguished by an "**`s`**" suffix on the opcode.

Vector instructions execute repeatedly for each of a series of operands, and are distinguished by a "**`v`**" suffix on the opcode. Vector operations start with the specified register or memory address operand(s) and then step through succeeding locations determined by the *vector stride* and *vector length*:

- The *vector stride* determines the number of registers or memory addresses a vector operation advances at each step. The default vector stride depends on the type of operation (memory or arithmetic).
- The *vector length* determines the number of registers or memory addresses affected by a vector instruction. The vector length defaults to the value of the VU register **`dp_vector_length`**, unless a different vector length is specified explicitly.

Note: If a DPEAC statement includes both a memory instruction and an arithmetic instruction, the two must agree in form: they must be either both scalar instructions or both vector instructions.

3.3.2 Register Operands

The register operands of arithmetic and memory instructions are indicated by the following symbols, indicating arbitrary VU registers:

rS1, rS2	First and second source registers.
rLS	Load/store (or third source) register.
rD	Destination register.
rIA	Indirect addressing (used in Register Indirect format).

When an instruction format requires vector (**`V`***nn*) register arguments, the symbols *vS1*, *vS2*, *vLS*, *vD*, and *vIA* are used instead. Similarly, when scalar (**`S`***nn*) register arguments are required, the symbols *sS1*, *sS2*, *sLS*, *sD*, and *sIA* are used.

3.3.3 Data Types

The data type of a DPEAC instruction is typically indicated by one of the following prefixes on the instruction opcode:

`i`	Signed 32-bit integer	**`u`**	Unsigned 32-bit integer
`di`	Signed 64-bit integer	**`du`**	Unsigned 64-bit integer
`f`	Single (32-bit) float	**`df`**	Double (64-bit) float

3.3.4 Arithmetic Instructions

An arithmetic instruction causes the VUs to perform a register arithmetic operation. Arithmetic instructions have the following general forms, where *opcode* is: {**`i`**,**`di`**,**`u`**,**`du`**,**`f`**,**`df`**}*operation*{**`v`**,**`s`**}

Monadic (one source argument):	*opcode*	*rS1*, *rD*
Dyadic (two source arguments):	*opcode*	*rS1*, *rS2*, *rD*
Triadic (three source arguments):	*opcode*	*rS1*, *rLS*, *rS2*, *rD*

Note: In the statement format descriptions in Section 3.4, the arithmetic operation is always shown in triadic form. Dyadic and monadic forms are obtained simply by omitting the appropriate operand symbols (*rLS* and *rS2*).

(Appendix D describes the VU arithmetic instructions in detail, and describes the VU status bits that are affected by each instruction.)

Vector instructions have a default stride of 1 (singleword) or 2 (doubleword) for register operands, unless the instruction explicitly specifies a different stride.

rS2 Operand Restrictions: The *rS2* operand of an arithmetic instruction has the following restrictions:

- For vector operations, *rS2* cannot be any of **`R0`** through **`R7`**, by any name (**`S0`**, **`V0`**, etc.).
- In scalar operations, *rS2* cannot be any of **`R`***nn*, where *nn* is any multiple of **`16`** (for single-precision) or **`32`** (for double-precision).

This restriction is imposed by the internal representation of DPEAC operations.

Triadic/Memory Register Restriction Note: When a triadic arithmetic operation and a memory operation are **`join`**ed, the *rLS* operand of the arithmetic operation must be identical to the *rLS* operand of the memory operation.

3.3.5 Memory Instructions

A memory instruction causes the VUs to move data between memory and VU registers. The operands of a memory instruction are a memory address and a VU register. Memory instructions have the following general form:

{**`i`**,**`di`**,**`u`**,**`du`**,**`f`**,**`df`**}*memory-operation*{**`v`**,**`s`**} *memory-operand*, *rLS*

The *rLS* operand can be any VU register, but if a triadic arithmetic operation and a memory operation are combined, the *rLS* operand of both must be the same, and the memory operation can only be a **`load`**, not a **`store`**. The default stride for the *rLS* register is determined by the arithmetic operation, and the stride required by the memory operation must agree.

The *memory-operand* can be any memory address that selects one or more VUs, and it is specified by SPARC register indirection, using Sun-4 Assembler syntax:

Syntax	**Memory Address**
`[`*as-register*`]`	Contents of *as-register*
`[`*as-register1* `+` *as-register2*`]`	Sum of *as-register1* and *as-register2*
`[`*as-register* `+` *offset*`]`	Contents of *as-register* + *offset*

where *offset* is limited to the range **`-4096`** to **`+4095`**. Note that double precision memory references must be doubleword (8-byte) -aligned.

The stride of vector instructions is either specified explicitly in the instruction, or else defaults to the **`dp_stride_memory`** register value.

Singleword / Doubleword Performance Note: Doublewords are the natural word size for the VUs. Singleword operations require a read-modify-write step. Thus, singleword operations are less efficient than doubleword operations.

3.3.6 Modifiers

Modifiers: Modifiers are keywords, such as **`pad`**, **`maddr`**, **`vmcurrent`**, etc., that modify the assembly or execution of a DPEAC statement. The modifiers permitted in a DPEAC statement are determined by the statement's format. The available modifiers are listed below, and described in more detail in Section 4.3.

Modifiers That Can Be Used in All (or Most) Formats:

[**`no`**]**`pad`**[**`:`***pad-size*]	Pad vector length
`maddr=`*memory-operand*	Default memory address
{**`vmrotate`**, **`vmcurrent`**}	Packing mode for vector mask bits
[**`no`**]**`align`**	Doubleword alignment declaration
`vmmode:`[**`=`**]*mode-keyword*	Conditionalization mode selector

Conditionalization Modifiers (Mode Set Format Only):

{**`vminvert`**, **`vmtrue`**}	Conditionalization bit sense selector
{**`vmold`**, **`vmnew`**, **`vmnop`**}	Vector mask copying mode

Special Modifiers (Mode Set Format Only):

[**`d`**]**`epc`**{**`v`**,**`s`**} **`(`***vLS***`)=`***rIA***`:`***stride*	Population count
`vmcount`[**`s`**]**`=`***reg***`:`***stride*	Accumulated context count
[**`no`**]**`exchange`**	On-chip VU data exchange

3.4 DPEAC Statement Formats

There are two main classes of statement formats, *short format* and *long format*. The distinction comes from the way the two formats are assembled into SPARC operations.

A *short format statement* assembles into a singleword (32-bit) operation. Short format instructions execute faster than those in long format, but lack some of the features provided by the long format.

A *long format statement* assembles into a doubleword (64-bit) operation. Long format instructions are slower to issue, but use the extra word to provide additional operand types and modifiers that are not permitted by the short format. Specifically, the long instruction format comes in three varieties:

Immediate format allows an immediate operand in the arithmetic operation.

Register stride format allows register striding in the arithmetic operation.

Memory stride format allows address striding in the memory operation.

Mode set format provides access to a number of VU features, including register/memory indirection and overriding of many VU instruction defaults.

