Chapter 5

The CDPEAC Instruction Set

The CDPEAC instruction set is a set of preprocessor macros implemented in the C programming language. These macros are based on the C **`asm`** statement, which allows a programmer to directly insert a line of assembly code as a statement in a C procedure. (See Appendix G for more information.)

When a C program containing CDPEAC statements is compiled, each CDPEAC statement is translated into an **`asm`** statement that inserts a line of DPEAC code. This DPEAC code is further compiled into SPARC assembly code, which sends appropriately assembled instruction word(s) to the VU hardware.

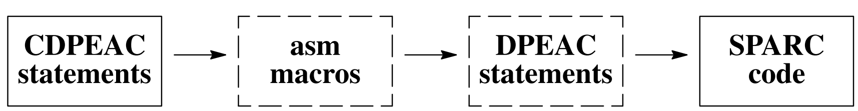

Figure 15. Process of translation used for CDPEAC code.

The most common use of CDPEAC is for efficient arithmetic functions: a main program written in a high-level CM language (such as C* or CM Fortran) defines parallel CM arrays using its own operators, and then calls a CDPEAC subroutine to perform a specific arithmetic operation on the contents of the arrays.

Note: CDPEAC is C interface for DPEAC programmers. There are a few lesser-used features of DPEAC syntax that have no analogues in CDPEAC syntax — these are noted in the appropriate sections of this chapter.

Also, because CDPEAC statements expand into DPEAC code with little internal translation, some familiarity with DPEAC is very helpful for effective CDPEAC programming. In particular, the syntax of CDPEAC arguments is virtually the same as that for DPEAC (see Section 5.2).

5.1 CDPEAC Code

A CDPEAC procedure is just a C procedure that includes CDPEAC statements. A *CDPEAC statement* is one of the following:

- a *VU instruction*
- a *VU accessor instruction*
- a *special instruction*

5.1.1 VU Instructions

A CDPEAC *VU instruction* corresponds to a scalar or vector operation executed on the vector units. (In other words, a CDPEAC instruction corresponds to a single DPEAC statement.)

A VU instruction is either:

- an *arithmetic instruction*, which causes the VUs to perform a register arithmetic operation:

```
addv( i, V0, V1, V2 )   /* vector add (V2=V0+V1) */
```

- a *memory instruction*, which moves data between VU registers and parallel memory:

```
loadv( i, address , V0 )  /* load values into V0 */
```

- a *statement modifier*, which affects the compilation and/or execution of a CDPEAC statement:

```
vmmode(cond)  /* Vector mask conditionalization */
```

- or some combination of the above instruction types, made with the CDPEAC **join***n* operator:

```
join3( addv(i,V0,V1,V2),
       loadv(i,address,V0),
       vmmode(cond) )
```

5.1.2 VU Accessor Instructions

A CDPEAC *VU accessor instruction* is a CDPEAC instruction that doesn't correspond to a VU arithmetic or memory operation (that is, an instruction executed by the SPARC). 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, i.e., **dpset**, **dpget**, etc.:

```
/* Get memory argument stride */
dpget( i, DP_1, dp_stride_memory, sp_dest )

/* Read VU data register into SPARC register */
dprd( i, ALL_DPS, R0, sp_dest )
```

5.1.3 VU Special Instructions

A CDPEAC *special instruction* is an instruction not belonging in either of the other classes but that peforms some useful operation on the SPARC and/or VUs:

```
set_vector_length(8)    /* Set default vector length */
ldvm(R0)   /* Set contents of dp_vector_mask register */
```

5.1.4 The Join Macro

CDPEAC includes a macro named (**join***n*) that *joins* two or more CDPEAC VU instructions into a more efficient single instruction:

join[*1-9*](instruction1,...,instruction*n*) — *n*-way **join**

This joining is not arbitrary, however; it is based on the underlying statement syntax of DPEAC.

A single CDPEAC VU instruction represents a single VU operation, so a **join***n* statement can include no more than one memory instruction and one arithmetic instruction. Either or both can be omitted (in which case appropriate no-ops are generated). A **join***n* statement may also include any number (including zero) of modifiers, as permitted by the statement format in use (see Section 5.4).

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

`join`*n* **`(`** *arithmetic-inst*`,` *memory-inst*`,` *modifier-1*`,` *modifier-2 ...* **`)`**

This order is recommended because while 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 greatest interest.

Note: The *n* in the **`join`***n* macro name must match the number of arguments. It can range from 1 to 9. If there are only two arguments, the *n* can be omitted. Also, **`join`** statements cannot be used as arguments to other **`join`** statements (for example, you can't apply **`join2`** to two other **`join`** statements).

Chain Loading

When a join 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.

When a join 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 6.7 lists the CDPEAC statement modifiers and indicates which can and can't chain load.

5.1.5 Instruction Suffixes

CDPEAC instructions often use special suffixes, such as "**`_i`**", "**`_v`**", etc., to indicate alternate forms of a single arithmetic or memory instruction:

```
loadv_i(i,source,V2,V0) /* memory indirection */
movev_v(i,16,V0,V2)     /* explicit vector length */
```

These suffixes are introduced in the syntax and statement format sections below.

5.1.6 Argument Macros

There are also *argument macros* that apply to a single argument of a CDPEAC instruction, and provide some modification of the instruction's effects on that argument:

```
dreg_u( V8, 8 )     /* Register stride of 8 */
dreg_x( V2, 5 )     /* Register offset of 5 */
```

These macros are described in more detail in the syntax sections below.

5.2 CDPEAC Syntax

5.2.1 General Syntax

Since CDPEAC procedures are written in C, standard C syntax is followed for the overall structure of a CDPEAC procedure and declaration of its arguments. However, the arguments to a CDPEAC macro have their own syntax, which is derived from the underlying DPEAC syntax. CDPEAC expression syntax is the same as the DPEAC syntax described in Section 3.2, with one exception: the SPARC register syntax of DPEAC is replaced by references to C variables, as described below.

CDPEAC operations that need to refer to parallel memory addresses or SPARC registers, in particular the memory instruction operations **load** and **store**, take C variables as the parallel memory address or SPARC register argument. (These variables are converted internally into appropriate references for DPEAC.) Thus, for example, in the CDPEAC fragment:

```
unsigned source, dest;
loadv( f, source , V0 );
storev( f, dest , V0 );
```

the variables **source** and **dest** must be pointers to arrays in parallel memory of values (floating-point values, in this example). The length of these arrays must be as least as large as the current value of **dp_vector_length**. The contents of the **source** array are copied into the vector register **V0** of the VUs, and then read back out and stored in the **dest** array.

Note: Typically, the C variables used in this fashion will be addresses supplied by a C* or CM Fortran program, representing a subgrid of an array argument.

5.2.2 Vector Unit Data Registers

CDPEAC code refers 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: CDPEAC statements in immediate format use the `R0` and `R1` registers to store immediate arguments, 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" argument, the argument 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 16. 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.

