Chapter 2

The CM-5 Vector Units

2.1 CM-5 Vector Unit Accelerators

The vector unit (VU) accelerators are located between the SPARC CPU and the memory banks of the processing node (see Figure 4).

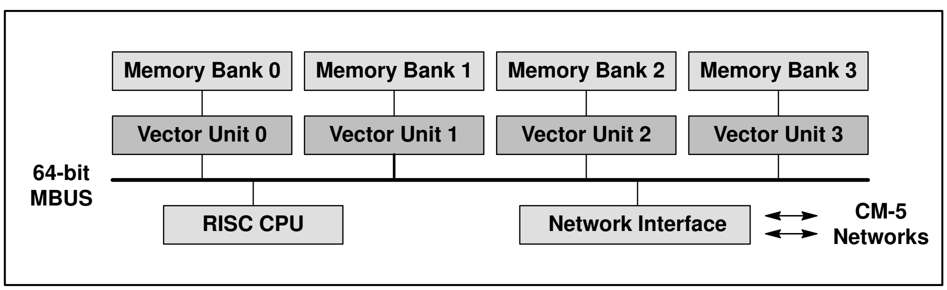

Figure 4. A typical CM-5 processing node, showing the location of the 4 VUs.

The VUs act as memory controllers, handling memory store and fetch operations as required by the SPARC. However, some memory operations are interpreted as instructions by the VUs: the value written is interpreted as a VU arithmetic and/or memory instruction, and the address to which it is written determines which of the four VUs on the node will execute the instruction.

VU instructions can be *strided*, or made to operate step-wise across many memory or VU register locations; hence the term "vector unit" for the accelerator hardware. (This striding is specified either by explicit instructions, or by a default value stored in a VU control register.)

2.1.1 Vector Unit Hardware

There are actually only two VU chips in each processing node; each chip contains the hardware necessary to simulate the operations of a pair of VUs. Thus, VU instructions that select groups of VUs can only select them as follows: one VU alone, all VUs at once, or fixed pairs (0/1 or 2/3).

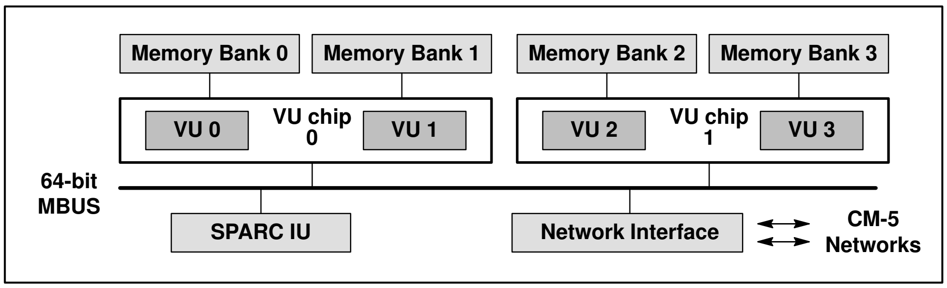

Figure 5. Internal arrangement of VU chips in CM-5 processing node.

For the Curious: The VU chips operate at approximately 32 MHz, while the memory chips operate at 16MHz. Thus, each VU chip performs two memory operations per cycle, one for each of the two attached memory chips.

2.1.2 VU Virtual Memory Layout

Each vector unit instruction can be performed either by a single VU, or by two or four of the VUs operating in parallel (this parallel operation typically provides the best performance).

The vector units that perform a given VU instruction are selected by the VU memory address to which the instruction is written. There is a set of virtual addresses for each VU and permitted combination of VUs (see Figure 6).

These VU memory regions all correspond to the same physical memory region, but each VU region selects a different VU or set of VUs to execute a DPEAC statement. (There are also memory regions devoted to ordinary SPARC serial memory references, which don't trigger VU operations.)