Each of the varieties of long format represents a modification of the short format. In terms of DPEAC source code, you can think of the short format as the backbone of features that all DPEAC source lines share, with each of the long formats representing some modification of or addition to those features.

Important! Because of the way that DPEAC code is assembled, the modifications provided by each of the long formats cannot be combined. You can use only one of the long formats, or none of them (that is, use the short format) in a single statement.

For the Curious: Each DPEAC statement is assembled into a word (or doubleword) containing fields for each of the opcodes and operands in the statement. Each of the long formats is assembled as a doubleword, and uses the extra word for a different purpose; thus the extensions provided by the long formats are physically incompatible within a single DPEAC statement.

Note: In the syntax descriptions below, escaped linebreaks (indicated by "****") are sometimes inserted for clarity when a statement's syntax is long and/or complex. These linebreaks are *not* a syntax requirement — all statement formats occupy one line in a DPEAC program.

3.5 The Short Format

The *short statement format* is:

Vector Instuctions:

arith-opcode rS1, vLS, vS2, vD `;` *mem-opcode mem-operand, vLS* `;` *modifier* `;` *...*

Scalar Instuctions:

arith-opcode rS1, sLS, sS2, sD `;` *mem-opcode mem-operand, sLS* `;` *modifier* `;` *...*

With one exception (the mode set statement format, see Section 3.9), the *rS1* operand can only have one of the following explicit stride forms:

rS1	Use register *rS1*, with unit stride for vector ops.
rS1`:mode`	Use register *rS1*, with **`dp_stride_rs1`** stride.
sS1:0	Use scalar register *sS1* with 0 stride.

The remaining register operand(s) must be aligned vector (**`V`***nn*) registers for a vector operation, or scalar (**`S`***nn*) registers for a scalar operation. Vector instructions always use unit striding, so stride markers are not allowed in short format (see register stride format, Section 3.7).

The *mem-operand* must have one of the following forms:

mem-operand	Use **`dp_stride_memory`** stride.
mem-operand[`:`*tempstride*]	Use restricted *tempstride*.

The optional *tempstride* is restricted to **`4`** or **`8`** for single word operations, **`8`** or **`16`** for doubleword operations (see memory stride format, Section 3.8). If *tempstride* is specified, the *rS1* operand must be an aligned vector (**`V`***nn*) register or a scalar (**`S`***nn*) register with a stride of 0.

The vector length is taken from **`dp_vector_length`**. This cannot be overridden in the short format (see mode set format, Section 3.9)

Only the following modifiers are permitted by the short format:

[**`no`**]**`pad`**[`:`*pad-size*]	Vector length padding (default is 4).
`maddr=`*mem-operand*	Memory operand specifier.
[**`no`**]**`align`**	Doubleword alignment guarantee.
{**`vmrotate`**, **`vmcurrent`**}	Status bit rotation mode.
`vmmode:`[**`=`**]*mode-keyword*	Conditionalization mode selector

Note: {**`vmcurrent`**, **`vmrotate`**} are useful only for comparison operations, where the result of the comparison produces status bits that can be rotated.

Examples:

```
imovev   V0,V1                    ! Integer monadic
imovev   V0,V1; iloadv [%i0],V0   ! same, chain-loaded
imovev   V0,V1; iloadv [%i0]:8,V0 ! same, temp stride
imovev   V0:mode,V1               ! Default reg. stride
imovev   S0:0,V1                  ! Scalar reg, 0 stride
imoves   S0:0,S1                  ! Scalar operation

iloadv   [%i0],V1                 ! memory operation
iloadv   [%i0],V1; noalign        ! same, non-aligned
floadv   [%i0]:4,V0               ! unit stride
floadv   [%i0]:8,V0               ! double stride
dfloadv  [%i0]:8,V0               ! unit stride
dfloadv  [%i0]:16,V0              ! double stride

itestv   V0,V1; maddr=[%i0]       ! maddr modifier
dfgtv    V0,V1                    ! Conditional
dfgtv    V0,V1; vmcurrent         ! same, with modifier

faddv    V0,V1,V2                 ! Float dyadic
faddv    V0,V1,V2; nopad          ! No vlen padding
fmadav   V0,V1,V2                 ! Mult-add
fmadiv   V0,V1,V2                 ! Mult-add, inverted
fmadtv   V0,V1,V2,V3; floadv [%i0],V1
                           ! True triadic, chain-loaded
```

3.6 Immediate (Long) Format

The *immediate format* modifies the short format by replacing one source operand in the arithmetic instruction with an immediate value. (The operand replaced depends on the arithmetic instruction in use — see the instruction listings in Chapter 4.) The immediate value is loaded into **`R0`** (singleword operations) or **`R0`** and **`R1`** (doubleword operations) prior to use.

Vector Instuctions: (*rS2* replaced with immediate value)
arith-opcode rS1, vLS, imm, vD **`;`** *mem-opcode mem-operand, vLS* **`;`** *modifier* **`;`** *...*
Scalar Instuctions: (*rS2* replaced with immediate value)
arith-opcode rS1, sLS, imm, sD **`;`** *mem-opcode mem-operand, sLS* **`;`** *modifier* **`;`** *...*

The *imm* operand is a 32-bit immediate value, either an *as-register* or a general expression. Immediate values are sign-extended in double integer arithmetic (zero-extended for double unsigned operations). For double-precision constants, only the upper 32 bits are included in the instruction. Thus, only floating-point numbers with 0s in the 32 least significant bits of their mantissas are allowed.

Older syntax required an immediate value expression to be preceded by a dollar sign (**`$`**). This syntax is still supported, but is discouraged in new code.

Restrictions: With the exception of the immediate operand, all register and memory operands have the same restrictions as in the short format. Vector length comes from **`dp_vector_length`**, and the permitted modifiers are the same.

Examples:

```
imovev   29,V1                      ! Monadic immed.
imovev   %i0,V1                     ! SPARC register
imovev   29,V1; iloadv [%i0],V2     ! with memory op.
imovev   29,V1; iloadv [%i0]:8,V0 ! with temp stride
imoves   29,S1                      ! Scalar operation

faddv    R0:0,29,V1                 ! Immed arithmetic
fmadtv   R0:0,V1,29,V3; dfloadv [%i0],V1
                              ! Triadic immediate
```

3.7 Register Stride (Long) Format

The *register stride format* modifies the short format by allowing arbitrary stride markers on the *rS2*, *rLS*, and *rD* register operands. (*rS1* format doesn't change.)

Vector Instuctions:

arith-op rS1, vLS[`:`*stride2*], *vS2*[`:`*stride3*], *vD*[`:`*stride4*] `;` `\`
mem-op mem-operand, vLS[`:`*stride2*]`;` `\`
modifier `;` ...

Scalar Instuctions:

arith-op rS1, sLS[`:`*stride2*], *sS2*[`:`*stride3*], *sD* [`:`*stride4*]`;` `\`
mem-op mem-operand, sLS[`:`*stride2*]`;` `\`
modifier `;` ...

The *stride* markers can be any of the register stride markers in Section 3.2.5, except those that apply to *rS1* only. If a triadic arithmetic operation is used, the *rLS* stride must be the same for both the arithmetic and memory operations.