5.2.3 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`. CDPEAC `join`*n* 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 5.9) for examples.

5.2.4 Register Offset Macro

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

```
dreg_x( dreg, index )
```

For example, **dreg_x**(V2, 5) refers to **R21**, that is, **V2** (=**R16**) + 5 = **R21**

5.2.5 VU Register Stride Macros

VU arithmetic operations can stride, or step through, a group of data registers. The stride increment is indicated by a *stride macro* applied to the appropriate register argument. The general syntax is shown below:

Macro Syntax	**Effect**
register	Unit stride (1 for singlewords, 2 for double).
dreg_u(*register*, *stride*)	Temporarily use specified *stride*.
scalar(*register*, *stride*)	Scalar striding, same as **dreg** (*register*, 0).
dreg_u(*register*, **mode**)	Use stride value stored in **dp_stride_rs1**.
dreg_s(*register*, *stride*)	Set **dp_stride_rs1** to *stride* and use it.
dreg_u_s(*register*, *stride*, *set-stride*)	Set **dp_stride_rs1** to *set-stride*, use *stride*.

In the above, *register* is any valid VU data 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.

Important: The stride marker forms shown here are not valid for all statement formats — most statement formats restrict the types of stride markers that are allowed. In particular, the latter four forms 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, **dreg_u**(R0,0)), the *rS1* register must be one of the scalar registers **S0** through **S15**, or **S30** for double-precision.

Note for DPEAC Programmers: There is no CDPEAC macro equivalent to the DPEAC *register=stride* stride marker format.

5.2.6 VU Memory Striding

VU memory operations can also stride through memory locations. The stride of vector instructions is either specified explicitly in the instruction, or else defaults to the value of the **dp_stride_memory** control register. Typically, the default memory stride (**dp_stride_memory**) is used, but CDPEAC memory instructions also allow you to specify the memory stride as part of the instruction.

For each memory instruction, there are a number of suffixes you can add that change the striding of the memory address argument. (**Note:** These memory stride instruction forms are mainly of use for CDPEAC instructions written in memory stride format; see Section 5.8.)

Instruction Suffix	Effect
mem-inst(*type*, *memop*, *reg*)	Use default stride on *memop*.
*mem-inst*_**u**(*type*, *memop*, *stride*, *reg*)	Use stride *stride* on *memop*.
*mem-inst*_**s**(*type*, *memop*, *stride*, *reg*)	Set default stride to *stride* and use it.
*mem-inst*_**u_s**(*type*, *memop*, *stride*, *set-stride*, *reg*)	Set default to *set-stride*, but use *stride* for this instruction.

For all the above suffix formats, the *stride* values that can be specified are restricted to **4** or **8** for singleword operations, and **8** or **16** for doubleword operations. When you write CDPEAC 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 CDPEAC accessor instruction **dpset** for this purpose; for example:

```
dpset (  i, ALL_DPS, 8, DP_STRIDE_MEMORY )
```

5.2.7 VU Selection in CDPEAC Statements

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

- the value of the *memory-argument* 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 high-level language compiler and/or the **`dpcc`** compiler generate these addresses for you.

5.2.8 VU Selection in CDPEAC Accessor Instructions

The VU(s) referenced by a CDPEAC 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).
C variable	Use value of specified variable.
`*`	Select all VUs.
`*`*n*	Use both VUs on chip *n* (0=VU's 0&1, 1=VU's 2&3).

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 by the header file **`cdpeac.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`**

DPEAC Usage Note: There is no CDPEAC equivalent of the modifier "**`<`**" for VU selectors in DPEAC, which selects bits **`12:15`** of the value.

5.3 CDPEAC Instructions

5.3.1 Scalar and Vector Instructions

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

Scalar instructions execute just once, for the supplied arguments, and are distinguished by an "**s**" suffix on the instruction name.

Vector instructions execute repeatedly for each of a series of arguments, and are distinguished by a "**v**" suffix on the instruction name. Vector instructions start with the specified register or memory address argument(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 instruction 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 CDPEAC **join** 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.

5.3.2 Register Arguments

The register arguments of CDPEAC 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.

5.3.3 Data Type Argument

Virtually every CDPEAC instruction has an initial *type* argument, which specifies the data type of the instruction. This argument must be one of the following data-type symbols:

`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

5.3.4 Arithmetic Instructions

An arithmetic instruction causes the VUs to perform a register arithmetic operation. Arithmetic instructions have the following general forms:

Monadic (one source):	*instruction*{**`v`**,**`s`**} (*type*, *rS1*, *rD*)
Dyadic (two sources):	*instruction*{**`v`**,**`s`**} (*type*, *rS1*, *rS2*, *rD*)
Triadic (three sources):	*instruction*{**`v`**,**`s`**} (*type*, *rS1*, *rLS*, *rS2*, *rD*)

Note: In the statement format descriptions in Section 5.4, the arithmetic operation is always shown in triadic form. Dyadic and monadic forms are obtained simply by omitting the appropriate arguments (*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 arguments, unless the argument explicitly specifies a different stride (see Section 5.2.5.)

rS2 Argument Restrictions: The *rS2* argument of an arithmetic instruction has these restrictions, imposed by the internal representation of the instruction:

- 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).

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.

5.3.5 Memory Instructions

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

memory-operation{**v**,**s**} (*type*, *memory-argument*, *rLS*)

The *rLS* argument can be any VU register, but if a triadic arithmetic operation and a memory operation are combined, the *rLS* argument 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-argument* can be any memory address that selects one or more VUs, and it is specified by a C variable containing the address as an unsigned integer.

The stride of vector instructions is always the default value given by the **dp_stride_memory** register.

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.

5.3.6 Modifiers

Modifiers: Modifiers are keywords, such as **pad**, **maddr**, **vmcurrent**, etc., that modify the assembly or execution of a CDPEAC statement. The modifiers permitted in a CDPEAC statement are determined by the statement's format. The available modifiers are listed below, and described in more detail in Section 6.7.

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

nopad, **pad**[**(***pad-size***)**]	Vector length padding (default is 4).
maddr(*memory-argument***)**	Default memory address.
{**vmrotate**, **vmcurrent**}	Packing mode for vector mask bits.
[**no**]**align**	Doubleword alignment declaration.
vmmode[**_s**]**(***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):

epc{**v**,**s**}**(***type***,** *sreg***,** *dreg***)**	Population count.
vmcount[**s**]**(***dreg***)**	Accumulated context count.
[**no**]**exchange**	On-chip VU data exchange.

5.4 CDPEAC Statement Formats

As noted in Section 5.1.4, the permissible arguments to a **`join`** statement are constrained by the DPEAC code that the **`join`** statement turns into. Thus, there are two main classes of **`join`** statement formats, *short format* and *long format*.

