Appendix A

VU Memory Mapping

This appendix describes in more detail the relationship between the physical and virtual memory mappings of the CM-5 vector units. **Note:** The diagrams shown here are a simplification of the detailed memory maps provided in Appendix B.

A.1 VU Physical Memory Mapping

The SPARC IU's physical memory is divided up into memory regions, one for each possible VU grouping. The memory regions are located at physical address *N*`00000000` hex, where *N* is one of:

Memory Region (*N*)	**Purpose**
`0-3`	VU 0-3 memory and data regs (read/write).
`8`	All VUs (write only).
`9`	VUs 0 and 1 (write only).
`B`	VUs 2 and 3 (write only).
`F`	VU control registers and ROM.

Within each of the VU memory regions (with the exception of the control register region, described separately below) there are three subdivisions, indicated by the second hex digit of the physical address:

Physical Address (hex)	**Purpose**
N`80`*mmmmmm*	Instruction memory space.
N`00`*mmmmmm*	Data memory space.
N`40000`*mmm*	VU data registers.

In each case, the *mmmmmm* indicates the range of addresses permitted within the corresponding memory space.

A.1.1 VU Memory Spaces

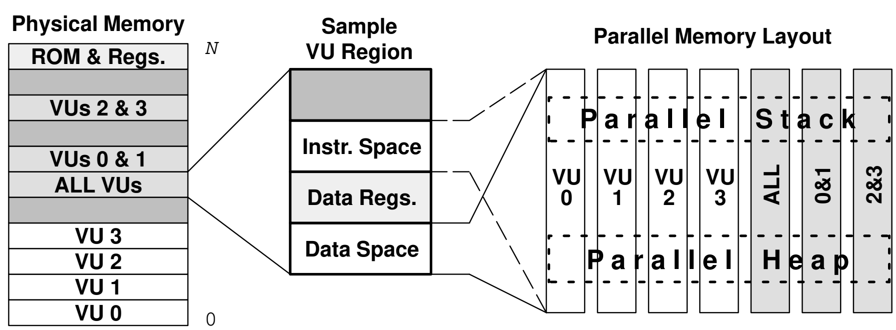
The data space and instruction space of a VU memory region in fact refer to the same piece of VU memory. A single memory location can thus be accessed in two ways: by an *instruction space* address, which triggers a VU operation, or by a *data space* address, which does not.

VU instruction space memory addresses trigger VU operations. A VU operation begins when a singleword or doubleword DPEAC instruction is written to an address in instruction space memory. The address written to provides the memory operand for the DPEAC instruction. The VU space in which the address is located selects the VUs that execute the instruction.

VU data space memory provides access to the parallel memory of the VUs without an accompanying VU operation. Data space memory operations are treated as normal memory accesses.

A.1.2 VU Parallel Stack and Heap

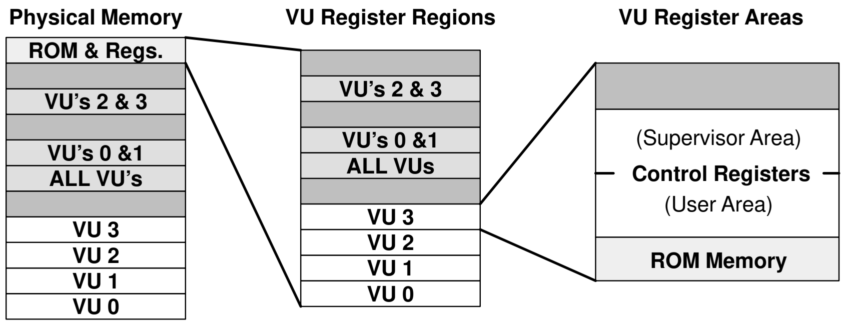
The memory region referred to by the data and instruction areas includes two regions: *parallel stack* and *parallel heap*. These occupy “stripes” of memory across the memory regions of all possible VU groupings:



A.1.3 VU Register Spaces

The VU *data register region* occupies 128 words of space, from physical address *N*`40000000` to *N*`400001FF` hex. This memory region corresponds to the 128 registers (`R0` – `R127`) accessible through DPEAC.

The VU *control register region* of VU physical memory is itself subdivided into regions for each possible VU combination.



Physical Address (hex)	**Purpose**
`FF`*N*`00`*mmmm*	ROM memory.
`FF`*N*`80`*mmmm*	Control registers (supervisor area).
`FF`*N*`88`*mmmm*	Control registers (user area).

Again, *N* represents the seven possible combinations of VUs, as listed above. Remember that the pair of VUs on a single chip share all control registers except for `dp_vector_mask` and `dp_vector_mask_buffer`. Any change to a shared register affects *both* VUs that share it.

A.2 VU Virtual Memory Mapping

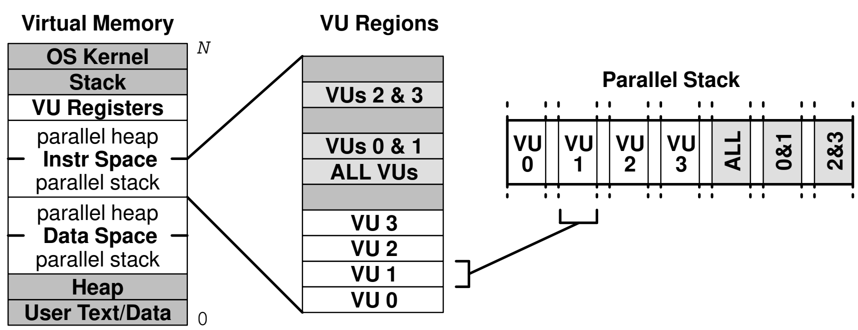
The virtual memory mapping for each CM node is established by CMOST, the CM-5 operating system. The VU memory and register regions are mapped into virtual memory by function, rather than by VU:

Virtual Address (hex)	**Purpose**
`40000000`	Instruction space stack regions.
`60000000`	Instruction space heap regions.
`80000000`	Data space stack regions.
`A0000000`	Data space heap regions.
`C0000000`	VU register (control and data) regions.

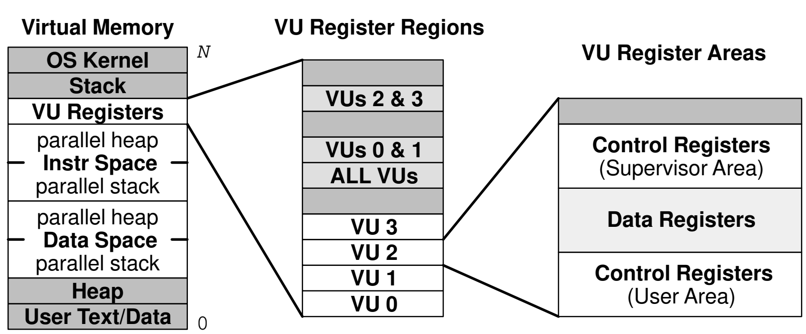
Each of the five virtual memory regions is divided into VU regions, with offsets as follows:

Address Offset (hex)	**Purpose**
`00000000`	VU 0 region.
`04000000`	VU 1 region.
`08000000`	VU 2 region.
`0C000000`	VU 3 region.
`10000000`	All VUs region.
`14000000`	VU 0/1 region.
`18000000`	VU 2/3 region.

Pictorially, the virtual memory mapping is as follows:



The VU control and data registers for each VU combination are mapped together into a single region:



A.3 VU Virtual Memory Symbolic Constants

For each virtual memory region and VU combination, there is a corresponding symbolic constant that specifies the starting address of the corresponding memory region. These constants are defined in the header file **`<cmsys/dp.h>`**. The names and current values of these constants are shown in the tables below:

Instruction Space Stack:

VU Region	Programming Constant Name	Address (hex)
VU 0	**`DPV_STACK_INST_PORT_0`**	**`0x40000000`**
VU 1	**`DPV_STACK_INST_PORT_1`**	**`0x44000000`**
VU 2	**`DPV_STACK_INST_PORT_2`**	**`0x48000000`**
VU 3	**`DPV_STACK_INST_PORT_3`**	**`0x4c000000`**
All VUs	**`DPV_STACK_INST_PORT_ALL`**	**`0x50000000`**
VU 0/1	**`DPV_STACK_INST_PORT_0_AND_1`**	**`0x54000000`**
VU 2/3	**`DPV_STACK_INST_PORT_2_AND_3`**	**`0x58000000`**

Instruction Space Heap:

VU Region	Programming Constant Name	Address (hex)
VU 0	**`DPV_HEAP_INST_PORT_0`**	**`0x60000000`**
VU 0	**`DPV_HEAP_INST_PORT_1`**	**`0x64000000`**
VU 0	**`DPV_HEAP_INST_PORT_2`**	**`0x68000000`**
VU 0	**`DPV_HEAP_INST_PORT_3`**	**`0x6c000000`**
All VUs	**`DPV_HEAP_INST_PORT_ALL`**	**`0x70000000`**
VU 0/1	**`DPV_HEAP_INST_PORT_0_AND_1`**	**`0x74000000`**
VU 2/3	**`DPV_HEAP_INST_PORT_2_AND_3`**	**`0x78000000`**

Data Space Stack:

VU Region	Programming Constant Name	Address (hex)
VU 0	**`DPV_STACK_DATA_0`**	**`0x80000000`**
VU 1	**`DPV_STACK_DATA_1`**	**`0x84000000`**
VU 2	**`DPV_STACK_DATA_2`**	**`0x88000000`**
VU 3	**`DPV_STACK_DATA_3`**	**`0x8c000000`**
All VUs	**`DPV_STACK_DATA_ALL`**	**`0x90000000`**
VUs 0/1	**`DPV_STACK_DATA_0_AND_1`**	**`0x94000000`**
VUs 2/3	**`DPV_STACK_DATA_2_AND_3`**	**`0x98000000`**

Data Space Heap:

VU Region	Programming Constant Name	Address (hex)
VU 0	`DPV_HEAP_DATA_0`	`0xa0000000`
VU 1	`DPV_HEAP_DATA_1`	`0xa4000000`
VU 2	`DPV_HEAP_DATA_2`	`0xa8000000`
VU 3	`DPV_HEAP_DATA_3`	`0xac000000`
All VUs	`DPV_HEAP_DATA_ALL`	`0xb0000000`
VU 0/1	`DPV_HEAP_DATA_0_AND_1`	`0xb4000000`
VU 2/3	`DPV_HEAP_DATA_2_AND_3`	`0xb8000000`

VU Data Registers:

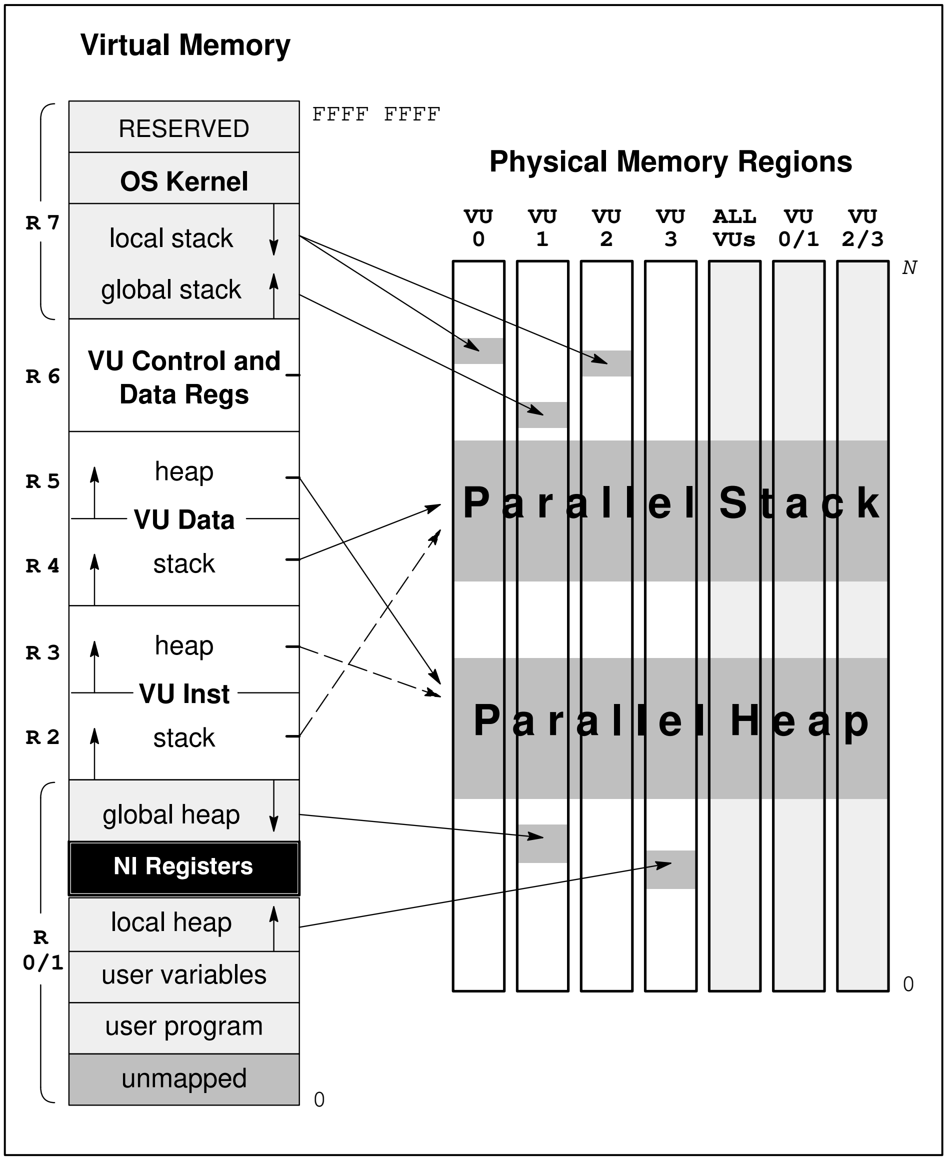
VU Region	Programming Constant Name	Address (hex)
VU 0	`DPV_DATA_REGS_0`	`0xc0800000`
VU 1	`DPV_DATA_REGS_1`	`0xc4800000`
VU 2	`DPV_DATA_REGS_2`	`0xc8800000`
VU 3	`DPV_DATA_REGS_3`	`0xcc800000`
All VUs	`DPV_DATA_REGS_ALL`	`0xd0800000`
VU 0/1	`DPV_DATA_REGS_0_AND_1`	`0xd4800000`
VU 2/3	`DPV_DATA_REGS_2_AND_3`	`0xd8800000`

VU Control Registers (user area):

VU Region	Programming Constant Name	Address (hex)
VU 0	`DPV_CTL_REGS_0`	`0xc0000000`
VU 1	`DPV_CTL_REGS_1`	`0xc4000000`
VU 2	`DPV_CTL_REGS_2`	`0xc8000000`
VU 3	`DPV_CTL_REGS_3`	`0xcc000000`
All VUs	`DPV_CTL_REGS_ALL`	`0xd0000000`
VU 0/1	`DPV_CTL_REGS_0_AND_1`	`0xd4000000`
VU 2/3	`DPV_CTL_REGS_2_AND_3`	`0xd8000000`

A.4 VU Physical/Virtual Memory Correspondence

The diagram below summarizes the above description of the relationship between physical and virtual VU memory regions:



Appendix B

VU Memory Maps

On the following pages are a two-sided memory and register map showing the overall layout of VU virtual memory and of the VU data and control registers, a one-page diagram showing the relationship between VU physical and virtual memory, and a quick-reference sheet showing the starting memory addresses of the various VU stack and heap regions.