The register operands do not have to be vector-aligned, and thus can be any of the 128 data registers.

The short format's operand, vector length, and modifier restrictions apply.

Examples:

```
imovev   V0,R4:4              ! Integer monadic
imovev   V0,R4:4; iloadv [%i0],V0   ! same, chain-loaded
imovev   V0,R4:4; iloadv [%i0]:8,V0  ! same, temp stride
iloadv   [%i0],R4:4;          ! memory operation
imovev   V0:mode,R4:4         ! Default reg. stride
imovev   S0:0,R4:4            ! Scalar reg, 0 stride
imoves   S0:0,S3:2            ! Scalar operation
dfgtv    V0,R12:10            ! Conditional
faddv    V0,R20:4,R6:3        ! Float dyadic
fmadtv   R0:0,V1:2,R20:4,R60:7; dfloadv [%i0],V1:2
                         ! True triadic, chain-loaded
```

3.8 Memory Stride (Long) Format

The *memory stride format* modifies the short format by allowing an arbitrary stride marker on the memory operand.

Vector Instuctions:

arith-op rS1, vLS, vS2, vD `;` *mem-op mem-operand*[`:`*stride*], *vLS* `;` *modifier* `;` ...

Scalar Instuctions:

arith-op rS1, sLS, sS2, sD `;` *mem-op mem-operand*[`:`*stride*], *sLS* `;` *modifier* `;` ...

The *stride* marker can be any of the memory stride markers in Section 3.2.5.

The short format's operand, vector length, and modifier restrictions apply.

Examples:

```
iloadv   [%i0]:=8,V1;         ! use and set 8
iloadv   [%i0]:8=4,V1;        ! use 8, set 4
iloads   [%i0]=4,S0;          ! set 4 (scalar op)
imovev   V0,V1; iloadv [%i0]:=8,V0     ! Chain-loaded
```

3.9 Mode Set (Long) Format

The *mode set format* is the most complex of the long formats. It allows you to do any or all of the following:

- Override and/or set the default vector length in **`dp_vector_length`**.
- Override the default conditionalization mode (**`vmmode`**).
- Override the default conditionalization sense (**`vminvert`**, **`vmtrue`**).
- Override the default vector mask copy mode (**`vmold`**, **`vmnew`**, **`vmnop`**).
- Use any of the modifiers permitted by the short format.

Mode set format also allows you to use one (and only one) of the following mutually incompatible extensions to the short format:

- Register stride markers on the *rS1* operand.
- Register indirection on the *rS1* operand.
- Memory indirection on the *memory-operand*.
- Exchange of data between the two VUs on a single chip ([**`no`**]**`exchange`**).
- Accumulated count of conditionalization bits (**`vmcount`**[**`s`**]).
- Population counts ([**`d`**]**`epc`**{**`v`**,**`s`**}).

The mode set "format" is actually a family of distinct but related variants, determined by the appearance of one of the incompatible features listed above. These variants are presented, with examples, in the sections below.

3.9.1 Mode Set Format Variants

The legal mode set variants are:

Vector Length Variant

arith-op[*vec-len*] *rS1, vLS, vS2, vD* `;` `\`
mem-op[*vec-len*] *mem-operand, vLS*`;` `\`
modifier`;` ...

(The syntax for the *vec-len* specifier is described in Section 3.9.2.)

This is the basic mode set variant, in which the only features used are those that are allowed in all mode set variants. In other words, this variant lets you specify an arbitrary vector length for a vector operation, and use general mode set modifiers like **vmnew**, **vminvert**, and **vmcurrent**.

Examples:

```
imovev*16     V0,V2           ! Integer monadic
imovev*16     V0,V2; iloadv [%i0],V0
                              ! same, chain-loaded
iloadv*16     [%i0],V1;       ! memory operation
iloadv*16     [%i0],V1; noalign
                              ! same, non-aligned
faddv*16 V0,V2,V4             ! Float dyadic
fmadtv*16     V0,V2,V4,V6; dfloadv [%i0],V2
                         ! True triadic, chain-loaded

imovev*=16    V0,V2           ! Use and set len.
imovev*%i2    V0,V2           ! SPARC register
imovev*=%i2   V0,V2           ! Use and set
imovev*%i2<   V0,V2           ! 4 bit length
imovev*=%i2<  V0,V2           ! 4 bit use/set
imoves=16     S0,S8           ! Scalar set


faddv    V0,V1,V2; vmcurrent;     ! Current mode
faddv    V0,V1,V2; vmnew;         ! New mask copy
faddv    V0,V1,V2; vmnop;         ! No mask copy
faddv    V0,V1,V2; iloadv [%i0],V0; vminvert;
                                  ! Inverted conditional
```

rS1 Stride Variant

arith-op[*vec-len*] *rS1*[`:`*stride*], *vLS, vS2, vD* `;` `\`
mem-op[*vec-len*] *mem-operand, vLS*`;` `\`
modifier`;` ...

This variant lets you specify an arbitrary stride marker for the *rS1* operand. This stride marker can be any of the register stride markers in Section 3.2.5.

Examples:

```
imovev*16      V0:2,V2         ! Use stride 2
imovev         V0:1=4,V2       ! Use 1, set 4
imoves         R0=4,R6         ! Set 4 (scalar)
faddv*16       V0:2,V2,V4      ! Float dyadic
fmadtv         V0:1=0,V2,V4,V6; \
               dfloadv [%i0],V2    ! Triadic
```

Register Stride Indirection Variant

arith-op[*vec-len*] *rS1*[`(`*rIA*`:`*stride*`)`], *vLS, vS2, vD* `;` `\`
mem-op[*vec-len*] *mem-operand, vLS*`;` `\`
modifier `;` ...

This variant allows the use of an arbitrary VU register to specify the *rS1* stride. Register indirection format is described in Section 3.9.3.

Examples:

```
imovev         V0(V2),V4 ! Reg. indirection
imovev*16      V0(V2),V4; iloadv [%i0],V0 \
                                   ! same, chain-loaded
imovev         V0(V2:2),V4 ! Indirect. with stride
```

Memory Stride Indirection Variant

arith-op[*vec-len*] *rS1, vLS, vS2, vD* **;** ****
mem-op[*vec-len*] *mem-operand*[**(***rIA***:***stride***)**], *vLS***;** ****
modifier **;** ...

This variant allows the use of a VU register to specify the *mem-operand* stride. Memory indirection format is described in Section 3.9.4.

Examples:

```
iloadv*16      [%i0](V2),V0;        ! Mem. Indirect
iloadv*16      [%i0](V2:4),V0;      ! with stride
imovev*16     V0,V4; iloadv [%i0](V2),V0
                                    ! Chain-loading
```

Population Count Variant

arith-op[*vec-len*] *rS1, vLS, vS2, vD* **;** ****
[**d**]**epc**{**v**,**s**} **(***vLS*[**:unit**]**)=***rIA*[**:***stride*] **;** ****
*other-modifier***;** ...