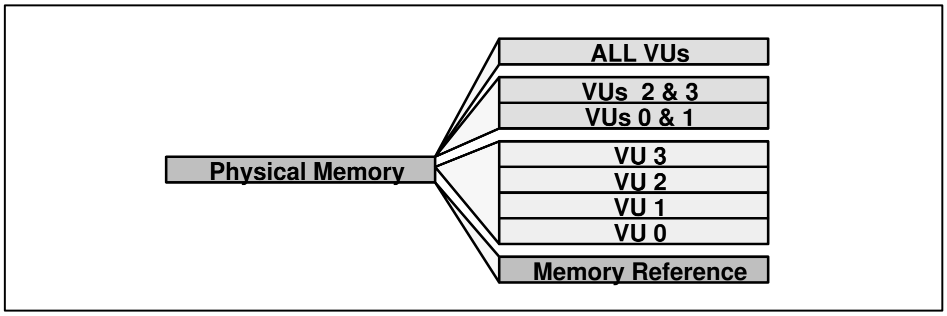

Figure 6. VU virtual memory regions.

Each of the VU regions includes two separate address spaces, called *data space* and *instruction space*, that refer to the same physical VU memory, but have different effects on the VUs (see Figure 7). Data space addresses allow the SPARC to perform normal load/store operations on VU parallel memory. Instruction space addresses cause the VU(s) to perform an operation using the instruction space address as the memory operand.

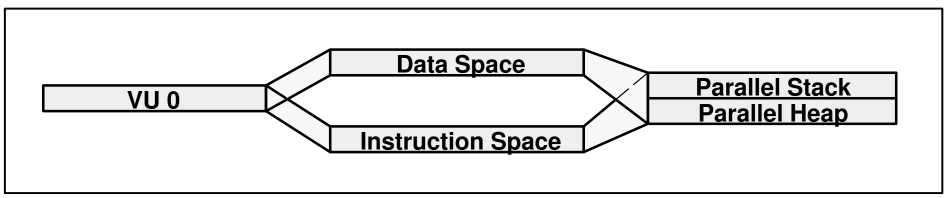

Figure 7. Contents of a single VU memory region.

The instruction and data spaces of each VU virtual memory region refer to a single physical memory region that includes a *parallel stack* and a *parallel heap*. The stack and heap are "striped" across the VU memory banks in such a way that they occupy the same locations in each VU memory region.

Note: The diagrams above are a simplification. Refer to Appendix A for a more detailed description of the way the vector units are mapped into the physical and virtual memory of the SPARC CPU.

2.2 VU Registers

Each VU has an internal set of registers, along with hardware for both memory accessing and register arithmetic. Thus, a single VU operation can involve a memory operation, a register arithmetic operation, or both at once.

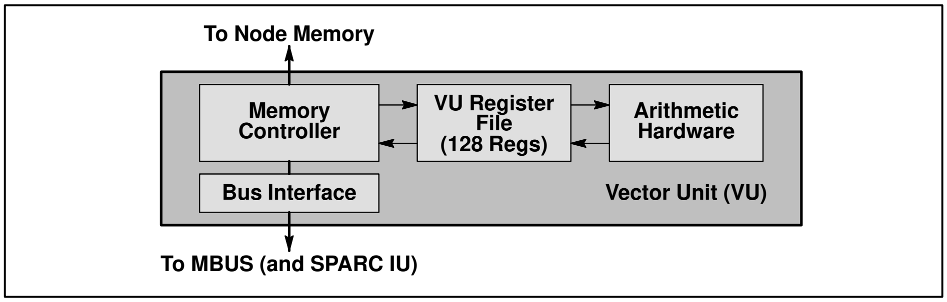

Figure 8. VU internal components.

2.2.1 VU Data Registers

Each VU has a *register file* containing 128 *data registers*, each 32 bits long, which are used as operands for arithmetic and memory operations. These registers are typically addressed as *vectors*, that is, blocks of registers that are either adjacent to each other or are located a constant distance (or *stride*) apart.

Depending on the data type in use, the data registers may be accessed individually as singleword (32-bit) values, or in pairs as doubleword (64-bit) values. The typical way to view these data registers is as 16 vectors of 8 elements:

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 9. VU data registers: 16 vectors of 8 registers.

2.2.2 VU Control Registers

Each VU also has internal *control registers* that affect VU instruction execution.

VU Vector Mask Registers:

`dp_vector_mask` — Vector mask register:
Source of context bits (see below) and storage for arithmetic status bits.