VU Memory Map Goes Here
(side 1)

VU Memory Map Goes Here
(side 2)

VU Physical/Virtual Diagram Goes Here

VU Memory Addressing Diag. Goes Here

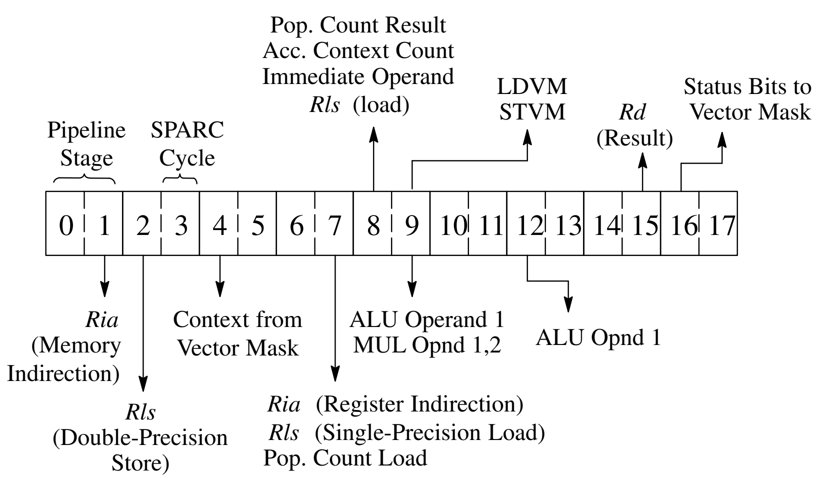
Appendix C

VU Pipeline

C.1 VU Instruction Pipeline

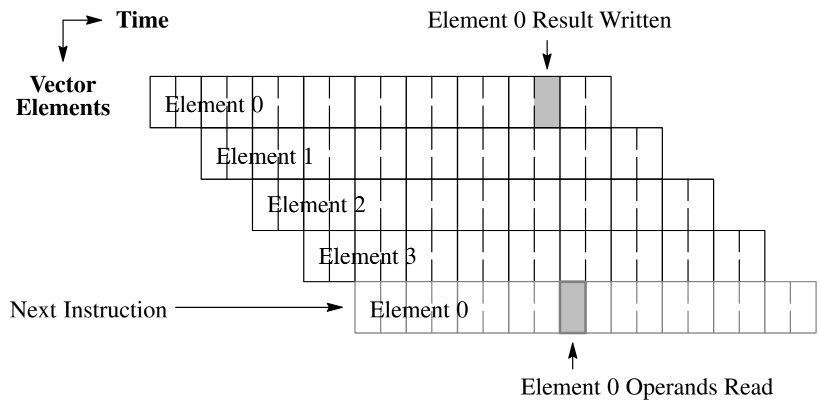
The VU accelerator chips execute vector instructions in a pipelined fashion, so that the operations on successive elements of vector operands can begin on successive clock cycles.

There are 9 stages to the VU pipeline, each two SPARC cycles long, through which a VU operation must pass for each element of a vector operand.



A new vector element is started at each VU pipeline stage. Thus, the result of the operation for element 0 is not generally available until four operations later.

For example, assuming a four-element vector length, the pipeline pattern is:



Generally, a destination register (*rD*) is readable four vector element operations later. For example, the upper marked box in the diagram above shows a register written in the second half of the eighth VU pipeline stage for element 0. The lower marked box shows the same register being read as an operand in the first half of the fourth stage for element 0 of the next vector operation. The read takes place in the succeeding SPARC cycle to the register write, so the value read will be the value written to the register. Any attempt to read the register earlier than shown above will return the prior contents of the register instead.

C.1.1 Pipeline Hazards

When two or more vector operations are executed in sequence, the VU chips attempt to execute them without breaking the pipeline, so that their execution can overlap. This creates the potential for *pipeline hazards*, conditions in which otherwise correctly written instructions can execute improperly when overlapped in the VU pipeline. These pipeline hazards can be corrected in one of two ways:

- by inserting a **`dpsynch`** instruction (see Section 4.4.1) between the offending instructions to prevent pipeline overlap
- by using the [**`no`**]**`pad`** modifier to insert padding between instructions
- by rewriting the instructions so the hazard condition no longer exists

There are seven possible pipeline hazards:

Hazard 1: Reading from a register written fewer than 4 VU operations ago.

The results of an operation typically cannot be used until 4 operations later. By default, the VUs pad all operations to a vector length of 4 (including scalar operations). Thus, an operation of vector length 2 is executed as: *operation0*, *operation1*, **`NOP`**, **`NOP`**. This avoids hazards as long as vectored data is always accessed from ascending registers, since the result from the first vector element of one vectored instruction will not be needed until the beginning of the next vectored instruction, which is guaranteed (by the padding) to be 4 operations later. There are three exceptions to this rule, described under Hazards 2, 3, and 4 below.

Hazard 2: Storing a register value to memory after fewer than 7 SPARC cycles.

If an instruction stores data to memory from a vector of registers, and the data in the registers is the result of an arithmetic operation, the data must be written to the registers 7 pipeline stages before it is stored in memory (for double-precision), or 5 pipeline stages (for single-precision data). This implies that a vector length of 7 or more is required for the data to be stored correctly, assuming registers are computed in the order in which they are stored on the subsequent instruction (for example, in ascending **`R`***nn* order).

Performance Note: The **`dpas`** assembler, by default, avoids this hazard by inserting an **`fnops`** operation before all instructions that perform stores. Such operations are padded to 8 cycles. This is effectively the same as assembling the instruction with a pad modifier of **`pad:8`**.

Hazard 3: Memory indirection after fewer than 8 SPARC cycles.

If an instruction uses indirect addressing of memory, and the vector of indirection offsets (*Ria*) is calculated by an arithmetic operation, each offset must be written to its register at last 8 pipeline stages before it is used. Thus, assuming the offsets are used in the order in which they were computed (for example, in ascending **`R`***nn* order), a vector length of 8 is required for this instruction sequence to work correctly. This does not apply to indirect regster accessing, which has the normal latency of 4 operations.

Performance Note: The **`dpas`** assembler, by default, avoids this hazard by inserting an **`fnops`** operation before each instruction that uses memory indirection, effectively padding it out to 8 cycles. (This is effectively the same as assembling the instruction with a pad modifier of **`pad:8`**.)

Hazard 4: Loading from second ALU operand after fewer than 3 cycles.

A relaxation of the 4-cycle latency rule is that a number of arithmetic operations (specifically, those that use the ALU circuitry of the accelerator chip) only require their second ALU operand (the first operand for shift operations) to have been written at least 2 operations previously rather than 4. This is because the second operand is read 2 cycles later in the pipeline than operands are normally read. The operations for which this exception applies are:

- **`neg`** and **`not`**
- **`enc`**, **`clas`**, **`exp`**, and **`mant`**
- type conversion operations: **`cvtf`**, **`dftof`**, **`ftodu`**, etc.
- shifts: **`shl`**, **`shr`**, etc. (first operand, value to be shifted)
- **`add`**, **`sub`**
- **`sub`**
- 2-operand bitwise logicals: **`and`**, **`nand`**, **`xor`**, etc.
- comparisons: **`cmp`**, **`le`**, **`gt`**, etc.
- triadics: **`dum`**, **`mad`**, **`msb`**, etc. (the operand that is not in the multiply)

An additional hazard occurs with this 2-cycle latency operand because it is read 2 cycles later in the pipeline. In a vector operation of these instructions, in the last two element calculations, the second operand register (as it is being strided) is vulnerable to loads (from memory, or from ACC or EPC operations) of that register done in the first two cycles of the next instruction.

Hazard 5: Accessing a VU register from the SPARC while the reg is in use.

Another hazard occurs when the SPARC reads or writes registers (via the **`dprd`** and **`dpwrt`** instruction) that are being read or written by a currently active instruction. The **`dprd`** and **`dpwrt`** operations are not synchronized to the pipelined instruction stream by the accelerator chip, and therefore can interact unpredictably. When the SPARC reads a register, as many as three of the previous instructions can be actively writing the register. When the SPARC writes a register, only the immediately preceding instruction can be active.

Performance Note: **`dpas`** automatically inserts synchronization code before **`dprd`** and **`dpwrt`** instructions to avoid this hazard, though this can be disabled.

Hazard 6: Generating status bits in a VU that is subsequently disabled.

The current implementation of the accelerator architecture has hazard potential when collecting status. Two VUs are housed on each accelerator chip. If an instruction is issued that collects meaningful status into **vector_mask** for one VU of a pair, and the next instruction is issued only to the other VU on the same chip, the **vector_mask** register will be corrupted in the disabled VU, effectively losing the status collected by the first instruction. To correct for this, rather than disabling the VU, you can operate it with conditionalization enabled and a vector mask of 0's.

Hazard 7: Chain loading into destination register of a VU **exchange**.

The current implementation of the assembler architecture disallows chain loading into a register that is also used as the destination register when the **exchange** modifier is used. In particular, the following is illegal:

```
dumovev V2,V2; duloadv [%i1],V2; exchange
```

This must be recoded to use different source and destination registers in the arithmetic operations. For example,

```
dumovev V2,V6; duloadv [%i1],V2; exchange
```

C.1.2 Avoiding Pipeline Hazards

You can avoid hazards by applying the following usage guidelines in writing your code:

1. Always use aligned vectors: start operations on vector register boundaries (that is, refer to vector registers by name, **V0**, **V1**, etc.) and process vectors in ascending order (always use a positive stride).
2. Do not use scalar registers (**S0** through **S31**) simultaneously as scalar and as vector operands.
3. Don't override the **sync** default in **dprd** and **dpwrt** instructions, and don't override the **pad:8** default in DPEAC instructions.
4. When executing an operation that generates status bits, don't execute a subsequent instruction that deselects some of the VUs.
5. Don't chain load into the rD operand in an **exchange** operation.

Appendix D

VU Arithmetic Operations

This appendix presents a description of each of the arithmetic operations provided by the CM-5 vector units, including information about the VU status bits that are modified by each operation.

D.1 Arithmetic Status Results

The computation of each element in a scalar or vector arithmetic operation generates status information. Arithmetic status is written to the **`dp_status`** mode register as an 18-bit value after each individual computation. Each bit of this status word indicates a particular item of status.

The **`dp_status`** mode register is overwritten after each individual computation. Therefore, one cannot retrieve the status bits for each vector element in a vector operation. Instead, bits can be chosen to contribute to (be logically OR-ed into) a single status bit that is shifted into the vector mask.

The table below lists the bits in the **`dp_status`** control register, along with their programming mask symbols, as defined by the DPEAC and CDPEAC header files. The symbols shown are defined as integer masks for the indicated bit.

The first five status bits, marked with (**`*`**), are the exceptions defined by the IEEE754 Floating-Point Arithmetic Standard.

Note: The opcode descriptions in Section D.2 include a list of status bits for each opcode, indicating which of the status bits *may* be set to 1 by the opcode. Any status bits not included in an opcode's list are always set to zero by the operation.

Bit	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)

The status bits are defined as follows:

- **`inexact`** is asserted when the delivered result after rounding differs from what would have been computed were both the exponent range and precision unbounded. **`inexact`** is never asserted when **`invalid`** is active.

- **`divide_by_zero`** is active when a division by zero is attempted, and the operands are not invalid.

- **`underflow`** is active when an IEEE floating-point underflow is detected after rounding. This flag is also active when an IEEE denormalized number is clipped to zero in fast mode. **`underflow`** is never active when an integer result is produced.

  Implementation Note: Currently, the underflow signal is generated by the logical OR of the **`under`** status with the logical AND of the **`deno`** and **`zero`** status bits.

