Chapter 5

Instructions and Opcodes

DPEAC provides several types of instructions:

- VU instructions

  These instructions provide arithmetic and/or memory operations.

- VU accessor instructions

  These instructions read and write the control and data registers of the accelerator chips.

- VU Indirect Execution instructions

  These instructions allow the indirect execution of VU instructions (See Chapter 6 for a discussion of indirect execution.)

- Control instructions

  These instructions control the IU (the SPARC). They have no effect on the accelerator chip.

This chapter briefly discusses or summarizes each instruction type.

5.1 Arithmetic Opcodes For VU Instructions

The tables below list the opcodes that are supported for arithmetic operations. Operations are generally prefixed with the type of operands they expect. The supported types are:

f	single float	**df**	double float
i	signed 32–bit integer	**di**	signed 64–bit intger
u	unsigned 32–bit integer	**du**	unsigned 64–bit intger

Opcodes for arithmetic operations end either with the letter **v** or the letter **s**. Operations whose opcodes end in **v** are vector operations; those that end with **s** are scalar operations.

5.1.1 Monadic Operations

Opcode	Function
{**i**,**di**,**u**,**du**,**f**,**df**}**move**{**v**,**s**}	Move (pass) No status
{**i**,**di**,**u**,**du**,**f**,**df**}**test**{**v**,**s**}	Pass operand and test
{**u**,**du**}**not**{**v**,**s**}	Bitwise Invert
{**f**,**df**}**clas**{**v**,**s**}	Classify Operand
{**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)
cvt{**f**,**fi**,**i**,**ir**}{**v**,**s**}	General Convert
{**f**,**df**}**exp**{**v**,**s**}	Extract exponent
{**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
{**i**,**di**,**f**,**df**}**abs**{**v**,**s**}	Absolute Value
{**f**,**df**}**inv**{**v**,**s**}	Invert (*Rd* = 1 / *Rs1*)
{**f**,**df**}**sqrt**{**v**,**s**}	Square Root
{**f**,**df**}**isqt**{**v**,**s**}	Inverse Root (*Rd* = 1/SQRT(*Rs1*))

5.1.2 Dyadic Operations

Opcode	Function
{**i**,**di**,**u**,**du**,**f**,**df**}**add**{**v**,**s**}	Add
{**i**,**di**,**u**,**du**}**addc**{**v**,**s**}	Integer add with carry
{**i**,**di**,**u**,**du**,**f**,**df**}**sub**{**v**,**s**}	Subtract
{**i**,**di**,**u**,**du**}**subc**{**v**,**s**}	Integer subtract with carry
{**i**,**di**,**u**,**du**,**f**,**df**}**subr**{**v**,**s**}	Subtract reversed
{**i**,**di**,**u**,**du**}**sbrc**{**v**,**s**}	Integer subtract reversed with carry
{**i**,**di**,**u**,**du**,**f**,**df**}**mul**{**v**,**s**}	Multiply (low 32 or 64 bits for int)
{**di**,**du**}**mulh**{**v**,**s**}	Integer multiply (high 64 bits)
{**f**,**df**}**div**{**v**,**s**}	Divide
{**u**,**du**}**enc**{**v**,**s**}	Make float from exp and mant (*Rs1*, *Rs2*)
{**u**,**du**}**shl**{**v**,**s**}	Logical Shift left
{**u**,**du**}**shlr**{**v**,**s**}	Logical Shift left, reversed
{**i**,**di**,**u**,**du**}**shr**{**v**,**s**}	Shift right
{**i**,**di**,**u**,**du**}**shrr**{**v**,**s**}	Shift right, reversed
{**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**}**gt**{**v**,**s**}	Test for Greater Than
{**i**,**di**,**u**,**du**,**f**,**df**}**ge**{**v**,**s**}	Test for Greater Than or Equal
{**i**,**di**,**u**,**du**,**f**,**df**}**lt**{**v**,**s**}	Test for Less Than
{**i**,**di**,**u**,**du**,**f**,**df**}**le**{**v**,**s**}	Test for Less Than or Equal
{**i**,**di**,**u**,**du**,**f**,**df**}**eq**{**v**,**s**}	Test for Equal
{**i**,**di**,**u**,**du**,**f**,**df**}**ne**{**v**,**s**}	Test for Not Equal or Undordered
{**i**,**di**,**u**,**du**,**f**,**df**}**lg**{**v**,**s**}	Test for Ordered and Not Equal
{**i**,**di**,**u**,**du**,**f**,**df**}**un**{**v**,**s**}	Test for Unordered
{**i**,**di**,**u**,**du**,**f**,**df**}**cmp**{**v**,**s**}	Compare
{**i**,**di**,**u**,**du**,**f**,**df**}**mrg**{**v**,**s**}	If (mask bit == 1) then *Rs1* else *Rs2*
fnop{**v**,**s**}	No arithmetic operation

5.1.3 Two-Operation Instructions

DPEAC offers a number of two–operation instructions. All of these perform a multiply plus some other operation (for example, an add).

