CDPEAC Quick–Reference

CDPEAC: CM-5 Vector Unit Programming in C

This document describes the CDPEAC instruction set, used for writing C programs that access the CM–5's Vector Unit (VU) accelerators.

Note: This is a ***preliminary version*** of a forthcoming document on CDPEAC. Please send any comments and/or corrections to: `traveler@think.com`

Syntax Conventions Used In This Document:

`{a,b...}`	=	Selection; you *must* choose a or b or...
`[x]`	=	Optional part; you *may* include x
`bold`	=	Indicates opcode or suffix that can be added to opcode
`register`		(Also used to indicate register names.)
name	=	Metavariable; replaced by a value or symbol (typically indicated by a list of valid replacements)

1 CDPEAC Syntax

A CDPEAC program consists of C code with embedded CDPEAC statements. These statements are expanded during compilation into code that controls the CM-5's Vector Units.

A CDPEAC statement is one of the following:

- a *VU Instruction*
- a *VU Accessor Instruction*
- a *VU Special Instruction*

A VU Instruction corresponds to a scalar or vector operation performed by the Vector Units, and is either:

- a **VU Arithmetic operator**, which performs an ALU operation:

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

- a **VU Memory operator**, which performs a memory load or store:

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

- a **VU Statement Modifier**, which affects statement compilation:

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

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

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

A VU Accessor Instruction is an instruction that executes on the CM-5 node microprocessor (the SPARC), but modifies the contents of VU registers or parallel memory:

```
dpwrt(i,ALL_DPS,sp_src,R0) /* Write VU data register */
dpget(i,DP_1,dp_stride_memory) /* Get memory stride */
```