A *short format statement* assembles into a single word (32-bit) operation. Short format statements 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 statements are slower is issue, but use the extra word to provide additional argument types and modifiers that are not permitted by the short format. Specifically, the long format comes in four varieties:

> *Immediate format* allows an immediate argument 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. You can think of the short format as the backbone of features that all CDPEAC **`join`** statements are allowed to have, with each of the long formats representing some modification of or addition to those features.

Important! Because of the way that CDPEAC code is compiled and 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 **`join`** statement.

Note: You do not have to use the **`join`** macro to make use of the statement formats described below. It is perfectly legal to write a CDPEAC statement consisting of a single arithmetic or memory instruction using a modifier or macro allowed by any of the statement formats. Just be sure that you don't try to use more than one statement format in the same instruction.

5.5 The Short Format

The *short statement format* is:

Vector Instuctions:

```
joinn( arith-inst( type, rS1, vLS, vS2, vD ),
       mem-inst[_u]( type, mem-argument, [stride,] vLS ),
       modifier ... )
```

Scalar Instuctions:

```
joinn( arith-inst( type, rS1, sLS, sS2, sD ),
       mem-inst[_u]( type, mem-argument, [stride,] sLS ),
       modifier ... )
```

With one exception (the mode set statement format, see Section 5.9), the *rS1* argument can only have one of the following explicit stride forms:

rS1	Use register *rS1*, with unit stride for vector ops.
dreg_u(*rS1*, **mode**)	Use register *rS1*, with **dp_stride_rs1** stride.
dreg_u(*sS1*, **0**)	Use scalar register *sS1* with 0 stride.

The remaining register argument(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 5.7).

The *mem-inst* instruction can be in the "**_u**" (explicit memory stride) form, but if a memory stride is specified then the *rS1* argument must be either an aligned vector (**V***nn*) register, or a scalar (**S***nn*) register with an explicit stride of 0. The *mem-argument* must be a C variable (unsigned integer) giving a valid memory address.

The vector length for a vector operation is taken from **dp_vector_length**. This cannot be overridden in the short format (see mode set format, Section 5.9).

Only the following modifiers are permitted by the short format.

nopad, **pad**[**(***pad-size***)**]	Vector length padding (default is 4).
maddr(*memory-argument***)**	Default memory address.
{**vmrotate**, **vmcurrent**}	Packing mode of vector mask bits.
[**no**]**align**	Doubleword alignment declaration.
vmmode[**_s**]**(***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:

```
movev(i,V0,V1)                /* Integer monadic */
join2(movev(i,V0,V1),loadv(i,source,V0))
                              /* Same, chain-loaded */
movev(i,dreg_u(V0,mode),V1)   /* Default reg. stride */
movev(i,dreg_u(S0,0),V1)      /* Scalar reg, 0 stride */
moves(i,dreg_u(S0,0),S1)      /* Scalar operation */

loadv(i,source,V1)            /* memory operation */
join2(loadv(i,source,V1), noalign)
                              /* same, non-aligned */
loadv_u(f,source,4,V0)        /* unit stride, singleword */
loadv_u(f,source,8,V0)        /* unit stride, singleword */
loadv_u(df,source,8,V0)       /* unit stride, doubleword */
loadv_u(df,source,16,V0)      /* unit stride, doubleword */

join2(testv(i,V0,V1), maddr(source) )
                              /* maddr modifier */

gtv(df,V0,V1)                 /* Conditional */
join2(gtv(df,V0,V1), vmcurrent)
                              /* Conditional, with modifier */

addv(f,V0,V1,V2)              /* Float dyadic */
join2(addv(f,V0,V1,V2),nopad) /* No vlen padding */
madav(f,V0,V1,V2)             /* Mult-add */
madiv(f,V0,V1,V2)             /* Mult-add, inverted */

join2(madtv(f,V0,V1,V2,V3),loadv(f,source,V1))
                              /* True triadic, chain-loaded */
join2(addv(f,V0,V1,V2),load(i,source,V0),vmmode(condmem))
                              /* Conditional mem. op. */
join2(addv(f,V0,V1,V2),load(i,source,V0),vmmode(condalu))
                              /* Conditional arith. op. */
```

5.6 Immediate (Long) Format

The *immediate format* (indicated by an "**`i`**" suffix on the arithmetic operator) modifies the short format by replacing one source argument in the arithmetic instruction with an immediate value. (The operand replaced depends on the arithmetic instruction in use — see the instruction listings in Chapter 6.) The immediate value is loaded into **`R0`** (singleword operations) or **`R0`** and **`R1`** (doubleword operations) prior to use.

Vector Instuctions:

`join`*n* **`(`** *arith-inst***`i(`** *type, rS1, vLS, imm, vD* **`),`**
mem-inst[**`_u`**]**`(`** *type, mem-argument,* [*stride*,] *vLS* **`),`**
modifier ... **`)`**

Scalar Instuctions:

`join`*n* **`(`** *arith-inst***`i(`** *type, rS1, sLS, imm, sD* **`),`**
mem-inst[**`_u`**]**`(`** *type, mem-argument,* [*stride*,] *sLS* **`),`**
modifier ... **`)`**

The *imm* argument is a 32-bit immediate value, either a C variable 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 0's in the 32 least significant bits of their mantissas are allowed.

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

Examples:

```
movevi(i,29,V1)                         /* Monadic immed. */
movevi(i,value,V1)                      /* C variable */
join2(movevi(i,29,V1),loadv(i,source,V2)) /* with mem. op. */
join2(movevi(i,29,V1),loadv_u(i,source,8,V0))
                                        /* with mem stride */
movesi(i,29,S1)                         /* Scalar operation */
addvi(f,dreg_u(R0,0),29,V1)             /* Immed arithmetic */
join2(madtvi(f,dreg_u(R0,0),V1,29,V3),loadv(df,source,V1))
                                        /* Triadic immediate */
```

5.7 Register Stride (Long) Format

The *register stride format* modifies the short format by allowing any of the register stride macros (**`dreg_u`**, **`dreg_s`**, **`dreg_u_s`**, etc.) on the *rS2*, *rLS*, and *rD* register arguments. (The *rS1* format does *not* change.)

Vector Instuctions:

`join`*n*(*arith-inst*(*type, rS1,* {*stride vLS*}, {*stride vS2*}, {*stride vD*}),
mem-inst[**`_u`**](*type, mem-argument,* [*stride*,] {*stride vLS*}),
modifier ...)

Scalar Instuctions:

`join`*n*(*arith-inst*(*type, rS1,* {*stride sLS*}, {*stride sS2*}, {*stride sD*}),
mem-inst[**`_u`**](*type, mem-argument,* [*stride*,] {*stride sLS*}),
modifier ...)

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

The *stride* macros on the *rS2*, *rLS*, and *rD* can be any of the register stride macros described in Section 5.2.5, 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 short format's argument, vector length, and modifier restrictions apply.