`dp_vector_mask_direction` — Vector mask shift direction:
One-bit register, 0 means shift right (towards LSB), 1 means shift left

`dp_vector_mask_buffer` — Vector mask copy buffer:
Copy of vector mask register loaded or stored prior to each operation.

`dp_vector_mask_mode` — Default vector mask conditionalization mode:
Indicates which of ALU and memory instructions are conditionalized.

VU Arithmetic Status Registers:

`dp_status` — Status register:
Holds status bits produced by arithmetic operations.

`dp_status_enable` — Status enable register:
Selects status bits that are ORed and stored in vector mask register.

C/DPEAC Instruction Default Registers:

`dp_vector_length` — Vector length register:
Default length of vectors (number of steps) for vector operations.

`dp_stride_rs1` — *Rs1* register operand stride:
Default stride for *Rs1* operand in arithmetic operations.

`dp_stride_memory` — Memory operand stride:
Default stride for memory operations.

`dp_alu_mode` — Arithmetic mode register:
Selects Fast or IEEE mode for arithmetic operations.

Important: The pair of VUs on a single chip (that is, VUs 0/1 and 2/3) actually share all control 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.

2.3 Effects of VU Control Registers

The VU control registers are used for a number of purposes:

- Conditionalization of VU instructions (described in Section 2.3.1).
- Contextualization, or collection of status bits (described in Section 2.3.2).
- Default registers for DPEAC and CDPEAC operators. (These are are described in the chapters on DPEAC and CDPEAC, along with the instructions that use and modify these registers.)

Figure 10 summarizes the effects of the control registers (and some instruction modifiers) on the ALU and memory components of VU instructions:

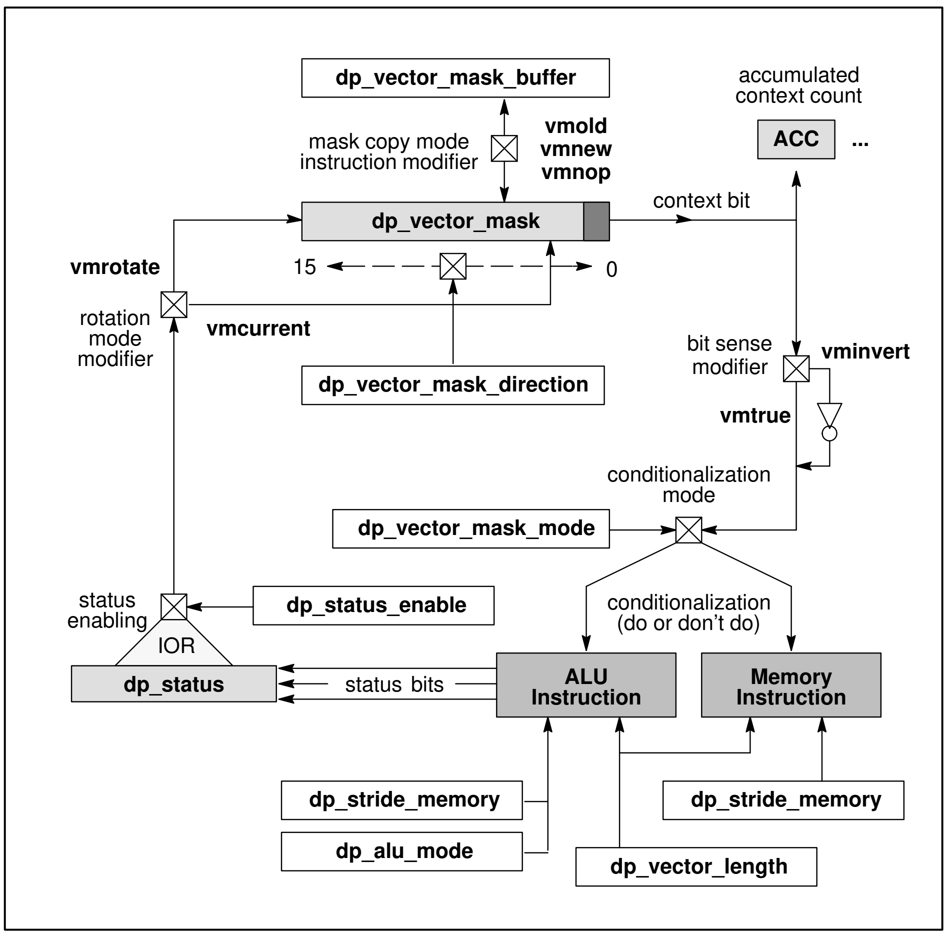

Figure 10. Effects of VU control registers on ALU and memory instructions.

