The Connection Machine System

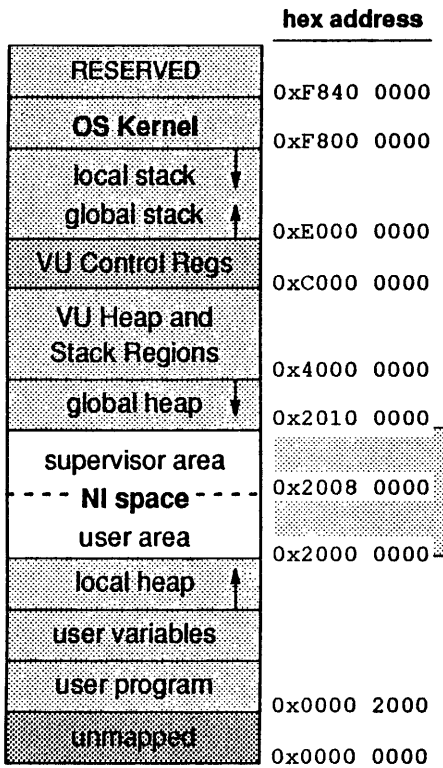
CM5

VU Programmer's Handbook

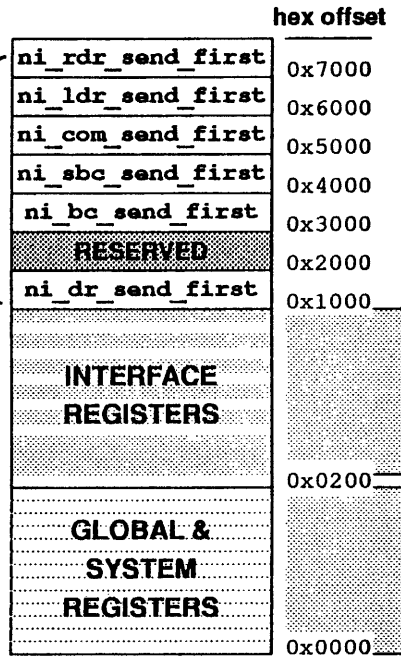
CMOST Version 7.2

Thinking Machines Corporation

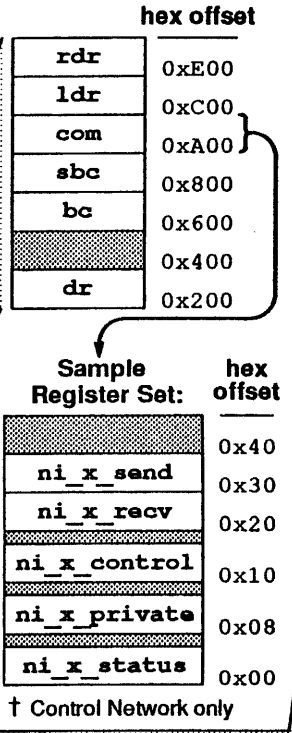
Node Virtual Memory Map
(with or without VUs installed)



NI Virtual Memory Area
(user or supervisor)



Interface Registers

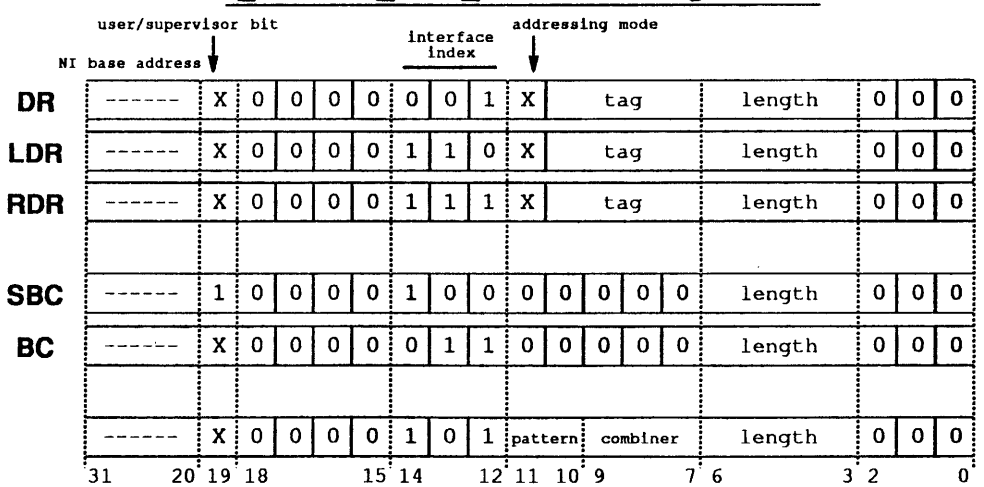


Global & System Registers

		hex offset
	RESERVED	0x180
		0x0F0
*	ni_bad_address	0x0E8
	ni_scan_start	0x0E0
	ni_interrupt_now	0x0D8
*	ni_interrupt_clear_green	0x0D0
*	ni_interrupt_clear	0x0C8
	ni_sync_global_send	0x0C0
*	ni_hodgepodge	0x0B8
*	ni_async_sup_global	0x0B0
*	ni_async_global	0x0A8
	ni_com_flush_send	0x0A0
	ni_sync_global_abstain	0x098
*	ni_sync_global	0x090
	ni_serial_number	0x088
	ni_interrupt_send	0x080
	ni_configuration	0x078
	ni_time	0x070
		0x068
	ni_user_tag_mask	0x060
	ni_rec_interrupt_mask	0x058
	ni_count_mask	0x050
	ni_dr_message_count	0x048
	ni_chunk_size	0x040
	ni_chunk_table_data	0x038
	ni_chunk_table_address	0x030
	ni_partition_size	0x028
	ni_partition_base	0x020
	ni_physical_self	0x018
*	ni_interrupt_level	0x010
*	ni_interrupt_cause_green	0x008
*	ni_interrupt_cause	0x000

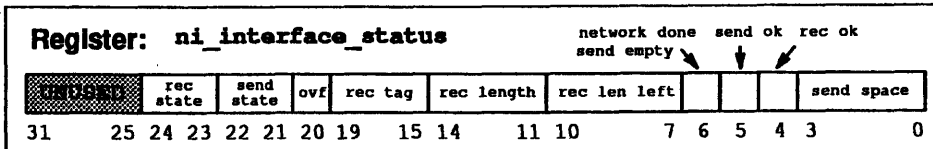
* Indicates register with subfields
(See listings on reverse side)

ni_interface_send_first Addressing Patterns



Register & Memory Map

Register: ni_interface_status

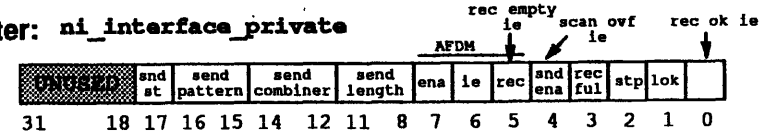


Field Name:	Pos:	Size:	DR	L/RDR	S/BC	COM
ni_send_space	0	4	√	√	√	√
ni_rec_ok	4	1	√	√	√	√
ni_send_ok	5	1	√	√	√	√
ni_router_done_complete	6	1	√			
ni_send_empty	6	1			√	√
ni_rec_length_left	7	4	√	√	√	√
ni_rec_length	11	4	√	√		√
ni_dr_rec_tag	15	4	√	√		
ni_com_scan_overflow	20	1				√
ni_dr_send_state	21	2	√			
ni_dr_rec_state	23	2	√			

Register: ni_interface_control

Field Name:	Pos:	Size:	DR	L/RDR	S/BC	COM
ni_rec_abstain	0	1			√	√
ni_reduce_rec_abstain	1	1				√

Register: ni_interface_private



Field Name:	Pos:	Size:	DR	L/RDR	S/BC	COM
ni_rec_ok_ie	0	1	√	√	√	√
ni_lock	1	1	√	√	√	√
ni_rec_stop	2	1	√			√
ni_send_stop	2	1			√	
ni_rec_full	3	1	√	√	√	√
ni_send_enable	4	1			√	
ni_com_scan_overflow_ie	4	1				√
ni_dr_rec_all_fall_down	5	1	√	√		
ni_com_rec_empty_ie	5	1				√
ni_all_fall_down_ie	6	1	√			
ni_all_fall_down_enable	7	1	√			
ni_com_send_length	8	4				√
ni_com_send_combiner	12	3				√
ni_com_send_pattern	15	2				√
ni_com_send_start	17	1				√

Registers: (same bit positions, all flags)
ni_interrupt_cause/clear

Field Name:	Pos:
ni_cause/clear_internal_fault	0
ni_cause/clear_mc_error	1
ni_cause/clear_cmu_error	2
ni_cause/clear_bc_interrupt_red	3
ni_cause/clear_cn_checksum_error	4
ni_cause/clear_cn_hard_error	5
ni_cause/clear_dr_checksum_error	6
ni_cause/clear_timer_interrupt	7
ni_cause/clear_bc_interrupt_orange	8
ni_cause/clear_bc_interrupt_yellow	9
ni_cause/clear_bc_or_com_collision	10
ni_cause/clear_com_abstain_changed	11
ni_cause/clear_dr_count_negative	12
ni_cause/clear_bad_relative_address	13
ni_cause/clear_bad_memory_access	14

Register: ni_hodgepodge

Field Name:	Pos:	Size:
ni_global_rec_ie	0	1
ni_supervisor_global_rec_ie	1	1
ni_flush_complete	2	1
ni_interrupt_send_ok	3	1
ni_configuration_complete	4	1
ni_interrupt_rec_enable	5	1
ni_sync_global_rec_ie	6	1
ni_timer_ie	7	1
ni_cn_stop_send	8	1

Register: ni_sync_global

Field Name:	Pos:	Size:
ni_sync_global_rec	0	1
ni_sync_global_complete	1	1

Register: ni_async_global

Field Name:	Pos:	Size:
ni_global_send	0	1
ni_global_rec	1	1

Register: ni_async_sup_global

Field Name:	Pos:	Size:
ni_supervisor_global_send	0	1
ni_supervisor_global_rec	1	1

Registers: (same bit positions, all flags)
ni_interrupt_cause/clear_green

Field Name:	Pos:
ni_cause/clear_bc_interrupt_green	0
ni_cause/clear_scan_overflow	1
ni_cause/clear_bc_rec_ok	2
ni_cause/clear_sbc_rec_ok	3
ni_cause/clear_com_rec_ok	4
ni_cause/clear_com_rec_empty	5
ni_cause/clear_sync_global_rec	6
ni_cause/clear_global_rec	7
ni_cause/clear_supervisor_global_rec	8
ni_cause/clear_dr_rec_ok	9
ni_cause/clear_ldr_rec_ok	10
ni_cause/clear_rdr_rec_ok	11
ni_cause/clear_dr_rec_tag	12
ni_cause/clear_dr_rec_all_fall_down	13

Register: ni_interrupt_level

Field Name:	Pos:	Size:
ni_interrupt_level_green	0	1
ni_interrupt_level_yellow	8	1
ni_interrupt_level_orange	16	1
ni_interrupt_level_red	24	1

Register: ni_bad_address

Field Name:	Pos:	Size:
ni_bad_address_low	0	20
ni_bad_address_type	20	12

The Connection Machine System

VU Programmer's Handbook

CMOST Version 7.2,
August 1993

Thinking Machines Corporation

First printing, August 1993

Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142–1264
(617) 234–1000

Contents

Appendixes

About This Manual

Objectives

This manual provides a concise collection of the information required for writing programs that directly access the CM-5 vector unit (VU) accelerators. There are two low-level instruction sets available on the CM-5: DPEAC and CDPEAC. A program that directly manipulates the VU accelerators will typically include subroutines written in either DPEAC or CDPEAC. Both methods of programming the vector unit accelerators are described in this manual.