Each instruction comes in three forms, with each form offering an alternative mapping of the operands to the computation inputs. Two forms, the accumulative and the inverted forms, are dyadic: that is, they operate on two operands. The third form, the true triadic form, takes three input operands. (See Section 4.9 for details on specifying the three operands and special rules which apply.)

A letter suffix to the opcode indicates which of the three forms is desired, as shown in the following chart (the operation in this case is a multiply/add):

Form	Operation	Opcode Suffix
Accumulative Form	*Rd* = (*Rs1* * *Rs2*) + *Rd*	**a**
Inverted Form	*Rd* = (*Rs2* * *Rd*) + *Rs1*	**i**
True Triadic Form	*Rd* = (*Rs1* * *Rls*) + *Rs2*	**t**

The following tables list the two–operation instructions. Notice that the opcode suffix precedes the vector/scalar suffix (**v** or **s**). For example, **dfmadiv** is the inverted form of the double precision multiply–add (**mad**) vector operation.

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*

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*

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*

5.1.4 Special Control Instructions

The following instructions exert special control over the accelerator chip. In all of these, the current vector length settings are ignored (they are all essentially scalar). No load/store operations may be executed concurrently with these instructions, and only the Short instruction format may be used with them. These instructions are not affected by conditionalization.

Opcode	Function
trap	Generate trap unconditionally.
etrap	Generate trap if the **set_enb** bit in the **dp_interrupt_cause_green** register is set.
ldvm	Move *Rs1* to **dp_vector_mask**
stvm	Move **dp_vector_mask** to *Rs1*

5.2 Memory Opcodes for VU Instructions

The following opcodes are supported for Memory operations. Data types are as in the arithmetic operations:

f	single float	**df**	double float
i	signed 32–bit integer	**di**	signed 64–bit intger
u	unsigned 32–bit integer	**du**	unsigned 64–bit intger

Opcodes for memory operations end either with the letter **v** or the letter **s**. Operations whose opcodes end in **v** are vector operations; those that end with **s** are scalar operations.

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
memnop	No memory Operation

The **memnop** operation is useful when an operation should have the side effects of the memory syntax without actually performing a memory operation. For example, the following instruction does not perform a memory operation, but sets the **dp_stride_memory** control register to the value 100:

```
faddv V1,V2,V3 ; memnop []:=100
```

Note that the assembler allows the memory address to be unspecified. In this case, the default Instruction Space Address (**DPV_STACK_INST_PORT_ALL**) is used, as if no memory operation syntax had been given. If given, however, it will be used as the Instruction Space Address for executing the instruction, to control which VU executes the instruction (as the **maddr** modifier, described below, would do).

5.3 VU Accessor Instructions

Several special instructions are provided for reading and writing the control and data registers of the accelerator chips.

Instruction	Function
dpwrt[**d**]	Write data registers
dprd[**d**]	Read data registers
dpset[**d**]	Write control registers
dpget[**d**]	Read control registers
dpchgsp	Convert between data and instruction space
dpchgbk	Convert address from one VU to another
dpld[**d**]	Read parallel memory
dpst[**d**]	Write parallel memory
dpsync	Synchronize VU pipelines

These accessors are also implemented as macros to be used from the C language. The following sections describe the VU accessor instructions, and then describe the macros.

5.3.1 Writing the Data Registers: dpwrt, dpwrtd

The **`dpwrt`** and **`dpwrtd`** instuctions write the register files of one or all accelerator chips. Use **`dpwrt`** to write any of the 128 single precision registers; use **`dpwrtd`** to write any of the 64 double precision register pairs.

The syntax is:

> **dpwrt** *VU–selector*, *value*, *VU–register* [, {**`sync`**, **`nosync`**}]
> or
> **dpwrt**[**d**] *VU–selector*, *as–register*, *VU–register* [, {**`sync`**, **`nosync`**}]

where

VU–selector indicates which of the accelerator chips on each PN are to be written. (For *VU–selector* syntax, see Section 3.15.)

value is the value to be written; a constant or *constant–expression*.

as–register is a SPARC IU register containing the value to be written.

VU–register is the register to be written. Any legal VU register specifier may be used, with the exception that for **`dpwrtd`**, only even–numbered registers are allowed.

The **`sync`** modifier causes additional code to be emitted, prior to the register write, to synchronize the write with the previous accelerator instructions. The alternative **`nosync`** modifier prevents the inclusion of synchronizing code. The default is **`sync`**, which synchronizes the instructions. The **`sync`** and **`nosync`** keywords can be set in upper or lower, but not mixed case.