This variant allows you to specify the [**d**]**epc**{**v**,**s**} modifier, which cannot be combined with a memory operation, or with any other mode set variant. (See Section 4.3.3.)

Examples:

```
epcv     (V0)=V1             ! Unit stride
epcv     (V0)=V1:2           ! Explicit stride
depcv    (V0)=V1:2           ! Double op
faddv*16 V0,V1,V2; epcv (V0)=V1;       ! Chain-loading
dfaddv*16 V0,V1,V2; depcv (V0)=V4;     ! Double op
```

Special Modifier Variant

arith-op[*vec-len*] *rS1, vLS, vS2, vD* **;** ****
mem-op[*vec-len*] *mem-operand, vLS*; ****
{[**no**]**exchange**, **vmcount**[**s**]**=***reg*[**:***stride*]} **;**
other-modifier; ...

This variant allows you to specify one (and only one) of the [**no**]**exchange** or **vmcount**[**s**] modifiers, which cannot be combined with any other mode set variant. (See Section 4.3.3.)

Examples:

```
faddv    V0,V1,V2; exchange;       ! exchange values
faddv    V0,V1,V2; \
    floadv [%i0],V0; exchange;     ! chain-load

vmcount=V0;   ! Context count
vmcount=V0:2; ! with stride

faddv    V0,V1,V2; vmcount=V0;     ! chain-loaded
faddv    V0,V1,V2; \
    floadv [%i0],V0; vmcount=V0;   ! chain-loaded
faddv    V0,V1,V2; vmcount=V0:2;   ! strided
```

Scalar Instruction Variant

arith-op[*vec-len*] *rS1*[**:***stride*]*, sLS, sS2, sD* **;** ****
mem-op[*vec-len*] *mem-operand, vLS*; ****
modifier **;** ...

This variant lets you use a scalar DPEAC operation to set the default vector length for future instructions (and specify an arbitrary *rS1* stride marker). This mode set variant is much more efficient than using the **dpset** accessor instruction to modify the **dp_vector_length** register.

Examples:

```
fadds    V0,V1,V2; exchange;       ! exchange values
fadds    V0,V1,V2; \
    floads [%i0],V0; exchange;     ! chain-load
```

3.9.2 Vector Length Modifier

In all mode set format variants, the *vec-len* modifier specifies the vector length for the operation, and can also be used to modify the default vector length stored in the register **`dp_vector_length`**. The syntax of the *vec-len* modifier is:

Syntax	Effect
opcode`*`*vlen*	Use constant length *vlen*.
opcode`*=`*vlen*	Use/set **`dp_vector_length`** to *vlen*.
opcode`*`*as-register*	Use length from *as-register* (all bits, + 1).
opcode`*=`*as-register*	Use/set **`dp_vector_length`** from *as-register.*
opcode`*`*as-register*`<`	Use length from *as-register* (bits **`19:22`**, + 1).
opcode`*=`*as-register*`<`	Use/set **`dp_vector_length`** from *as-register.*
opcode`=`*vlen*	Set **`dp_vector_length`** to *vlen* (scalar ops.)

where *vlen* is a *constant-expression*. The length specified must always be an integer from **`1`** to **`16`**. Any unused bits of a referenced *as-register* must be 0. The modifier "**`<`**" makes *as-register* references faster (fewer SPARC operations) because only 4 bits (**`19`** through **`22`**) of the register are used.

The *vec-len* modifier can be attached to either the arithmetic opcode or memory opcode, or both, and it applies to both. (If a *vec-len* modifier is specified on both the arithmetic and memory opcodes, the two modifiers must be identical.)

Note: All forms that obtain a length value from a register implicitly add 1 to the value before use. All forms that store a value into **`dp_vector_length`** store the value in decremented form, so that this implicit incrementing will work properly.

3.9.3 Register Stride Indirection

For register stride indirection, the *rS1* operand format is:

Syntax	Effect
rS1`(`*rIA*`)`	Indirect addressing, unit stride.
rS1`(`*rIA*`:`*stride*`)`	Indirect addressing, constant stride.

The *rIA* register operand contains offsets that are separately added to the *rS1* base register to obtain the actual **`R`***nn* register containing the *rS1* stride. (**Note:** This offset addition is *not* cumulative.)

The register offsets are packed *four to a register* in the specified *rIA* register and in subsequent registers at the specified *stride*. Since offsets cannot exceed 127 (7 bits), the eighth bit of each offset byte must be zero:

0	*offset 1*	**0**	*offset 2*	**0**	*offset 3*	**0**	*offset 4*
31	30 24	23	22 16	15	14 8	7	6 0

Note: If a *stride* is not specified, then the "unit" stride is always 1 register for both single- and doubleword operations; one doubleword "register" corresponds to two singleword registers.

3.9.4 Memory Indirection

For memory stride indirection, the *mem-operand* format is:

Syntax	**Effect**
mem-operand **(***register***)**	Memory indirection, unit stride.
mem-operand **(***register* **:** *stride***)**	Memory indirection, constant stride.

The indirection modifier replaces the [**:***tempstride*] modifier of the short format.

The specified single-precision VU *register* contains offsets that are separately added to the memory address to obtain each operand location. The addition is done in two's-complement, so negative offsets will work correctly. (**Note:** This offset addition is *not* cumulative.) The memory offsets are stored *one byte per register*, taken from the specified single-precision *register* and subsequent registers at the specified *stride*.

Note: If a *stride* is not specified, then the "unit" stride is 1 single-precision register for single-precision memory operations, and 2 single-precision registers (1 double-precision register) for double-precision memory operations.

3.9.5 Mode Set Format Modifiers

The following modifiers are permitted by the mode set format:

These modifiers are permitted by the short format:

[**no**]**pad**[**:***pad-size*]	Vector length padding (default is 4).
maddr=*mem-operand*	Memory operand specifier.
[**no**]**align**	Doubleword alignment guarantee.
{**vmrotate**, **vmcurrent**}	Status bit rotation mode.
vmmode:[**=**]*mode-keyword*	Conditionalization mode selection.

These are the mutually-compatible modifiers added by the mode set format:

{**vminvert**, **vmtrue**}**;**	Conditionalization bit sense selection.
{**vmold**, **vmnew**, **vmnop**}**;**	Vector mask copy mode.

These are only allowed in the pop. count and special modifier variants:

[**d**]**epc**{**v**,**s**} **(***vLS*[**:unit**]**)=***rIA***:***stride*	Population count.
vmcount[**s**]**=***reg***:***stride*	Accumulated context count.
[**no**]**exchange**	VU on-chip data swapping.

These modifiers are all described in more detail in Section 4.3.

Chapter 4

DPEAC Instruction Set Reference

This chapter presents a quick-reference list of the DPEAC instruction set, including DPEAC instructions, instruction modifiers, and accessor instructions.

4.1 DPEAC Arithmetic Instructions

4.1.1 Monadic (One-Source) Arithmetic Instructions

These operators perform an arithmetic operation on *rS1*, storing the result in *rD*. (**Note:** In immediate format, the *rS1* source argument is the immediate value.)