IMPORTANT

You do *not* have to use the methods described in this book to write programs that access the CM-5 VUs. The compilers for high-level CM languages (such as CM Fortran and C*) automatically take advantage of the VUs where possible, without the need for explicit instructions. The information presented here is intended for knowledgeable users who want to hand-code specific low-level subroutines for execution on the VUs.

Intended Audience

This is a programmer's handbook, not a tutorial. This document describes the DPEAC and CDPEAC instruction sets in detail, and provides some examples of their use, but is intended to be used by knowledgeable CM programmers in writing low-level code. For the most part, this handbook contains concise summaries of information that these low-level programmers will find helpful.

Revision Information

This is a new manual.

Organization of This Manual

Chapter 1 Introduction
Presents an overview of the CM-5 and the two low-level instruction sets DPEAC and CDPEAC.

Chapter 2 The CM-5 Vector Units
Describes the design and features of the CM-5's vector unit accelerators.

Chapter 3 The DPEAC Instruction Set
Explains the syntax and structure of the DPEAC instruction set.

Chapter 4 DPEAC Instruction Set Reference
Lists the arithmetic, memory, modifier, and accessor instructions of the DPEAC instruction set.

Chapter 5 The CDPEAC Instruction Set
Explains syntax and structure of the CDPEAC instruction set.

Chapter 6 CDPEAC Instruction Set Reference
Lists the arithmetic, memory, modifier, and accessor instructions of the CDPEAC instruction set.

Chapter 7 Using DPEAC/CDPEAC in Programs
Presents an example of using a DPEAC (or CDPEAC) subroutine in a CM Fortran program.

Appendixes:

Appendix A VU Memory Mapping
Explains the layout of VU parallel memory.

Appendix B VU Memory Maps
A three-page VU memory map and register quick-reference.

Appendix C VU Pipeline
Describes of the operation of the VU instruction pipeline, and its effects on execution of VU vector instructions.

Appendix D VU Arithmetic Operations
Describes the arithmetic instruction set of the VUs, with special emphasis on the status bits that are modified by each instruction.

Appendix E The dpas Assembler
Describes `dpas`, the DPEAC assembler.

Appendix F The dpcc Compiler
Describes `dpcc`, the CDPEAC compiler.

Appendix G How CDPEAC Works
Describes the implementation of CDPEAC via the GCC compiler's `asm` statement and macro facility.

Appendix H CMRTS and CM Memory Allocation
Describes the CM Run-Time system, CM parallel array data structures, and methods for allocating parallel memory either through the CMRTS or by other means.

Related Documents

These documents are part of the Connection Machine documentation set.

- *Programming the NI*, Version 7.1.
- *DPEAC Reference Manual*, CMOST Version 7.1.

Notation Conventions

The table below displays the notation conventions observed in this manual.

Convention	Meaning
bold typewriter	UNIX and CM System Software commands, command options, and filenames, when they appear embedded in text. Also programming language elements, such as keywords, operators, and function names, when they appear embedded in text.
italics	Argument names and placeholders in function and command formats.
`typewriter`	Code examples and code fragments.
`% bold typewriter` `regular typewriter`	In interactive examples, user input is shown in **bold typewriter** and system output is shown in `regular typewriter` font.

Customer Support

Thinking Machines Customer Support encourages customers to report errors in Connection Machine operation and to suggest improvements in our products.

When reporting an error, please provide as much information as possible to help us identify and correct the problem. A code example that failed to execute, a session transcript, the record of a backtrace, or other such information can greatly reduce the time it takes Thinking Machines to respond to the report.

If your site has an applications engineer or a local site coordinator, please contact that person directly for support. Otherwise, please contact Thinking Machines' home office customer support staff:

Internet Electronic Mail:	`customer-support@think.com`
uucp Electronic Mail:	`ames!think!customer-support`
U.S. Mail:	Thinking Machines Corporation Customer Support 245 First Street Cambridge, Massachusetts 02142-1264
Telephone:	(617) 234-4000

Chapter 1

Introduction

1.1 Programming the CM-5 Vector Units (VUs)

Writing a program that takes explicit control of the vector unit (VU) accelerators of the Connection Machine CM-5 system requires an understanding of the CM-5's hardware design (in particular, the design and function of the VUs themselves), and how to construct programs that contain assembly-level CM-5 code.

This chapter presents a brief overview of the CM-5's hardware design, along with a description of the assembly-level instruction sets (DPEAC and CDPEAC) that are available on the CM-5.

IMPORTANT

You do *not* have to use the methods described in this book to write programs that access the CM-5 VUs. The compilers for high-level CM languages (such as CM Fortran and C*) automatically take advantage of the VUs where possible, without the need for explicit instructions. The information presented here is intended for knowledgeable users who want to hand-code specific low-level subroutines for execution on the VUs.

1.2 The CM-5 Hardware

The CM-5 computing environment consists of a *partition* of processing nodes (each of which has its own memory) together with a *partition manager* (PM). These components are linked together by the CM-5's internal communication networks: the *Data Network* and *Control Network* (see Figure 1).

Depending on how the CM-5's processing nodes have been configured by the system administrator, there may be one or several partitions active in a CM-5 at any one time. A partition of processing nodes is treated as a single computing system for the purpose of assigning and swapping processes.

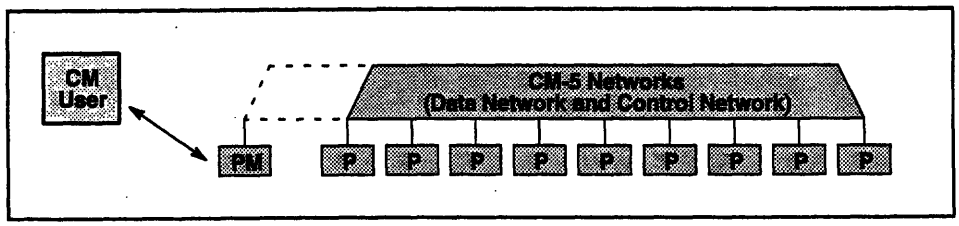


Figure 1. The CM-5 computing environment.

The partition manager (PM) contains a RISC CPU and connecting hardware that allows the PM to interact with other computers and with users on terminals. Thus, the PM is the "gateway" by which a programmer gains access to the processing nodes of the CM-5 and instructs the CM-5 to execute a program.

1.2.1 The CM-5 Networks

The CM-5's processors can exchange information with each other through the machine's internal networks.

The *Data Network* is a high-speed, high-bandwith network for data transmission. It is the primary means for sending large blocks of information between the nodes and/or the PM.

The *Control Network* is a high-speed internal network for control functions, such as broadcasting a value to the nodes, parallel-prefix computations, and node synchronization.

1.2.2 The CM-5 Processing Nodes

A CM-5 processing node consists of a RISC processor, a Network Interface chip, 4 memory units, 4 vector unit arithmetic accelerators, and a 64-bit MBUS that links the various components together.

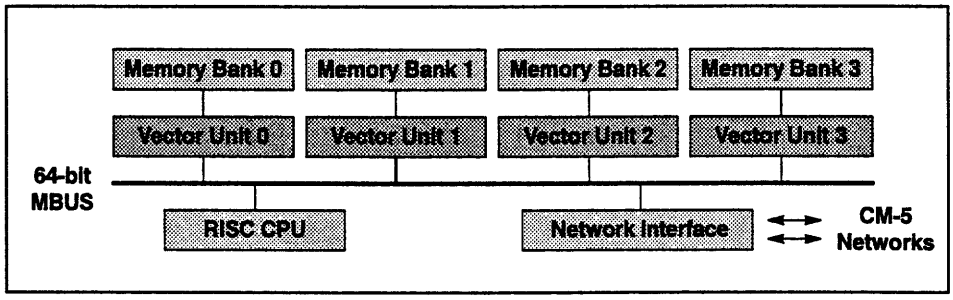


Figure 2. A typical CM-5 processing node.

The RISC processor (CPU) is a SPARC chip in the current implementation, and will hereafter be referred to as the "SPARC" or the "SPARC CPU".

The Network Interface (NI) is the node's link to the CM-5 networks, and is used by the SPARC IU to send messages to other nodes and to the PM.

1.2.3 The CM-5 Vector Units

The vector unit (VU) accelerators are located between the SPARC CPU and node memory, and typically act as memory controllers, handling memory store and fetch operations as required by the SPARC.

However, some memory operations are interpreted as instructions by the VUs: the value written is interpreted as a VU arithmetic and/or memory instruction, and the address to which it is written determines which of the four VUs on the node will execute the instruction. Thus, VU computations are invoked by (and look like) SPARC memory operations.

Chapter 2 provides more detail on the vector units, and describes features of their internal design that are important for DPEAC and CDPEAC programmers.

1.3 The DPEAC and CDPEAC Instruction Sets

The Connection Machine system provides a number of layers of software that are used to write CM-5 programs. The basic structure is shown below. Typically, a user-written CM-5 program depends on high-level software for the majority of its data structures and control flow, and only directly calls low-level code for hand-crafted subroutines that must execute as efficiently as possible.

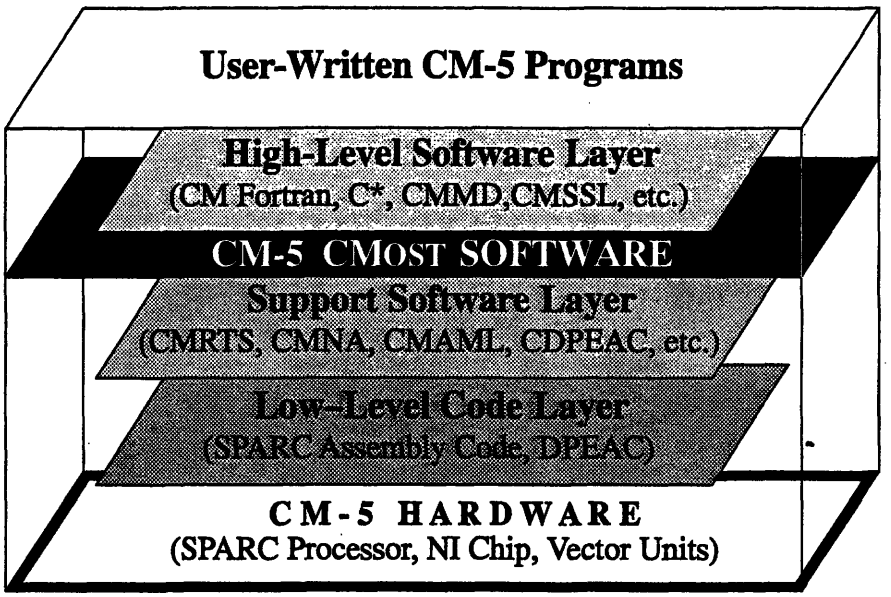


Figure 3. Structure of software layers on the CM-5.