Usage Note: The **`nosync`** modifier should be used only when the previous instruction does not use the register that is to be written by **`dpwrt`**. (The instruction executes much faster (11 pipeline stages) if synchronization is not performed, but can overwrite registers used as operands in the previous operation before it reads them from the register file.

Examples:

```
dpwrt *,0f1.0,R3    ! set R3 to 1.0 (single precision)
                    ! on all VUs
dpwrtd  DP_1,%i4,R20,nosync  ! write %i4/%i5 to R20/R21
                             ! on VU 1
                             ! (not synchronized with
                             !  previous instruction)
```

The following code sets up a VU selector in the SPARC register **%i2** in the shifted position, and uses the shifted form of the VU selector register syntax. Note the use of a symbol to specify the bit position:

```
! Following defines DPV_DATA_REGS_DP_NUMBER__POS (12)
#include <cmsys/dp.h>

sll   %i2,DPV_DATA_REGS_DP_NUMBER__POS,%l0
dpwrt %l0<,0f1.3,S4
```

5.3.2 Reading the Data Registers: dprd, dprdd

The **dprd** and **dprdd** instructions read registers from the accelerator register file. Their syntax is similar to that of **dpwrt**:

dprd[**d**] *vu–selector*, *VU–register*, *as–register* [, {**sync**, **nosync**}]

where *vu–selector* and *VU–register* are as in **dpwrt**.

The optional **sync** and **nosync** keywords control whether code is to be emitted prior to the register read to synchronize it with the previous accelerator instructions. The default is **sync**, i.e. to synchronize. If synchronization is not performed, the instruction executes much faster (11 pipeline stages), but can read registers before they are written by as many as 3 previous operations. Use **nosync** only when the three previous instructions do not write the register being read by **dprd**. These keywords can be set in upper or lower, but not mixed, case.

Example:

```
dprd DP_0,S12,%o3  ! read S12 (R12) from VU 0 into %o3
                   ! (synchronized)
```

The following code sets up a VU selector in the SPARC register **%i2** in the shifted position, and uses the shifted form of the VU selector register syntax. Note the use of a symbol to specify the bit position:

```
! Following defines DPV_DATA_REGS_DP_NUMBER__POS (12)
#include <cmsys/dp.h>

sll    %i2,DPV_DATA_REGS_DP_NUMBER__POS,%l0
dprdd  %l0<,S2,%l2
```

5.3.3 Writing the Control Registers: dpset, dpsetd

The **dpset** instruction writes the accelerator control registers. The syntax is:

> **dpset** *vu–selector*, *general–expression*, *ctl–reg–offset* [, **SUPERVISOR**]
> or
> **dpset**[**d**] *vu–selector*, *as–register*, *ctl–reg–offset* [, **SUPERVISOR**]

where

vu–selector indicates which of the accelerator chips on each PN are to be written. By default, all chips are written; supplying a "*****" codes this explicitly.

general–expression is an expression containing the value to be written.

as–register is a SPARC IU register containing the value to be written.

ctl–reg–offset is the (positive) numerical offset of the control register from the base address (that is, from the base of the accelerator register block). It can be given as a constant–expression or as an as–register that contains the offset. As a convenience, the **#include** file "**dp.h**" includes symbolic definitions for the control register offsets. For example, the vector length control register offset is defined as **dp_vector_length**.

The **SUPERVISOR** modifier indicates that **dpas** should use the virtual address at which the supervisor maps the accelerator control register into its address space to do the access. This should only be used in supervisor mode.

Which VUs are written implies the register file base address, which is taken from the external symbol **dpv_ctl_regs_***xxx*.

The **dpsetd** instruction is provided for setting control registers that are 2 words in length, or pairs of control registers. However, the mode value must be specified as an *as–register*, and may not be an immediate value.

Examples:

```
dpset *,7,dp_vector_length   ! set vector length to 8
                             ! (off by one!)
dpset DP_1,%i4,dp_stride_memory   ! set mem stride on
                                  ! VU 1 to the value
                                  ! in %i4
dpset DP_1,%i4,%l3      ! set ctl reg whose offset is
                        ! in %l3 to the value in %i4
```

Note the special consideration used for the **dp_vector_length** control register. The accelerator chip expects this register to be loaded with the vector length minus one.