2.3.1 Vector Mask and Conditionalization

The VUs have the ability to *conditionalize* vector operations. The vector mask register (**`dp_vector_mask`**) is used to mask individual ALU and memory instructions in a vector operation. At each step of a vector operation, a *context bit* is shifted out of the mask register (see Figure 11). This bit can be used to mask out (prevent) the ALU operation, the memory operation, both operations, or neither of them. (**Note:** In the current implementation, **`dp_vector_mask`** is a 32-bit register, but only the least significant 15 bits are used.)

By default, the vector mask mode register (**`dp_vector_mask_mode`**) determines which, if any, of the ALU and/or memory operations are conditionalized. Initially, the mode register is set so that no conditionalization is done. A 0 context bit masks the corresponding ALU and/or memory operation, preventing the results from being stored in the destination register. A 1 context bit allows the results to be written. (**Note:** Scalar operations are *never* conditionalized.)

C/DPEAC instruction modifiers let you override the mode register and/or change its value while executing an instruction. There is also a C/DPEAC instruction modifier that allows you to *invert* the sense of the context bit, so that a 1 bit masks the operation, and a 0 bit allows the operation to proceed.

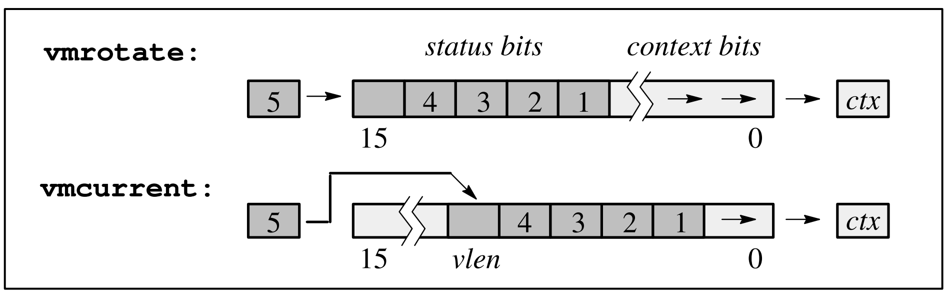

Figure 11. Bit-shifting modes of vector mask register.

2.3.2 ALU Status and Contextualization

Every ALU operation sets the flags in the status register (**`dp_status`**) to indicate the results of the operation. There is a similar set of flags in the status enable register (**`dp_status_enable`**), indicating which of the **`dp_status`** flags are ORed together to make the status bit of a vector operation.

At each step of a vector operation, the ALU sets the flags in **`dp_status`**, and then the flags selected by 1 bits in **`dp_status_enable`** are ORed together into a single *status bit*, indicating whether or not the ALU operation completed successfully. This status bit is shifted into the vector mask register.

Status bits are typically rotated into the vector mask register at the end opposite to that from which condition bits are drawn (this is known as *rotate* mode.) However, a C/DPEAC instruction modifier can cause the context bits to be inserted into the vector mask register in numerical order at the same end from which they are drawn (*current* mode). See Figure 11 above.

The vector mask shift direction register, **`dp_vector_mask_direction`**, determines which way the vector mask bits are shifted. If it is 0, the default, the bits are shifted right (toward the low end of the register, as shown in Figure 11). If the mask direction is 1, bits are shifted left (toward the high end).

2.3.3 Status Register Flags

The current flags in the **`dp_status`** and **`dp_status_enable`** registers, together with their symbolic names as defined by the C/DPEAC header files, are shown in the table below. (Starred status flags are the IEEE-defined exceptions.)