Note: There is nothing inherently inefficient about a program written in a high-level CM-5 programming language. The CM-5 language compilers themselves make use of efficient low-level routines wherever possible.

1.3.1 The CM-5 Assembly Code Level

Because the instruction units of the CM-5 processing nodes are SPARC chips, the SPARC assembler instruction set is the CM-5's "native" machine language instruction set. However, there is an entirely different instruction set used to compose instructions for the CM-5's vector units. This instruction set is called *DPEAC*. There is also a C interface to DPEAC, called *CDPEAC*. Which instruction set you use depends on your experience and programming needs.

1.3.2 DPEAC — Vector Unit Assembly Code

The DPEAC instruction set is the "assembly code" of the CM-5 vector unit accelerators. DPEAC looks much like standard assembly code, in that it consists of instructions that perform arithmetic and memory operations:

```
Start:   floadv    [%i0]:4, V2
         fmulv     V2, 0r3.69, V3
         fmadav    V2, 0r25.0, V4
         floadv    [%i1]:4, V5; fmadav V3,V4,V5
         fstorev   [%i2]:4, V5
```

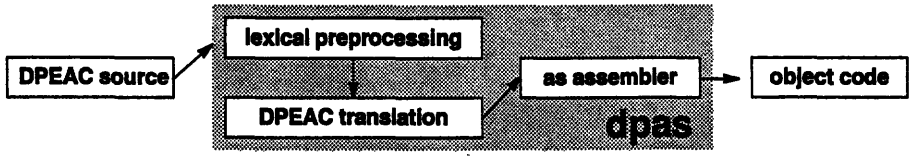
However, DPEAC instructions are not executed directly by the SPARC. Instead, they are assembled into singleword or doubleword values that can be written to the VUs to cause them to execute the appropriate arithmetic and/or memory operations.

DPEAC code and SPARC assembly code can be intermixed freely; the SPARC code is executed by the SPARC processor, and the DPEAC code is sent to the VUs for execution.

Coding in DPEAC is best for a programmer with some experience in coding at the assembly-code level. It requires skill in managing the SPARC registers and a firm knowledge of the SPARC ABI calling conventions, which describe how subroutines pass values to each other at the SPARC assembly code level.

dpas — The DPEAC Assembler

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



For a more detailed description of the **dpas** assembler, see Appendix E.

1.3.3 CDPEAC — DPEAC Written in C

For those programmers who would rather not code at the DPEAC level, there is an alternative: the CDPEAC instruction set. CDPEAC is a set of macros written in the C programming language, which can be used to insert DPEAC instructions into the body of a standard C function or subroutine.

A CDPEAC routine generally consists of both C and CDPEAC code:

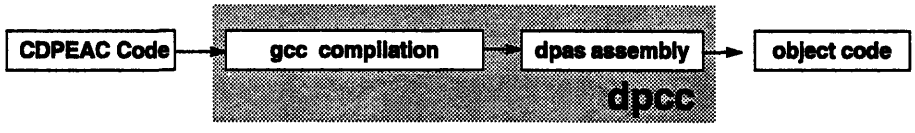
```
CDPEAC_routine(aloc,bloc,size)
unsigned aloc,bloc,size;
{ dpsetup();
  for ( ; size ; size -= 8 ); {
    loadv_u(f,aloc,4,V2);
    join2( loadv_u(f,bloc,4,V3), madav(f,V2,V2,V3) );
    storev_u(f,bloc,4,V3);
    aloc += (4*8); bloc += (4*8);
  }
}
```

CDPEAC instructions expand directly into corresponding DPEAC instructions; the two instruction sets are best seen as two ways of accomplishing the same thing. Both produce assembly-level code, but CDPEAC lets this code be written in a form that is familiar to, and readily understandable by, C programmers.

Coding in CDPEAC is best for C programmers who want to use DPEAC instructions without having to write a DPEAC assembly code routine. CDPEAC still requires an understanding of the basic vector unit operations being performed, but does not require as much attention to assembly-level details as does direct DPEAC coding.

dpcc — The CDPEAC 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.



For a more detailed description of `dpcc`, see Appendix F.

1.4 Using DPEAC and CDPEAC

The most common use of DPEAC or CDPEAC in a CM-5 program is for writing highly efficient subroutines to be called from programs written in a high-level language (such as CM Fortran). The high-level program uses its own operators to define parallel CM arrays, and then calls DPEAC (or CDPEAC) routines to perform efficient arithmetic operations on those arrays.

This is the best way to make use of DPEAC: let the high-level language compiler manage the details of memory management and data layout, so that the DPEAC or CDPEAC subroutines can be focused on exactly those parts of the program that require large amounts of efficient computation.

1.4.1 The DPEAC Header File

To have access to the DPEAC instruction set, including the symbolic constants defined for the locations of registers, etc., as described later in this book, your DPEAC source file should include the DPEAC header file:

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

This header file is only required in the DPEAC source code file; the other source files in your program (see Chapter 7) should include whatever other header files are needed.

1.4.2 The CDPEAC Header File

Similarly, to have access to the CDPEAC instruction set, including the symbolic constants defined for the locations of registers, etc., as described later in this book, your CDPEAC source file should include the CDPEAC header file:

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

This header file is only required in the CDPEAC source code file; the other source files in your program (see Chapter 7) should include whatever other header files are needed.

1.5 Using This Handbook

All programmers should read through Chapter 2, which describes the design and features of the CM-5 vector units.

Programmers who feel comfortable working with SPARC assembly code should read through Chapters 3 and 4, which describe the DPEAC instruction set. Programmers who prefer working in C should read Chapters 5 and 6, which describe CDPEAC. The DPEAC and CDPEAC chapters present basically the same information, but describe it in terms of the appropriate instruction set. (The CDPEAC chapter includes occasional notes describing DPEAC features that are not currently implemented in CDPEAC.)

Both DPEAC and CDPEAC programmers should read through Chapter 7, which presents an example of a CM Fortran program that calls a DPEAC (or CDPEAC) subroutine.

The appendixes contain useful information about the vector units and about the `dpas` assembler and `dpcc` compiler, which are used to assemble/compile DPEAC and CDPEAC source code. Appendix D, in particular, provides detailed descriptions of the VU arithmetic operations and their effects on the flags in the VU status register.

Chapter 2

The CM-5 Vector Units

2.1 CM-5 Vector Unit Accelerators

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

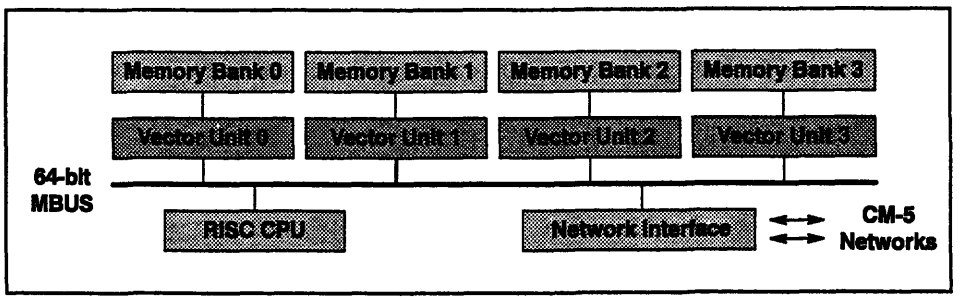


Figure 4. A typical CM-5 processing node, showing the location of the 4 VUs.

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

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

2.1.1 Vector Unit Hardware

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

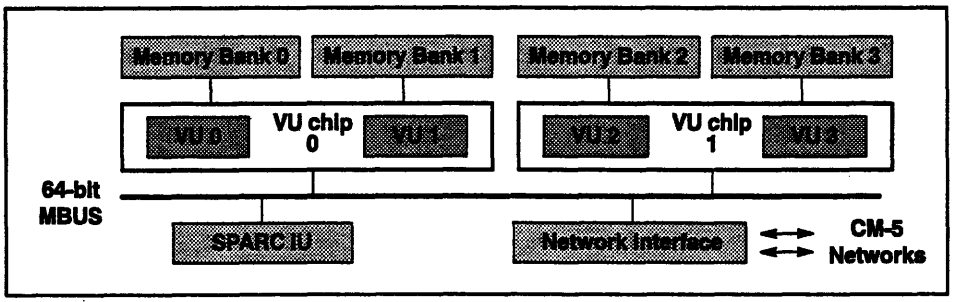


Figure 5. Internal arrangement of VU chips in CM-5 processing node.

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

2.1.2 VU Virtual Memory Layout

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

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

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

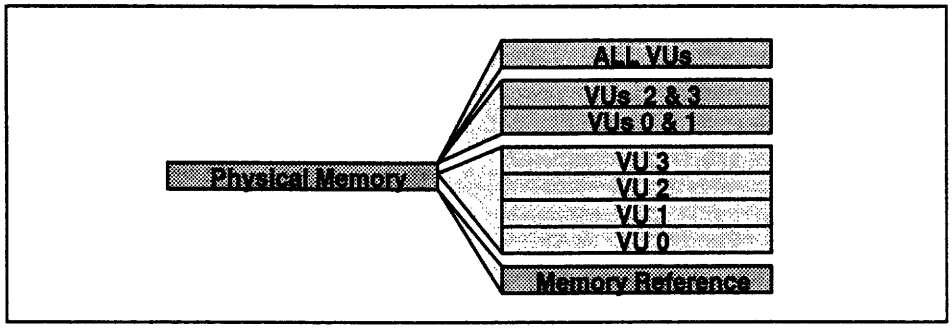


Figure 6. VU virtual memory regions.

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

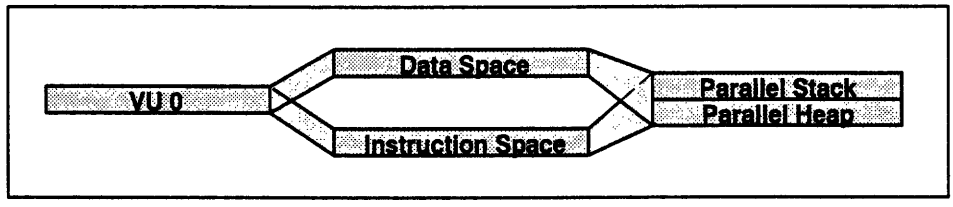


Figure 7. Contents of a single VU memory region.

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

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

2.2 VU Registers

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

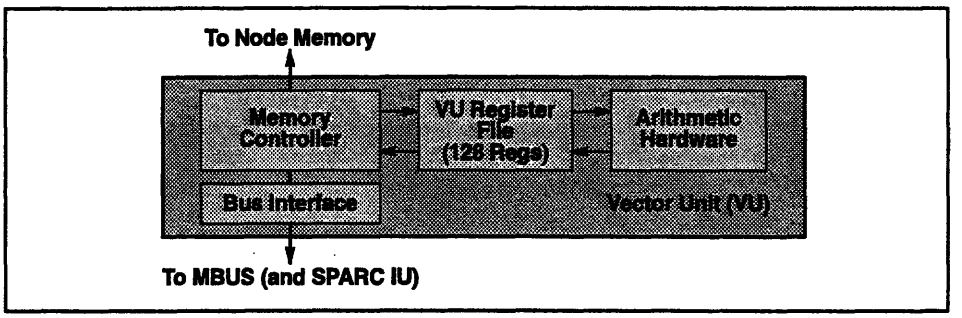


Figure 8. VU internal components.