The following code sets up a VU selector in the SPARC register **%i2** in the shifted position. It uses the shifted form of the VU Selector register syntax. Note the use of a symbol to specify the bit position:

```
! Following defines DPV_CTL_REGS_DP_NUMBER__POS (25)
#include <cmsys/dp.h>

sll   %i2,DPV_CTL_REGS_DP_NUMBER__POS,%l0
dpset %l0<,1234,DP_STRIDE_MEMORY
```

5.3.4 Reading the Control Registers: dpget, dpgetd

The **`dpget`** instruction reads a control register. The syntax is similar to **`dpset`**:

`dpget`[**`d`**][**`s`**]*vu–selector*, *ctl–reg–offset*, *as–register* [, **`SUPERVISOR`**]

where

vu–selector selects the VU from which the control register is read.

ctl–reg–offset is the (positive) numerical offset of the control register from the base address (that is, from the base of the accelerator register block). It can be given as a constant–expression or as an as–register that contains the offset. As a convenience, the **`#include`** file "**`dp.h`**" includes symbolic definitions for the control register offsets. For example, the vector length control register offset is defined as **`dp_vector_length`**.

as–register is a SPARC IU register into which the value is to be read.

The **`SUPERVISOR`** modifier indicates that **`dpas`** should use the virtual address at which the supervisor maps the accelerator control register into its address space to do the access. This should only be used in supervisor mode.

Examples:

```
dpget DP_0,dp_vector_mask,%o3     ! read vector mask
                                  ! from VU 0 into %o3
dpget DP_0,%l4,%o3      ! read ctl reg at offset %l4
                        ! into %o3
```

The following code sets up a VU selector in the SPARC register **%i2** in the shifted position, using the shifted form of the VU Selector register syntax. Note the use of a symbol to specify the bit position:

```
! Following defines DPV_CTL_REGS_DP_NUMBER__POS (25)
#include <cmsys/dp.h>

sll   %i2,DPV_CTL_REGS_DP_NUMBER__POS,%l0
dpget %l0<,DP_STRIDE_MEMORY,%l3
```

5.3.5 Converting Between Data and Instruction Space: dpchgsp

The **dpchgsp** (change space) instruction converts a virtual address from a Parallel Instruction Space address into the corresponding Parallel Data Space address or vice versa. The syntax is:

dpchgsp *inreg*, *outreg*

where both *inreg* and *outreg* are *as–registers*. The conversion is symmetric: if the input is an Instruction Space address, the output is a Data Space address, and vice versa.

For example, given an **%i1** register containing an address that is some offset *n* from **dpv_heap_instruction_port_all**, the base of the virtual region, then

```
dpchgsp %i1,%i2
```

causes **%i2** to be written with the address **dpv_heap_data_all** + *n*.

For details on virtual mapping of parallel data, see Chapter 2.

5.3.6 Selecting VUs: dpchgbk

The **dpchgbk** (change bank) instruction converts a virtual address from either the Parallel Instruction Space or the Parallel Data Space and adjusts it to point to the corresponding address in the memory bank of another VU. The syntax is:

dpchgbk [*inreg*], *vu–selector*, *outreg*

where both *inreg* and *outreg* are as–registers, and *vu–selector* indicates the new VU bank number (see Section 3.15).

For example, if the **%i1** register contains an address which is some offset *n* from the base of the virtual region **dpv_heap_data_all**, then

```
dpchgbk  %i1,DP_3,%i2
```

causes **%i2** to be written with the address **dpv_heap_data_3** + *n*.

If inreg is omitted, the address is constructed from the default instruction space address (the same address used for issuing instructions which do not specify memory operations). This address is the base of the all–VUs Stack Instruction Space (virtual location **DPV_STACK_INST_PORT_ALL**). This is used principally for constructing an instruction space address within one of the instruction space regions associated with a single VU for the purpose of executing an instruction on a particular VU.

Usage Note: While you may omit the first argument, the comma immediately following it is significant, and must *not* be omitted.

The following example shows the execution of a **faddv**/**floadv** instruction on VU 3:

```
dpchgbk ,3,%i0                  ! construct default VU 3
                                !   address in %i0
faddv V1,V2,V3 ; maddr=[%i0]    ! Issue instruction
```

Note the use of the **maddr** modifier for executing an instruction using a particular memory address, when no memory operation is desired.

The **dpchgbk** instruction is used to convert instruction space addresses into corresponding data pointer addresses, as used by the **dpld** and **dpst** instructions, described below.

For details on virtual mapping of parallel data, see Chapter 2.

5.3.7 Reading Parallel Memory: dpld, dpldd

The **dpld** instruction loads a single word from the parallel data space into an IU register. The syntax is:

dpld *as–mem–operand*, *as–register*

where *as–mem–operand* is a SPARC memory address specifier and *as–register* is a register on the IU (floating point registers are allowed). This instruction is used to read data from memory when the memory address might be a parallel data space address. (e.g. **`dpv_stack_data_n`**).

The **`dpldd`** instruction is equivalent, but moves a doubleword of data.