Bit	Flag Mask Symbol	Status
0	**DP_STATUS_ENABLE_MASK_INEXACT**	Float result is inexact (*)
1	**DP_STATUS_ENABLE_MASK_DIVIDE_BY_ZERO**	Division by zero (*)
2	**DP_STATUS_ENABLE_MASK_UNDERFLOW**	Float underflow (*)
3	**DP_STATUS_ENABLE_MASK_OVERFLOW**	Float overflow (*)
4	**DP_STATUS_ENABLE_MASK_INVALID_OPERATION**	Invalid operation (*)
5	**DP_STATUS_ENABLE_MASK_INT_OVERFLOW**	Integer overflow
6	**DP_STATUS_ENABLE_MASK_NEGATIVE_UNSIGNED**	Negative integer result
7	**DP_STATUS_ENABLE_MASK_DENORM_INPUT**	Float input denormalized
8	**DP_STATUS_ENABLE_MASK_ZERO**	Float/integer result of zero
9	**DP_STATUS_ENABLE_MASK_POSITIVE**	Float/integer result positive
10	**DP_STATUS_ENABLE_MASK_NEGATIVE**	Float/integer result negative
11	**DP_STATUS_ENABLE_MASK_INTEGER_CARRY**	Integer carry
12	**DP_STATUS_ENABLE_MASK_INFINITY**	Float result is +/– infinity
13	**DP_STATUS_ENABLE_MASK_NAN**	Float result is a NaN
14	**DP_STATUS_ENABLE_MASK_DENORM**	Float result is denormal
15	**DP_STATUS_ENABLE_MASK_UNORDERED**	(Internal, do not use)
16	**DP_STATUS_ENABLE_MASK_UNDER**	(Internal, do not use)
17	**DP_STATUS_ENABLE_MASK_DENO**	(Internal, do not use)

See Appendix D for a more detailed description of the meanings of the status flags, and for descriptions of the VU arithmetic operations that modify them.

2.3.4 The Vector Mask Buffer

Prior to each DPEAC operation, the contents of the vector mask register may be stored to, or copied from, the vector mask buffer register (**dp_vector_mask_buffer**). By default, no such copying is done. The vector mask buffer can be useful, for example, for keeping a fixed vector mask handy so that it can be copied into the mask register before each DPEAC operation.

A C/DPEAC instruction modifier allows you to override the value of this register for a given instruction, or modify its value to affect future instructions.

2.4 Other VU Features

2.4.1 Accumulated Context Count

The C/DPEAC format modifier **vmcount** causes the individual context bits shifted out of the vector mask register to be stored in a series of VU data registers. This accumulated context count feature can be useful for determining which instructions in a VU operation were masked out. For more information, see the discussion of the **vmcount** modifier (Section 4.3 for DPEAC, and Section 6.7 for CDPEAC).

2.4.2 Population Count

The C/DPEAC format modifier [**d**]**epc** causes the vector units to do a *population count*, or count of the number of 1 bits, on a register. This is a strided operation, and acts like a memory instruction in a VU operation. (Section 4.3 for DPEAC, and Section 6.7 for CDPEAC).

2.5 VU Control Register Constants

The following constants are defined by the C/DPEAC header files, giving the offsets of the VU control registers.

Register	Register Constant	Current Value
`dp_stride_rs1`	`DP_STRIDE_RS1`	0x10C
`dp_stride_memory`	`DP_STRIDE_MEMORY`	0x108
`dp_vector_length`	`DP_VECTOR_LENGTH`	0x104
`dp_alu_mode`	`DP_ALU_MODE`	0x100
`dp_status`	`DP_STATUS`	0x124
`dp_status_enable`	`DP_STATUS_ENABLE`	0x120
`dp_vector_mask`	`DP_VECTOR_MASK`	0x110
`dp_vector_mask_direction`	`DP_VECTOR_MASK_DIRECTION`	0x11C
`dp_vector_mask_buffer`	`DP_VECTOR_MASK_BUFFER`	0x114
`dp_vector_mask_mode`	`DP_VECTOR_MASK_MODE`	0x118

Note: These offsets are for use only with accessor instructions such as **`dpset`** and **`dpget`**. C/DPEAC statement formats also allow you to implictly use and/or set the value of one or more control registers while executing a VU operation. See the mode set format in particular (Section 3.9 for DPEAC, Section 5.9 for CDPEAC) for examples.