2.2.1 VU Data Registers

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

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

V0	V1	V2	V3	V4	V5	V6	V7	...	V14	V15
R0	R8	R16	R24	R32	R40	R48	R56	...	R112	R120
R1	R9	R17	R25	R33	R41	R49	R57	...	R113	R121
R2	R10	R18	R26	R34	R42	R50	R58	...	R114	R122
R3	R11	R19	R27	R35	R43	R51	R59	...	R115	R123
R4	R12	R20	R28	R36	R44	R52	R60	...	R116	R124
R5	R13	R21	R29	R37	R45	R53	R61	...	R117	R125
R6	R14	R22	R30	R38	R46	R54	R62	...	R118	R126
R7	R15	R23	R31	R39	R47	R55	R63	...	R119	R127

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

2.2.2 VU Control Registers

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

VU Vector Mask Registers:

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

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

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

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

VU Arithmetic Status Registers:

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

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

C/DPEAC Instruction Default Registers:

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

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

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

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

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

2.3 Effects of VU Control Registers

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

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

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

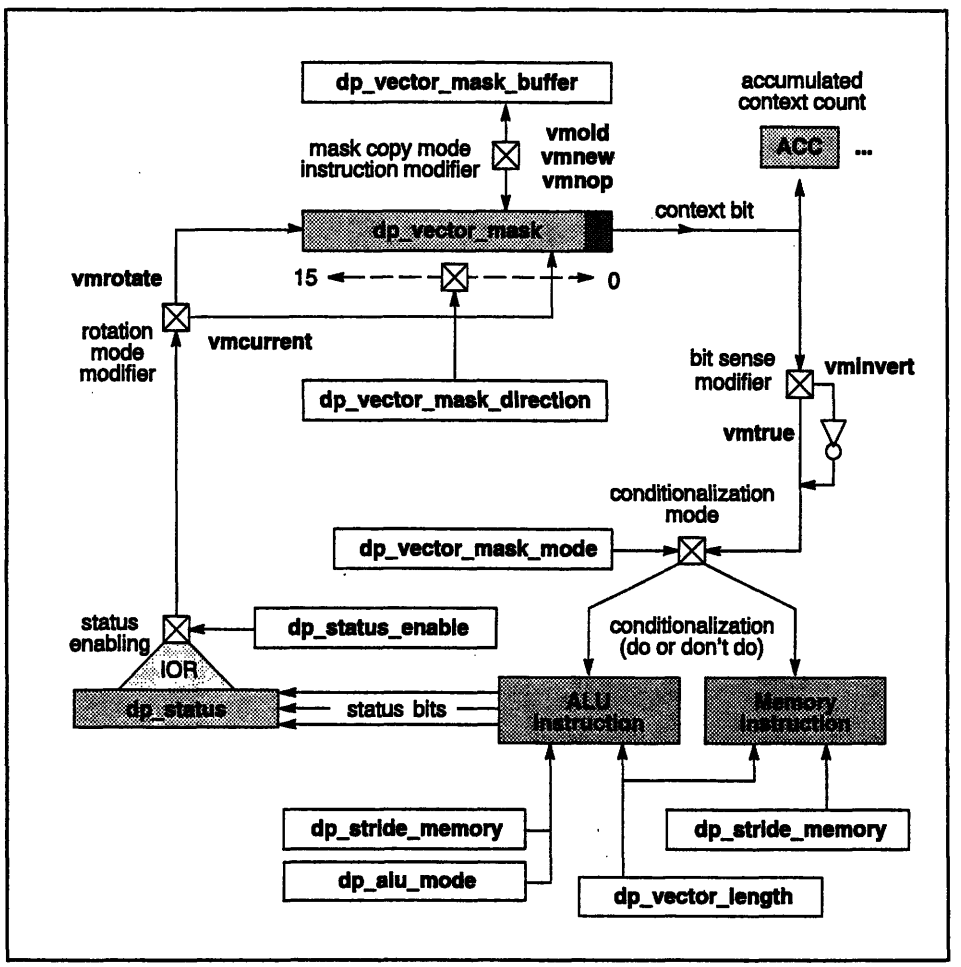


Figure 10. Effects of VU control registers on ALU and memory instructions.

2.3.1 Vector Mask and Conditionalization

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

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

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

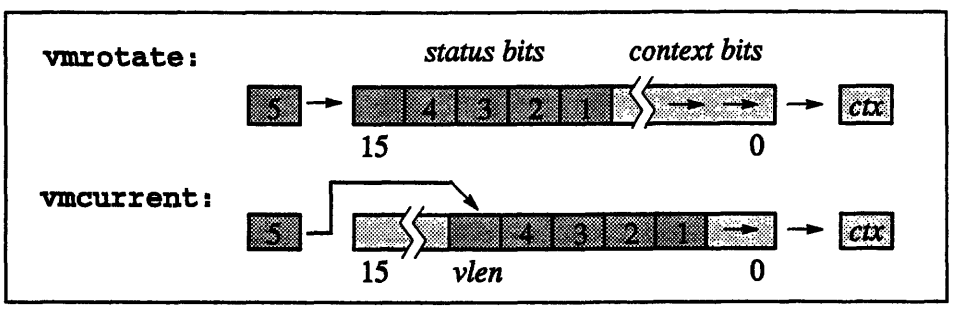


Figure 11. Bit-shifting modes of vector mask register.

2.3.2 ALU Status and Contextualization

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

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

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

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

2.3.3 Status Register Flags

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

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

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

2.3.4 The Vector Mask Buffer

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

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

2.4 Other VU Features

2.4.1 Accumulated Context Count

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

2.4.2 Population Count

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

2.5 VU Control Register Constants

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

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

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

Chapter 3

The DPEAC Instruction Set

The DPEAC instruction set is an extension of SPARC assembly code, providing extra instructions that are used to manipulate the vector units. When a routine containing DPEAC code is assembled, each DPEAC instruction is translated into one or more SPARC memory operations that send appropriately assembled instruction word(s) to the VU hardware.

3.1 DPEAC Code

A DPEAC routine consists of a series of *statements*. Each statement is either a SPARC instruction, a *DPEAC statement*, or a *DPEAC accessor instruction*. A DPEAC statement can occupy either a single text line or several text lines, with a "\" character immediately preceding each linebreak but the last.

A *DPEAC statement* consists of one or more *DPEAC instructions*, separated by semicolons. (An optional extra semicolon can follow the last instruction.) DPEAC instructions are grouped in three categories:

- *arithmetic instructions*, which cause the VUs to perform register arithmetic

```
dfaddv   V0,V2,V4
```

- *memory instructions*, which move data between VU registers and memory

```
floadv   [%i0]:4,V0
fstorev  [%i1]:8,R16
```

- *modifiers*, which alter the assembly or execution of the DPEAC statement

```
vmcurrent ; noalign ; vmnew
```

A single DPEAC statement represents a single VU operation, and thus can contain both a memory instruction and an arithmetic instruction (but no more than one of each type). At least one memory or arithmetic instruction must be present. When a DPEAC statement includes both memory and arithmetic instructions, the memory instruction executes first, and any value it obtains from memory can be used by the arithmetic instruction.

A DPEAC statement may also include any number (including zero) of modifiers, as permitted by the statement's format.

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

arithmetic-op ; *memory-op* ; *modifier-1* ; *modifier-2* ...

This order is recommended because although the memory operation and modifiers are usually executed and/or applied before the arithmetic operation, it is the arithmetic part of the instruction that is typically of the greatest interest.

3.1.1 Chain Loading

When a DPEAC statement refers to the same register in both the memory and arithmetic operations, and when the memory operation is a `load`, the loaded value from the memory operation is used in the arithmetic operation. This is called *chain loading*. In a vector operation, this can happen for each step in the vector operation.

There are some modifier operations (such as population counting), that can also chain load, and some modifier operations that *cannot* chain load. Section 4.3 lists the DPEAC modifiers and indicates any that can or cannot chain load.

3.1.2 DPEAC Accessor Instructions

A DPEAC accessor instruction is a DPEAC instruction that doesn't correspond to a VU arithmetic/memory operation. DPEAC accessor instructions are typically utility operations such as reading and writing VU registers from the SPARC, directly reading and writing parallel memory locations, etc. Accessor instructions can be recognized by their "`dp`" prefix: `dpset`, `dpget`, etc.

3.2 DPEAC Syntax

3.2.1 General Syntax

Numbers are 64-bit constants, parsed as in C. Numbers starting with 0 are octal by default, and numbers starting with a non-zero digit are decimal. The `0x` (hex), `0b` (binary), `0o` (octal), and `0n` (decimal) integer forms are provided, as well as `0f` (32-bit), `0r` (32-bit) and `0d` (64-bit) IEEE float forms.

ASCII constants appear in single quotes ('ABC'), and represent the integer obtained by concatenating the character bytes (the first byte is most significant). Comments are denoted by a "`!`", and extend to the end of the line (as in **as**). C-like */*comment*/* and *#comment* forms are provided by **dpas** itself.

Expressions in DPEAC evaluate to a constant when assembled. There are three classes: *constant-expressions*, *as-expressions*, and *general-expressions.*

A *constant-expression* is evaluated at **dpas** assembly time, using 64-bit integer arithmetic (signed for products/divisions, else unsigned). The following operators are supported, and are evaluated in the order shown (first across, then down):

+	Unary plus (no-op)	-	Negate (2's complement)
!	Logical not	~	Invert (1's complement)
%lo	Low 10 bits	%hi	High 22 bits
&	Bitwise AND	|	Bitwise OR
^	Bitwise XOR		
*	Signed multiply	/	Signed divide
<<	Logical Left shift	>>	Logical Right shift
+	Addition	-	Subtraction
<	Less than	<=	Less than or equal
==	Equal to	!=	Not equal to (<> also allowed)
>	Greater than	>=	Greater than or equal
&&	Logical AND	||	Logical OR

A constant-expression can include symbols only if they can be translated into constants by the **dpas** preprocessor. Floating-point constants are allowed, but are "cast" as integers. (Note that floating-point constants and the operators `<=` and `&&` are **dpas** extensions to **as** expression syntax.)

An *as-expression* is passed directly to the **as** assembler, and must follow **as** syntax. It is evaluated as a 32-bit integer. It can contain any symbols processed by **as**, but cannot contain either float constants or the operators `<=` and `&&`.

A *general-expression* can be either a constant-expression or an as-expression. If **dpas** cannot parse a general-expression as a constant-expression, it assumes it is an as-expression, and passes it directly to **as**.