It is never correct to load from or store to the Parallel Instruction Space. Only addresses within single VU banks can be used. Thus, for example, an address within the **`dpv_stack_data_all`** space will cause a Bus Error trap.

It is perfectly legal to use this instruction with addresses not in parallel data space at all, in which case it is equivalent to the SPARC "**`ld`**" instruction.

Examples:

```
dpld  [%i3+4],%l2
dpld  [%i5],%f0
dpldd [%i2],%i4         ! Note both registers are
                        !      doubleword aligned
```

It is common for DPEAC programs to manipulate Parallel Instruction Space addresses rather than Parallel Data Space addresses. Your code can convert from one to the other using the following:

In C:

```
outaddr = inaddr ^ DP_ID_XOR_VALUE;
                /* DP_ID_XOR_VALUE is defined in dp.h */
```

In DPEAC:

```
dpacvt %inreg,%outreg   ! "dpacvt" is described below
```

Note that the conversion is symmetric: the examples above convert from a Parallel Data Space address to a Parallel Instruction Space address or vice versa.

5.3.8 Writing Parallel Memory: dpst, dpstd

The **`dpst`** instruction stores a single word from an IU register to the parallel data space. The syntax is:

`dpst` *as–register*, *as–mem–operand*

where *as–mem–operand* is a SPARC memory address specifier and *as–register* is a register on the IU (floating point registers are allowed). This instruction is used to store data to parallel memory. See the description above of **dpld** for more information on addressing parallel data.

The **dpstd** instruction is equivalent, but moves a doubleword of data.

Examples:

```
dpst  %i2,[%i3+4]
dpst  %f0,[%i5]
dpstd %i4,[%i2]     ! Note both registers are
                    !      doubleword aligned
```

Implementation Note: In the current implementation, **dpld** and **dpst** translate into simple SPARC instructions (**ld** and **st**). However, they are included in DPEAC to allow simulation tools to correctly perform reads and writes of parallel memory. You are strongly urged to use these instructions, rather than SPARC **ld** and **st** instructions for this purpose.

Note For C programmers: The CMCOM runtime system library provides the following routines for performing parallel memory load and store operations:

```
void cmcom_dpld[d](pmem_address, p)
    unsigned pmem_address;
         /* Parallel Data (not instruction!) address */
    unsigned *p;
         /* ptr to word or double word to load into */

void cmcom_dpst[d](pmem_address, p)
    unsigned pmem_address;
         /* Parallel Data (not instruction!) address */
    unsigned *p;
         /* ptr to word or double word to store */
```

5.3.9 Synchronizing the VU Pipelines: dpsynch

The **dpsynch** control instruction generates code to cause the previous and next VU Instructions to be synchronized, that is, to be executed separately with no pipeline overlap. This may be required in cases where the results of the previous instruction are used as inputs to the next instruction. This is also required before **dprd** (or **dprdd**) instructions, as these instructions read the VU data registers and thus access registers which have been written more recently than four operations ago.

The **dpsync** instruction generates a dummy write to the **dp_sync** control register. Thus it is equivalent to:

```
sethi %hi(0xc0008000),%l5    ! All-DP ctl regs base
dpset *,%g0,0x128            ! Write DP_SYNC
```

A description of pipeline overlap in the accelerator chip can be found in Section 9.1.4.

5.3.10 C Versions of VU Accessors

Macros exist that implement the accessors described above, designed for use from the C programming language. These macros can be found in the header file **/usr/include/cmsys/dpc.h**, which is distributed with the CM–5 system software. The C macros defined here are:

CHANGE_DP
TOGGLE_DPV_SPACE

dpst[**d**]	**dpld**[**d**]	**dpstore**[**d**]	**dpload**[**d**]
dpwrt[**d**]	**dprd**[**d**]	**dpset**[**d**]	**dpget**[**d**]

The CHANGE_DP Macro

The **CHANGE_DP** macro takes an instruction space or data space parallel memory address and modifies it to point to a different VU or set of VU's. The syntax is

```
CHANGE_DP(address, vu-selector)
```

The *address* argument is the parallel address to be converted. The *vu–selector* argument indicates the VU, and must be a valid VU selector (e.g., **DP_0**, **ALL_DPS**, etc.)

This macro evaluates into the converted address as if it were a function. This is exactly analogous to the DPEAC instruction, **dpchgbk**.

The TOGGLE_DPV_SPACE Macro

The **`TOGGLE_DPV_SPACE`** macro converts an instruction space parallel memory address to a data space parallel memory address or vice versa. (The operation is its own inverse.) The syntax is

```
TOGGLE_DPV_SPACE(address)
```

The address argument is the parallel address to be converted. The macro evaluates to the converted address as if it were a function. This is exactly analogous to the DPEAC instruction, **`dpchgsp`**.