- **`overflow`** is active when a floating-point result exceeds in magnitude, after rounding, the largest finite number in the destination format, were the exponent range unbounded. It is never active when the result of a computation is an integer.

- **`invalid_operation`** becomes active when any of the following occur:

  1. A floating-point operation that generates status has a signaling NaN as an operand.
  2. Two infinities with opposite signs are added.
  3. An infinite floating-point number is an operand in a floating-point–to–integer conversion. In this case, the result is saturated to the maximum integer of the proper sign, and **`int_overflow`** is set.
  4. An attempt is made to convert to an integer a floating-point number that is out of the range of the integer. In this case, the result is saturated to the maximum integer of the proper sign, and the **`int_overflow`** bit is set.
  5. A NaN is an operand in a floating-point–to–integer conversion. The result will be a zero, and the **`zero`** flag will be raised.
  6. An attempt is made to multiply 0 times infinity.
  7. An attempt is made to divide 0 by 0 or infinity by infinity.
  8. A square root of a non-zero number less than zero is attempted.

- The **`int_overflow`** flag is raised when an integer result is to be produced from an arithmetic operation or conversion, and an overflow occurs. This occurs in the following situations:

  1. For two's complement addition, this is the XOR of the *msb* and the *msb*+1 bits.
  2. For unsigned results, this is the value of the *msb*+1 bit, if the operands are both positive.
  3. For conversions, this occurs when a floating-point number outside the range of a destination integer is converted to integer. (This is similar to the "**`V`**" overflow bit that one would see on a microprocessor.)
  4. For integer multiplication, this occurs when a 1 is found in the upper half of the result.

- **`negative_unsigned`** is active when a negative result is generated for an unsigned integer during arithmetic and conversions. The result is forced to zero, and the **`zero`** flag is set.

- **`denorm_input`** is active when a denormal operand is detected. This flag can be active regardless of the underflow handling mode.

- **`zero`** is active when the integer or floating-point result is zero. Negative floating-point zero is also indicated with this flag.

- **`positive`** is active when the integer result is not zero or positive, or the floating-point result is not zero, positive, or a NaN. In the current implementation, the pos-result signal is generated by the logical NOR of the **`zero`**, **`negative`**, and **`nan-result`** bits (i.e., **`positive`** is set when these are all clear).

- **`negative`** is used during comparison operations, to indicate that the second operand is larger than the first operand. (For integer arithmetic operations, this flag is similar to the "**`S`**" or "**`N`**" flag found on microprocessors.) When an answer is in two's complement format, the **`negative`** flag will be the value of the *msb* of the result. When an unsigned result is produced, the **`negative`** flag will always be zero. In floating-point operations that produce status other than comparisons, the **`negative`** flag will be equal to the sign bit in the result. During conversions, **`negative`** will be equal to the sign of the result.

- **`integer_carry`** indicates that a carry has been generated from the *msb* of the result in the ALU's adder during an integer arithmetic instruction. For shift instructions, **`integer_carry`** is equal to the bit to the left of the *msb* or the bit to the right of the *lsb*, depending on the direction of the shift.

- **`infinity`** indicates the floating-point result is an infinity of either sign.

- **`nan`** indicates that the result is a quiet NaN. It is valid for instructions that produce floating-point results in the ALU, with the exception of the **`enc`** and **`move`** instructions. The **`enc`** instruction will not set this status bit, even if a quiet or signaling NaN is created.

- **`denorm`** is active when the result after rounding and after fast mode clipping is an IEEE denormal number. It is not active when a result after rounding is an inexact zero. **`denorm`** indicates the class of the result and not necessarily the occurrence of the IEEE *tiny* condition.

  In the current implementation, the **`denorm`** signal is generated by the logical AND of **`deno`** and the logical NOT of **`zero`**.

Implementation Note

The following signals are only used in the current accelerator implementation, and are included merely to facilitate testing.

- **`unordered`** is active for the comparison operation when at least one of the operands is a signaling or quiet NaN.
- **`under`** is active when an IEEE floating-point underflow is detected after rounding. It is never active when an integer result is produced.
- **`deno`** is active when the result after rounding and before fast mode clipping is an IEEE denormal number. It is not active when a result after rounding is an inexact zero. **`deno`** indicates the class of the result and not necessarily the occurrence of the IEEE tiny condition.

D.2 VU Arithmetic Operations

D.2.1 {i,di,f,df}abs

Takes the absolute value of its operand.

Possible status outputs:

`invalid`	Invalid operand; operand was a signaling NaN.
`int-overflow`	Result overflows the destination integer format.
`zero`	Result is zero.
`positive`	Result is not zero, negative, or a quiet NaN.
`integer-carry`	Integer carry out was generated during negation of a two's complement integer.
`infinity`	Result is an infinity.
`nan`	Result is a quiet NaN.
`denorm`	Result is a denormal number.
`(deno)`	Result is a denormal number before fast mode clipping.

D.2.2 {i,di,u,du,f,df}add, sub, subr

The **`add`** operation adds its first two operands.
The **`sub`** operation subtracts the second operand from the first.
The **`subr`** operation subtracts the first operand from the second.

Integer overflows during addition are signaled, but the result is not saturated. Negative results for unsigned integers are saturated to zero, and the negative-unsigned flag is asserted.

Note: The accelerator chip handles unsigned subtraction in a nonstandard way. Whenever a larger number is subtracted from a smaller number, the result is zero, rather than wrapping. This can occur in the unsigned variants of the subtract instructions and the triadics with negated operands.

Possible status outputs:

`inexact`	Result cannot be represented exactly.
`underflow`	Result underflows the destination format.
`overflow`	Result overflows the destination floating-point format.
`invalid`	Invalid operand; operand was a signaling NaN or operands are oppositely signed infinities.
`int-overflow`	Result overflows the destination integer format.
`negative-unsigned`	A negative result was generated for an unsigned integer.
`zero`	Result is zero.
`positive`	Result is not zero, negative, or a quiet NaN.
`negative`	Result has a negative sign.
`integer-carry`	Integer carry out was generated.
`infinity`	Result is an infinity.
`nan`	Result is a quiet NaN.
`denorm`	Result is a denormal number.
`(deno)`	Result is a denormal number before fast mode clipping.

D.2.3 {i,di,u,du}addc, subc, sbrc

These three functions are similar to the **add**, **sub**, and **subr** operations, except that the **addc**, **subc**, and **sbrc** operations include a carry bit in the computation. The bits being shifted off of the vector mask (normally used for conditionalization) are used here as the carry input.

As a result, these operations always operate unconditionally, regardless of the setting of the **vector_mask_mode** control register and ignoring any use of the modifiers affectiong conditionalization (**always**, **condmem**, and **condalu**). If the **vminvert** modifier is used, the bit shifted from the vector mask is complemented before being used in the computation. If a memory operation accompanies these arithmetic operations, it is conditionalized normally by the vector mask bits.

Note: The accelerator chip handles unsigned subtraction in a nonstandard way. Whenever a larger number is subtracted from a smaller number, the result is zero, rather than wrapping. This can occur in the unsigned variants of the subtract instructions and the triadics with negated operands.

Possible status outputs:

int-overflow	Result overflows the destination integer format.
negative-unsigned	A negative result was generated for an unsigned integer.
zero	Result is zero.
positive	Result is not zero or negative.
negative	Result has a negative sign.
integer-carry	Integer carry out was generated.

Note: In the current implementation, for **subc** and **sbrc**, if the operands and carry are zero, then **integer-carry** is set to 1.

D.2.4 {u,du}and, andc, nand, or, nor, xor

These operations perform, respectively, a bitwise logical AND, a bitwise AND with the first operand complemented, a bitwise NAND, a bitwise OR, a bitwise NOR, and a bitwise XOR of the two operands.

Possible status outputs:

zero	Result is zero.
positive	Result is not zero.

D.2.5 {f,df}clas

Floating-point number class function. Produces an integer indicating the number class of the operand. (The size of the integer will be the size of the operand.) Does not flag signaling NaNs as an exception. Encodes the integer result as:

Integer Result	Interpretation
Hex 0001	Signaling NaN.
Hex 0002	Quiet NaN.
Hex 0004	Negative infinity.
Hex 0008	Negative normalized finite non-zero.
Hex 0010	Negative denormalized.
Hex 0020	Negative zero.
Hex 0040	Positive zero.
Hex 0080	Positive denormalized.
Hex 0100	Positive normalized finite non-zero.
Hex 0200	Positive infinity.

Possible status outputs:

`positive`	Always asserted.

D.2.6 {i,di,u,du,f,df}cmp

The generic **`cmp`** operation is supported only for completeness. More specific compares (such as **`fgtv`**) are preferred. For **`cmp`**, a third argument is given that specifies the compare code (0–7).

Possible status outputs:

`invalid`	Invalid operand; at least one operand is NaN.
`zero`	Operands are equal.
`positive`	Result is not zero, negative, or a quiet NaN.
`negative`	Second operand is greater than the first.
`unordered`	At least one operand is a NaN.

D.2.7 cvtf, cvtfi, cvti, cvtir

The general conversion opcodes (**`cvtf`**, **`cvtfi`**, **`cvti`**, and **`cvtir`**) are supported only for completeness. The specific conversion opcodes, such as **`ftoiv`**, are preferred. The specific opcodes expect two operands, a source and a destination. The general opcodes take three operands: a source, a convert code, and a destination. The convert code specifies the conversion done.

The **`cvtir`** opcode performs the same float-to-integer conversions as the **`cvti`** opcode, except that **`cvti`** truncates its result and **`cvtir`** rounds the result to the nearest integer.

Possible status outputs:

`inexact`	Result cannot be represented exactly.
`underflow`	Result underflows the destination format.
`overflow`	Result overflows the destination format.
`invalid`	Invalid operand; operand could be a NaN, infinite, or outside the range of the destination integer.
`int-overflow`	Integer overflow; operand is outside the range of the destination integer.
`negative-unsigned`	Attempt to convert negative value to unsigned.
`zero`	Result is zero.
`positive`	Result is not zero, negative, or a quiet NaN.
`negative`	Result has a negative sign.
`infinity`	Result is an infinity.
`nan`	Result is a quiet NaN.
`denorm`	Result is a denormal number.
`(deno)`	Result is a denormal number before fast mode clipping.

D.2.8 {f,df}div

This instruction divides the first operand by the second operand. When an operand is NaN, infinity, or zero, the division timing will be the same as for normalized operands and results. **Note:** In the current accelerator chip, the **`div`** function cannot be used in conjuunction with a memory operation.

These operations do not execute one-per-cycle as do the other operations. Specifically, for a vector length of N, these instructions will take $2N*k$ Mbus (SPARC) cycles to execute, rather than the usual $2N$, where k is taken from the following table:

Operation	*k* Value
fdiv{**v**,**s**}	4
dfdiv{**v**,**s**}	5

Possible status outputs:

inexact	Result cannot be represented exactly.
divide-by-zero	Division of zero into a non-zero finite number.
overflow	Result too large to be represented in destination format.
underflow	Result too small to be represented in destination format.
invalid	Result cannot be represented exactly; may be caused by a signaling NaN, 0/0, or infinity / infinity.
denorm-input	Denormal input.
zero	Result is zero.
positive	Result is not zero, negative or a quiet NaN.
negative	Result has a negative sign.
infinity	Result is an infinity.
nan	Result is a quiet NaN.
(deno)	Result is a denormal number before fast mode clipping.

D.2.9 dum[h]{s,m,o,x}{a,i,t}

These multiply-logical operations combine a 64-bit multiply and a 64-bit bitwise logical operation. These can be used to implement "shift-and-mask" type constructions. The use of multiply rather than a true shift allows more complicated shifting patterns. Either the least significant or the most significant 64 bits of the multiply result can be used.

These operations all work on 64-bit unsigned values (type **du**). The logical function is based on the "**s**,**m**,**o**,**x**" choice:

s (select)	gives the AND function
m (mask)	gives the AND-NOT function
o	gives the OR function
x	gives the XOR function

The last letter of the opcode indicates the triadic form:

`a`	accumulative
`i`	inverted
`t`	triadic

The optional "**`h`**" causes the most significant 64 bits of the multiply result to be used as the first operand to the logical operation, rather than the least significant 64 bits. The high half can simulate a right shift. For example, multiplying by 2^{64-*N*} and using the high half of the result is effectively a right shift by *N*.

The result status is reported for the ALU boolean only. The other status is derived entirely from the multiplication.

Possible status outputs:

`int-overflow`	A 1 was found in the upper half of the result (MULT).
`zero`	Result is zero (ALU).
`positive`	Result is not zero (ALU).

D.2.10 {u,du}enc

This operation generates a floating-point number by placing the first operand in the exponent field and the second operand in the mantissa field. For both operands, the values are given as unsigned integers in least significant bits. The exponent is given in biased form. The output is a floating-point value.

For floating-point values in the normalized range, the mantissa operand must contain the hidden bit. Denormal numbers are constructed from an exponent equal to 1 and a mantissa with the hidden bit cleared. The resulting denormal value will have a zero in its exponent field. No checking is performed on the resulting floating-point number.

Possible status outputs:

`positive`	Always asserted.

D.2.11 etrap

This operation is not a vector operation. It forces a trap to occur if the result of logically ANDing the value of **`dp_status`** with the bits in the register **`dp_interrupt_enable_green`** is non-zero. Since the **`dp_status`** register contains the status from the last element computed, this is really only useful for diagnostic purposes. This operation sets no status flags.

D.2.12 {f,df}exp

This operation extracts the biased exponent of its operand, a floating-point value. The *lsb* of the exponent becomes the *lsb* of the resulting integer. The precision of the floating point operand becomes the precision of the resulting integer. The format of the result is the same as that required for the **`enc`** instruction.