3.2.2 SPARC CPU Registers

DPEAC memory instructions refer to SPARC registers by these symbols:

`%r0 - %r31`	IU registers	`%0 - %31`	IU registers (alternate form)
`%i0 - %i7`	in registers	`%o0 - %o7`	out registers
`%f0 - %f31`	FP registers	`%g0 - %g7`	global registers
`%psr`	Processor state register	`%fsr`	FP state register
`%wim`	Window invalid register	`%tbr`	Trap base register
`%y`	Multiply-step (Y) register	`%fq`	FP queue register

(**Note:** Later descriptions denote an arbitrary SPARC register as *as-register*.)

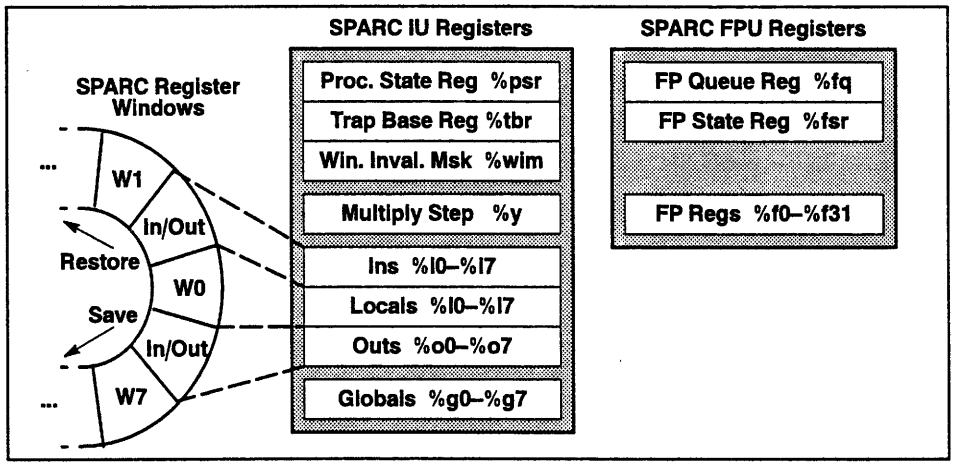


Figure 12. SPARC CPU registers accessible from DPEAC.

Register Restrictions: The following SPARC registers are used by DPEAC operations to store default values, so these registers should be avoided:

Register	Usage	
`%l6`	(Reserved)	Default memory operand (Selects all VU's)
`%l7`	(Reserved)	`dp_instruction_ext` register pointer
`%g2, %g3`	(Temporaries)	Used as temporaries in VU instructions

`%l6` and `%l7` are initialized by `dpas`, which expects them to be preserved. `dpas` assumes that these registers are no longer correct if a `.seg` directive or a `dpunset` accessor instruction is included. `%g2` and `%g3` are overwritten by `dpas` code execution, but are not expected to be preserved. Thus, these registers can be used as temporaries in code that has no VU instructions.

3.2.3 Vector Unit Data Registers

DPEAC arithmetic and memory instructions refer to the 128 VU *data registers* by the following names:

`R0 - R127`	All 128 registers in sequential order
`V0 - V15`	Vector regs (first in each vector, same as `R0, R8 ... R120`)
`S0 - S15`	Scalar regs (single precision), same as `R0 - R15`
`S0 - S30` (even)	Scalar regs (double precision), same as `R0 - R30` (even)

Restrictions: DPEAC statements in immediate format use the `R0` and `R1` registers to store immediate operands, so these registers should be avoided.

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

V0	V1	V2	V3	V4	V5	V6	V7							V14	V15
R0	R8	R16	R24	R32	R40	R48	R56							R112	R120
R1	R9	R17	R25	R33	R41	R49	R57							R113	R121
R2	R10	R18	R26	R34	R42	R50	R58							R114	R122
R3	R11	R19	R27	R35	R43	R51	R59							R115	R123
R4	R12	R20	R28	R36	R44	R52	R60							R116	R124
R5	R13	R21	R29	R37	R45	R53	R61							R117	R125
R6	R14	R22	R30	R38	R46	R54	R62							R118	R126
R7	R15	R23	R31	R39	R47	R55	R63							R119	R127

Figure 13. VU data registers: 16 vectors of 8 registers.

A subset of these registers is designated as the *scalar registers.* These are `S0 - S15` (singleword), or the even registers from `S0 - S30` (doubleword). (The `Snn` names are equivalent to `Rnn`, and explicitly show use of scalar registers.) Scalar operations that use scalar registers assemble into efficient instructions.

You can apply a *register offset* to a data register to access one of the registers succeeding it in `Rnn` order (this is mainly useful for accessing the elements of `Vnn` vectors):

Regn [*k*]	Refers to register *Regn* + *k*. (Ex: `V2[5] = R16+5 = R21`)

3.2.4 Vector Unit Control Registers

There are symbols for all the VU control registers, as described in Section 2.5. However, these symbols are typically only used for accessor instructions such as `dpset` and `dpget`. DPEAC statement formats allow you to implicitly use and/or set the value of one or more control registers while executing a VU operation. See the mode set format in particular (Section 3.9) for examples.

3.2.5 VU Register and Memory Stride Markers

Some VU arithmetic and memory operations can stride through a group of registers or memory addresses. The stride length is indicated by a *stride marker* attached to the appropriate register or memory operand. The generic syntax of these markers is shown below.

Important: The stride markers shown here are not valid for all statement formats; most statement formats restrict the types of stride markers that are allowed.

Register Stride Markers

The general syntax of register stride markers is shown below, where *register* is any valid VU register, and *stride* and *set-stride* are constant expressions in the range -128 to +128. Register striding is always in terms of the `R`*nn* ordering, even when a `V`*nn* register name is specified. A stride of zero causes the same register to be used at each step.

Syntax	Effect
register	Use unit stride (1 for words, 2 for doublewords).
register`:`*stride*	Temporarily use specified *stride*.
register`:mode`	Use stride value stored in `dp_stride_rs1`.
register`:=`*stride*	Set `dp_stride_rs1` to *stride* and use it.
register`:`*stride*`=`*set-stride*	Set `dp_stride_rs1` to *set-stride*, but use *stride*.
register`=`*stride*	Set `dp_stride_rs1` to *stride* (scalar ops only).

Note: The last four stride marker forms shown above are valid only for the *rS1* register argument of an arithmetic instruction.

rS1 Stride Restriction: When you apply a stride of 0 to the *rS1* argument of an arithmetic operation (for example, `R0:0`), the *rS1* register must be one of the scalar registers `S0` through `S15`, or `S30` for double-precision.

VU Memory Stride Markers

The general syntax of memory stride markers is shown below, where *n* and *set-n* is a general expression or *as-register* giving the stride in bytes. Note that the *stride* value is limited to a 24-bit signed integer. A stride of zero causes the same address to be used at each step.

Syntax	Effect
memory-operand	Use default stride in `dp_stride_memory`.
memory-operand`:`*n*	Temporarily stride by *n* bytes.
memory-operand`:=`*n*	Set `dp_stride_memory` to *n* and use it.
memory-operand`:`*n*`=`*set-n*	Set `dp_stride_memory` to *set-n*, but use *n*.
memory-operand`=`*n*	Set `dp_stride_memory` to *n* (scalar ops only).

In the above formats, *n* and *set-n* are either `4` or `8` for singleword data types, or `8` or `16` for doubleword data types.

When you write DPEAC code by hand, you should make sure the default memory stride register `dp_stride_memory` is set to the stride you require (for example, 4 bytes for single-precision or 8 bytes for double-precision). You can use the DPEAC accessor instruction `dpset` for this purpose; for example:

```
dpset    ALL_DPS, 8, DP_STRIDE_MEMORY
```

3.2.6 VU Selection in DPEAC Statements

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

- the value of the *memory-operand* in the memory instruction
- the value specified by the `maddr` modifier, if any
- If neither of these is supplied, a default address that selects all the VUs. The default address used is `DPV_STACK_INST_PORT_ALL`.

Typically, you won't construct these memory addresses yourself; your compiler and/or the `dpas` assembler generate these addresses for you.

3.2.7 VU Selection in DPEAC Accessor Instructions

The VU(s) referenced by a DPEAC accessor instruction are determined by the "VU selector" argument. This argument must be a valid VU selector as described below.

A *VU selector* is an integer or symbolic constant that specifies one or more VUs to perform a given accessor instruction. The syntax is:

Syntax	Immediate Value
constant-expression	Use the specified selector constant (see table below).
as-register	Use value from a SPARC register (all bits).
as-register<	Use value from SPARC register (bits `12:15`).
*	Select all VUs.
**n*	Use both VUs on chip *n* (0=VU's 0&1, 1=VU's 2&3).

The modifier "`<`" makes *as-register* references faster (fewer SPARC operations) because only 4 bits (`12` through `15`) of the register are used. The *constant-expression* form can be either an integer VU selector value, a physical VU selector (an integer preceded by a "`$`"), or one of the symbols defined in the header file `dp.h` for these values. (Use of predefined symbols is recommended.)

The legal VU selector values, and their corresponding symbols, are:

VU	VU Selector		Physical VU Selector	
Number(s)	Value	Symbol	Selector	Symbol
VU *n*	2**n*	`DP_n`	`$n`	`DP_PHYS_NUM_n`
ALL VUs	8	`ALL_DPS`	`$8`	`ALL_PHYS_NUM_DPS`
VUs 0 and 1	10	`DPS_0_AND_1`	`$9`	`DP_PHYS_NUM_0_AND_1`
VUs 2 and 3	12	`DPS_2_AND_3`	`$11`	`DP_PHYS_NUM_2_AND_3`

3.3 DPEAC Instructions

3.3.1 Scalar and Vector Instructions

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

Scalar instructions execute just once for the supplied operands, and are distinguished by an "`s`" suffix on the opcode.

Vector instructions execute repeatedly for each of a series of operands, and are distinguished by a "`v`" suffix on the opcode. Vector operations start with the specified register or memory address operand(s) and then step through succeeding locations determined by the *vector stride* and *vector length*:

- The *vector stride* determines the number of registers or memory addresses a vector operation advances at each step. The default vector stride depends on the type of operation (memory or arithmetic).
- The *vector length* determines the number of registers or memory addresses affected by a vector instruction. The vector length defaults to the value of the VU register `dp_vector_length`, unless a different vector length is specified explicitly.

Note: If a DPEAC statement includes both a memory instruction and an arithmetic instruction, the two must agree in form: they must be either both scalar instructions or both vector instructions.

3.3.2 Register Operands

The register operands of arithmetic and memory instructions are indicated by the following symbols, indicating arbitrary VU registers:

rS1, rS2	First and second source registers.
rLS	Load/store (or third source) register.
rD	Destination register.
rIA	Indirect addressing (used in Register Indirect format).

When an instruction format requires vector (`v`*nn*) register arguments, the symbols *vS1*, *vS2*, *vLS*, *vD*, and *vIA* are used instead. Similarly, when scalar (`s`*nn*) register arguments are required, the symbols *sS1*, *sS2*, *sLS*, *sD*, and *sIA* are used.