Examples:

```
movev(i,V0,dreg_u(R4,4))   /* Integer monadic */
join2(movev(i,V0,dreg_u(R4,4)),loadv(i,source,V0))
                           /* Chain-loaded */
join2(movev(i,V0,dreg_u(R4,4)),loadv_u(i,source,8,V0))
                /* same, temp stride */
loadv(i,source,dreg_u(R4,4))  /* memory operation */
movev(i,dreg_u(V0,mode),dreg_u(R4,4)) /* Default reg. stride */
movev(i,dreg_u(S0,0),dreg_u(R4,4)) /* Scalar reg, 0 stride */
moves(i,dreg_u(S0,0),dreg_u(S3,2))    /* Scalar operation */
addv(f,V0,dreg_u(R20,4),dreg_u(R6,3)) /* Float dyadic */
join2(madtv(f,dreg(R0,0),dreg_u(V1,2),dreg_u(R20,4),dreg_u(R60,7)),
     loadv(df,source,dreg_u(V1,2)))
                              /* True triadic, chain-loaded */
```

5.8 Memory Stride (Long) Format

The *memory stride format* modifies the short format by allowing any of the memory stride variants (**`_u`**,**`_s`**,**`_u_s`**), of the memory instructions to be used. (See Section 5.2.6.)

Vector Instuctions:

`join`*n* **`(`** *arith-inst* **`(`** *type, rS1, vLS, vS2, vD* **`),`**
*mem-inst*_{**`u`**,**`s`**,**`u_s`**}**`(`***type,mem-argument,*[*stride*,*set-stride*,] *vLS***`),`**
modifier ... **`)`**

Scalar Instuctions:

`join`*n* **`(`** *arith-inst* **`(`** *type, rS1, sLS, sS2, sD* **`),`**
*mem-inst*_{**`u`**,**`s`**,**`u_s`**}**`(`***type, mem-argument,*[*stride*, *set-stride*,] *sLS***`),`**
modifier ... **`)`**

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

Examples:

```
loadv_s(i,source,8,V1)  /* use and set 8 */
loadv_u_s(i,source,8,4,V1) /* use 8, set 4 */
join2(movev(i,V0,V1),loadv_s(i,source,8,V0))  /* Chain-loaded */
```

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

The mode set "format" is actually a family of distinct but related variants, each determined by the appearance of one of the incompatible features listed above.

Note for DPEAC Users: There is no CDPEAC counterpart to the "scalar modifier variant" of the mode set format in DPEAC (described in Section 3.9.1).

5.9.1 Mode Set Format Variants

The legal mode set variants are:

Vector Length Variant

> **join***n* **(** *arith-inst*_{**v**,**vs**,**vh**,**vhs**}
> **(** *type, vlen, rS1, vLS, vS2, vD* **) ,**
> *mem-inst*_{**v**,**vs**,**vh**,**vhs**}[**_u**]
> **(** *type, vlen, mem-argument,* [*stride*,] *vLS* **) ,**
> *modifier ...* **)**

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**. (The syntax for the *vec-len* specifier is described in Section 5.9.2.)

Examples:

```
movev_v(i,16,V0,V2)        /* Integer monadic */
join2(movev_v(i,16,V0,V2),loadv(i,source,V0))
                           /* same, chain-loaded */
loadv_v(i,16,source,V1)    /* memory operation */
join2(loadv_v(i,16,source,V1), noalign)
                           /* same, non-aligned */
addv_v(f,16,V0,V2,V4)      /* Float dyadic */
join2(madtv_v(f,16,V0,V2,V4,V6),loadv(f,source,V2))
                           /* True triadic, chain-loaded */

movev_v(i,vlen,V0,V2)      /* C variable */
movev_vs(i,16,V0,V2)       /* Use and set len. */
movev_vs(i,vlen,V0,V2)     /* Use and set */
moves_vs(i,16,S0,S8)       /* Scalar set */
movev_vh(i,vlen,V0,V2)     /* 4 bit length */
movev_vhs(i,vlen,V0,V2)    /* 4 bit use/set */

join2(addv(f,V0,V1,V2), vmcurrent) /* Current mode */
join2(addv(f,V0,V1,V2), vmnew)     /* New mask copy */
join2(addv(f,V0,V1,V2), vmnop)     /* No mask copy */
join3(addv(f,V0,V1,V2), loadv(i,source,V0), vminvert)
                           /* Inverted conditional */
```

rS1 Stride Variant

join*n* **(** *arith-inst*[**_**{**v**,**vs**,**vh**,**vhs**}]
(*type,* [*vlen,*] {*stride rS1*}, *vLS, vS2, vD* **),**
mem-inst[**_**{**v**,**vs**,**vh**,**vhs**}][**_u**]
(*type,* [*vlen,*] *mem-argument,* [*stride*,] *vLS* **),**
modifier ... **)**

This variant lets you apply an arbitrary register stride macro to the *rS1* argument. This macro can be any of the stride macros described in Section 5.2.5.

Examples:

```
movev_v(i,16,dreg_u(V0,2),V2)   /* Use stride 2 */
movev(i,dreg_u_s(V0,1,4),V2)    /* Use 1, set 4 */
addv_v(f,16,dreg_u(V0,2),V2,V4) /* Float dyadic */

join2(madtv(f,dreg_u_s(V0,1,0),V2,V4,V6),loadv(df,source,V2))
                                /* Triadic */
```

Register Stride Indirection Variant

join*n* **(** *arith-inst*[**_**{**v**,**vs**,**vh**,**vhs**}]
(*type,* [*vlen,*] **dreg_i(***rS1,rIA***)**, *vLS, vS2, vD* **),**
mem-inst[**_**{**v**,**vs**,**vh**,**vhs**}][**_u**]
(*type,* [*vlen,*] *mem-argument,* [*stride*,] *vLS* **),**
modifier ... **)**
arith-op[*vec-len*] *rS1*[**(***rIA***:***stride***)**], *vLS, vS2, vD* **;** ****
mem-op[*vec-len*] *mem-argument, vLS***;** ****
modifier **;** ...

This variant allows the use of an arbitrary VU register to specify the *rS1* stride. The macros used to specify the indirection register are described in Section 5.9.3.

Examples:

```
movev(i,dreg_i(V0,V2),V4)       /* Reg. indirection */
join2(movev_v(i,16,dreg_i(V0,V2),V4),loadv(i,source,V0))
                                /* Same, chain-loaded */
movev(i,dreg_i(V0,dreg_u(V2,2)),V4)
                                /* Indirect. with stride */
```

Memory Stride Indirection Variant

join*n* **(** *arith-inst*[**_**{**v**,**vs**,**vh**,**vhs**}]
(*type,* [*vlen,*] *rS1, vLS, vS2, vD* **),**
mem-inst[**_**{**v**,**vs**,**vh**,**vhs**}][**_i**]
(*type,* [*vlen,*] *mem-argument, rIA, vLS* **),**
modifier ... **)**

This variant allows the use of memory stride indirection (indicated by an "**_i**" suffix on the memory instruction). Memory indirection format is described in Section 5.9.4.