Formats:

opcode *rS1*, *rD*

Monadic Opcodes	Function
{**i,di,u,du,f,df**}**move**{**v,s**}	Move *rS1* to *rD*, no status generated.
{**i,di,u,du,f,df**}**test**{**v,s**}	Move *rS1* to *rD* and test.
{**u,du**}**not**{**v,s**}	Bitwise invert (*rD* = ~*rS1*).
{**f,df**}**clas**{**v,s**}	Classify operand (*rD* = class of *rS1*).
{**f,df**}**exp**{**v,s**}	Extract exponent from float.
{**f,df**}**mant**{**v,s**}	Extract mantissa with hidden bit.
{**u,du**}**ffb**{**v,s**}	Find first “1” bit.
{**i,di,f,df**}**neg**{**v,s**}	Negate (*rD* = 0 – *rS1*).
{**i,di,f,df**}**abs**{**v,s**}	Absolute value (*rD* = |*rS1*|).
{**f,df**}**inv**{**v,s**}	Invert (*rD* = 1/*rS1*).
{**f,df**}**sqrt**{**v,s**}	Square root (*rD* = SQRT(*rS1*)).
{**f,df**}**isqt**{**v,s**}	Inverse root (*rD* = 1/SQRT(*rS1*)).

The **to** operator converts between data types: *rS1* is of the first type in the opcode, and *rD* is of the second type. (In immediate format, *rS1* is an immediate value.)

Formats:

opcode *rS1*, *rD*

Monadic Opcodes	Function
{**i**,**di**,**u**,**du**}**to**{**f**,**df**}{**v**,**s**}	Convert integer to float.
{**f**,**df**}**to**{**f**,**df**}{**v**,**s**}	Convert to another precision.
{**f**,**df**}**to**{**i**,**di**,**u**,**du**}**r**{**v**,**s**}	Convert to integer (round).
{**f**,**df**}**to**{**i**,**di**,**u**,**du**}{**v**,**s**}	Convert to integer (truncate).

4.1.2 Dyadic (Two-Source) Instructions

These operators perform an arithmetic operation on the *rS1* and *rS2* arguments, and store the result in the *rD* argument. (In immediate format, *rS2* is an immediate value.)

Formats:

opcode *rS1*, *rS2*, *rD*

Dyadic Opcodes	Function
{**i**,**di**,**u**,**du**,**f**,**df**}**add**{**v**,**s**}	Add (*rD* = *rS1* + *rS2*).
{**i**,**di**,**u**,**du**}**addc**{**v**,**s**}	Integer add with carry bit from shift of vector mask register.
{**i**,**di**,**u**,**du**,**f**,**df**}**sub**{**v**,**s**}	Subtract (*rD* = *rS1* – *rS2*).
{**i**,**di**,**u**,**du**}**subc**{**v**,**s**}	Integer subtract with carry bit from shift of vector mask register.
{**i**,**di**,**u**,**du**,**f**,**df**}**subr**{**v**,**s**}	Subtract reversed (*rD* = *rS2* – *rS1*).
{**i**,**di**,**u**,**du**}**sbrc**{**v**,**s**}	Integer subtract reversed with carry bit from shift of vector mask register.
{**i**,**di**,**u**,**du**,**f**,**df**}**mul**{**v**,**s**}	Multiplication (low 32/64 bits for ints).
{**di**,**du**}**mulh**{**v**,**s**}	Integer multiply (high 64 bits).
{**f**,**df**}**div**{**v**,**s**}	Divide (*rD* = *rS1* / *rS2*).
{**u**,**du**}**enc**{**v**,**s**}	Make float from exp and mant (*rS1*, *rS2*).

Dyadic (Two-Source) Instructions *(continued)*

Formats:

opcode *rS1*, *rS2*, *rD*

Dyadic Opcodes	Function
{**u**,**du**} **shl**{**v**,**s**}	Shift left (*rD* = *rS1* << *rS2*).
{**u**,**du**} **shlr**{**v**,**s**}	Shift left, reversed (*rD* = *rS2* << *rS1*).
{**i**,**di**,**u**,**du**} **shr**{**v**,**s**}	Shift right (*rD* = *rS1* >> *rS2*).
{**i**,**di**,**u**,**du**} **shrr**{**v**,**s**}	Shift right, reversed (*rD* = *rS2* >> *rS1*).
{**u**,**du**} **and**{**v**,**s**}	Bitwise logical **AND**.
{**u**,**du**} **nand**{**v**,**s**}	Bitwise logical **NAND**.
{**u**,**du**} **andc**{**v**,**s**}	Bitwise logical **NOT**(*rS1*) **AND** *rS2*.
{**u**,**du**} **or**{**v**,**s**}	Bitwise logical **OR**.
{**u**,**du**} **nor**{**v**,**s**}	Bitwise logical **NOR**.
{**u**,**du**} **xor**{**v**,**s**}	Bitwise logical **XOR**.
{**i**,**di**,**u**,**du**,**f**,**df**} **mrg**{**v**,**s**}	If vector mask bit = 1 then *rS1* else *rS2*.

4.1.3 Arithmetic Comparisons

These operators perform an arithmetic comparison between the *rS1* and *rD* arguments, and set status flags accordingly. (In immediate format, *rD* is an immediate value.)

Formats:

opcode *rS1*, *rD*

Opcodes	Function
{**i**,**di**,**u**,**du**,**f**,**df**} **gt**{**v**,**s**}	Greater than.
{**i**,**di**,**u**,**du**,**f**,**df**} **ge**{**v**,**s**}	Greater than or equal.
{**i**,**di**,**u**,**du**,**f**,**df**} **lt**{**v**,**s**}	Less than.
{**i**,**di**,**u**,**du**,**f**,**df**} **le**{**v**,**s**}	Less than or equal.
{**i**,**di**,**u**,**du**,**f**,**df**} **eq**{**v**,**s**}	Equal.
{**i**,**di**,**u**,**du**,**f**,**df**} **ne**{**v**,**s**}	Not equal or unordered.
{**i**,**di**,**u**,**du**,**f**,**df**} **lg**{**v**,**s**}	Ordered and not equal.
{**i**,**di**,**u**,**du**,**f**,**df**} **un**{**v**,**s**}	Unordered.

4.1.4 Compare (Dyadic with rD constant)

The compare operation tests for a numeric relationship between the *rS1* and *rS2* arguments, as indicated by the supplied constant *code*. (In immediate format, *rS1* is an immediate value.)

Format:

{**i**,**di**,**u**,**du**,**f**,**df**}**cmp**{**v**,**s**} *rS1*, *rS2*, *code*

Code	Purpose
0	Test for greater than.
1	Test for equal.
2	Test for less than.
3	Test for greater than or equal.
4	Test for unordered (NaN present).
5	Test for ordered and not equal.
6	Test for not equal or unordered.
7	Test for less than or equal.

4.1.5 Dyadic Mult-Op Operators

These operations perform a muliplication and an arithmetic (or logical) operation on the *rS1*, *rS2*, and *rD* arguments, and store the result in *rD*. **Note:** The optional [**h**] suffix indicates that the high 64 bits of the multiplication are to be used in the logical operation, rather than the low 64 bits (the default). (In immediate format, *rS2* is an immediate value.)