A VU Special Instruction is an instruction not in either of the above two classes, which 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 */
```

1.1 The Join Operator

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*.

1.2 Registers

VU Data Registers: CDPEAC code generally refers to *VU data registers*. The 128 VU data registers are referenced by the following symbolic 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)

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

Scalar Registers: Scalar VU operations only accept the *scalar registers*. These are `S0 – S15` (single word), or the even registers from `S0 – S30` (double word). Scalar operations restrict their operands to the `S`*nn* (or equivalent `R`*nn*) registers.

Register Restrictions: The `R0` and `R1` registers are used to store immediate operands, so these registers should be used carefully.

Register Offsets: You can use an offset to a data register to access it and those succeeding it in `R`*nn* order as a vector (usually to access `V`*nn* elements). (See the `dreg_x` register modifier in Section 1.3 below.)

Internal Registers: There are some VU internal registers that influence the execution of DPEAC instructions. Some important examples are:

`dp_stride_rs1`	Stride of `src1` operand in arithmetic instruction.
`dp_stride_memory`	Stride of memory addresses in memory instruction.
`dp_vector_mask`	Context mask for vectored arithmetic operations.
`dp_vector_mask_mode`	Default vector conditionalization (masking) mode.
`dp_vector_length`	Default vector length for both types of instructions.
`dp_vector_mask_buffer`	Copy of `dp_vector_mask` used to save/restore it.

Important: The pair of VUs on a single chip (that is, VUs 0/1 and 2/3) actually share all these internal registers except for the two registers `dp_vector_mask` and `dp_vector_mask_buffer`. This means that any change to a shared register affects *both* VUs that share it.

1.3 Register Modifiers

These modifiers can be applied to any register argument in a CDPEAC operation to specify an offset, stride, or indirection for the register.

Register offsets:

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

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

Register striding: (Note: Unit stride is 1 for singles, 2 for doubles)

dreg	With no modifier, use unit striding
dreg_u(dreg,stride)	Use given **stride** once
scalar(dreg)	Scalar striding, same as **dreg_u**(dreg,0)
SCALAR(dreg)	Alternate name for **scalar**(dreg)

Src1 register striding: (Note: Default **src1** stride is **dp_stride_rs1**)

dreg_u(dreg,**mode**)	Use default stride (**mode** is a literal symbol)
dreg_s(dreg,stride)	Store stride as the **src1** default and use it
dreg_u_s(dreg,stride,set_stride)	Use **stride**, and store **set_stride** as default

Register indirection:

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

1.4 Common Abbreviations

Common CDPEAC opcode suffixes:

Type:	Meaning:	
`s`	Scalar operation — single elemental operation on given arguments	
`v`	Vector operation — multiple elemental operation with striding	
`_i`	Memory stride indirection	(for memory operations)
	Immediate value in `src2` argument	(for arithmetic operations)
`_v`	Use explicit vector length	(unsticky, vlen = constant or register)
`_vs`	Use and set vector length	(sticky, vlen = constant or register)
`_vh`	Vlen from register field	(unsticky, 1+(bits `19:22` of reg))
`_vhs`	Vlen from register field	(sticky, 1+(bits `19:22` of reg))

CDPEAC Operand type symbols:

Type:	Meaning:
`u`	Unsigned single-precision (32 bit) integer
`du`	Unsigned double-precision (64 bit) integer
`i`	Signed single-precision (32 bit) integer
`di`	Signed double-precision (64 bit) integer
`f`	Single-precision (32 bit) float
`df`	Double-precision (64 bit) float

1.5 Typical CDPEAC Operand Names

`address`	—	VU memory address
`type`	—	CDPEAC operation type
`src,src<n>`	—	source VU data registers (or immediate values)
`dest`	—	destination VU data register
`sp_src`	—	SPARC source register
`sp_dest`	—	SPARC destination register
`dreg`	—	VU data register
`ireg`	—	data register being used for indirection
`creg`	—	VU control register

2 CDPEAC Arithmetic Instructions

2.1 Monadic (One Source) Operators

These operators perform an arithmetic operation on the single **src** argument, and store the result in the **dest** argument.

Formats:

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

Opcodes:	Types:	Purpose:
move	{u, du, i, di, f, df}	Move **src** to **dest**, no status generated
test	{u, du, i, di, f, df}	Move **src** to **dest** and test
not	{u, du}	Bitwise invert (**dest** = ~**src**)
clas	{f, df}	Classify operand (**dest** = class of **src**)
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 (**dest** = 0 - **src**)
abs	{i, di, f, df}	Absolute value (**dest** = |**src**|)
inv	{f, df}	Invert (**dest** = 1/**src**)
sqrt	{f, df}	Square root (**dest** = sqrt(**src**))
isqt	{f, df}	Inverse root (**dest** = 1/sqrt(**src**))

2.1.1 Convert Operator (Monadic with extra type argument)

The **to** operator converts between data types (**src** is of *type1*, **dest** of *type2*).

Format:

opcode{s,v}[i](*type1*,*type2*[r],src,dest)
opcode{s,v}_{v,vs,vh,vhs}(*type1*,*type2*[r],vlen,src,dest)
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)

2.1.2 Dyadic (Two Source) Operators:

These operators perform an arithmetic operation on the **src1** and **src2** arguments, and store the result in the **dest** argument.