Consult the section ”Accelerator Chip Memory Mapping” for a discussion of the virtual mapping of parallel memory.

For details on virtual mapping of parallel data, see Chapter 2.

The dpst[d] and dpld[d] Macros

The **`dpst`**[**`d`**] macro moves a word or doubleword from a C variable into VU parallel memory. The **`dpld`**[**`d`**] macro moves data from VU parallel memory into a C variable.

```
void dpst (address, value)
void dpstd (address, dvalue)
unsigned dpld (address)
double dpldd (address)
    void *address;
    void value;
    double dvalue;
```

The *address* argument is the VU parallel memory address. It must be in one of the *data* virtual spaces, never in the *instruction* spaces. The *value* argument is the value to be transferred. For example:

```
/* store 123 to parallel address ”paddr”
    (currently in instruction space)      */

dpst(TOGGLE_DPV_SPACE(paddress), 123);
```

The dpstore[d] and dpload[d] Macros

These macros work identically to **`dpst`**[**`d`**] and **`dpld`**[**`d`**] except that they allow their address argument to be in either *data* or *instruction* virtual space. They also allow specification of the VU to or from which data is being transferred.

Usage Note: These macros are commonly used to accress memory using a parallel memory address that is stored (as the machine address) in a CMRT array descriptor or returned by the CMCOM memory allocation routines.

```
void dpstore (vu_selector, address, value);
void dpstored (vu_selector, address, dvalue);
unsigned dpload (address, vu_selector);
double dploadd (address, vu_selector);

unsigned address;  /* parallel address in data or
                               instruction space  */
unsigned value;
double dvalue;
int vu_selector;
```

The *vu_selector* argument indicates the VU, and must be a valid VU selector (e.g., **`DP_0`**, **`ALL_DPS`**, etc.)

For example:

```
/* store i to parallel address paddress */
dpstore(paddress, i);
```

The dpwrt[d] and dprd[d] Macros

dprd[d] moves a word or doubleword from a data register on a VU to a C variable. dpwrt[d] moves a word or doubleword from a C variable into a data register on one or more VUs.

```
void dpwrt(vu_selector, data_register, value)
void dpwrtd(vu_selector, data_register, dvalue)
unsigned dprd(vu_selector, data_register)
double dprdd(vu_selector, data_register)
    int vu_selector;
    int data_register;
    unsigned value;
    double dvalue;
```

The *vu_selector* argument indicates the VU, and must be a valid VU selector (e.g., **`DP_0`**, **`ALL_DPS`**, etc.) The *data_register* argument is a register number, an integer in the range 0 through 127. The *value* and *dvalue* arguments are the value to be transferred. For example:

```
/* read contents of R5 into i */
i = dprd(DP_2, 5);

/* write double 1.3 into R16/R17 */
double dfloat;
dfloat = 0d1.3;
dpwrtd(ALL_DPS, 16, dfloat);
```

The dpset[d] and dpget[d] Macros

dpget[d] moves a word or doubleword from a VU control register on a VU into a C variable. dpset[d] moves a word or doubleword from a C variable into a control register on one of more VUs.

```
void dpset(vu_selector, ctl_register, value)
void dpsetd(vu_selector, ctl_register, dvalue)
unsigned dpget(vu_selector, ctl_register)
double dpgetd(vu_selector, ctl_register)
    int vu_selector;
    int ctl_register;
    unsigned value;
    double dvalue;
```

The *vu_selector* argument indicates the VU, and must be a valid VU selector (e.g., **`DP_0`**, **`ALL_DPS`**, etc.). The **`ctl_register`** argument is the byte offset into the control register block for the selected register. Symbolic offsets are defined for the control register in the file **`/usr/include/cmsys/dp.h`** and are summarized the the earlier section on the **`dpset`** command. The *value* or *dvalue* argument is a word or doubleword value to be transferred. Doubleword transfers can be tricky; therefore, they are not recommended.

Examples:

```
/*  Set vector len for all VU's to 4 (by writing 3) */
dpset(ALL_DPs, dp_vector_length, 3);

/* Get VECTOR_MASK contents from VU into VMASK */
vmask = dpget(DP_0, dp_vector_mask);
```

5.4 Indirect Execution Of VU Instructions

DPEAC provides two instructions that execute a VU instruction indirectly, from a predetermined memory location. These instructions are as follows:

Instruction	Function
dpasm[**d**]	Assembles a VU instruction for indirect execution.
dpexec[**d**]	Execute an instruction assembled by **dpasm**[**d**].

For an explanation of how to do indirect execution, please see the next chapter.