3.3.3 Data Types

The data type of a DPEAC instruction is typically indicated by one of the following prefixes on the instruction opcode:

`i`	Signed 32-bit integer	`u`	Unsigned 32-bit integer
`di`	Signed 64-bit integer	`du`	Unsigned 64-bit integer
`f`	Single (32-bit) float	`df`	Double (64-bit) float

3.3.4 Arithmetic Instructions

An arithmetic instruction causes the VUs to perform a register arithmetic operation. Arithmetic instructions have the following general forms, where *opcode* is: `{i,di,u,du,f,df}`*operation*`{v,s}`

Monadic (one source argument):	*opcode*	*rS1*, *rD*
Dyadic (two source arguments):	*opcode*	*rS1*, *rS2*, *rD*
Triadic (three source arguments):	*opcode*	*rS1*, *rLS*, *rS2*, *rD*

Note: In the statement format descriptions in Section 3.4, the arithmetic operation is always shown in triadic form. Dyadic and monadic forms are obtained simply by omitting the appropriate operand symbols (*rLS* and *rS2*).

(Appendix D describes the VU arithmetic instructions in detail, and describes the VU status bits that are affected by each instruction.)

Vector instructions have a default stride of 1 (singleword) or 2 (doubleword) for register operands, unless the instruction explicitly specifies a different stride.

rS2 Operand Restrictions: The *rS2* operand of an arithmetic instruction has the following restrictions:

- For vector operations, *rS2* cannot be any of `R0` through `R7`, by any name (`S0`, `V0`, etc.).
- In scalar operations, *rS2* cannot be any of `R`*nn*, where *nn* is any multiple of `16` (for single-precision) or `32` (for double-precision).

This restriction is imposed by the internal representation of DPEAC operations.

Triadic/Memory Register Restriction Note: When a triadic arithmetic operation and a memory operation are `joined`, the *rLS* operand of the arithmetic operation must be identical to the *rLS* operand of the memory operation.

3.3.5 Memory Instructions

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

`{i,di,u,du,f,df}`*memory-operation*`{v,s}` *memory-operand, rLS*

The *rLS* operand can be any VU register, but if a triadic arithmetic operation and a memory operation are combined, the *rLS* operand of both must be the same, and the memory operation can only be a `load`, not a `store`. The default stride for the *rLS* register is determined by the arithmetic operation, and the stride required by the memory operation must agree.

The *memory-operand* can be any memory address that selects one or more VUs, and it is specified by SPARC register indirection, using Sun-4 Assembler syntax:

Syntax	Memory Address
[*as-register*]	Contents of *as-register*
[*as-register1* + *as-register2*]	Sum of *as-register1* and *as-register2*
[*as-register* + *offset*]	Contents of *as-register* + *offset*

where *offset* is limited to the range `-4096` to `+4095`. Note that double precision memory references must be doubleword (8-byte) -aligned.

The stride of vector instructions is either specified explicitly in the instruction, or else defaults to the `dp_stride_memory` register value.

Singleword / Doubleword Performance Note: Doublewords are the natural word size for the VUs. Singleword operations require a read-modify-write step. Thus, singleword operations are less efficient than doubleword operations.

3.3.6 Modifiers

Modifiers: Modifiers are keywords, such as **pad**, **maddr**, **vmcurrent**, etc., that modify the assembly or execution of a DPEAC statement. The modifiers permitted in a DPEAC statement are determined by the statement's format. The available modifiers are listed below, and described in more detail in Section 4.3.

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

[no]**pad**[:*pad-size*]	Pad vector length
maddr=*memory-operand*	Default memory address
{**vmrotate**, **vmcurrent**}	Packing mode for vector mask bits
[no]**align**	Doubleword alignment declaration
vmmode:[=]*mode-keyword*	Conditionalization mode selector

Conditionalization Modifiers (Mode Set Format Only):

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

Special Modifiers (Mode Set Format Only):

[d]**epc**{v,s} (*vLS*) =*rIA*:*stride*	Population count
vmcount[s]=*reg*:*stride*	Accumulated context count
[no]**exchange**	On-chip VU data exchange

3.4 DPEAC Statement Formats

There are two main classes of statement formats, *short format* and *long format*. The distinction comes from the way the two formats are assembled into SPARC operations.

A *short format statement* assembles into a singleword (32-bit) operation. Short format instructions execute faster than those in long format, but lack some of the features provided by the long format.

A *long format statement* assembles into a doubleword (64-bit) operation. Long format instructions are slower to issue, but use the extra word to provide additional operand types and modifiers that are not permitted by the short format. Specifically, the long instruction format comes in three varieties:

Immediate format allows an immediate operand in the arithmetic operation.

Register stride format allows register striding in the arithmetic operation.

Memory stride format allows address striding in the memory operation.

Mode set format provides access to a number of VU features, including register/memory indirection and overriding of many VU instruction defaults.

Each of the varieties of long format represents a modification of the short format. In terms of DPEAC source code, you can think of the short format as the backbone of features that all DPEAC source lines share, with each of the long formats representing some modification of or addition to those features.

Important! Because of the way that DPEAC code is assembled, the modifications provided by each of the long formats cannot be combined. You can use only one of the long formats, or none of them (that is, use the short format) in a single statement.

For the Curious: Each DPEAC statement is assembled into a word (or doubleword) containing fields for each of the opcodes and operands in the statement. Each of the long formats is assembled as a doubleword, and uses the extra word for a different purpose; thus the extensions provided by the long formats are physically incompatible within a single DPEAC statement.

Note: In the syntax descriptions below, escaped linebreaks (indicated by "\") are sometimes inserted for clarity when a statement's syntax is long and/or complex. These linebreaks are *not* a syntax requirement — all statement formats occupy one line in a DPEAC program.

3.5 The Short Format

The *short statement format* is:

Vector Instuctions:
arith-opcode rS1, vLS, vS2, vD ; mem-opcode mem-operand, vLS ; modifier ; ...
Scalar Instuctions:
arith-opcode rS1, sLS, sS2, sD ; mem-opcode mem-operand, sLS ; modifier ; ...

With one exception (the mode set statement format, see Section 3.9), the *rS1* operand can only have one of the following explicit stride forms:

rS1	Use register *rS1*, with unit stride for vector ops.
rS1`:mode`	Use register *rS1*, with `dp_stride_rs1` stride.
sS1:0	Use scalar register *sS1* with 0 stride.

The remaining register operand(s) must be aligned vector (`V`*nn*) registers for a vector operation, or scalar (`S`*nn*) registers for a scalar operation. Vector instructions always use unit striding, so stride markers are not allowed in short format (see register stride format, Section 3.7).

The *mem-operand* must have one of the following forms:

mem-operand	Use `dp_stride_memory` stride.
mem-operand[`:`*tempstride*]	Use restricted *tempstride*.

The optional *tempstride* is restricted to `4` or `8` for single word operations, `8` or `16` for doubleword operations (see memory stride format, Section 3.8). If *tempstride* is specified, the *rS1* operand must be an aligned vector (`V`*nn*) register or a scalar (`S`*nn*) register with a stride of 0.

The vector length is taken from `dp_vector_length`. This cannot be overridden in the short format (see mode set format, Section 3.9)

Only the following modifiers are permitted by the short format:

[`no`]`pad`[`:`*pad-size*]	Vector length padding (default is 4).
`maddr=`*mem-operand*	Memory operand specifier.
[`no`]`align`	Doubleword alignment guarantee.
{`vmrotate`, `vmcurrent`}	Status bit rotation mode.
`vmmode:`[`=`]*mode-keyword*	Conditionalization mode selector

Note: {`vmcurrent`, `vmrotate`} are useful only for comparison operations, where the result of the comparison produces status bits that can be rotated.

Examples:

```
imovev   V0,V1                        ! Integer monadic
imovev   V0,V1; iloadv [%i0],V0       ! same, chain-loaded
imovev   V0,V1; iloadv [%i0]:8,V0     ! same, temp stride
imovev   V0:mode,V1                   ! Default reg. stride
imovev   S0:0,V1                      ! Scalar reg, 0 stride
imoves   S0:0,S1                      ! Scalar operation

iloadv   [%i0],V1                     ! memory operation
iloadv   [%i0],V1; noalign            ! same, non-aligned
floadv   [%i0]:4,V0                   ! unit stride
floadv   [%i0]:8,V0                   ! double stride
dfloadv  [%i0]:8,V0                   ! unit stride
dfloadv  [%i0]:16,V0                  ! double stride

itestv   V0,V1; maddr=[%i0]           ! maddr modifier
dfgtv    V0,V1                        ! Conditional
dfgtv    V0,V1; vmcurrent             ! same, with modifier

faddv    V0,V1,V2                     ! Float dyadic
faddv    V0,V1,V2; nopad              ! No vlen padding
fmadav   V0,V1,V2                     ! Mult-add
fmadiv   V0,V1,V2                     ! Mult-add, inverted
fmadtv   V0,V1,V2,V3; floadv [%i0],V1
                              ! True triadic, chain-loaded
```

3.6 Immediate (Long) Format

The *immediate format* modifies the short format by replacing one source operand in the arithmetic instruction with an immediate value. (The operand replaced depends on the arithmetic instruction in use — see the instruction listings in Chapter 4.) The immediate value is loaded into `R0` (singleword operations) or `R0` and `R1` (doubleword operations) prior to use.

Vector Instuctions: (*rS2* replaced with immediate value)
arith-opcode rS1, vLS, imm, vD ; mem-opcode mem-operand, vLS ; modifier ; ...
Scalar Instuctions: (*rS2* replaced with immediate value)
arith-opcode rS1, sLS, imm, sD ; mem-opcode mem-operand, sLS ; modifier ; ...

The *imm* operand is a 32-bit immediate value, either an *as-register* or a general expression. Immediate values are sign-extended in double integer arithmetic (zero-extended for double unsigned operations). For double-precision constants, only the upper 32 bits are included in the instruction. Thus, only floating-point numbers with 0s in the 32 least significant bits of their mantissas are allowed.

Older syntax required an immediate value expression to be preceded by a dollar sign (`$`). This syntax is still supported, but is discouraged in new code.

Restrictions: With the exception of the immediate operand, all register and memory operands have the same restrictions as in the short format. Vector length comes from `dp_vector_length`, and the permitted modifiers are the same.

Examples:

```
imovev   29,V1                      ! Monadic immed.
imovev   %i0,V1                     ! SPARC register
imovev   29,V1; iloadv [%i0],V2     ! with memory op.
imovev   29,V1; iloadv [%i0]:8,V0 ! with temp stride
imoves   29,S1                      ! Scalar operation

faddv    R0:0,29,V1                 ! Immed arithmetic
fmadtv   R0:0,V1,29,V3; dfloadv [%i0],V1
                            ! Triadic immediate
```

3.7 Register Stride (Long) Format

The *register stride format* modifies the short format by allowing arbitrary stride markers on the *rS2*, *rLS*, and *rD* register operands. (*rS1* format doesn't change.)

Vector Instuctions:

arith-op rS1, vLS[:*stride2*], *vS2*[:*stride3*], *vD*[:*stride4*] ; \
mem-op mem-operand, vLS[:*stride2*]; \
modifier ; ...

Scalar Instuctions:

arith-op rS1, sLS[:*stride2*], *sS2*[:*stride3*], *sD* [:*stride4*]; \
mem-op mem-operand, sLS[:*stride2*]; \
modifier ; ...