The result of NaN and infinity operands is a left-justified string of ones equal to the width of the exponent field of the operand. Denormal numbers produce an integer with a value of 1.

Possible status outputs:

`invalid`	Invalid operand; first operand was a signaling NaN.
`positive`	Always asserted.

D.2.13 {u,du}ffb

The **`ffb`** (find first bit) instruction returns the number of leading (most significant) zeroes above the most significant 1 bit in the operand. For example, **`duffb`** of binary 0001... returns 3. If no 1s are present in the operand, a zero is returned. The single-precision version, **`duffb`**, views its operand as a 64-bit unsigned number constructed by padding the 32-bit argument with zeroes. As a result, **`uffb`**(0xFFFFFFFF) gives 32. The result of the **`ffb`** instruction on an operand equal to zero is zero.

Possible status outputs:

`zero`	Result is zero.
`positive`	Result is not zero.

D.2.14 ftodf, dftof

These operations convert a floating-point operand to a floating point result of another precision.

Possible status outputs:

`inexact`	Result cannot be represented exactly.
`underflow`	Result underflows the destination format.
`overflow`	Result overflows the destination floating-point format.
`invalid`	Invalid operand; operand was a signaling NaN.
`zero`	Result is zero.
`positive`	Result is not zero, negative, or a quiet NaN.
`negative`	Result has a negative sign.
`infinity`	Result is an infinity.
`nan`	Result is a quiet NaN.
`denorm`	Result is a denormal number.
`(under)`	Result underflows dest format before fast mode clipping.
`(deno)`	Result is a denormal number before fast mode clipping.

D.2.15 {f,df}inv

This operation takes the reciprocal of its operand. When the operand is NaN, infinity, or zero, the reciprocal timing will be the same as for normalized operands and results.

Note: In the current accelerator chip, the **`inv`** function cannot be used in conjunction with a memory operation.

These operations do not execute one-per-cycle as do the other operations. Specifically, for a vector length of N, these instructions will take $2N*k$ Mbus (SPARC) cycles to execute, rather than the usual $2N$, where k is taken from the following table:

Operation	*k* Value
`fdiv`{**`v`**,**`s`**}	4
`dfdiv`{**`v`**,**`s`**}	5

Possible status outputs:

inexact	Result is inexact.
divide-by-zero	1/0 was attempted.
overflow	Result too large to be represented in destination format.
underflow	Result too small to be represented in destination format.
invalid	Invalid operand; operand is a signaling NaN.
denorm-input	Denormal input.
zero	Result is zero.
positive	Result is not zero, negative, or a quiet NaN.
negative	Result has a negative sign.
infinity	Result is an infinity.
nan	Result is a quiet NaN.
(deno)	Result is a denormal number before fast mode clipping.

D.2.16 {f,df}isqt

This operation divides the first operand by the square root of the second operand. No status is generated by this instruction.

This operation does not obey the IEEE standard with respect to rounding or exception detection. First, **isqt** always rounds the result toward zero. The error in the result will be at most one *lsb* in the mantissa when compared to an infinitely precise answer, and the result will be equal to or smaller than the infinitely precise answer. Second, the only exception detected is when the operand is negative and non-zero or a NaN, in which cases a NaN result is generated and the **dp_status_nan_result** bit in the **dp_status** register is set. Notably, no divide-by-zero detection is done for the case when the argument is zero, nor is underflow signaled when the result is too small. The positive status bit is always set.

Note: In the current accelerator chip, the **isqt** function cannot be used in conjunction with a memory operation.

These operations do not execute one-per-cycle as do the other operations. Specifically, for a vector length of *N*, these instructions will take 2*N***k* Mbus (SPARC) cycles to execute, rather than the usual 2*N*, where *k* is taken from the following table:

Operation	*k* Value
`fisqt`{**`v`**,**`s`**}	5
`dfisqt`{**`v`**,**`s`**}	7

When an operand is NaN, infinity, or zero, the **`AINVSQRTB`** timing will be the same as for normalized operands and results.

Possible status outputs:

`positive` Always asserted.

D.2.17 lvdm

Required syntax: **`ldvm`** *VU-register*

This operation moves the low order 16 bits of a specified register into both the vector mask (**`dp_vector_mask`**) and the vector mask buffer (**`dp_vector_mask_buffer`**). This operation has a fairly significant cost both in execution speed and in pipeline delay, and should be used sparingly.

This instruction may not be combined with a memory operation (**`load`**, **`store`**), and is not affected by conditionalization. Possible status outputs: None.

D.2.18 {i,di,u,du,f,df}lt, le, gt, ge, eq, ne, un, lg

These operations compare the two operands. No result is written to the register file. The result of the compare (a 1 bit if true, otherwise a 0 bit) is shifted into the vector mask. The rotate mode (**`vmrotate`**) is used by default, but the **`vmcurrent`** modifier can be added to change to the current mode. (See Section 4.3.1 for DPEAC, Section 6.7.1 for CDPEAC.)

In addition, the status bits are set as if the two values were subtracted (operand 1 minus operand 2), as shown below. At most one of **`zero`**, **`negative`**, and **`unordered`** will be set.

Possible status outputs:

invalid	At least one operand is a signaling NaN.
zero	Operands are equal.
positive	Result is not zero, negative, or a quiet NaN.
negative	The second operand is greater than the first operand.
unordered	At least one operand is a NaN.

D.2.19 {i,di,u,du,f,df}mad, msb, msr, nma

These operations perform a multiply followed by some other operation (for example, an add). There are three forms, the accumulative, the inverted, and the triadic. These differ in the way in which they apply the operands in the computation. The form is signaled by the suffix on the instruction: **a =** accumulative, **i** = inverted, **t =** true triadic.

Integer overflows during addition are signaled, but the result is not saturated. Negative results for unsigned integers are saturated to zero, and the **negative-unsigned** flag is asserted. The result status (**zero**, **negative-result**, etc.) reflects the final result only. Exception status (e.g., **overflow**) is derived by OR-ing the status from both operations.

Possible status outputs:

inexact	Result cannot be represented exactly.
invalid	Invalid operand.
overflow	Result overflows the destination floating-point format.
underflow	Result underflows the destination format.
int-overflow	**MULT**: a '1' was found in the upper half of the result.
	ALU: result overflows the destination integer format.
negative-unsigned	A negative result was generated for an unsigned integer.
denorm-input	Denormal input.
zero	Result is zero.
positive	Result is not zero, negative, or a quiet NaN.
negative	Result has a negative sign.
integer-carry	Integer carry out was generated.
infinity	Result is an infinity.
nan	Result is a quiet NaN.
denorm	Result is a denormal number.
(under)	Result underflows dest format before fast mode clipping.
(deno)	Result is a denormal number before fast mode clipping.

D.2.20 {f,df}mant

The operation extracts the mantissa of its operand, a floating-point number. If the operand is normalized, the hidden bit is present in the result. The *lsb* of the mantissa becomes the *lsb* of the resulting integer. A double-precision floating-point operand produces a double-precision integer result, and a single-precision floating-point operand produces a single-precision result. The format of the result is the same as that required for the **`enc`** instruction. A NaN input returns the value of the mantissa, with the hidden bit set to 1, and sets the **`invalid`** status.

Possible status outputs:

`invalid`	Invalid operand; first operand was a signaling NaN.
`positive`	Always asserted.

D.2.21 {i,di,u,du,f,df}move

The first operand is passed through the ALU. Invalid operands are not converted into a quiet NaN. Possible status outputs:

`positive`	Always asserted.

D.2.22 {u,du}mrg

The merge instruction merges two vectors under control of the vector mask. Bits shifted off of the vector mask, normally used for conditionalization, are used in a different way. Specifically, for each element, the bits control which operand is moved to the destination. If the bit shifted from the vector mask is a ‘1’, the result is taken from the first operand; otherwise, it is taken from the second operand. If you use the **`vminvert`** modifier, which inverts the sense of the vector mask, the merge is done in the opposed fashion. As a result, this operation always operates unconditionally, regardless of the setting of the **`vector_mask_mode`** control register, and it ignores the modifiers affecting conditionalization (**`always`**, **`condmem`**, and **`condalu`**). Any memory operation that accompanies these arithmetic operations is conditionalized normally by the vector mask bits.

Possible status outputs:

`positive`	Always asserted.

D.2.23 {i,u,f,df}mul

This operation multiplies its two operands. Multiplication with the double integer format can be accomplished with the {**di**,**du**}**mul** and {**di**,**du**}**mulh** instructions. Multiplication of 32-bit integers will produce a 64-bit result. When an operand is NaN, infinity, or zero, the multiplication timing will be the same as for normalized operands and results.

Possible status outputs:

inexact	Result cannot be represented exactly.
overflow	Result overflows the destination floating-point format.
underflow	Result underflows the destination format.
invalid	Invalid operand; caused by 0 times infinity or by a signaling NaN operand.
int-overflow	A '1' was found in the upper half of the result.
denorm-input	Denormal input.
zero	Result is zero.
positive	Result is not zero, negative, or a quiet NaN.
negative	Result has a negative sign.
infinity	Result is an infinity.
nan	Result is a quiet NaN.
(under)	Result underflows dest format before fast mode clipping.
(deno)	Result is a denormal number before fast mode clipping.

D.2.24 {di,du}mul, mulh

These operations generate the low (**mul**) and high (**mulh**) 64 bits of the multiplication of the two integer operands.

Possible status outputs:

int-overflow	A '1' was found in the upper half of the result.
zero	Result is zero.
positive	Result is not zero, negative, or a quiet NaN.
negative	Result has a negative sign.

D.2.25 {i,di,f,df}neg

This operation subtracts the first operand from zero. Integer overflows are signaled, but the result is not saturated. Negative results for unsigned integers are saturated to zero, and the negative-unsigned flag is asserted.

Possible status outputs:

invalid	Invalid operand; operand was a signaling NaN.
int-overflow	Result overflows the destination integer format.
zero	Result is zero.
positive	Result is not zero, negative, or a quiet NaN.
negative	Result has a negative sign.
negative-unsigned	Attempt to convert negative value to unsigned.
integer-carry	Integer carry out was generated during negation of a two's complement integer.
infinity	Result is an infinity.
nan	Result is a quiet NaN.
denorm	Result is a denormal number.
(deno)	Result is a denormal number before fast mode clipping.

D.2.26 fnop{v,s}

This operation is, as its name suggests, a NOP. It does nothing.

Possible status outputs: None.

D.2.27 {u,du}not

This operation performs a bitwise logical NOT of its first operand.

Possible status outputs:

zero	Result is zero.
positive	Result is not zero, negative, or a quiet NaN.

D.2.28 {u,du}shl, shlr

This operation performs an integer logical left shift.

For **shl**, the first operand is the value to be shifted, and the unsigned value of the low 6 bits of the second operand is the shift distance. The bits in the second operand above the sixth bit are ignored. The reversed shift, **shlr**, shifts the second operand by the value in the low 6 bits of the first operand. These alternatives are provided so that the *Rs1* operand, which has improved accessibility and striding capability, can be used as either operand.

Zeros are shifted into the low end of the result. The status integer-carry will be the value of the bit to the left of the *msb*. A negative two's complement integer is sign-extended beyond the *msb*, so a zero shift on a negative two's complement number will produce **integer-carry**. As a result of the 6-bit shift value, it is not possible to shift a double-precision value by a full 64 bits. (It is, however, possible to shift a 32-bit integer by 32).

Possible status outputs:

integer-carry	Value of the *msb*+1 bit.
zero	Result is zero.
positive	Result is not zero, negative, or a quiet NaN.

D.2.29 {u,du,i,di}shr, shrr

This operation performs an integer shift right (logical or artihmetic). For **shr**, the first operand is the value to be shifted and the unsigned value of the low 6 bits of the second operand is the shift distance. The bits in the second operand above the sixth bit are ignored. For **shrr**, the reverse shift is performed: the second operand is the value to be shifted, and the low 6 bits of the first operand provide the shift distance.

For the unsigned data types (**u** and **du**), the shift is a logical shift; thus, zeros are shifted into the high end of the result. For the signed data types (**i** and **di**), an arithmetic shift is performed: i.e., the bits shifted into the upper end of the result are a copy of the original sign bit of the operand. For example, shifting the 32-bit hexidecimal value 80000008 right one bit by an arithmetic shift yields C0000004, while a logical shift of the same value yields 40000004. The **integer-carry** status is the value of the bit to the right of the *lsb*.

Possible status outputs:

`integer-carry`	Value of the LSB-1 bit.
`zero`	Result is zero.
`positive`	Result is not zero, negative, or a quiet NaN.
`negative`	Result has a negative sign.

D.2.30 stvm

Required syntax: **`stvm`** *VU-register*

This operation moves the 16-bit value in the vector mask (**`dp_vector_mask`**) into the specified *VU-register*. This instruction may not be combined with a memory operation and is not affected by conditionalization. This operation has a significant cost in speed and pipeline delay, and, should be used sparingly.

Possible status outputs: None.

D.2.31 {d,df}sqrt

The operation takes the IEEE square root of the first operand.

Note: In the current accelerator chip, the **`sqrt`** function cannot be used in conjunction with a memory operation.

These operations do not execute one-per-cycle as do the other operations. Specifically, for a vector length of *N*, these instructions will take 2*N* *k* Mbus (SPARC) cycles to execute, rather than the usual 2N, where *k* is taken from the following table:

Operation	*k* Value
`fsqrt`{**`v`**,**`s`**}	6
`dfsqrt`{**`v`**,**`s`**}	8

When the operand is a NaN, infinity, zero, or negative, the square root timing will be the same as for normalized operands and results.