Examples:

```
loadv_i(i,source,V2,V0)                 /* Mem. Indirect */
loadv_v_i(i,16,source,V2,V0)            /* Same, with vlen */
loadv_v_i(i,16,source,dreg_u(V2,4),V0)  /* Same, with stride */
join2(movev_v(i,16,V0,V4),loadv_i(i,source,V2,V0))
                                        /* Chain-loading */
```

Population Count Variant

join*n* **(** *arith-inst*[**_**{**v**,**vs**,**vh**,**vhs**}]**(** *type,* [*vlen,*] *rS1, vLS, vS2, vD* **),**
epc{**v**,**s**} **(** *type, vLS, rIA* **),**
modifier ... **)**

This variant allows you to specify the **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(u,V0,V1)             /* Unit stride */
epcv(u,V0,dreg_u(V1,2))   /* Explicit stride */
epcv(du,V0,dreg_u(V1,2))  /* Double op */
join2(addv_v(f,16,V0,V1,V2),epcv(u,V0,V1))    /* Chain-loading */
join2(addv_v(df,16,V0,V1,V2),epcv(du,V0,V4))  /* Double op */
```

Special Modifier Variant

join*n***(** *arith-inst*[**_**{**v**,**vs**,**vh**,**vhs**}]
(*type,* [*vlen,*] *rS1, vLS, vS2, vD* **),**
mem-inst[**_**{**v**,**vs**,**vh**,**vhs**}][**_u**]
(*type,* [*vlen,*] *mem-argument,* [*stride*,] *vLS* **),**
{[**no**]**exchange**, **vmcount**[**s**](*reg*)}
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:

```
join2(addv(f,V0,V1,V2),exchange)   /* exchange values */
join3(addv(f,V0,V1,V2),loadv(f,source,V0),exchange)
                                   /* chain-load */