Formats:

opcode{**s**,**v**}[**i**](*type*,**src1**,**src2**,**dest**)
opcode{**s**,**v**}_{**v**,**vs**,**vh**,**vhs**}(*type*,**vlen**,**src1**,**src2**,**dest**)
type = {**u**, **du**, **i**, **di**, **f**, **df**}

Opcodes:	Types:	Purpose:
add	{**u**, **du**, **i**, **di**, **f**, **df**}	Add (**dest** = **src1** + **src2**)
addc	{**u**, **du**, **i**, **di**}	Integer add with carry
sub	{**u**, **du**, **i**, **di**, **f**, **df**}	Subtract (**dest** = **src1** – **src2**)
subc	{**u**, **du**, **i**, **di**}	Integer subtract with carry
subr	{**u**, **du**, **i**, **di**, **f**, **df**}	Subtract reversed (**dest** = **src2** – **src1**)
sbrc	{**u**, **du**, **i**, **di**}	Integer subtract reversed with carry
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 (**dest** = **src1** / **src2**)
enc	{**u**, **du**}	Make float from exp and mant (**src1**, **src2**)
shl	{**u**, **du**}	Shift left (**dest** = **src1** << **src2**)
shlr	{**u**, **du**}	Shift left reversed (**dest** = **src2** << **src1**)
shr	{**u**, **du**, **i**, **di**}	Shift right (**dest** = **src1** >> **src2**)
shrr	{**u**, **du**, **i**, **di**}	Shift right reversed (**dest** = **src2** >> **src1**)
and	{**u**, **du**}	Bitwise logical **AND**
nand	{**u**, **du**}	Bitwise logical **NAND**
andc	{**u**, **du**}	Bitwise logical **AND**, **src1** complemented
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 **src1** else **src2**

2.1.3 Arithmetic Comparisons:

These operators perform an arithmetic comparison between the **src1** and **src2** arguments, and set status flags accordingly.

Format:

opcode{**s**,**v**}[**i**] (*type*,src1,src2)
opcode{**s**,**v**}_{**v**,**vs**,**vh**,**vhs**} (*type*,vlen,src1,src2)
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

2.1.4 Compare (Dyadic with Rd constant)

The Compare operation tests for a numeric relationship between the **src1** and **src2** arguments, as indicated by the supplied constant **code**.

Format:

opcode{**s**,**v**}[**i**] (*type*,src1,src2,code)
opcode{**s**,**v**}_{**v**,**vs**,**vh**,**vhs**} (*type*,vlen,src1,src2,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

2.1.5 Dyadic Mult-Op Operators

These operations perform a muliplication and an arithmetic (or logical) operation on the **src1**, **src2**, and **dest** arguments, and store the result in **dest**.

Format:

opcode{**s**,**v**}[**i**](*type*,src1,src2,dest)
opcode{**s**,**v**}_{**v**,**vs**,**vh**,**vhs**}(*type*,vlen,src1,src2,dest)
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}	**dest** = (**src1** * **src2**) + **dest**
msba	{u, du, i, di, f, df}	**dest** = (**src1** * **src2**) – **dest**
msra	{u, du, i, di, f, df}	**dest** = **dest** – (**src1** * **src2**)
nmaa	{u, du, i, di, f, df}	**dest** = –**dest** – (**src1** * **src2**)
m[h]sa	{du}	**dest** = (**src1** * **src2**) AND **dest**
m[h]ma	{du}	**dest** = (**src1** * **src2**) AND NOT **dest**
m[h]oa	{du}	**dest** = (**src1** * **src2**) IOR **dest**
m[h]xa	{du}	**dest** = (**src1** * **src2**) XOR **dest**

Inverted Operators

Opcodes:	Types:	Purpose:
madi	{u, du, i, di, f, df}	**dest** = (**src2** * **dest**) + **src1**
msbi	{u, du, i, di, f, df}	**dest** = (**src2** * **dest**) – **src1**
msri	{u, du, i, di, f, df}	**dest** = **src1** – (**src2** * **dest**)
nmai	{u, du, i, di, f, df}	**dest** = –**src1** – (**src2** * **dest**)
m[h]si	{du}	**dest** = (**src2** * **dest**) AND **src1**
m[h]mi	{du}	**dest** = (**src2** * **dest**) AND NOT **src1**
m[h]oi	{du}	**dest** = (**src2** * **dest**) IOR **src1**
m[h]xi	{du}	**dest** = (**src2** * **dest**) XOR **src1**

2.1.6 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 `dest` argument.

Format:

opcode`{s,v}[i]` (*type*`,src,code,dest`)
opcode`{s,v}_{v,vs,vh,vhs}` (*type*`,vlen,src,code,dest`)
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

2.1.7 True Triadic (Three Source) Operators

These operations perform a muliplication and an arithmetic (or logical) operation on the **src1**, **src2**, and **src3** arguments, and store the result in **dest**.

Format:

opcode{**s,v**}[**i**] (*type*,**src1,src2,src3,dest**)
opcode{**s,v**}_{**v,vs,vh,vhs**} (*type*,**vlen,src1,src2,src3,dest**)
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**}	**dest** = (**src1** * **src2**) + **src3**
msbt	{**u, du, i, di, f, df**}	**dest** = (**src1** * **src2**) – **src3**
msrt	{**u, du, i, di, f, df**}	**dest** = **src3** – (**src1** * **src2**)
nmat	{**u, du, i, di, f, df**}	**dest** = –**src3** – (**src1** * **src2**)
m[h]st	{**du**}	**dest** = (**src1** * **src2**) AND **src3**
m[h]mt	{**du**}	**dest** = (**src1** * **src2**) AND NOT **src3**
m[h]ot	{**du**}	**dest** = (**src1** * **src2**) IOR **src3**
m[h]xt	{**du**}	**dest** = (**src1** * **src2**) XOR **src3**

Important:
When a triadic operators is joined with a memory operator, the **src2** argument of the triadic *must* be identical to the **dreg** argument of the memory operator. (This restriction is imposed by the way such statements are assembled.)

2.1.8 No–op Operator

The untyped arithmetic no–op allows modifier side–effects without specifying an operation. The no-op takes no arguments. The suffixes are as described above.

Format:

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

3 CDPEAC Memory Operations

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 stride

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)
```