5.5 Control Instructions

The DPEAC language includes a few basic instructions to help with coding the SPARC IU. These do not generate code that affects the accelerator chip; they are provided as a convenience only.

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.

5.5.1 The dpentry Instruction

The **dpentry** instruction creates a callable routine in DPEAC. The syntax is as follows:

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

where *name* is an *as–symbol* which becomes the name of the routine. (Don't forget the leading underscore when naming routines to be called from C.)

The *argwords* argument is the number of stack words to be reserved for holding arguments (in excess of 6) to routines that will be called by this routine. If this routine calls no routines, or never passes more than 6 arguments to routines it calls, *argwords* can be given as zero. (Remember that doubleword arguments require two words, and so contribute 2 to this count.)

The *localbytes* argument is the number of additional 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 are referenced using negative offsets from the frame pointer, **`%fp`**.

Your DPEAC routine probably will be passed some arguments. Argument passing works as follows:

- The first 6 argument words are passed in the IN registers **`%i0`** through **`%i5`**. The callee may use the **`ARGPUSH`** area on the caller's stack to save these if desired. To do this, the callee references these at **`[%fp+ARGPUSH]`** for argument 1, **`[%fp+ARGPUSH+4]`** for argument 2, etc., through **`[%sp+ARGPUSH+20]`** for argument 6.

  The symbol **`ARGPUSH`** is defined as hexidecimal 68; thus the arguments are saved in **`[%fp+68]`**, **`[%fp+0x72]`**, etc., through **`[%fp+0x88]`**.

- If more than 6 argument words are passed, they are passed on the stack, in the caller's stack frame following the **`ARGPUSH`** area. The callee references these at **`[%fp+MINFRAME]`**, **`[%fp+MINFRAME+4]`**, and so forth. The symbol **`MINFRAME`** is defined to be 92 (0x5C), thus the excess arguments are at **`[%fp+92]`**, **`[%fp+96]`**, etc.

Remember that type **`double`** args are passed as two argument words, and thus occupy two registers or two locations (8 bytes) on the stack. Storage is always *big–endian*, i.e. with the most significant half in the lower numbered register or lower–addressed memory word. Be careful when passing values of type **`double`**, as they may not be aligned in the registers, or on the stack, and may in fact be split with one half in **`%i5`** and the other half on the stack.

Caller's Stack Frame Regions

Stack Location	Usage
`[%fp + ARGPUSH]` (**`ARGPUSH`** = 68)	First 6 arguments
`[%fp + MINFRAME]` (**`MINFRAME`** = 92)	Arguments beyond 6

To reference local stack space reserved by the *localbytes* operand to **`dpentry`**, you use negative offsets from the frame pointer. For example, if *localbytes* is given as 12, thus requiring three local temporary words, the words are referenced at **`[%fp-4]`**, **`[%fp-8]`**, and **`[%fp-12]`**.

The following table maps the resulting stack frame created by **`dpentry`**, listed in order of increasing memory address (i.e. from the stack top toward the stack base). This is the standard stack frame format for the SPARC processor:

Local Stack Frame Regions

Offset	Size (Bytes)	Usage
`[%fp - offset]`	*localbytes*	Local temporaries
`[%sp + MINFRAME]`	(*argwords*–6)*4	For passing arguments 7–*N*
`[%sp + ARGPUSH]`	**`ARGPUSHSIZE`** (24)	Callee's arg. 1–6 save area
`[%sp + WINDOWSIZE]`	4	Structure return address
`[%sp + 0]`	**`WINDOWSIZE`** (64)	O/S Usage (for saving **`IN`** and **`LOCAL`** registers)

The last three regions form the so–called "minimum stack frame", and are required. Together these consist of 92 (**`MINFRAME`**) bytes.

The following are defined in **`/usr/include/sun4/asm_linkage.h`**:

```
WINDOWSIZE = 64 (16 words)
ARGPUSHSIZE = 24 (6 words)
ARGPUSH = WINDOWSIZE + 4 = 68
MINFRAME = WINDOWSIZE + 4 + ARGPUSHSIZE = 92
```

The total frame size in bytes is **`MINFRAME`** (92), plus 4 for each word reserved for passing arguments in excess of 6, plus *localbytes*, all rounded up to a multiple of 8 bytes.

The **`dpentry`** instruction generates the following code:

```
    FRAMESIZEn = framesize
    .seg "text"
    .global name
name:
    sethi %hi(FRAMESIZEn),%g1
    add %g1,%lo(FRAMESIZEnnn),%g1
    save %sp,%g1,%sp
```

Where *nnn* is a unique 3 digit number and *framesize* is calculated as follows:

Round_Up_To_Doubleword(92 + (max(0,(*numargs*–6))*4) + *localbytes*)

Because it expands entrypoint definitions internally, **`dpas`** does not require you to include the file **`asm_linkage.h`**. You only need to include it if you use the stack offset symbols **`MINFRAME`**, **`ARGPUSH`**, etc.