Possible status outputs:

`inexact`	Result cannot be represented exactly.
`underflow`	Result underflows the destination format.
`invalid`	Invalid operand; operand is a negative non-zero number or a signaling NaN.
`denorm-input`	Denormal input.
`zero`	Result is zero.
`positive`	Result is not zero, negative, or a quiet NaN.
`negative`	Result has a negative sign.
`infinity`	Result is an infinity.
`nan`	Result is a quiet NaN.
`(under)`	Result underflows dest format before fast mode clipping.
`(deno)`	Result is a denormal number before fast mode clipping.

D.2.32 {i,di,u,du,f,df}test

This operation adds the first operand to zero. Unlike **`move`**, the **`test`** instructions do assert status outputs to reflect the value moved. Specifically, **`test`** is exactly like adding zero to the operand, as far as the setting of status flags is concerned.

Possible status outputs:

`invalid`	Invalid operand; operand was a signaling NaN.
`zero`	Result is zero.
`positive`	Result is not zero, negative, or a quiet NaN.
`negative`	Result has a negative sign.
`infinity`	Result is an infinity.
`nan`	Result is a quiet NaN.
`denorm`	Result is a denormal number.
`(deno)`	Result is a denormal number before fast mode clipping.

D.2.33 {f,df}to{i,di,u,du}{[r]}

These operations convert a floating-point operand to integer format. Overflows are saturated to the maximum integer value in the output format, and underflows are forced to zero. Converting a negative floating-point number to unsigned causes the **`negative-unsigned`** status and saturates the result to zero. Conversions from NaN formats to integers create a zero result, and set the **`invalid`** status. Conversions from infinities are saturated like overflows; they, too, cause the **`invalid`** status. The "**`r`**" forms round by the current rounding mode (set by default to "nearest"); the non-"**`r`**" forms simply truncate toward zero.

Possible status outputs:

`inexact`	Result cannot be represented exactly.
`invalid`	Invalid operand; operand could be a NaN, infinity, or outside the range of the destination integer.
`int-overflow`	Result overflows the destination integer format.
`negative-unsigned`	Attempt to convert negative value to unsigned.
`zero`	Result is zero.
`positive`	Result is not zero, negative, or a quiet NaN.
`negative`	Result has a negative sign.
`nan`	The first operand is a NaN.

D.2.34 {i,di,u,du}to{f,df}

These operations convert an integer operand to floating-point format.

Possible status outputs:

`inexact`	Result cannot be represented exactly.
`negative`	Result has a negative sign.
`positive`	Result is not zero, negative, or a quiet NaN.
`zero`	Result is zero.

D.2.35 trap

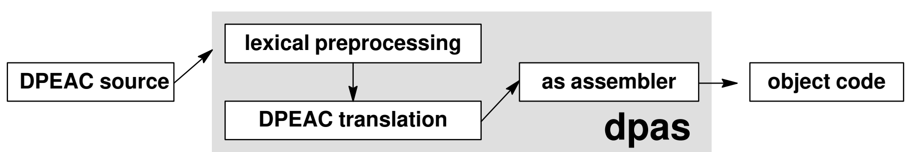
This operation, which is not a vector operation, forces a trap to occur. This trap will be a green interrupt identical to the trap caused by a masked status trap. Possible status outputs: None.

Appendix E

The dpas Assembler

E.1 The dpas Assembler

The `dpas` assembler is used to assemble a DPEAC source file. **`dpas`** is an extension of the SPARC **`as`** assembler; it translates DPEAC instructions into SPARC instructions, then passes the translated instructions to **`as`** for assembly.



The **`dpas`** command line format is

`dpas` [*switches...*] [*source-file*] [*switches...*]

where *source-file* is a text file containing a DPEAC program, and having a filename extension of "**`.dp`**" (if omitted, **`stdin`** is used). Assembled object code is written to a file with the same name but with an extension of "**`.o`**".

Optional *switches* can precede and follow the *source-file* argument. Typing "**`dpas -h`**" gives a list of the current switches.

Typing "**`dpas -Fw`**" puts **`dpas`** in "filter" mode; you can type in a DPEAC statement to see if its syntax is correct, and to see what SPARC code it produces.

`dpas` also includes its own preprocessor that provides C-like lexical directives (**`#define`**, **`#ifdef`**, etc.) and macro definitions.

The **dpas** assembler provides the following lexical directives:

#comment	*comment-text...*	— Comment line.
#define	*symbol text...*	— Preprocessor symbol.
#undef	*symbol*	— Undefine preprocessor symbol.
#set	*symbol expression*	— Sets symbol to value of expression.
#define	*name(params) body...*	— Preprocessor macro.
#macro	*name parameter*[=*default*]...	— Multi-line macro.
#endmacro		— End of macro body.
#include	*filename*	— Include named file (as in C code).
#if	*expression*	— Assemble if *expression* is non-zero.
#ifz	*expression*	— Assemble if *expression* is zero.
#ifdef	*symbol*	— Assemble if *symbol* is **#define**d.
#ifndef	*symbol*	— Assemble if *symbol* is not **#define**d.
#ifsame	*string1, string2*	— Assemble if strings are exactly identical.
#ifnsame	*string1, string2*	— Assemble if strings are different.
#ifblank	[*text*]	— Assemble if rest of line is blank.
#ifnblank	[*text*]	— Assemble if rest of line is not blank.
#ifz	*expression*	— Assemble if *expression* is zero.
#else	*expression*	— Else case for **#if** directive.
#elif	*expression*	— Else-if case.
#else if*xxx*	*expression*	— Else case, starts new **#if***xxx* directive.
#else		— Final else clause for **#if***xxx* directive.
#endif		— End of **#if** directive.
#repeat	*count*	— Repeat block.
#endrepeat	*count*	— End of repeat block.
#print	*item, item,...*	— Print items (strings or expressions).
#warning	*item, item,...*	— Print items, signal assembler warning.
#error	*item, item,...*	— Print items, signal assembler error.
#ident	*text...*	— Entire line passed to output unchanged.

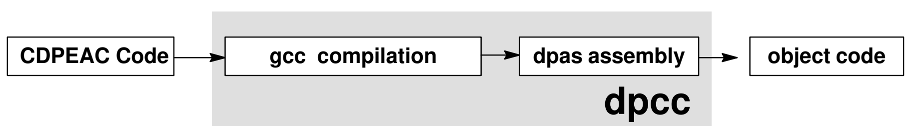
dpas pre-defines the symbol "**dpas**" to the string "**1**" before preprocessing a file; this symbol can be used to conditionally assemble code depending on whether or not **dpas** is being used as the assembler.

Appendix F

The dpcc Compiler

F.1 The dpcc Compiler

The **`dpcc`** compiler is used to compile a CDPEAC program. **`dpcc`** is an extension of the GNU C compiler **`gcc`**; it translates a CDPEAC procedure into the corresponding DPEAC code, then calls **`dpas`** to assemble the code.



The **`dpcc`** command line format is

> **`dpcc`** [*switches*...] [*source-file*] [*switches*...]

where *source-file* is a text file containing a CDPEAC program, and having a filename extension of "**`.cdp`**". Assembled object code is written to a file with the same name but with an extension of "**`.o`**".

Optional *switches* can precede and follow the *source-file* argument. Typing "**`dpcc -h`**" gives a list of the current switches.

Appendix G

How CDPEAC Works

This appendix describes the way that GNU CC's **`asm`** statement and macro facilities are used to define the CDPEAC instruction set.

Note: This information is provided for those readers interested in how CDPEAC operates — this is not essential knowledge for simply using CDPEAC, however.

G.1 GNU CC's ASM Statement

GNU CC (or "GCC", as it is often abbreviated) is a C compiler provided with the GNU operating system. The full description of GNU CC's **`asm`** statement is best left to the GCC user's manual. The description below concentrates on how GCC is currently used to permit DPEAC programming from C.

GCC's ASM statement has the basic form:

`asm``(` *pattern* **`:`** *outputs* **`:`** *inputs* **`:`** *clobbered* **`);`**

where

- *pattern* is a string containing an assembly-language instruction.
- *inputs* and *outputs* are descriptions of C variables that represent the operands passed into the instruction and the values (if any) that are returned.
- *clobbered* is a series of strings naming any internal chip registers that are modified (or "clobbered") by the instruction.

Note: For the purposes of CDPEAC, the *outputs* argument can be ignored.

For example:

```
asm( "dfloadv %0, V5" : : "m" (*source) : "%g2", "%g3" )
```

In this example, the DPEAC instruction **dfloadv** directs each VU to load into vector register **V5** a vector's worth of data from the memory location specified by the C variable **source**. (The **"m"** indicates that the **source** variable is a pointer into memory.) As with most DPEAC operations, this instruction clobbers the SPARC registers **%g2** and **%g3**.

When a C program containing this **asm** statement is compiled by GCC, the **asm** statement might translate into the following actual DPEAC code:

```
dfloadv [%i1], V5
```

where the **[%i1]** indicates that the pointer contained in the **source** variable has been moved into the SPARC register **%i1** for use by DPEAC.

G.1.1 The Pattern and Input Arguments

In an **asm** statement, the *pattern* argument is a string (or a series of strings) containing a "template" for an instruction. This template can contain pattern variables **%0**, **%1**, **%2**, etc., indicating where the *input* and *output* arguments of the **asm** statement should appear in the final assembled instruction.

The pattern variables **%0**, **%1**, **%2**, etc., enumerate in order the arguments appearing in the *output* and *input* fields of the **asm** statement. (Since the *output* field is not used by CDPEAC, these variables effectively enumerate the *input* arguments.)

Each *input* argument consists of two parts:

- a *constraint* string, which indicates the type of the input argument
- a C expression defining the argument's *value*

The constraint typically contains a single letter giving the argument's type. For example, "**m**" indicates a memory operand, "**r**" a general register operand, "**i**" an immediate integer value, etc. (The GCC documentation includes a list of these constraint strings and their specific meanings.)

Note: If the *pattern* argument consists of multiple strings, these strings are concatenated in order when the **asm** statement is compiled.

G.2 Using GCC Macros to Produce ASM Statements

The preprocessor macro facility of GCC makes it easy to construct **asm** statements for DPEAC instructions. For example, the **asm** statement presented above could be rewritten as:

```
asm( "df" "loadv %0, " "V5" : : "m" (*source) : \
     "%g2", "%g3" )
```

Since the pattern argument of this statement is now separated into parts, these parts can be provided by macro arguments. For example, a general C macro for defining **loadv** instructions might be defined as follows:

```
#define loadv(type, source, data_register) \
    asm(#type "loadv %0, " data_register : : \
         "m" (*source) : "%g2", "%g3")
```

The **loadv** macro expects *type* to be a literal symbol representing the data type of the load operation (the # in front of *type* converts it into a string). The *source* and *data_register* arguments are assumed to be strings representing the source variable and the VU data register, respectively.

The **loadv** macro can be called from a C program like this:

```
loadv( df, source, "V5" )
```

Note: To reduce the number of quotation marks, CDPEAC defines macro symbols for all the VU data registers. For example, the **"V5"** string in the example above could be replaced by CDPEAC's literal **V5** symbol, which is defined as:

```
#define V5 "V5"
```

G.2.1 Going Generic with Macros

Because many DPEAC instructions have similar formats, one can define a generic macro in which the instruction opcode is also provided as an argument. For example:

```
#define mem(mnemonic, source, data_register) \
    asm(mnemonic " %0, " data_register : : \
         "m" (*source) : "%g2", "%g3")
```

CDPEAC defines a number of these internal, generic macros, and uses them to define the macros for the CDPEAC instruction set. The definitions of the loadv and storev instructions, for example, might be:

```
#define loadv(type, source, data_register) \
    mem(#type "loadv", source, data_register)

#define storev(type, source, data_register) \
    mem(#type "storev", source, data_register)
```

G.2.2 Handling Argument Syntax with Macros

DPEAC instructions often indicate special strides or modes by attaching markers to instruction operands. In CDPEAC, this is handled by macros that construct the appropriate DPEAC syntac. For example:

```
/* Data register offset syntax */
#define dreg_x(dreg, index) \
    dreg ## [ ## index ## ]

/* Data register indirection */
#define dreg_i(dreg, ireg) \
    dreg ## ( ## ireg ## )

/* Stride marker macros */
#define dreg_u(dreg, stride) \
    dreg ## : ## stride
#define dreg_s(dreg, setstride) \
    dreg ## := ## setstride
#define dreg_u_s(dreg, stride, setstride) \
    dreg ## : ## stride ## = ## setstride
```

(In these examples, the "**##**" operator is an ANSI C convention that concatenates the surrounding arguments together into a single C symbol, eliminating any space in between.)

G.2.3 CDPEAC: Macros on Macros

Similar definitions are used for the other elements of the CDPEAC instruction set, making it, in essence, a very large macro package.

Appendix H

CMRTS and CM Memory Allocation

This appendix describes the features of the CM Run-Time System (CMRTS) that you are likely to use in DPEAC and CDPEAC programming. It also discusses methods for allocating parallel CM memory, both through the CMRTS and by other means.

Note: To make direct use of the CMRTS functions and data structures described in this chapter, you must include the CMRTS header file in your program:

```
#include <cm/rts.h>
```

H.1 The CM Run-Time System (CMRTS)

The CMRTS is a set of low-level CM code libraries that define and manipulate array data structures in CM parallel memory. CMRTS functions allocate blocks of CM memory and manipulate their contents "at run time" (that is, during execution of CM programs), hence the name of the library. The CMRTS is divided into three main libraries of functions:

- CMRT — The CM Run-Time Library.
- CMCOM — The CM Communications Library.
- CMIP — The CM In-Processor Library.