4 CDPEAC Statement Modifiers

This section describes the statement modifiers that can be **joined** 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.

General Modifiers:

nopad, pad(n)	Vector length padding (n = new length, default is 4)
maddr(address)	Memory address for statement lacking memory load/store
[**no**]**align**	Doubleword alignment guarantee on memory operand

Vector Mask Modifiers:

vmmode[**_s**](mode) Vector mask conditionalization mode
(**_s** version sets value of **dp_vector_mask_mode**)

Mode:	Meaning:
vmmode	Use default vector mask mode (**dp_vector_mask_mode**)
cond	Full conditionalization
condalu	Arithmetic operation only
condmem	Memory operation only
always	No conditionalization

vmrotate, vmcurrent	Vector mask bit rotation
vminvert, vmtrue	Vector mask bit sense
vmold, vmnew, vmnop	Vector mask copy mode

Accumulated Context Count:

vmcount[{**v,s**}](**dreg**) Set **dreg** to count of 1's in vector mask

Note: **_s** version is for scalar ops, **_v** for vector ops. (**_v** is the default.)

VU Pair Data Exchange:

exchange, noexchange Arithmetic results exchanged by pairs of VUs on the same chip

Population Count:

epc{**s,v**} (*type*, **src**, **dest**) Counts 1 bits in src, stores total in dest
type = {**u, du**}

5 VU 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.

Data Register Read/Write Operators: These move data between SPARC Registers and VU Data Registers:

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

Control Register Read/Write Operators: These move values between SPARC Registers and VU Control Registers:

```
dpset[_supervisor](type,selector,sp_src,creg)
dpget[_supervisor](type,selector,creg,sp_dest)
   type = {u, du, i, di, f, df}
   supervisor = get/set in supervisor region
```

Parallel Memory Load/Store Operators: These move values between SPARC registers and VU parallel memory:

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

Memory Space/Bank Conversions: These operators modify the memory address in the **src** register to point to a different space/bank of VU memory, and store the modified address in **dest**.

```
dpchgsp(src,dest)            Toggle between data/instruction spaces
dpchgbk(src,selector,dest)   Change referenced VU region
```

VU Pipeline Sync: This operator prevents the preceding and following CDPEAC statements from overlapping in the VU pipeline:

```
dpsync()
```

CDPEAC Function Setup/Cleanup:

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

6 VU 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 **src1** 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)

7 Special Notes and Restrictions

Register Stride Restrictions:

When you apply a stride of 0 to the `src1` argument of an arithmetic operation (for example, `dreg_u(R0,0)`), the `src1` register must be one of the scalar registers `S0` through `S15` (or `S30` for double precision).

Src2 Operand Restrictions:

The `src2` operand of an arithmetic instruction has the following restrictions:

- For vector operations, `src2` cannot be any of `R0` through `R7`, by any name (`S0`, `V0`, etc.).
- In scalar operations, `src2` cannot be any of `Rnn`, where *nn* is any multiple of `16` (for single-precision) or `32` (for double-precision).

(**For the Curious:** This restriction is imposed by the way CDPEAC operations are represented internally.)

Triadic Operator Restrictions:

When a triadic arithmetic operation and a memory operation are `joined`, the src2 operand of the arithmetic operation must be identical to the dreg operand of the memory operation.

Double Precision Move Immediate:

Double-precision move operations only use the upper 32 bits of an immediate source operand. Thus, operands with any non-zero bits in the lower 32 bits cannot be specified.