The *stride* markers can be any of the register stride markers in Section 3.2.5, except those that apply to *rS1* only. If a triadic arithmetic operation is used, the *rLS* stride must be the same for both the arithmetic and memory operations.

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

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

Examples:

```
imovev   V0,R4:4                ! Integer monadic
imovev   V0,R4:4; iloadv [%i0],V0   ! same, chain-loaded
imovev   V0,R4:4; iloadv [%i0]:8,V0  ! same, temp stride
iloadv   [%i0],R4:4;            ! memory operation
imovev   V0:mode,R4:4           ! Default reg. stride
imovev   S0:0,R4:4              ! Scalar reg, 0 stride
imoves   S0:0,S3:2              ! Scalar operation
dfgtv    V0,R12:10              ! Conditional
faddv    V0,R20:4,R6:3          ! Float dyadic
fmadtv   R0:0,V1:2,R20:4,R60:7; dfloadv [%i0],V1:2
                        ! True triadic, chain-loaded
```

3.8 Memory Stride (Long) Format

The *memory stride format* modifies the short format by allowing an arbitrary stride marker on the memory operand.

Vector Instuctions:

arith-op rS1, vLS, vS2, vD ; mem-op mem-operand[*:stride*]*, vLS ; modifier ; ...*

Scalar Instuctions:

arith-op rS1, sLS, sS2, sD ; mem-op mem-operand[*:stride*]*, sLS ; modifier ; ...*

The *stride* marker can be any of the memory stride markers in Section 3.2.5.

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

Examples:

```
iloadv   [%i0]:=8,V1;          ! use and set 8
iloadv   [%i0]:8=4,V1;         ! use 8, set 4
iloads   [%i0]=4,S0;           ! set 4 (scalar op)
imovev   V0,V1; iloadv [%i0]:=8,V0     ! Chain-loaded
```

3.9 Mode Set (Long) Format

The *mode set format* is the most complex of the long formats. It allows you to do any or all of the following:

- Override and/or set the default vector length in `dp_vector_length`.
- Override the default conditionalization mode (`vmmode`).
- Override the default conditionalization sense (`vminvert`, `vmtrue`).
- Override the default vector mask copy mode (`vmold`, `vmnew`, `vmnop`).
- Use any of the modifiers permitted by the short format.

Mode set format also allows you to use one (and only one) of the following mutually incompatible extensions to the short format:

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

The mode set "format" is actually a family of distinct but related variants, determined by the appearance of one of the incompatible features listed above. These variants are presented, with examples, in the sections below.

3.9.1 Mode Set Format Variants

The legal mode set variants are:

Vector Length Variant

arith-op[*vec-len*] *rS1, vLS, vS2, vD ;* \
mem-op[*vec-len*] *mem-operand, vLS;* \
modifier; ...

(The syntax for the *vec-len* specifier is described in Section 3.9.2.)

This is the basic mode set variant, in which the only features used are those that are allowed in all mode set variants. In other words, this variant lets you specify an arbitrary vector length for a vector operation, and use general mode set modifiers like **vmnew**, **vminvert**, and **vmcurrent**.

Examples:

```
imovev*16      V0,V2            ! Integer monadic
imovev*16      V0,V2; iloadv [%i0],V0
                                ! same, chain-loaded
iloadv*16      [%i0],V1;        ! memory operation
iloadv*16      [%i0],V1; noalign
                                ! same, non-aligned
faddv*16 V0,V2,V4               ! Float dyadic
fmadtv*16      V0,V2,V4,V6; dfloadv [%i0],V2
                          ! True triadic, chain-loaded

imovev*=16     V0,V2            ! Use and set len.
imovev*%i2     V0,V2            ! SPARC register
imovev*=%i2    V0,V2            ! Use and set
imovev*%i2<    V0,V2            ! 4 bit length
imovev*=%i2<   V0,V2            ! 4 bit use/set
imoves=16      S0,S8            ! Scalar set


faddv     V0,V1,V2; vmcurrent;      ! Current mode
faddv     V0,V1,V2; vmnew;          ! New mask copy
faddv     V0,V1,V2; vmnop;          ! No mask copy
faddv     V0,V1,V2; iloadv [%i0],V0; vminvert;
                                    ! Inverted conditional
```

rS1 Stride Variant

arith-op[*vec-len*] *rS1*[:*stride*], *vLS, vS2, vD* ; \
mem-op[*vec-len*] *mem-operand, vLS*; \
modifier; ...

This variant lets you specify an arbitrary stride marker for the *rS1* operand. This stride marker can be any of the register stride markers in Section 3.2.5.

Examples:

```
imovev*16       V0:2,V2           ! Use stride 2
imovev          V0:1=4,V2         ! Use 1, set 4
imoves          R0=4,R6           ! Set 4 (scalar)
faddv*16        V0:2,V2,V4        ! Float dyadic
fmadtv          V0:1=0,V2,V4,V6; \
                dfloadv [%i0],V2     ! Triadic
```

Register Stride Indirection Variant

arith-op[*vec-len*] *rS1*[(*rIA*:*stride*)], *vLS, vS2, vD* ; \
mem-op[*vec-len*] *mem-operand, vLS*; \
modifier ;...

This variant allows the use of an arbitrary VU register to specify the *rS1* stride. Register indirection format is described in Section 3.9.3.

Examples:

```
imovev          V0(V2),V4 ! Reg. indirection
imovev*16       V0(V2),V4; iloadv [%i0],V0 \
                                    ! same, chain-loaded
imovev          V0(V2:2),V4 ! Indirect. with stride
```

Memory Stride Indirection Variant

arith-op[*vec-len*] *rS1, vLS, vS2, vD* ; \
mem-op[*vec-len*] *mem-operand*[(*rIA* : *stride*)], *vLS*; \
modifier ; ...

This variant allows the use of a VU register to specify the *mem-operand* stride. Memory indirection format is described in Section 3.9.4.

Examples:

```
iloadv*16      [%i0](V2),V0;        ! Mem. Indirect
iloadv*16      [%i0](V2:4),V0;      ! with stride
imovev*16      V0,V4; iloadv [%i0](V2),V0
                                    ! Chain-loading
```

Population Count Variant

arith-op[*vec-len*] *rS1, vLS, vS2, vD* ; \
[d]**epc**{**v,s**} (*vLS*[:**unit**]) =*rIA*[:*stride*] ; \
other-modifier; ...

This variant allows you to specify the [d]**epc**{**v,s**} modifier, which cannot be combined with a memory operation, or with any other mode set variant. (See Section 4.3.3.)

Examples:

```
epcv      (V0)=V1              ! Unit stride
epcv      (V0)=V1:2            ! Explicit stride
depcv     (V0)=V1:2            ! Double op
faddv*16 V0,V1,V2; epcv (V0)=V1;       ! Chain-loading
dfaddv*16 V0,V1,V2; depcv (V0)=V4;     ! Double op
```

Special Modifier Variant

arith-op[*vec-len*] *rS1, vLS, vS2, vD* ; \
mem-op[*vec-len*] *mem-operand, vLS*; \
{[no]**exchange**, **vmcount**[**s**]=*reg*[:*stride*]} ;
other-modifier; ...

This variant allows you to specify one (and only one) of the **[no]exchange** or **vmcount[s]** modifiers, which cannot be combined with any other mode set variant. (See Section 4.3.3.)

Examples:

```
faddv    V0,V1,V2; exchange;       ! exchange values
faddv    V0,V1,V2; \
    floadv [%i0],V0; exchange;     ! chain-load

vmcount=V0;    ! Context count
vmcount=V0:2; ! with stride

faddv    V0,V1,V2; vmcount=V0;     ! chain-loaded
faddv    V0,V1,V2; \
    floadv [%i0],V0; vmcount=V0;   ! chain-loaded
faddv    V0,V1,V2; vmcount=V0:2;   ! strided
```

Scalar Instruction Variant

arith-op[*vec-len*] *rS1*[:*stride*], *sLS, sS2, sD* ; \
mem-op[*vec-len*] *mem-operand, vLS*; \
modifier ;...

This variant lets you use a scalar DPEAC operation to set the default vector length for future instructions (and specify an arbitrary *rS1* stride marker). This mode set variant is much more efficient than using the **dpset** accessor instruction to modify the **dp_vector_length** register.

Examples:

```
fadds    V0,V1,V2; exchange;       ! exchange values
fadds    V0,V1,V2; \
    floads [%i0],V0; exchange;     ! chain-load
```

3.9.2 Vector Length Modifier

In all mode set format variants, the *vec-len* modifier specifies the vector length for the operation, and can also be used to modify the default vector length stored in the register `dp_vector_length`. The syntax of the *vec-len* modifier is:

Syntax	Effect
*opcode*vlen*	Use constant length *vlen*.
opcode=vlen*	Use/set `dp_vector_length` to *vlen*.
*opcode*as-register*	Use length from *as-register* (all bits, + 1).
opcode=as-register*	Use/set `dp_vector_length` from *as-register*.
*opcode*as-register<*	Use length from *as-register* (bits `19:22`, + 1).
opcode=as-register<*	Use/set `dp_vector_length` from *as-register*.
opcode=vlen	Set `dp_vector_length` to *vlen* (scalar ops.)

where *vlen* is a *constant-expression*. The length specified must always be an integer from `1` to `16`. Any unused bits of a referenced *as-register* must be 0. The modifier "`<`" makes *as-register* references faster (fewer SPARC operations) because only 4 bits (`19` through `22`) of the register are used.

The *vec-len* modifier can be attached to either the arithmetic opcode or memory opcode, or both, and it applies to both. (If a *vec-len* modifier is specified on both the arithmetic and memory opcodes, the two modifiers must be identical.)

Note: All forms that obtain a length value from a register implicitly add 1 to the value before use. All forms that store a value into `dp_vector_length` store the value in decremented form, so that this implicit incrementing will work properly.

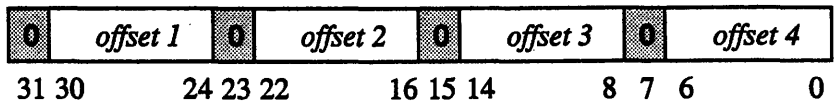
3.9.3 Register Stride Indirection

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

Syntax	Effect
rS1 (*rIA*)	Indirect addressing, unit stride.
rS1 (*rIA*:*stride*)	Indirect addressing, constant stride.

The *rIA* register operand contains offsets that are separately added to the *rS1* base register to obtain the actual `Rnn` register containing the *rS1* stride. (**Note:** This offset addition is *not* cumulative.)

The register offsets are packed *four to a register* in the specified *rIA* register and in subsequent registers at the specified *stride*. Since offsets cannot exceed 127 (7 bits), the eighth bit of each offset byte must be zero:



Note: If a *stride* is not specified, then the "unit" stride is always 1 register for both single- and doubleword operations; one doubleword "register" corresponds to two singleword registers.

3.9.4 Memory Indirection

For memory stride indirection, the *mem-operand* format is:

Syntax	Effect
mem-operand(*register*)	Memory indirection, unit stride.
mem-operand(*register*:*stride*)	Memory indirection, constant stride.

The indirection modifier replaces the [:*tempstride*] modifier of the short format.