vmcount(V0)                         /* Context count */
vmcount(dreg_u(V0,2))               /* with stride */
join2(addv(f,V0,V1,V2),vmcount(V0))   /* chain-loaded */
join3(addv(f,V0,V1,V2),loadv(f,source,V0),vmcount(V0))
                                    /* chain-loaded */
join2(addv(f,V0,V1,V2),vmcount(dreg_u(V0,2)))
                                    /* strided */
```

Scalar Instruction Variant

Note for DPEAC Users: There is no CDPEAC counterpart to the "scalar modifier variant" of the mode set format in DPEAC. However, you can use the special instructions described in Section 6.9 to accomplish the same effect.

5.9.2 Vector Length Instruction Suffixes

In all mode set format variants, either (or both) of the arithmetic and memory instructions can explicitly specify a vector length. This is indicated by a special suffix attached to the instruction, and by an extra *vlen* argument. These suffixes can also be used to modify the default vector length stored in the register **`dp_vector_length`**. The defined vector length suffixes are:

Syntax	Effect
operator`_v`	Use constant vector length *vlen*.
operator`_vs`	Use/set `dp_vector_length` to *vlen*.
operator`_vh`	Use length from bits `19:22` of *vlen*.
operator`_vhs`	Use/set `dp_vector_length` from bits `19:22` of *vlen*.

Note: The vector length suffixes listed above are, in some mode set variants, combined with the `_i` (indirection) and `_u` (explicit stride suffixes), as in the form *operator*`_vhs_u`.

The *vlen* argument is either a constant-expression or a C variable. The length specified must always be an integer from `1` to `16`.

Either or both of the arithmetic and memory instructions in a `join` statement may be given a vector length suffix; the specified vector length applies to both instructions. If a vector length is specified in both instructions, both the suffix and vector length for both instructions must be the same.

Implementation Note: If you specify the vector length with a C variable (which is translated into a SPARC register reference at the DPEAC level) or by defaulting to the value of `dp_vector_length`, then 1 is added to the length before it is used. Whenever a value is stored into `dp_vector_length` by one of the suffix forms above, it is stored in decremented form, so that this implicit incrementing by 1 will work properly.

5.9.3 Register Stride Indirection

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

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

The *rIA* register argument 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:

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

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.

5.9.4 Memory Indirection

For the memory stride indirection (`_i` suffix) form of CDPEAC memory operations, the *rIA* argument format is one of:

Syntax	Effect
register	Memory indirection, unit stride.
`dreg_u(`*register, stride*`)`	Memory indirection, constant *stride.*

The specified single-precision VU *register* contains offsets that are separately added to the memory address to obtain each argument 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* in the specified *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.

5.9.5 Mode Set Format Modifier

The following modifiers are permitted by the mode set format:

These modifiers are permitted by the short format:

nopad, **pad**[**(***pad-size***)**]	Vector length padding (default is 4).
maddr(*memory-argument***)**	Default memory address.
{**vmrotate**, **vmcurrent**}	Packing mode for vector mask bits.
[**no**]**align**	Doubleword alignment declaration.
vmmode[**_s**]**(***mode-keyword***)**	Conditionalization mode selector.

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

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

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

epc{**v**,**s**}**(***type*, *sreg*, *dreg***)**	Population count.
vmcount[**s**]**(***dreg***)**	Accumulated context count.
[**no**]**exchange**	On-chip VU data exchange.

These modifiers are all described in more detail in Section 6.7.

Chapter 6

CDPEAC Instruction Set Reference

This chapter presents a quick-reference list of the CDPEAC instruction set, including CDPEAC instructions, argument macros, and accessor instructions.

6.1 The CDPEAC Join Macro

The **join** operator connects arithmetic operations, memory operations, and statement modifiers to form compound CDPEAC statements:

> **join**`(instruction1,instruction2)` — default **join**, same as **join2**
> **join***N*`(instruction1,...,instruction`*N*`)` — *N*-way **join**
> *N* `= {1,2,3,4,5,6,7,8,9}`

A **join** can have at most one arithmetic and one memory operation, but any number of modifiers from 0 to 7. The *N* of a **join***N* must match the total number of instructons (operations and modifiers) supplied to the **join***N*.

6.2 CDPEAC Type Abbreviations

These symbols can be used as the *type* argument of a CDPEAC instruction:

Type	Meaning
u , **du**	Unsigned integer, singleword (32-bit) and doubleword (64-bit).
i , **di**	Signed integer, singleword and doubleword.
f , **df**	Float value, single-precision (32-bit) and double-precision (64-bit).

6.3 CDPEAC Argument Macros

Data Register Offsets:

dreg_x(*dreg*, *index*)	Data register offset (*index* must be a constant). If *dreg* is **R***nn*, this form refers to **R**(*nn*+*index*).

Note: The **dreg_x** form can be the `dreg` argument in any modifier below.

Data Register Striding:

dreg	With no modifier, use unit striding. (Unit stride is 1 for singles, 2 for doubles.)
dreg_u(*dreg*, *stride*)	Use given *stride* once.
scalar(*dreg*)	Scalar striding, same as **dreg_u**(*dreg*, 0).
dreg_u(*dreg*, **mode**)	Use default stride (**dp_stride_rs1**).
dreg_s(*dreg*, *stride*)	Store *stride* as the *rS1* default and use it.
dreg_u_s(*dreg*, *stride*, *set_stride*)	Use *stride*, and store *set_stride* as default.

Data Register Indirection:

dreg_i(*dreg*, *ireg*)	Simple register indirection.
dreg_i(*dreg*, **dreg_u**(*ireg*, *stride*))	Register indirection, *ireg* striding.

6.4 Instruction Suffixes

These suffixes appear at the end of long format CDPEAC instructions, and indicate an alternate instruction form and/or argument list:

Type	Meaning	
i	Immediate value argument.	(arithmetic operations only)
_i	Memory stride indirection.	(memory operations only)
_u	Use explicit memory stride.	(memory operations only)
_s	Use and set memory stride.	(memory operations only)
_u_s	Use *stride* and set *set_stride* as default.	(mem. operations only)
_v	Use explicit vector length.	(unsticky)
_vs	Use and set vector length.	(sticky)
_vh	Vector length from variable.	(unsticky, **1+**(bits **19:22**))
_vhs	Vector length from variable.	(sticky, **1+**(bits **19:22**))

6.5 CDPEAC Arithmetic Instructions

6.5.1 Monadic (One-Source) Arithmetic Instructions

These operators perform an arithmetic operation on the single *rS1* argument, and store the result in the *rD* argument. (**Note:** In immediate format, indicated by the **`i`** suffix, the first source argument, *rS1*, is the immediate value.)

Formats:

opcode`{`**`s,v`**`}[`**`i`**`](`*type*`,`*rS1*`,`*rD*`)`
opcode`{`**`s,v`**`}_{`**`v,vs,vh,vhs`**`}(`*type*`,`*vlen*`,`*rS1*`,`*rD*`)`
type = {**`u`**, **`du`**, **`i`**, **`di`**, **`f`**, **`df`**}

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

The **`to`** operators have an extra *type* argument, and convert between the two types: *rS1* is of *type1*, *rD* of *type2*. (In immediate, **`i`**, format, *rS1* is immediate.)

Format:

opcode`{`**`s,v`**`}[`**`i`**`](`*type1*`,`*type2*`[`**`r`**`],`*rS1*`,`*rD*`)`
opcode`{`**`s,v`**`}_{`**`v,vs,vh,vhs`**`}(`*type1*`,`*type2*`[`**`r`**`],`*vlen*`,`*rS1*`,`*rD*`)`
type1, type2 = {**`u`**, **`du`**, **`i`**, **`di`**, **`f`**, **`df`**}

Opcode	Type1	Type2	Purpose
`to`	{**`u`**, **`du`**, **`i`**, **`di`**}	{**`f`**, **`df`**}	Convert integer to float.
`to`	{**`f`**, **`df`**}	{**`f`**, **`df`**}	Convert to another precision.
`to`	{**`f`**, **`df`**}	{**`u`**, **`du`**, **`i`**, **`di`**}**`r`**	Convert to integer (round).
`to`	{**`f`**, **`df`**}	{**`u`**, **`du`**, **`i`**, **`di`**}	Convert to integer (truncate).

6.5.2 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, **i**, format, the *rS2* argument is the immediate value.)

Formats:

opcode{**s,v**}[**i**](*type*, *rS1*, *rS2*, *rD*)
opcode{**s,v**}_{**v,vs,vh,vhs**}(*type*, *vlen*, *rS1*, *rS2*, *rD*)
type = {**u**, **du**, **i**, **di**, **f**, **df**}

Opcodes	Types	Purpose
add	{**u**, **du**, **i**, **di**, **f**, **df**}	Add (*rD* = *rS1* + *rS2*).
addc	{**u**, **du**, **i**, **di**}	Integer add with carry bit from shift of vector mask register.
sub	{**u**, **du**, **i**, **di**, **f**, **df**}	Subtract (*rD* = *rS1* – *rS2*).
subc	{**u**, **du**, **i**, **di**}	Integer subtract with carry bit from shift of vector mask register.
subr	{**u**, **du**, **i**, **di**, **f**, **df**}	Subtract reversed (*rD* = *rS2* – *rS1*).
sbrc	{**u**, **du**, **i**, **di**}	Integer subtract reversed with carry bit from shift of vector mask register.
mul	{**u**, **du**, **i**, **di**, **f**, **df**}	Multiplication (low 32/64 bits for ints).
mulh	{**du**, **di**}	Integer multiply (high 64 bits).
div	{**f**, **df**}	Divide (*rD* = *rS1* / *rS2*).
enc	{**u**, **du**}	Make float from exp and mant (*rS1*, *rS2*).
shl	{**u**, **du**}	Shift left (*rD* = *rS1* << *rS2*).
shlr	{**u**, **du**}	Shift left, reversed (*rD* = *rS2* << *rS1*).
shr	{**u**, **du**, **i**, **di**}	Shift right (*rD* = *rS1* >> *rS2*).
shrr	{**u**, **du**, **i**, **di**}	Shift right, reversed (*rD* = *rS2* >> *rS1*).
and	{**u**, **du**}	Bitwise logical **AND**.
nand	{**u**, **du**}	Bitwise logical **NAND**.
andc	{**u**, **du**}	Bitwise logical **NOT**(*rS1*) **AND** *rS2*.
or	{**u**, **du**}	Bitwise logical **IOR**.
nor	{**u**, **du**}	Bitwise logical **NOR**.
xor	{**u**, **du**}	Bitwise logical **XOR**.
mrg	{**u**, **du**, **i**, **di**, **f**, **df**}	If vector mask bit = 1 then *rS1* else *rS2*.

6.5.3 Arithmetic Comparisons

These operators perform an arithmetic comparison between the *rS1* and *rD* arguments, and set status flags accordingly. *rD* is not modified. (In immediate, **i**, format, the *rD* argument is the immediate value.)

Format:

opcode{**s,v**}[**i**](*type*,*rS1*,*rD*)
opcode{**s,v**}_{**v,vs,vh,vhs**}(*type*,*vlen*,*rS1*,*rD*)
type = {**u**, **du**, **i**, **di**, **f**, **df**}

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

6.5.4 Compare (Dyadic with rD constant)

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

Format:

opcode{**s,v**}[**i**](*type*,*rS1*,*rS2*,*code*)
opcode{**s,v**}_{**v,vs,vh,vhs**}(*type*,*vlen*,*rS1*,*rS2*,*code*)
type = {**u**, **du**, **i**, **di**, **f**, **df**}

Opcode	Types	Code	Purpose
cmp	{**u**, **du**, **i**, **di**, **f**, **df**}	0	Test for greater than.
cmp	{**u**, **du**, **i**, **di**, **f**, **df**}	1	Test for equal.
cmp	{**u**, **du**, **i**, **di**, **f**, **df**}	2	Test for less than.
cmp	{**u**, **du**, **i**, **di**, **f**, **df**}	3	Test for greater than or equal.
cmp	{**u**, **du**, **i**, **di**, **f**, **df**}	4	Test for unordered (NaN present).
cmp	{**u**, **du**, **i**, **di**, **f**, **df**}	5	Test for ordered and not equal.
cmp	{**u**, **du**, **i**, **di**, **f**, **df**}	6	Test for not equal or unordered.
cmp	{**u**, **du**, **i**, **di**, **f**, **df**}	7	Test for less than or equal.

6.5.5 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*. (In immediate, **i**, format, the *rS2* argument is the immediate value.)

Format:

opcode{**s,v**}[**i**](*type*,*rS1*,*rS2*,*rD*)
opcode{**s,v**}_{**v,vs,vh,vhs**}(*type*,*vlen*,*rS1*,*rS2*,*rD*)
type = {**u**, **du**, **i**, **di**, **f**, **df**}

Note: In the opcode descriptions below, 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).

Accumulative Operators

Opcodes	Types	Purpose
mada	{**u**, **du**, **i**, **di**, **f**, **df**}	*rD* = (*rS1* * *rS2*) + *rD*
msba	{**u**, **du**, **i**, **di**, **f**, **df**}	*rD* = (*rS1* * *rS2*) – *rD*
msra	{**u**, **du**, **i**, **di**, **f**, **df**}	*rD* = *rD* – (*rS1* * *rS2*)
nmaa	{**u**, **du**, **i**, **di**, **f**, **df**}	*rD* = –*rD* – (*rS1* * *rS2*)
m[**h**]**sa**	{**du**}	*rD* = (*rS1* * *rS2*) **AND** *rD*
m[**h**]**ma**	{**du**}	*rD* = (*rS1* * *rS2*) **AND NOT** *rD*
m[**h**]**oa**	{**du**}	*rD* = (*rS1* * *rS2*) **IOR** *rD*
m[**h**]**xa**	{**du**}	*rD* = (*rS1* * *rS2*) **XOR** *rD*

Inverted Operators

Opcodes	Types	Purpose
madi	{**u**, **du**, **i**, **di**, **f**, **df**}	*rD* = (*rS2* * *rD*) + *rS1*
msbi	{**u**, **du**, **i**, **di**, **f**, **df**}	*rD* = (*rS2* * *rD*) – *rS1*
msri	{**u**, **du**, **i**, **di**, **f**, **df**}	*rD* = *rS1* – (*rS2* * *rD*)
nmai	{**u**, **du**, **i**, **di**, **f**, **df**}	*rD* = –*rS1* – (*rS2* * *rD*)
m[**h**]**si**	{**du**}	*rD* = (*rS2* * *rD*) **AND** *rS1*
m[**h**]**mi**	{**du**}	*rD* = (*rS2* * *rD*) **AND NOT** *rS1*
m[**h**]**oi**	{**du**}	*rD* = (*rS2* * *rD*) **IOR** *rS1*
m[**h**]**xi**	{**du**}	*rD* = (*rS2* * *rD*) **XOR** *rS1*

6.5.6 Convert Operation (Dyadic with Rs2 Constant)

These operations convert the **src** 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 **cdpeac.h** header file. (In immediate, **i**, format, the *rS1* argument is the immediate value.)

Format:

opcode{**s,v**}[**i**](*type*,*rS1*,*code*,*rD*)
opcode{**s,v**}_{**v,vs,vh,vhs**}(*type*,*vlen*,*rS1*,*code*,*rD*)
type = {**i**[**r**], **f**, **fi**}
code = a C constant from the list below

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.
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.

6.5.7 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, **i**, format, the *rS2* argument is the immediate value.)

Format:

opcode{**s,v**}[**i**](*type*,*rS1*,*rLS*,*rS2*,*rD*)
opcode{**s,v**}_{**v,vs,vh,vhs**}(*type*,*vlen*,*rS1*,*rLS*,*rS2*,*rD*)
type = {**u**, **du**, **i**, **di**, **f**, **df**}

Note: In the opcode descriptions below, 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).

Opcodes	Types	Purpose
madt	{**u**, **du**, **i**, **di**, **f**, **df**}	*rD* = (*rS1* * *rLS*) + *rS2*
msbt	{**u**, **du**, **i**, **di**, **f**, **df**}	*rD* = (*rS1* * *rLS*) – *rS2*
msrt	{**u**, **du**, **i**, **di**, **f**, **df**}	*rD* = *rS2* – (*rS1* * *rLS*)
nmat	{**u**, **du**, **i**, **di**, **f**, **df**}	*rD* = –*rS2* – (*rS1* * *rLS*)
m[**h**]**st**	{**du**}	*rD* = (*rS1* * *rLS*) **AND** *rS2*
m[**h**]**mt**	{**du**}	*rD* = (*rS1* * *rLS*) **AND NOT** *rS2*
m[**h**]**ot**	{**du**}	*rD* = (*rS1* * *rLS*) **IOR** *rS2*
m[**h**]**xt**	{**du**}	*rD* = (*rS1* * *rLS*) **XOR** *rS2*

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.

6.5.8 No-Op Operator

The untyped arithmetic no-op allows modifier side effects without specifying an operation. The no-op takes no arguments (except for the *vlen* argument in the vector-length cases). The suffixes are as described above.

Format:

fnop{**s,v**}()
fnop{**s,v**}_{**v,vs,vh,vhs**}(*vlen*)

6.6 CDPEAC Memory Instructions

These operations move data between VU memory and data registers.
Note: The default memory stride is stored in **`dp_stride_memory`**.

Formats:

opcode{**s,v**}(*type*,*address*,*dreg*)
— use default memory stride.
opcode{**s,v**}**_u**(*type*,*address*,*stride*,*dreg*)
— use stride once.
opcode{**s,v**}**_s**(*type*,*address*,*stride*,*dreg*)
— use *stride* and store it as default.
opcode{**s,v**}**_u_s**(*type*,*address*,*stride*,*set_stride*,*dreg*)
— use *stride*, and store *set_stride* as default.
opcode{**s,v**}**_i**(*type*,*address*,*ireg*,*dreg*)
— memory stride indirection.
opcode{**s,v**}**_i**(*type*,*address*,**dreg_u**(*ireg*,*stride*),*dreg*)
— memory indirection with *stride* on *ireg*.
opcode{**s,v**}**_**{**v,vs,vh,vhs**}(*type*,*vlen*,*address*,*dreg*)
— explicit vector length for CDPEAC statement.
opcode{**s,v**}_{**v,vs,vh,vhs**}**_i**(*type*,*vlen*,*address*,*ireg*,*dreg*)
— vector length and memory stride indirection.
opcode{**s,v**}_{**v,vs,vh,vhs**}**_u**(*type*,*vlen*,*address*,*cstride*,*dreg*)
— vector length and use-once *cstride*.
type = {**u**, **du**, **i**, **di**, **f**, **df**}

Opcode	Types	Purpose
load	{**u**, **du**, **i**, **di**, **f**, **df**}	Load from memory to VU data register.
store	{**u**, **du**, **i**, **di**, **f**, **df**}	Store from VU data register to memory.

No-Op Instruction: Untyped memory no-op allows modifier side effects without a load or store. Suffixes and arguments are as in the load/store formats above.