The CMRT layer is the topmost layer of the RTS software, and represents an "external," machine-independent interface for the RTS. CMRT functions and data types provide access to all CM operations defined in the RTS. The CMCOM and CMIP layers are "internal" support software for the CMRT layer. CMCOM functions perform CM communication operations (sends, scans, etc.). CMIP functions perform in-processor operations (arithmetic and logical computations, etc., that don't require communication between CM processors). If your program makes any direct calls to the CMRTS, it is typically through the CMRT functions. However, references to data structures defined at the CMCOM level (in particular, CMCOM *machine geometries*) are common.

H.1.1 Arrays in the CMRTS

A high-level CM array, as defined in a parallel programming language such as CM Fortran or C*, is represented in the RTS by an *array descriptor*, of type **`CMRT_desc_t`**. This array descriptor is the topmost level of a hierarchy of data structures (see Figure 18) that form a bridge between high-level arrays and the physical memory of the CM hardware.

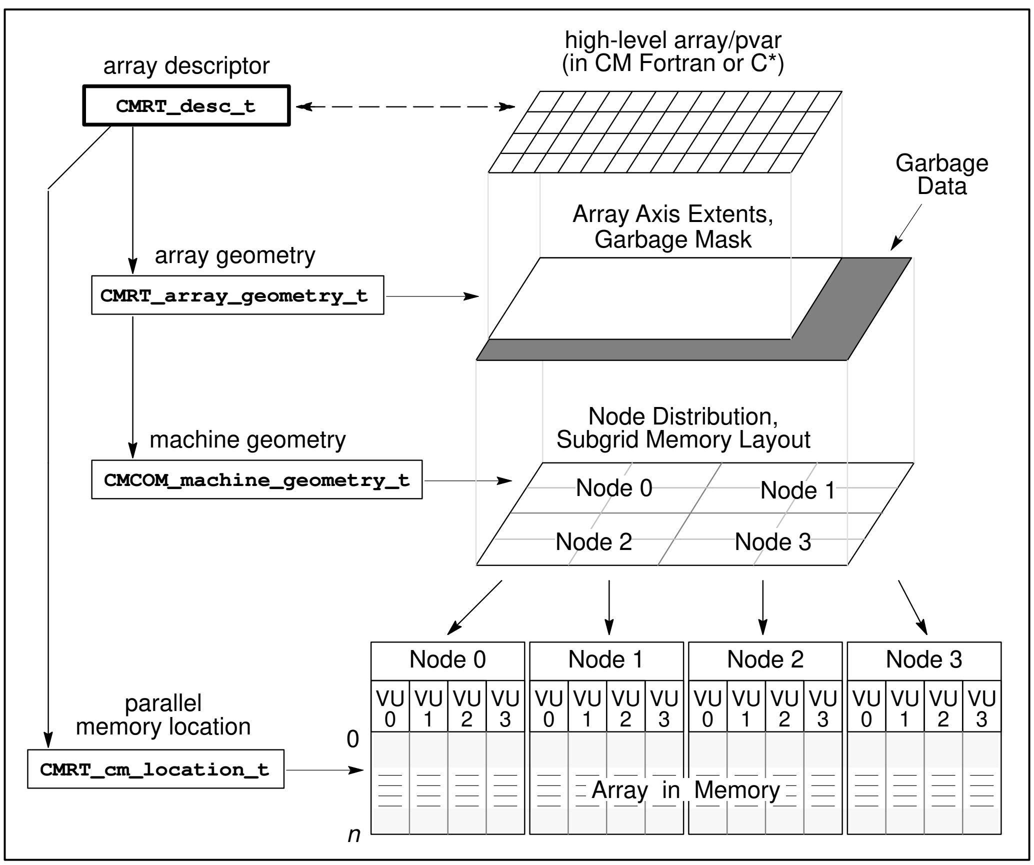


Figure 18. Internal Structure of a CM Array.

A high-level parallel array can have any reasonable shape and size, as permitted by the syntax of the programming language in use, and by the memory space available on the CM. However, the number of processing nodes available on the CM and the amount of memory available within each node typically remains fixed, placing a physical constraint on the sizes and shapes of arrays that can be conveniently stored in parallel memory.

The RTS allocates array memory in bands across the individual memory banks of the nodes, so that the starting memory address and size of the array region is the same for each node. (If the CM has vector units, these bands are across the individual memory banks of the VUs, as shown in the above figure.)

Implementation Note: The RTS memory allocation routines ensure that each allocated region of memory is double-word aligned (that is, starts at an address that is a multiple of 8 bytes).

An array with a number of elements that is an exact multiple of the number of processing nodes (or VUs) can be stored very neatly in such a memory stripe. But if the array is not an exact multiple of the machine size (or if the program that uses the array is run on a CM of a different size), then the array cannot be stored in CM memory without wasting some space on one or more of the nodes. This *garbage space* must be kept track of, so that the invalid values it contains are not confused with the actual data of the array.

The many-layered structure of RTS arrays deals with these hardware constraints by using data structures known as *geometries* to define the arrangement of array data in CM memory. Two types of geometry are used to define the layout of a high-level array:

- A *machine geometry*, of type **`CMCOM_machine_geometry_t`**, which describes the structure of an arbitrary array that is sized and shaped to fit exactly into a stripe of CM memory.

- An *array geometry*, of type **`CMRT_array_geometry_t`**, which refers to a specific machine geometry and selects a region of it to represent the actual data of a high-level array. (The unused space defined by the machine geometry is considered *garbage data*, and is ignored.)

A CM array descriptor (a **`CMRT_desc_t`** data structure) includes:

- An array geometry, which defines the layout of the array.
- A parallel memory location, of type **`CMRT_cm_location_t`**, which defines the start of the band of parallel memory that holds the array data.

For the Curious: This multi-layer array data structure saves storage space, since a single machine geometry data structure can be shared between the descriptors of many array geometries. It also makes it possible to use a simple pointer comparison to determine whether two arrays have the same machine geometry.

H.1.2 An Example of A CMRTS Array

Let's take a specific example. Suppose that we have a CM-5 with a very small number of processing nodes — four, to be exact (see Figure 19). This CM-5 has vector units, so that the number of *processing elements* in the machine is 16: four nodes with four VUs per node. (If this CM-5 did not have vector units, then there would be only four processing elements — the four processing nodes themselves.)

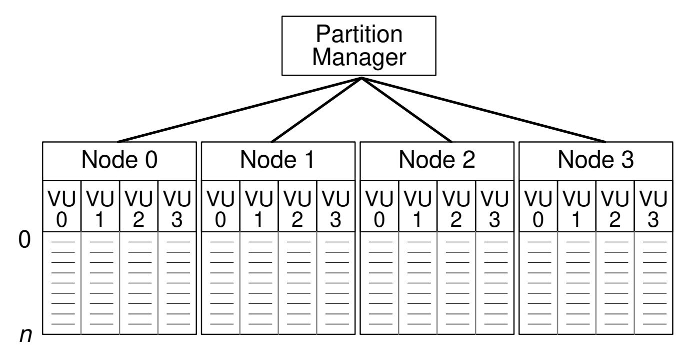
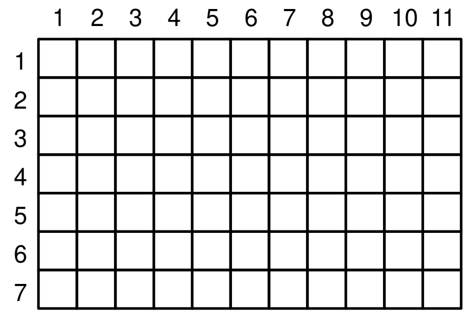


Figure 19. A hypothetical 4-node CM-5 system (with VUs).

Suppose further that we compile and run a CM Fortran program on this machine, defining a floating-point array as follows:

```
DOUBLE PRECISION LUCKY (7,11)
```

How is this array stored in the memory of the CM? First, look at the array itself:



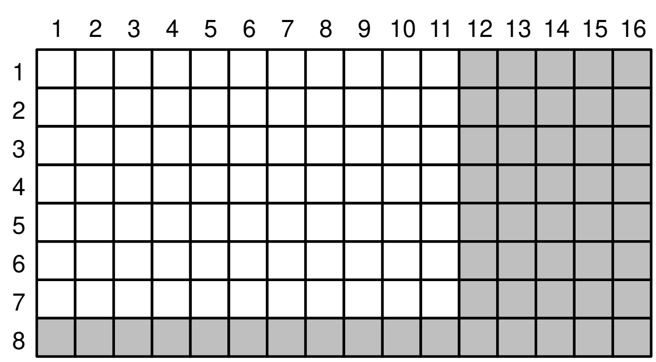
It's a two-dimensional array of 77 elements, each of which is a double-precision floating-point number. This is not evenly divisible across 16 VUs, so a small number of garbage elements will need to be added. In general, the garbage space of an array is added by extending the axes of the array, adding garbage elements at the high ends of one or more axes.

The amount of garbage space to add is determined as follows:

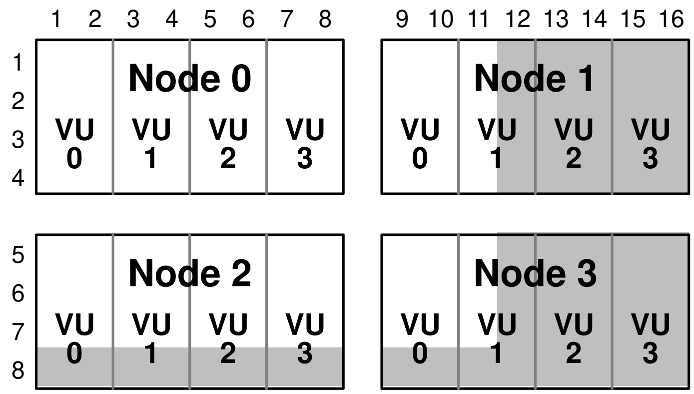
- Enough garbage space must be added so that the array can be divided into pieces of equal size and shape for each CM node.
- The part of the array assigned to each node must furthermore be divisible into 4 parts of equal size and shape, one for each of the node's VUs.
- Each node's portion of the array should be of the same rank as the entire array, and should have the same basic shape.
- The amount of added garbage space should be as small as possible.

In addition, there is an implementation-dependent restriction: the number of array elements assigned to each VU must be a multiple of 8. (In a forthcoming version of CM Fortran, you will be able to supply a compiler switch, **`-nopadding`**, which removes this restriction.)

Following these guidelines, a 7-by-11 array is padded out into an 8-by-16 array,



and is divided among the nodes and VUs as follows:

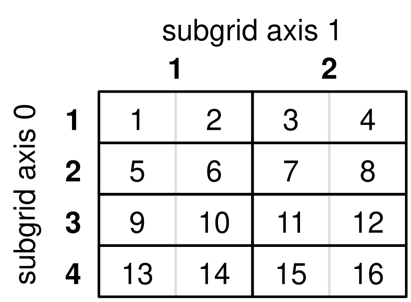


Note that this assigns a 4-by-2 portion of the array to each vector unit.

The portion of the array assigned to each vector unit is called the array's *subgrid*. This subgrid is the same size and shape for each VU, and basically represents the part of the array that is stored in the memory of each VU. The size of the subgrid determines the total amount of parallel memory allocated for the array. Exactly enough memory for one subgrid is allocated in the memory bank of each VU.

The subgrid of our sample array contains 8 double-precision floating-point values. A double-precision float in Fortran occupies two words of memory, so 16 words of parallel memory must be allocated on each CM node to contain the array. The *n*-dimensional subgrid stored on each VU is "unwrapped" and stored as a one-dimensional series of VU memory words.

As you might expect, this is done systematically by defining a single memory storage order for all subgrids. In the case of the sample (7,11) array, the memory storage order is as shown below (remember that each double-precision subgrid element is stored as 2 words in memory):



The resulting actual memory distribution of the array is shown below:

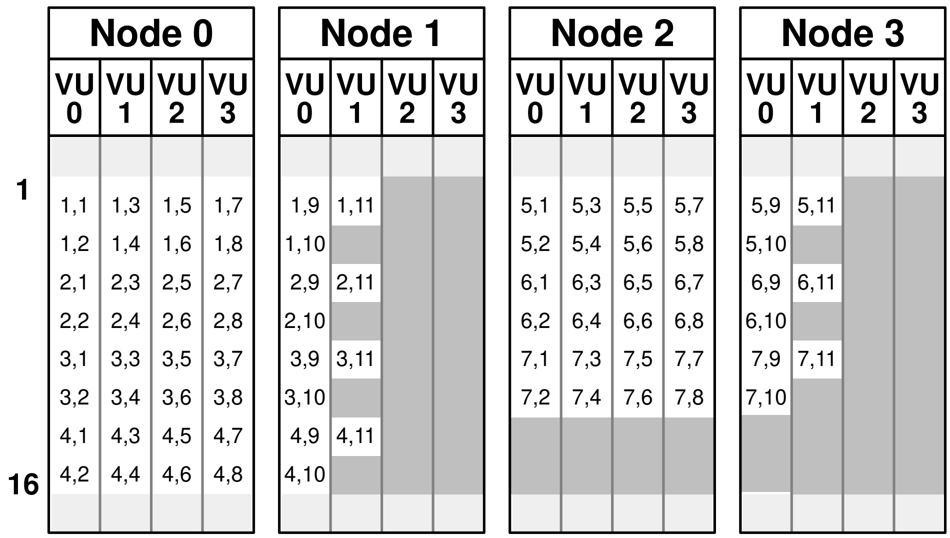


Figure 20. The actual memory distribution of a 7-by-11 CM Fortran array.