Format:

opcode *rS1*, *rS2*, *rD*

Accumulative Opcodes	Function
{**i**,**di**,**u**,**du**,**f**,**df**}**mada**{**v**,**s**}	*rD* = (*rS1* * *rS2*) + *rD*
{**i**,**di**,**u**,**du**,**f**,**df**}**msba**{**v**,**s**}	*rD* = (*rS1* * *rS2*) – *rD*
{**i**,**di**,**u**,**du**,**f**,**df**}**msra**{**v**,**s**}	*rD* = *rD* – (*rS1* * *rS2*)
{**i**,**di**,**u**,**du**,**f**,**df**}**nmaa**{**v**,**s**}	*rD* = –*rD* – (*rS1* * *rS2*)
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*

Dyadic Mult-Op Operators *(continued)*

Format:

opcode *rS1*,*rS2*,*rD*

Inverted Opcodes	Function
{**i**,**di**,**u**,**du**,**f**,**df**}**madi**{**v**,**s**}	*rD* = (*rS2* * *rD*) + *rS1*
{**i**,**di**,**u**,**du**,**f**,**df**}**msbi**{**v**,**s**}	*rD* = (*rS2* * *rD*) – *rS1*
{**i**,**di**,**u**,**du**,**f**,**df**}**msri**{**v**,**s**}	*rD* = *rS1* – (*rS2* * *rD*)
{**i**,**di**,**u**,**du**,**f**,**df**}**nmai**{**v**,**s**}	*rD* = –*rS1* – (*rS2* * *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*

4.1.6 Convert Operation (Dyadic with rS2 constant)

These operations convert the *rS1* argument to the type indicated by the constant *code* argument, and store the result in the *rD* argument. The symbolic code constants listed below are defined by the **dp.h** header file. (In immediate format, *rS1* is an immediate value.)

Format:

cvt{**f**,**fi**,**i**,**ir**}{**v**,**s**} *rS1*,*code*,*rD*

Opcode/Type		Code	Purpose
cvt	**i**[**r**]	**CVTICD_F_I** (4)	Single float to single signed integer.
cvt	**i**[**r**]	**CVTICD_F_U** (5)	Same, to unsigned integer.
cvt	**i**[**r**]	**CVTICD_F_DI** (6)	Single float to double signed integer.
cvt	**i**[**r**]	**CVTICD_F_DU** (7)	Same, to unsigned integer.
cvt	**i**[**r**]	**CVTICD_DF_I** (12)	Double float to single signed integer.
cvt	**i**[**r**]	**CVTICD_DF_U** (13)	Same, to unsigned integer.
cvt	**i**[**r**]	**CVTICD_DF_DI** (14)	Double float to double signed integer.
cvt	**i**[**r**]	**CVTICD_DF_DU** (14)	Same, to unsigned integer.
cvt	**f**	**CVTFCD_F_DF** (3)	Single float to double float.
cvt	**f**	**CVTFCD_DF_F** (9)	Double float to single float.

Convert Operation, *Continued*

Format:

cvt{**f**,**fi**,**i**,**ir**}{**v**,**s**} *rS1*, *code*, *rD*

Opcode/Type		Code	Purpose
cvt	**fi**	**CVTFICD_I_F** (1)	Single signed integer to single float.
cvt	**fi**	**CVTFICD_U_F** (5)	Same, but from unsigned integer.
cvt	**fi**	**CVTFICD_I_DF** (3)	Single signed integer to double float.
cvt	**fi**	**CVTFICD_U_DF** (7)	Same, but from unsigned integer.
cvt	**fi**	**CVTFICD_DI_F** (9)	Double signed integer to single float.
cvt	**fi**	**CVTFICD_DU_F** (13)	Same, but from unsigned integer.
cvt	**fi**	**CVTFICD_DI_DF** (11)	Double signed integer to double float.
cvt	**fi**	**CVTFICD_DU_DF** (15)	Same, but from unsigned integer.

4.1.7 True Triadic (Three-Source) Operators

These operations perform a muliplication and an arithmetic (or logical) operation on the *rS1*, *rS2*, and *rLS* arguments, and store the result in *rD*. (In immediate format, *rS2* is an immediate value.)

Format:

opcode *rS1*, *rLS*, *rS2*, *rD*

True Triadic Opcodes	Function
{**i**,**di**,**u**,**du**,**f**,**df**}**madt**{**v**,**s**}	*rD* = (*rS1* * *rLS*) + *rS2*
{**i**,**di**,**u**,**du**,**f**,**df**}**msbt**{**v**,**s**}	*rD* = (*rS1* * *rLS*) – *rS2*
{**i**,**di**,**u**,**du**,**f**,**df**}**msrt**{**v**,**s**}	*rD* = *rS2* – (*rS1* * *rLS*)
{**i**,**di**,**u**,**du**,**f**,**df**}**nmat**{**v**,**s**}	*rD* = –*rS2* – (*rS1* * *rLS*)
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*

Note: In the opcode descriptions above, the optional [**h**] indicates that the high 64 bits of the multiplication are to be used in the logical operation, rather than the low 64 bits (the default).

Triadic/Memory Register Restriction Note: When a triadic arithmetic operation and a memory operation are **join**ed, the *rLS* operand of the arithmetic operation must be identical to the *rLS* operand of the memory operation.

4.1.8 No-Op Operator

The untyped arithmetic no-op allows modifier side effects without specifying an operation. The no-op takes no arguments.

Formats:

fnop{**v**,**s**}	(No arithmetic operation.)

4.2 DPEAC Memory Instructions

The following opcodes are supported for memory operations.

Format:

opcode *memory-address*`,`*rD*

Opcodes	Function
{**i**,**di**,**u**,**du**,**f**,**df**}**load**{**v**,**s**}	Load *Vls* from memory.
{**i**,**di**,**u**,**du**,**f**,**df**}**store**{**v**,**s**}	Store *Vls* to memory.

4.2.1 No-Op Operator

The following opcodes are supported for memory operations.

Format:

memnop *memory-address*`,`*rD*

Opcodes	Function
memnop	No memory operation.

The **memnop** operation allows the side effects of memory syntax (setting of stride defaults by stride markers, etc.) to happen without an actual memory operation.

4.3 DPEAC Instruction Modifiers

This section describes the statement modifiers that can be combined with arithmetic and memory operations to affect their assembly and/or execution. **Note:** Some of these modifiers (such as the last three) can be used on their own.

4.3.1 Modifiers That Can Be Used in All (or Most) Formats

[**`no`**]**`pad`**[**`:`***pad-size*] Default: **`pad:4`**

Vector Length Padding: Pads vector length of instruction to at least *pad-size*. Has no effect if vector length is already that size. Used to avoid instruction pipeline hazards. If not supplied, defaults to **`pad:4`**. The nopad variant is the same as **`pad:0`**. Pads between **`0`** and **`4`** are allowed, but have the same effect as **`pad:4`**.