```
memnop(address)
memnop_u(address, ustride)
memnop_s(address, stride)
memnop_u_s(address, stride, set_stride)
memnop_i(address, idreg)
memnop_{v,vs,vh,vhs}(vlen, address)
memnop_{v,vs,vh,vhs}_i(vlen, address, idreg)
memnop_{v,vs,vh,vhs}_u(vlen, address, ustride)
```

6.7 CDPEAC Statement Modifiers

This section describes the statement modifiers that can be **`join`**ed 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.

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

`nopad`, **`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-address* **`)`** 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 17.) **Note:** this modifier is allowed by the short format for conditional operations only. Otherwise, it can only be used in the mode set format.

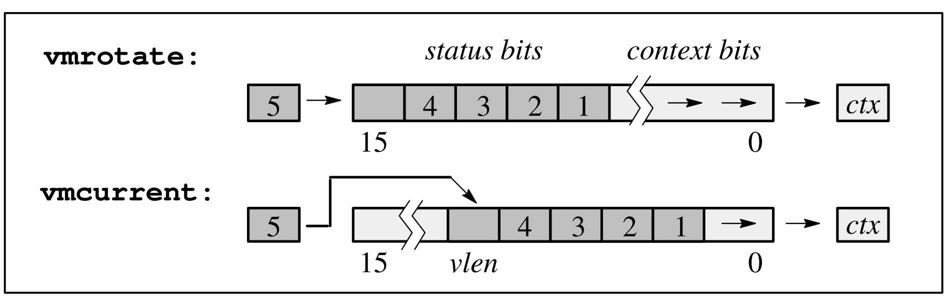

Figure 17. Bit-shifting modes of vector mask register.

[**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**.)

6.7.2 Conditionalization Modifiers

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

vmmode[**_s**]**(***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_s(always)	Set **dp_vector_mask_mode** for no conditionalization.
vmmode(condmem)	Conditionalize loads and stores in this instruction.
vmmode_s(condmem)	Set **dp_vector_mask_mode** for conditionalization.
vmmode(condalu)	Conditionalize arithmetic in this instruction.
vmmode_s(condalu)	Set **dp_vector_mask_mode** for condit. arithmetic.
vmmode_s(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 allowed only in the mode set statement format.

6.7.3 Special Modifiers (Mode Set Format Only)

epc{**v**,**s**}(*type*, *vLS*, *rIA*) Default: None

Population Count: The **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 **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 **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 **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***)** 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 **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 used only in the mode set format. However, it is incompatible with register stride indirection, memory stride indirection, and with the **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.

6.8 CDPEAC 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.

6.8.1 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[**_sync,_nosync**](*type*,selector,sp_src,vu_dreg)
dpwrt[**_sync,_nosync**](*type*,selector,value,vu_dreg)
dprd[**_sync,_nosync**](*type*,selector,vu_dreg,sp_dest)
type = {**u**, **du**, **i**, **di**, **f**, **df**}
sync/**nosync** = whether to sync VU pipeline (default is **sync**)