5.5.2 The dpretn Instruction

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

```
ret
restore
```

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

5.5.3 The load[d] Instruction

The **load** instruction loads one of the SPARC IU registers with a constant value. **loadd** loads an aligned pair of registers with a 64–bit constant value. These are provided as a convenience, since the SPARC instructions for loading registers with constants are cumbersome, especially when loading larger or floating point values. The syntax is:

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

The most common usage is for putting floating point constants into SPARC IU registers. The following example shows **loadd** being used to put the double precision constant 1.3 into **%o0** and **%o1**. The example also clearly shows why these instructions are useful:

```
loadd 0d1.3,%o2
```

generates the following SPARC code:

```
sethi   %hi(0x3ff4cccc),%o3
or      %o3,%lo(0x3ff4cccc),%o3
sethi   %hi(0xcccccccd),%o2
or      %o2,%lo(0xcccccccd),%o2
```

5.5.4 SPARC Register Usage and Control

On the SPARC, **`dpas`** must reserve the use of four registers for the code it emits as follows:

Register	Usage
`%l6`	(Reserved) Pointer to Parallel Instruction Space (all VUs)
`%l7`	(Reserved) Pointer to **`dp_instruction_ext`** (VU–selector=0)
`%g2`	(Temporary) Used to buffer VU Instruction
`%g3`	(Temporary) Used to buffer VU Instruction

The two ”reserved” registers, **`%l6`** and **`%l7`**, are set up with specific contents by **`dpas`**, which expects them to be preserved throughout the DPEAC routine. However, **`dpas`** will assume that the registers are no longer correct if a **`.seg`** directive is seen, or if the **`dpunset`** directive (below) is encountered.

The two ”temp” registers are overwritten by the code that **`dpas`** generates to execute VU instructions, and so are not preserved by **`dpas`**. The use of these registers as temporaries is allowed within code lacking VU instructions.

No other registers are read or written by **`dpas`**.

5.5.5 The dpunset Instruction

The **`dpunset`** instruction signals **`dpas`** that one or both of the reserved registers may have been overwritten. This information is used by **`dpas`** to reinitialize the registers when necessary. It is never necessary to preserve, or declare as overwritten, the two temp registers.

5.5.6 The dpregs Instruction

The **`dpregs`** control instruction allows you to override the default SPARC register usage by **`dpas`**. The syntax is:

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

The arguments refer to the register that **`dpas`** uses to do the following:

Argument	Register Usage
MaddrReg	Default Maddr Base Register Register containing the address **`DPV_STACK_INSTR_PORT_ALL`**, used for executing instructions that do not give a memory address.
InstExtPtrReg	Instruction Extension Pointer Register Register containing 0xC0000176, a value used to quickly compute pointers to the **`dp_instruction_ext`** control register.
TempRegs	Aligned pair of registers used as temps to execute VU instructions.

Each argument is one of:

Argument	Register Usage
as–register	Use this reg (and mark it as not currently set up)
as–register`=`	Use this register and generate code now to set it up
`=`	Generate code to set up the register in effect
`–`	Tell the code generator not to use any register
{blank}	Do not change the register in effect (NOP)

The default at the beginning of an assembly is:

```
dpregs %l6,%l7,%g2
```

i.e. the registers are so declared, but marked as not set up yet.

A DPENTRY instruction implies the following code:

```
dpregs =,=,
```

that is, the instruction sets up the Default Maddr Base and the Instruction Extension Pointer registers.

The Maddr Register and Instruction Extension Pointer Register can be totally turned off using the "`–`" syntax. If the Maddr Register is turned off, subsequent VU instructions that do not specify a memory address, either in a Memory Operation or via the MADDR modifier, will be flagged with an error.

If the Instruction Extension Pointer Register is turned off, **dpas** will generate code two cycles slower in the case where an long format instruction performs a memory operation to a possibly non–doubleword–aligned memory address (one for which neither the the **align** modifier nor the **–a** switch to **dpas** were given).

The "**=**" syntax (with or without a register) causes code to be generated to initialize the associated register to its correct value. The temps cannot be initialized, as they are used as temporaries only.

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 being no longer set up with the proper value. This forces subsequent code (i.e. later in the assembly, not later in execution) that requires that register to set it up. This can be used when a register has (possibly) been overwritten, to declare that **dpas** should not assume its contents.

TempRegs must be an even (aligned) register, and actually specifies that both registers of the pair can be used as temporaries. Thus, for example, specifying **%g4** indicates that both **%g4** and **%g5** can be used as temps.