`maddr=`*memory-operand* Default: None

Memory Operand Specifier: Used to supply a default memory operand for DPEAC statements that omit the memory instruction — this memory operand is used solely to determine VU selection.

{**`vmrotate`**, **`vmcurrent`**} Default: **`vmrotate`**

Status Bit Rotation Mode: Determines how status bits from vector operations are stored in the register **`dp_vector_mask`**. **`vmrotate`** "rotates" them in, **`vmcurrent`** inserts them in bit order. (See Figure 14.) Note: this modifier is allowed by the short format for conditional operations only. Otherwise, it can only be used in the mode set format.

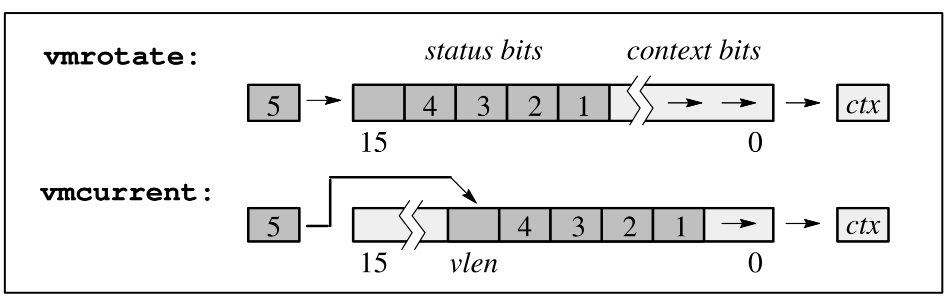

Figure 14. Bit-shifting modes of vector mask register.

[**no**]**align** Default: **noalign**

Doubleword Alignment Guarantee: Declares whether or not the memory operand is doubleword-aligned (even for singleword operations). If alignment is guaranteed, **dpas** can generate more efficient code. (**Note:** The default setting of this modifier can be reversed by providing the **-a** command line switch to **dpas**.)

4.3.2 Conditionalization Modifiers

These modifiers are used to control the conditionalization mechanism. For more information, see Section 2.3.1.

vmmode:[**=**]*mode-keyword* Default: **vmmode:vmmode**

Conditionalization Mode: The **vmmode** modifier overrides the value of the **dp_vector_mask_mode** register, which affects whether arithmetic operations and/or memory operations are to be conditionalized. The permitted *mode-keyword* operands are:

Mode	Effect
vmmode:vmmode	Use current value of **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 legal to override **dp_vector_mask_mode** for full conditionalization. Thus, "**vmmode:cond**" is not allowed.

Usage Note: Scalar instructions are executed without conditionalization, so you may add **vmmode:always** to any scalar instruction in any format with no effect. Similarly, you may add **vmmode:vmmode** to any vector instuction in any format since it represents the default action taken by the hardware.

{**vminvert**, **vmtrue**} Default: **vmtrue**

Conditionalization Bit Sense: The **vminvert** and **vmtrue** modifiers control whether the conditionalization bits shifted out of the **dp_vector_mask** are inverted. If inverted, the sense of these bits is reversed; i.e., 0 selects a vector element, and 1 deselects it.

Modifier	Effect
vminvert	Invert sense of vector mask bits for conditionalization.
vmtrue	Do not invert sense of vector mask bits.

Note: This modifier is only allowed in the mode set statement format.

{**vmold**, **vmnew**, **vmnop**} Default: **vmold**

Vector Mask Copy Mode: The **vmold**, **vmnew**, and **vmnop** modifiers control the copying of the vector mask and vector mask buffer registers prior to instruction execution:

Modifier	Effect
vmold	Copy **dp_vector_mask_buffer** to **dp_vector_mask**.
vmnew	Copy **dp_vector_mask** to **dp_vector_mask_buffer**.
vmnop	No copy.

Note: This modifier is only allowed in the mode set statement format.

4.3.3 Special Modifiers (Mode Set Format Only)

[**d**]**epc**{**v**,**s**} **(***vLS*[**:unit**]**)=***rIA*[**:***stride*] Default: None

Population Count: The [**d**]**epc**{**v**,**s**} modifier enables the population count feature. Specifically, the single- or double-precision register *vLS* (and subsequent registers at a unit stride) are read and the "1" bits in each are counted. The results, each a single-precision unsigned integer between 0 and either 32 (single-precision) or 64 (double-precision), are written to the register *Ria* (and subsequent single-precision registers at the specified *stride*, a *constant-expression* that defaults to the unit stride for the data type).

The [**d**]**epc**{**v**,**s**} modifier effectively replaces the normal memory operation in a DPEAC statement. The *Vls* register operand is used, so population counting cannot be combined with any memory operation. Population counting also cannot be used in conjunction with register or memory indirection or the

vmcount[**s**] or [**no**]**exchange** modifiers. The population count result is written before the operands are read for the arithmetic operation, so the [**d**]**epc**{**v**,**s**} modifier chain loads. The *vLS* operand is always strided with a unit (1 or 2 register) stride, so the **:unit** keyword is optional and has no effect other than to emphasize the unit striding.

Implementation Note: Currently, the [**d**]**epc**{**v**,**s**} modifier 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**.

vmcount[**s**]**=***reg* **:** *stride* Default: None

Accumulated Context Count: The **vmcount** modifier enables the VU chip's *accumulated context count* feature. The single-precision VU register *reg* (and subsequent registers at the given *stride*, a *constant-expression*) is loaded with the accumulated count of "1" bits in the vector mask at each step in the vector operation. This accumulation is inclusive; the count includes the bit that is shifted out of the vector mask register for each element. The scalar version, **vmcounts**, is intended for use with scalar operations. It is an error to use **vmcounts** with any vector operation.

For each element in the vector, the **vmcount** result is written before the operands are read for the arithmetic operation, so this modifier chain loads. This modifier cannot be used in conjunction with either register or memory indirection, nor with the [**d**]**epc**{**v**,**s**}, or [**no**]**exchange** modifiers.

[**no**]**exchange** Default: **noexchange**

VU On-Chip Data Swapping: Controls exchange of data between two VUs on the same chip. Specifying **exchange** causes arithmetic results on each VU to be written to the destination register(s) of the other VU. In conditionalized ALU operations, deselected elements are not written to the opposite VU. Selected elements *are* written, even if the corresponding element in the opposite VU is deselected.

The [**no**]**exchange** modifier is only used in the mode set format. However, it is incompatible with register stride indirection, memory stride indirection, and with the [**d**]**epc**{**v**,**s**}, and **vmcount**[**s**] modifiers.

Implementation Note: This modifier is implementation-dependent, and may not be available in the future. Also, the current implementation of exchanging does not allow chain loading into the arithmetic destination register.

4.4 DPEAC Accessor Instructions

These accessor instructions are always used as single statements, execute on the node microprocessor (the SPARC), and generally move data between the SPARC and the VU, or affect values stored in SPARC registers.