```
dpwrt_sync(i,ALL_DPS,%i1,V0)
dpwrt_nosync(i,DPS_0_AND_1,29,V0)
dprd(i,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[**_supervisor**](*type*,selector,sp_src,vu_creg)
dpset[**_supervisor**](*type*,selector,sp_src,vu_creg)
dpget[**_supervisor**](*type*,selector,vu_creg,sp_dest)
type = {**u**, **du**, **i**, **di**, **f**, **df**}
supervisor = get/set in supervisor region

```
dpset(i,DP_3,%i0,DP_VECTOR_MASK)
dpset(i,ALL_DPS,0,DP_VECTOR_MASK)
dpget(i,DPS_0_AND_1,DP_VECTOR_MASK,%i0)
```

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

dpchgsp(src,dest) Toggle between data/instruction spaces.

```
dpchgsp(R5,R6)
```

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

dpchgbk(src,selector,dest) Change referenced VU region.

```
dpchgbk(R5,DPS_0,R6)
```

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

dpld(*type*,address,sp_dest)
dpst(*type*,sp_src,address)
type = {**u**, **du**, **i**, **di**, **f**, **df**}

```
dpld(i, [%i0], %i1)
dpst(i, %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()

```
addv(f,V0,V1,V2)
dpsync()
mulv(f,V1,V2,V3)
```

6.9 CDPEAC Special Instructions

These control operations are always used as single statements, and typically perform some useful operation on VU or SPARC registers and/or memory locations.

VU Internal Register Modifiers: These operations expand into CDPEAC instructions with special modifier flags that set the values of one or more of the following VU internal registers:

dp_vector_mask_mode	Default vector mask mode
dp_stride_memory	Default memory stride
dp_stride_rs1	Default *rS1* register stride
dp_vector_length	Default vector length

set_vmmode(vmmode)	Sets **dp_vector_mask_mode** to vmmode
set_mem_stride(stride)	Sets **dp_stride_memory** to stride
set_rs1_stride(rs1_stride)	Sets **dp_stride_rs1** to rts_stride
set_vector_length(vlen)	Sets **dp_vector_length** to vlen

set_vector_length_and_vmmode(vlen,vmmode)
set_vector_length_and_rs1_stride(vlen,rs1_stride)
set_vector_length_and_rs1_stride_and_vmmode
(vlen,rs1_stride,vmmode)

Vector Mask Load/Store: These operators move the value of the vector mask register to or from the specified VU data register (dreg).

ldvm(dreg)
stvm(dreg)

```
ldvm    V1
stvm    V1
```

CDPEAC Function Setup/Cleanup: These functions set up (and clean up) the VU registers before and after a user-written CDPEAC routine. (**Usage Note:** These operators are not always necessary, depending on the use of a CDPEAC routine, but it is not harmful to include them. Their use is recommended.)

dpsetup()	Initializes the SPARC registers for use with CDPEAC code; must appear at start of block of CDPEAC code.
dpcleanup()	Restores state of VU control registers for CM Run-Time System code. Must appear at end of a block of CDPEAC code that can be called by the CMRTS.