As you can see, this arrangement of array data values in CM memory bears almost no relation to the shape of the original high-level array. It is the array's geometry alone that determines how the array data in CM memory is interpreted. With a different array geometry, the values of this two-dimensional array could just as readily be accessed as a one-dimensional array or a three-dimensional array, by suitably adjusting the axis lengths.

A complete discussion of the algorithms used to determine the layout of a CM array in memory is beyond the scope of this appendix, but the above example should help you understand the information found in CMRTS array descriptors and geometry objects, as described in the following sections.

H.1.3 CMRTS Data Structures

CMRT_desc_t

This is the top-level array descriptor data structure. (See Figure 18.) It contains references to all component data structures that define a single high-level array.

The user-accessible structure slots are:

`CMRT_cm_location_t` **`cm_location`**

This is the parallel memory location (the same address on each VU) at which the array’s allocated memory region begins. The amount of memory allocated is determined by the array size and shape specified in the **`array_geometry`** structure.

`CMRT_array_geometry_t` **`array_geometry`**

This is the array geometry, which specifies the size, shape, and garbage region of the array. See the description of the **`CMRT_array_geometry_t`** data structure below.

`int4` **`element_size`**

This is the size, in words of memory, of a single array element.

The CMRTS functions used to access these structure slots are:

```
CMRT_cm_location_t CMRT_desc_get_cm_location(descriptor)
    CMRT_desc_t descriptor;

CMRT_array_geometry_t CMRT_desc_get_geometry(descriptor)
    CMRT_desc_t descriptor;

int4 CMRT_desc_get_element_size(descriptor)
    CMRT_desc_t descriptor;
```

CMRT_array_geometry_t

This is the array geometry, which specifies an array's extents, garbage space, and machine geometry (only the first two are directly specified by the array geometry; the machine geometry is specified by referring to a CMCOM data structure).

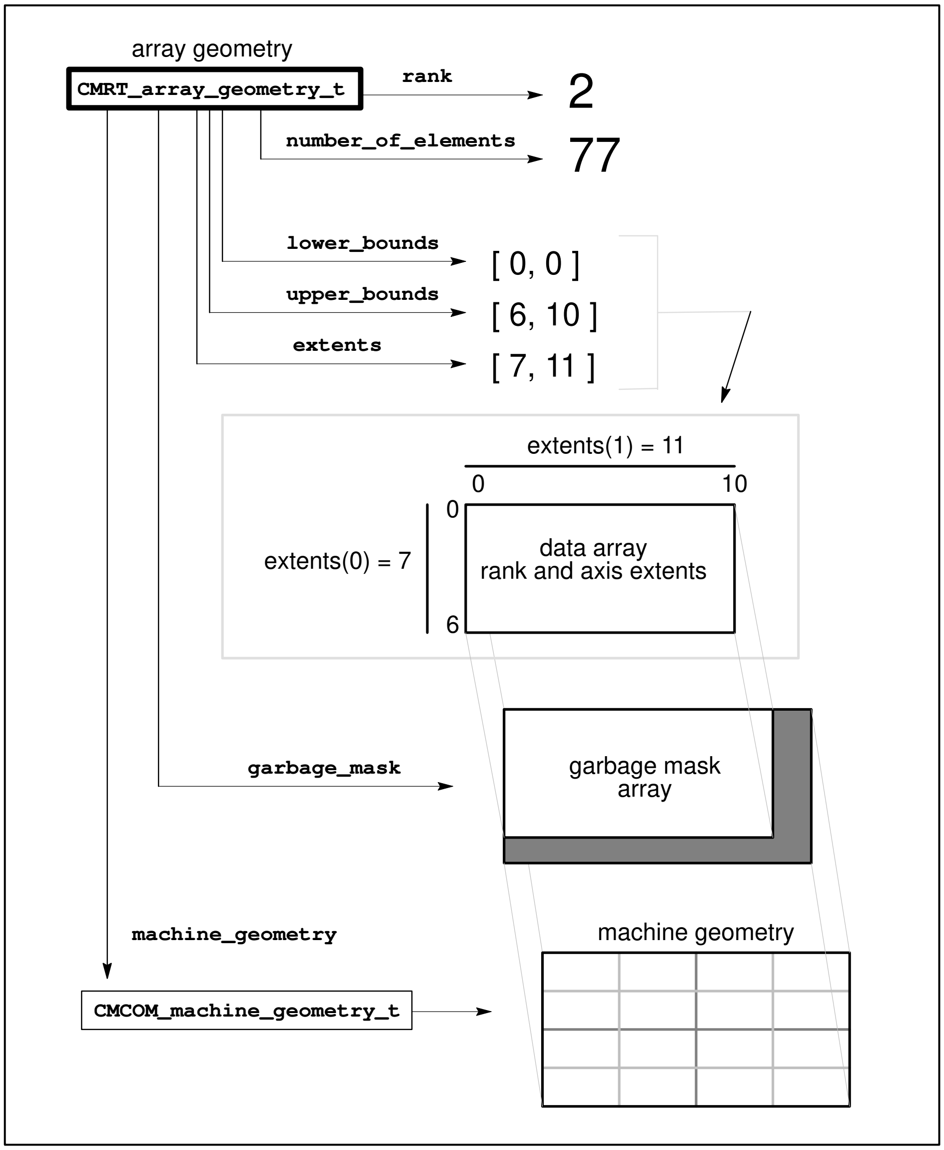


Figure 21. The CMRT_array_geometry data structure.

The user-accessible structure slots are:

int4 **rank**

This is the rank (number of dimensions) of the array. This is typically the same as the rank specified by the **machine_geometry**.

int8 **number_of_elements**

This is the total number of actual (non-garbage) data elements in the array (basically the product of the array's axis extents, defined below).

int8 ***extents, *lower_bounds, *upper_bounds**

These are integer arrays specifying the lengths and lower and upper axis indices for each array axis. Note that these values are completely independent of the array extents specified by the **machine_geometry** — the bounds arrays specify what part of the array is used for actual data.

Note: The lower and upper bound values in these arrays are *zero*-based, unlike Fortran array indices which are one-based. The CMRTS is coded in C, and thus follows the C conventions for array indexing.

CMCOM_machine_geometry_t **machine_geometry**

This is the machine geometry, which specifies the parallel memory layout for the array. See the description of the **CMCOM_machine_geometry_t** data structure below.

CMRT_cm_location_t **garbage_mask**

This is the *garbage mask* array, a region of parallel memory that defines the garbage space of the array. The garbage mask contains boolean values, with a **TRUE** value representing garbage elements. Essentially, the garbage mask provides the same information as the extents and bounds arrays described above, but in an array format that is more convenient for some CMRTS operations. The garbage mask is typically stored in a compressed format to save space, so extracting the appropriate boolean value for a given array element is non-trivial.

To access these slots, you can use the following accessor functions. (Note that these functions are applied to the *array descriptor*, not to the geometry object.)

```
int4 CMRT_desc_get_rank(descriptor)
    CMRT_desc_t descriptor;

int8 CMRT_desc_get_number_of_elements(descriptor)
    CMRT_desc_t descriptor;
```

(The following accessors take an extra **axis** argument, zero-based as in C. The appropriate value for the specified array axis is returned.)

```
int8 CMRT_desc_get_lower_bound(descriptor, axis)
    CMRT_desc_t descriptor;
    int4 axis;

int8 CMRT_desc_get_upper_bound(descriptor, axis)
    CMRT_desc_t descriptor;
    int4 axis;
```

You can also access the structure slots directly (you must do this to access the **extents** and **machine_geometry** slots, for example):

```
( array_descriptor->array_geometry ) -> extents[axis]

( array_descriptor->array_geometry ) -> machine_geometry
```

CMRT_machine_geometry_t

This is the machine geometry, which specifies the actual parallel memory layout of the array (the division of the array among the nodes and VUs, and the size and shape of the subgrid stored in each VU's memory).

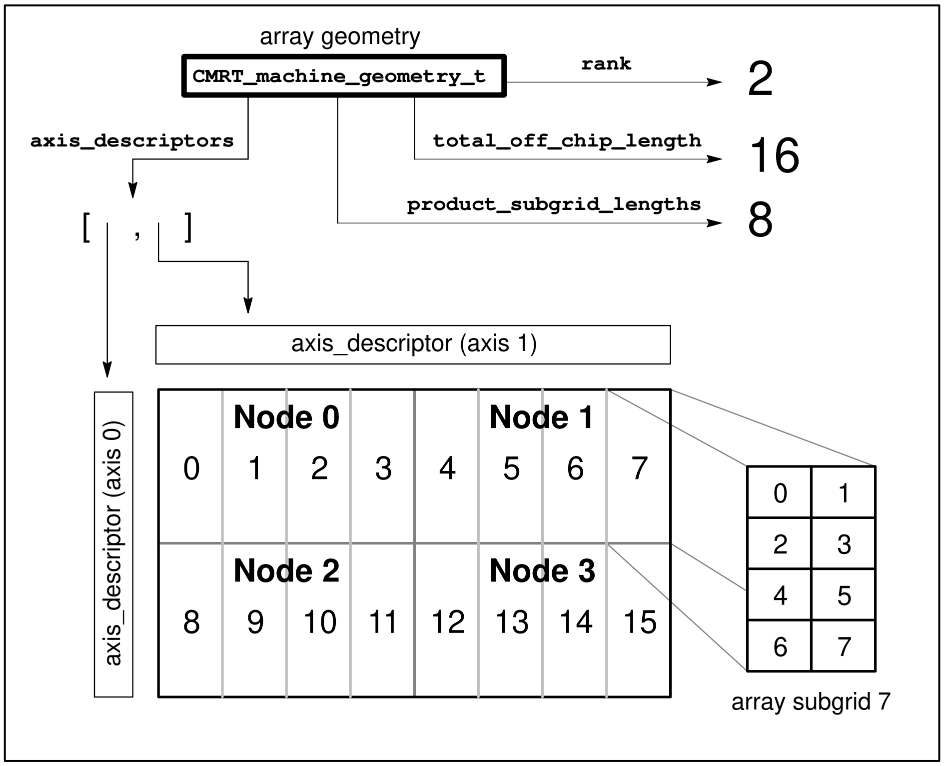


Figure 22. The CMCOM_machine_geometry data structure.

The user-accessible structure slots are:

`int4` **`rank`**

This is the rank (number of dimensions) of the array. This is typically the same as the rank specified by the array geometry.

`unsigned char` **`total_off_chip_length`**

This is the logarithm (base 2) of the total number of subgrids (specifically, this is the total number of physical, or "off-chip" bits required in send addresses of array elements). **Note:** While this value typically represents the number of VUs in your CM, it may sometimes be less. This is particularly the case for small arrays that do not use all of the nodes of the CM to store array data.

`int4` **`product_subgrid_lengths`**

This is the total number of elements in each subgrid (that is, the product of all the subgrid axis lengths).

`CMCOM_axis_descriptor *`**`axis_descriptors`**

This is an array of axis descriptor data structures, one for each axis of the array. See the description of the **`CMCOM_axis_descriptor`** data structure below.

To access the **`product_subgrid_lengths`** slot, you can use the following accessor function:

```
int4 CMRT_desc_get_subgrid_size(descriptor)
    CMRT_desc_t descriptor;
```

You can also access the structure slots directly (you must do this to access the remaining slots):

```
CMCOM_machine_geometry_t m_geometry =
  array_descriptor->array_geometry -> machine_geometry

m_geometry -> rank

m_geometry -> total_off_chip_length

m_geometry -> axis_descriptors[axis]
```

CMCOM_axis_descriptor

This is the array descriptor data structure, which defines the geometry information for a single axis of a machine geometry.

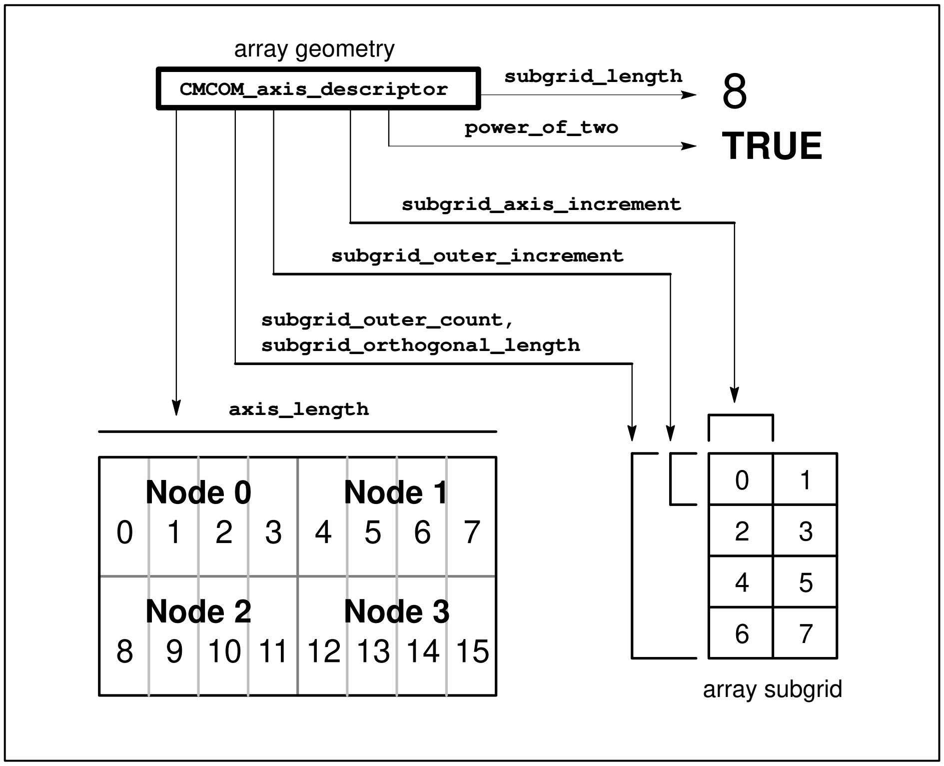


Figure 23. The CMCOM_axis_descriptor data structure.