4.4.1 VU Register Accessor Instructions

Instruction(s)	Function(s)
dpwrt[**d**], **dprd**[**d**]	Write and read VU data registers.
dpset[**d**], **dpget**[**d**]	Write and read VU control registers.
dpchgbk	Convert address from one VU region to another.
dpchgsp	Convert between VU data and instruction spaces.
dpld[**d**], **dpst**[**d**]	Read and write VU parallel memory.
dpsync	Synchronize instruction pipelines of VUs.

These instructions move data between VU data registers and SPARC registers:

dpwrt[**d**] *VU-selector*, *as-src-reg*, *VU-dest-reg* [, {**sync**, **nosync**}]
dpwrt[**d**] *VU-selector*, *value*, *VU-dest-reg* [, {**sync**, **nosync**}]
dprd[**d**] *VU-selector*, *VU-src-reg*, *as-dest-reg* [, {**sync**, **nosync**}]

```
dpwrt   ALL_DPS,%i1,V0,sync
dpwrt   DPS_0_AND_1,29,V0,nosync
dprd    DP_3,V0,%i0
```

These instructions move data between VU control registers and SPARC registers. (See Section 2.5 for a list of predefined control register constants.)

dpset[**d**] *VU-selector*, *as-src-reg*, *ctl-reg-offset* [, **SUPERVISOR**]
dpset[**d**] *VU-selector*, *value*, *ctl-reg-offset* [, **SUPERVISOR**]
dpget[**d**][**s**] *VU-selector*, *ctl-reg-offset*, *as-dest-reg* [, **SUPERVISOR**]

```
dpset   DP_3,%i0,DP_VECTOR_MASK
dpset   ALL_DPS,0,DP_VECTOR_MASK
dpget   DPS_0_AND_1,DP_VECTOR_MASK,%i0
```

This instruction converts a VU memory address between the data and instruction virtual memory spaces:

dpchgsp *srcreg*, *destreg*

```
dpchgsp      R5, R6
```

This instruction modifies a VU memory address to refer to a different VU memory region:

dpchgbk *srcreg*, *VU-selector*, *destreg*

```
dpchgbk      R5, DPS_0, R6
```

These instructions move data between VU parallel memory and a SPARC IU register:

dpld[**d**] *as-mem-operand*, *as-dest-register*
dpst[**d**] *as-src-register*, *as-mem-operand*

```
dpld    [%i0], %i1
dpst    %i1, [%i0]
```

This instruction generates code to prevent the preceding and following instructions from overlapping in the instruction pipeline of the VUs (see Appendix C):

dpsync

```
faddv V0,V1,V2
dpsync
fmulv V1,V2,V3
```

4.4.2 VU Trap Instructions

These instructions generate traps and provide direct SPARC access to the **dp_vector_mask** register:

Opcode	Function
trap	Generate trap unconditionally.
etrap	Generate trap if **set_enb** bit in the **dp_interrupt_cause_green** register is set.

4.4.3 Vector Mask Instructions

Opcode	Function
ldvm *rS1*	Move *rS1* to **dp_vector_mask**.
stvm *rS1*	Move **dp_vector_mask** to *rS1*.

```
ldvm V1
stvm V1
```

4.4.4 SPARC Accessor Instructions

These instructions assemble into SPARC-only code, and do not affect the VUs:

Instruction	Function
dpentry	Creates a callable DPEAC routine.
dpretn	Returns from DPEAC routine.
load[**d**]	Loads an IU register with a constant value.
dpunset	Signals that one or both reserved registers may have been overwritten.
dpregs	Overrides SPARC default register usage.

dpentry *name*, *argwords*, *localbytes*

```
_ROUTINENAME, 0, 0
```

The **dpentry** instruction creates a callable DPEAC routine. *name* is an *as-symbol*, the name of the routine. (Don't forget the leading underscore when naming routines to be called from C.)

argwords is the number of stack words reserved for arguments (in excess of 6) to subroutines. (Doubleword arguments count as two words.) If there are no subroutine calls (or none with more than 6 arguments), *argwords* is 0.

localbytes is the number of bytes beyond the standard frame size (**MINFRAME**, i.e., 92) to be allocated on the stack frame for local temporaries. (These are located at the top of the frame and referenced by negative offsets from **%fp**.)

The **dpentry** instruction implies a "**dpregs =,=,** " — that is, the Default Maddr Base and Instruction Extension Pointer registers are initialized.

dpretn

```
dpretn
```

This instruction generates a return from the DPEAC routine. It takes no arguments, and generates the following code:

```
ret
restore
```

To return values from a routine, place them in **%i0** and **%i1**, as per the C convention. Floats are returned as a double in the floating-point register **%f0**.

load[**d**] *general-expression*, *as-register*

```
load 1066, %i0
```

This instruction loads a SPARC IU register with a constant value, automatically generating the SPARC instructions needed to load the value. **loadd** loads an aligned pair of registers with a doubleword value.

dpunset

```
dpunset
```

The **dpunset** "instruction" informs the **dpas** assembler that one or both of the reserved registers (**%l6** and **%l7**) may have been overwritten. If succeeding code requires the original values of these registers, **dpas** inserts instructions to reinitialize them.

dpregs

```
dpregs    %l6,%l7,%g2
dpregs    %l6=,-,%g2+
```

The **dpregs** "instruction" modifies the default SPARC registers used by **dpas** for code construction. The syntax is:

dpregs *MaddrReg*, *InstExtPtrReg*, *TempRegs*

Argument	Register Usage
MaddrReg	Default Maddr Base Register, used for executing DPEAC statements lacking a memory address. Default value is **DPV_STACK_INSTR_PORT_ALL**.
InstExtPtrReg	Instruction Extension Pointer Register, used in non-doubleword-aligned memory references; contains 0xC0000176, a value used to compute a pointer to the **dp_instruction_ext** register.
TempRegs	An even-numbered SPARC register; declares that both the specified register and its immediate successor can be used as temporaries to execute VU instructions.

Each of these arguments can have any one of the following forms:

Argument	Meaning
as-register	Use this register and mark it as uninitialized.
as-register=	Use this register and generate code to initialize it.
=	Generate code to initialize the current register.
–	Tell **dpas** to not use any register for this feature.
{*as-register*}+	Tell **dpas** to use the register, but not alter its value.
(blank)	Do not change the current setting (NOP).

The default setting at the beginning of **dpas** assembly is:

```
dpregs %l6,%l7,%g2  ! declare regs, but don't initialize
```

If "–" syntax is used to turn off Maddr Register usage, subsequent VU instructions that don't specify a memory address will signal an error. If the Instruction Extension Pointer Register is turned off, **dpas** will generate code two cycles slower wherever a long format instruction performs a memory operation on a possibly non-doubleword-aligned memory address (one for which neither the the **align** modifier nor the **-a** switch to **dpas** were given).

The "–" (disable) and "=" (initialize) markers can only be applied to the *MaddrReg* and *InstExtPtrReg* operands.

Declaring a register but not immediately setting it up (i.e., specifying a register name, but not using the "=" syntax), causes **dpas** to mark that register as uninitialized. This causes initialization code to be inserted later in assembly when the value of the register is needed again. This can be used when a register may have been overwritten to declare that **dpas** should not assume its contents are valid.