The user-accessible structure slots are:

`int4` **`subgrid_length, power_of_two`**

The **`subgrid_length`** is the number of subgrid elements along the given axis. The **`power_of_two`** slot is a flag that is **`TRUE`** if and only if the subgrid length is an exact power of two.

`int8` **`axis_length`**

This is the total length (number of array elements) of the array axis. (This is basically the subgrid length along the given axis times the number of subgrids).

`int4` **`subgrid_axis_increment`**

This is the number of array elements in memory that must be skipped to move from each subgrid element along the given axis to the next element.

`int4` **`subgrid_outer_increment`**

This is the product of the **`subgrid_axis_increment`** and the **`subgrid_length`**; in other words the number of array elements in memory that must be skipped to move past *all* the subgrid elements along a single axis. (If **`subgrid_axis_increment`** is the distance between elements in a row, for example, then **`subgrid_outer_increment`** is the distance between the first elements of successive rows of the subgrid.)

`int4` **`subgrid_outer_count`**

This is the result of dividing the subgrid size (number of elements) by the value of **`subgrid_outer_increment`**. In other words, it is the number of iterations that would be needed to step through the entire subgrid using increments of **`subgrid_outer_increment`**. (This slot is used internally in the CMRTS to quickly calculate looping limits for operations that take place over the entire subgrid.)

`int4` **`subgrid_orthogonal_length`**

This is the product of the subgrid lengths of all other axes in the array. In other words, this is the number of subgrid elements in a single multi-dimensional "slice" through the array that is perpendicular to the given axis. (In a three-dimensional array, for example, this would be the number of rows and columns of elements in each horizontal "slice" of the vertical axis.)

(The remaining slots are used to specify the send addresses of array elements. These slots are not shown in Figure 23.)

```
unsigned char off_chip_positions, off_chip_length
unsigned4 off_chip_mask
```

These slots specify the physical, or "off-chip" part of an array element's send address that corresponds to the given axis. (Literally, these values are the starting position of the physical, or "off-chip" bits assigned to the axis, the number of bits assigned to the axis, and a binary mask that selects only those bits from a send address.)

```
unsigned char subgrid_bits_position, subgrid_bits_length;
int4 subgrid_bits_mask;
```

These slots specify the subgrid part of an array element's send address that corresponds to the given axis. (Literally, these values are the starting position of the subgrid bits assigned to the axis, the number of bits assigned to the axis, and a binary mask that selects only those bits from a send address.)

Note: The **subgrid_bits** slots are only valid if the **power_of_two** slot is **TRUE** — in other words, if the number of elements in the subgrid is an exact power of two.

To access the **subgrid_length** slot, you can use this accessor function:

```
int4 CMRT_desc_get_subgrid_dimension(descriptor, axis)
    CMRT_desc_t descriptor;
    int4 axis;
```

You can also access the structure slots directly (you must do this to access the remaining slots):

```
CMCOM_machine_geometry_t m_geometry =
  array_descriptor->array_geometry -> machine_geometry

CMCOM_axis_descriptor axis_d =
  m_geometry -> axis_descriptors[axis]

axis_d -> axis_length

axis_d -> subgrid_axis_increment

axis_d -> subgrid_outer_count
```

H.2 CMRTS Parallel Memory Allocation

For some special-purpose applications, it is necessary to allocate parallel CM memory other than by using a high-level language to define an array. (For example, you may need to allocate memory to hold a temporary value on each node.) To do this, you can use the memory allocation functions of the CMRTS.

H.2.1 Standard CMRTS Memory Allocation Functions

As described in Appendix A, parallel VU memory is mapped into two general regions of memory, the parallel stack and the parallel heap. Both the stack and the heap regions grow upward, toward higher memory addresses. When you allocate new space in these regions, it is allocated as a stripe across the physical memory of the VUs.

There are two ways to allocate either stack or heap memory space: by a *physical* block of memory, or by a *geometry* block of memory. The difference is essentially one of convenience. If you know exactly how many bytes of memory you want to allocate on each VU, use the physical memory allocation functions:

```
CMRT_cm_location_t
  CMRT_allocate_physical_stack_field(num_bytes)
    int4 num_bytes;

CMRT_cm_location_t
  CMRT_allocate_physical_heap_field(num_bytes)
    int4 num_bytes;
```

Both functions take a number of bytes as an argument, and return a **CMRT_cm_location_t** pointer to the allocated stack or heap memory region. To deallocate a memory region allocated in this way, use the functions:

```
CMRT_deallocate_physical_stack_through(field, num_bytes)
    CMRT_cm_location_t field;
    int4 num_bytes;

CMRT_deallocate_physical_heap_field(field, num_bytes)
    CMRT_cm_location_t field;
    int4 num_bytes;
```

These functions take a starting address and a number of bytes, and return the indicated space to free storage. (Note that deallocating a stack field implicitly deallocates all stack fields with higher stack addresses.)

If, on the other hand, you have a specific **CMRT_array_geometry_t** and want to allocate enough memory to give each element of the geometry a specific number of bytes, you should use these allocation functions:

```
CMRT_cm_location_t
  CMRT_allocate_stack_field(geometry, num_bytes)
    CMRT_array_geometry_t geometry;
    int4 num_bytes;
CMRT_cm_location_t
  CMRT_allocate_heap_field(geometry, num_bytes)
    CMRT_array_geometry_t geometry;
    int4 num_bytes;
```

Both functions take a geometry object and a number of bytes, and return a **CMRT_cm_location_t** pointer to the allocated stack or heap region. The difference between these functions and the physical ones shown above is that these functions allocated the specified number of bytes for *each element* of the array’s subgrid — the **num_bytes** argument is multiplied by the array’s subgrid size. Thus, the following equivalences hold:

```
CMRT_allocate_stack_field(geometry, num_bytes) ==
CMRT_allocate_physical_stack_field (num_bytes *
   geometry->machine_geometry->product_subgrid_lengths)

CMRT_allocate_heap_field(geometry, num_bytes) ==
CMRT_allocate_physical_heap_field (num_bytes *
   geometry->machine_geometry->product_subgrid_lengths)
```

To deallocate these fields, you can use the following deallocation functions:

```
CMRT_deallocate_stack_field_through
  (field, geometry, num_bytes)
    CMRT_cm_location_t field;
    CMRT_array_geometry_t geometry;
    int4 num_bytes;
CMRT_deallocate_heap_field
  (field, geometry, num_bytes)
    CMRT_cm_location_t field;
    CMRT_array_geometry_t geometry;
    int4 num_bytes;
```

H.2.2 Node-Level Stack Operations

When you are writing a node-level routine that is to be called as part of a global program, you can also use CMRTS functions to allocate temporary stack space independently on each node. The only condition to this is that you must ensure that the allocated space is freed before your node-level routine returns.

The following routines can be used at this level:

```
CMCOM_cm_address_t CMCOM_pe_get_stack_pointer ()

CMCOM_cm_address_t
  CMCOM_pe_allocate_stack_space (nbytes)
    int4 nbytes;

CMCOM_pe_set_stack_pointer (new_sp)
    CMCOM_cm_address_t new_sp;
```

The proper (and recommended) method to do this is:

- At the start of a node-level subroutine, get the current value of the stack pointer and store it in a temporary variable:

```
CMCOM_cm_address_t temp;
temp = CMCOM_pe_get_stack_pointer();
```

- When you need to allocate stack space, call the allocation function:

```
space = CMCOM_pe_allocate_stack_space (nbytes);
```

- At the end of the routine (or before any return point), free all allocated stack space by resetting the stack pointer to its original value.

```
CMCOM_pe_set_stack_pointer (temp);
```

Important: This method is only applicable for node routines that are called directly as part of a global (PM and nodes) program. If you are running code under the global/local version of the RTS, in which each node is treated as a parallel machine in and of itself, you can make calls to the standard RTS memory allocation routines as described in Section H.2.1 above. These will work in either the global or the local parts of a global/local program.

H.3 Non-RTS (CMMD) Parallel Memory Allocation

For some applications (in particular, when writing DPEAC or CDPEAC code that is to be called from CMMD message-passing routines), it is necessary to allocate parallel CM memory without using the standard memory allocation and deallocation routines provided by the CMRTS. Methods for allocating parallel memory without use of the CMRTS are described in the sections below.

Important: The methods described below must *not* be used in any application that makes calls to the CMRTS — directly accessing the stack and heap pointers as described here is incompatible with the CMRTS memory management code.

H.3.1 Parallel Memory Addressing

Using the memory allocation routines described here requires that you refer to memory regions by their actual memory addresses (as opposed to using a memory location data type, such as **`CMRT_cm_location_t`**, as a "handle").

Parallel memory locations are referenced by their *all-VU, instruction space* address. This is the address in the region of VU memory that causes all four VUs to execute a DPEAC instruction simultaneously. Both CMRTS routines and CMMD routines take addresses of this type as arguments.

When you are using and manipulating these kinds of addresses, whether you are coding in DPEAC or CDPEAC, you should include this header file:

```
#include <cmsys/dp.h>
```

This header file defines a number of symbolic constants that are helpful in constructing and interpreting addresses in the VU memory regions.

For example, the base of the parallel stack (in all-VU instruction space) is given by the symbol **`DPV_STACK_INST_PORT_ALL`** (**`0x50000000`**), and the base of the parallel heap region is given by the symbol **`DPV_HEAP_INST_PORT_ALL`** (**`0x70000000`**). You can construct an address within these regions by adding a byte offset to these base addresses.

Important: Before you can access a stack or heap word, the memory region must have been expanded to include the address (that is, you must allocate the memory before you can legally access it).

H.3.2 Expanding the Stack or Heap

When you want to expand the stack or heap, you make a CMOST system call to manipulate the pointer of the appropriate memory region. You can do this either from the partition manager or from a processing node. If you do this from a node, only one processing node must (and should) make the allocation call. To access the appropriate CMOST routines, include the header file:

```
#include <cmsys/cm_memory.h>
```

The memory pointer system calls from the partition manager are:

```
CM_memaddr_t
  CM_set_dp_stack_ptr (CM_memaddr_t new_limit)
CM_memaddr_t
  CM_set_dp_heap_ptr (CM_memaddr_t new_limit)

CM_memaddr_t  CM_get_dp_stack_ptr ()
CM_memaddr_t  CM_get_dp_heap_ptr ()
```

The equivalent calls from the node are:

```
CM_memaddr_t
  CMPE_set_dp_stack_ptr (CM_memaddr_t new_limit)
CM_memaddr_t
  CMPE_set_dp_heap_ptr (CM_memaddr_t new_limit)

CM_memaddr_t CMPE_get_dp_stack_ptr ()
CM_memaddr_t CMPE_get_dp_heap_ptr ()
```

All these routines return a **`CM_memaddr_t`** value, which is an *all-VU, instruction space* address, representing the current position of the memory pointer (in the case of the **`set`** routines, this is the value of the pointer after you have modified it). The value of the pointer is always one more than the highest allocated address in the memory region.

You cannot access allocated memory using the **`CM_memaddr_t`** values returned from these system calls, because they are in all-VU instruction space. You must translate this value into a *single-VU, data space* pointer, as described in Section H.3.3 below.

To use the **`set`** system calls, you pass in the highest address that you want to have allocated. The pointer value the call returns will always be greater than this value (unless there is insufficient memory remaining, in which case zero is returned), but it may not be exactly one more than the address you passed in.

Important: Don't make a "copy" of the stack or heap pointer and expect the copy to remain valid. Stack and heap memory can be allocated for reasons other than explicit system calls from your program. Thus, the stack and heap pointers can change without warning. You should always use the current value returned by the system calls mentioned above when determining the current size of the stack or heap.

If you want to deallocate parallel memory (in other words, shrink the stack or heap regions), call the appropriate **set** function with the new lower limit.

Note: CMOST currently does not allow the regions to shrink, and thus the call described above will have no effect, and the current limit will be returned. Nevertheless, it is sensible to include deallocation calls, for compatibility with later software versions.

H.3.3 Translating Stack and Heap Addresses

You can change **CM_memaddr_t** values into valid data space addresses using the following C macro, which is defined in **cmsys/dp.h**:

```
data_address = TOGGLE_DPV_SPACE(instruction_address);
```

Note that the returned data space address is still an *all-VU* address. It cannot be used to read from memory, and if used to store to memory, the stored value will be written to all four VUs (broadcast).

You can change the data space address to point to a single VU by using one of the following macros:

```
VU_0_address = CHANGE_DP(data_address, DP_0);
VU_1_address = CHANGE_DP(data_address, DP_1);
VU_2_address = CHANGE_DP(data_address, DP_2);
VU_3_address = CHANGE_DP(data_address, DP_3);
```

The resulting addresses are pointers to words (or doublewords) in stack or heap memory and can be used, for example, as a C pointer value to read or write memory values.

Note: Parallel memory accessed by the node processor is always mapped with caching *disabled*. Thus, access to words/doublewords in the above fashion will be 2 to 3 times slower than normal cached accesses. Also, all attempts to read/write parallel memory using pointers that are not word-aligned will result in memory faults.