The specified single-precision VU *register* contains offsets that are separately added to the memory address to obtain each operand location. The addition is done in two's-complement, so negative offsets will work correctly. (**Note:** This offset addition is *not* cumulative.) The memory offsets are stored *one byte per register*, taken from the specified single-precision *register* and subsequent registers at the specified *stride*.

Note: If a *stride* is not specified, then the "unit" stride is 1 single-precision register for single-precision memory operations, and 2 single-precision registers (1 double-precision register) for double-precision memory operations.

3.9.5 Mode Set Format Modifiers

The following modifiers are permitted by the mode set format:

These modifiers are permitted by the short format:

[no]pad[**:***pad-size*]	Vector length padding (default is 4).
maddr=*mem-operand*	Memory operand specifier.
[no]align	Doubleword alignment guarantee.
{vmrotate, vmcurrent}	Status bit rotation mode.
vmmode:[**=**]*mode-keyword*	Conditionalization mode selection.

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

{vminvert, vmtrue};	Conditionalization bit sense selection.
{vmold, vmnew, vmnop};	Vector mask copy mode.

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

[d]epc{v,s} (*vLS*[**:unit**])**=***rIA***:***stride*	Population count.
vmcount[s]=*reg***:***stride*	Accumulated context count.
[no]exchange	VU on-chip data swapping.

These modifiers are all described in more detail in Section 4.3.

Chapter 4

DPEAC Instruction Set Reference

This chapter presents a quick-reference list of the DPEAC instruction set, including DPEAC instructions, instruction modifiers, and accessor instructions.

4.1 DPEAC Arithmetic Instructions

4.1.1 Monadic (One-Source) Arithmetic Instructions

These operators perform an arithmetic operation on *rS1*, storing the result in *rD*. (**Note:** In immediate format, the *rS1* source argument is the immediate value.)

Formats:

opcode rS1, rD

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

The `to` operator converts between data types: *rS1* is of the first type in the opcode, and *rD* is of the second type. (In immediate format, *rS1* is an immediate value.)

Formats:

opcode rS1, rD

Monadic Opcodes	Function
`{i,di,u,du}to{f,df}{v,s}`	Convert integer to float.
`{f,df}to{f,df}{v,s}`	Convert to another precision.
`{f,df}to{i,di,u,du}r{v,s}`	Convert to integer (round).
`{f,df}to{i,di,u,du}{v,s}`	Convert to integer (truncate).

4.1.2 Dyadic (Two-Source) Instructions

These operators perform an arithmetic operation on the *rS1* and *rS2* arguments, and store the result in the *rD* argument. (In immediate format, *rS2* is an immediate value.)

Formats:

opcode rS1, rS2, rD

Dyadic Opcodes	Function
`{i,di,u,du,f,df}add{v,s}`	Add (*rD* = *rS1* + *rS2*).
`{i,di,u,du}addc{v,s}`	Integer add with carry bit from shift of vector mask register.
`{i,di,u,du,f,df}sub{v,s}`	Subtract (*rD* = *rS1* - *rS2*).
`{i,di,u,du}subc{v,s}`	Integer subtract with carry bit from shift of vector mask register.
`{i,di,u,du,f,df}subr{v,s}`	Subtract reversed (*rD* = *rS2* - *rS1*).
`{i,di,u,du}sbrc{v,s}`	Integer subtract reversed with carry bit from shift of vector mask register.
`{i,di,u,du,f,df}mul{v,s}`	Multiplication (low 32/64 bits for ints).
`{di,du}mulh{v,s}`	Integer multiply (high 64 bits).
`{f,df}div{v,s}`	Divide (*rD* = *rS1* / *rS2*).
`{u,du}enc{v,s}`	Make float from exp and mant (*rS1*, *rS2*).

Dyadic (Two-Source) Instructions *(continued)*

Formats:

opcode rS1, rS2, rD

Dyadic Opcodes	Function
{u,du} shl{v,s}	Shift left (*rD* = *rS1* << *rS2*).
{u,du} shlr{v,s}	Shift left, reversed (*rD* = *rS2* << *rS1*).
{i,di,u,du} shr{v,s}	Shift right (*rD* = *rS1* >> *rS2*).
{i,di,u,du} shrr{v,s}	Shift right, reversed (*rD* = *rS2* >> *rS1*).
{u,du} and{v,s}	Bitwise logical AND.
{u,du} nand{v,s}	Bitwise logical NAND.
{u,du} andc{v,s}	Bitwise logical NOT(*rS1*) AND *rS2*.
{u,du} or{v,s}	Bitwise logical OR.
{u,du} nor{v,s}	Bitwise logical NOR.
{u,du} xor{v,s}	Bitwise logical XOR.
{i,di,u,du,f,df} mrg{v,s}	If vector mask bit = 1 then *rS1* else *rS2*.

4.1.3 Arithmetic Comparisons

These operators perform an arithmetic comparison between the *rS1* and *rD* arguments, and set status flags accordingly. (In immediate format, *rD* is an immediate value.)

Formats:

opcode rS1, rD

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

4.1.4 Compare (Dyadic with rD constant)

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

Format:

`{i,di,u,du,f,df}cmp{v,s}` *rS1, rS2, code*

Code	Purpose
0	Test for greater than.
1	Test for equal.
2	Test for less than.
3	Test for greater than or equal.
4	Test for unordered (NaN present).
5	Test for ordered and not equal.
6	Test for not equal or unordered.
7	Test for less than or equal.

4.1.5 Dyadic Mult-Op Operators

These operations perform a muliplication and an arithmetic (or logical) operation on the *rS1*, *rS2*, and *rD* arguments, and store the result in *rD*. **Note:** The optional `[h]` suffix indicates that the high 64 bits of the multiplication are to be used in the logical operation, rather than the low 64 bits (the default). (In immediate format, *rS2* is an immediate value.)

Format:

opcode *rS1, rS2, rD*

Accumulative Opcodes	Function
`{i,di,u,du,f,df}mada{v,s}`	*rD* = (*rS1* * *rS2*) + *rD*
`{i,di,u,du,f,df}msba{v,s}`	*rD* = (*rS1* * *rS2*) - *rD*
`{i,di,u,du,f,df}msra{v,s}`	*rD* = *rD* - (*rS1* * *rS2*)
`{i,di,u,du,f,df}nmaa{v,s}`	*rD* = -*rD* - (*rS1* * *rS2*)
`dum[h]sa{v,s}`	*rD* = (*rS1* * *rS2*) AND *rD*
`dum[h]ma{v,s}`	*rD* = (*rS1* * *rS2*) AND NOT *rD*
`dum[h]oa{v,s}`	*rD* = (*rS1* * *rS2*) OR *rD*
`dum[h]xa{v,s}`	*rD* = (*rS1* * *rS2*) XOR *rD*

Dyadic Mult-Op Operators *(continued)*

Format:

opcode rS1, rS2, rD

Inverted Opcodes	Function
{i,di,u,du,f,df}madi{v,s}	*rD* = (*rS2* * *rD*) + *rS1*
{i,di,u,du,f,df}msbi{v,s}	*rD* = (*rS2* * *rD*) - *rS1*
{i,di,u,du,f,df}msri{v,s}	*rD* = *rS1* - (*rS2* * *rD*)
{i,di,u,du,f,df}nmai{v,s}	*rD* = -*rS1* - (*rS2* * *rD*)
dum[h]si{v,s}	*rD* = (*rS2* * *rD*) AND *rS1*
dum[h]mi{v,s}	*rD* = (*rS2* * *rD*) AND NOT *rS1*
dum[h]oi{v,s}	*rD* = (*rS2* * *rD*) OR *rS1*
dum[h]xi{v,s}	*rD* = (*rS2* * *rD*) XOR *rS1*

4.1.6 Convert Operation (Dyadic with rS2 constant)

These operations convert the *rS1* argument to the type indicated by the constant *code* argument, and store the result in the *rD* argument. The symbolic code constants listed below are defined by the `dp.h` header file. (In immediate format, *rS1* is an immediate value.)

Format:

`cvt{f,fi,i,ir}{v,s}` *rS1, code, rD*

Opcode/Type		Code	Purpose
cvt	i[r]	CVTICD_F_I (4)	Single float to single signed integer.
cvt	i[r]	CVTICD_F_U (5)	Same, to unsigned integer.
cvt	i[r]	CVTICD_F_DI (6)	Single float to double signed integer.
cvt	i[r]	CVTICD_F_DU (7)	Same, to unsigned integer.
cvt	i[r]	CVTICD_DF_I (12)	Double float to single signed integer.
cvt	i[r]	CVTICD_DF_U (13)	Same, to unsigned integer.
cvt	i[r]	CVTICD_DF_DI (14)	Double float to double signed integer.
cvt	i[r]	CVTICD_DF_DU (14)	Same, to unsigned integer.
cvt	f	CVTFCD_F_DF (3)	Single float to double float.
cvt	f	CVTFCD_DF_F (9)	Double float to single float.

Convert Operation, *Continued*

Format:

`cvt{f,fi,i,ir}{v,s}` *rS1, code, rD*

Opcode/Type	Code	Purpose
`cvt fi`	`CVTFICD_I_F` (1)	Single signed integer to single float.
`cvt fi`	`CVTFICD_U_F` (5)	Same, but from unsigned integer.
`cvt fi`	`CVTFICD_I_DF` (3)	Single signed integer to double float.
`cvt fi`	`CVTFICD_U_DF` (7)	Same, but from unsigned integer.
`cvt fi`	`CVTFICD_DI_F` (9)	Double signed integer to single float.
`cvt fi`	`CVTFICD_DU_F` (13)	Same, but from unsigned integer.
`cvt fi`	`CVTFICD_DI_DF` (11)	Double signed integer to double float.
`cvt fi`	`CVTFICD_DU_DF` (15)	Same, but from unsigned integer.

4.1.7 True Triadic (Three-Source) Operators

These operations perform a muliplication and an arithmetic (or logical) operation on the *rS1*, *rS2*, and *rLS* arguments, and store the result in *rD*. (In immediate format, *rS2* is an immediate value.)

Format:

opcode rS1, rLS, rS2, rD

True Triadic Opcodes	Function
`{i,di,u,du,f,df}madt{v,s}`	*rD* = (*rS1* * *rLS*) + *rS2*
`{i,di,u,du,f,df}msbt{v,s}`	*rD* = (*rS1* * *rLS*) - *rS2*
`{i,di,u,du,f,df}msrt{v,s}`	*rD* = *rS2* - (*rS1* * *rLS*)
`{i,di,u,du,f,df}nmat{v,s}`	*rD* = -*rS2* - (*rS1* * *rLS*)
`dum[h]st{v,s}`	*rD* = (*rS1* * *rLS*) `AND` *rS2*
`dum[h]mt{v,s}`	*rD* = (*rS1* * *rLS*) `AND NOT` *rS2*
`dum[h]ot{v,s}`	*rD* = (*rS1* * *rLS*) `OR` *rS2*
`dum[h]xt{v,s}`	*rD* = (*rS1* * *rLS*) `XOR` *rS2*

Note: In the opcode descriptions above, the optional [h] indicates that the high 64 bits of the multiplication are to be used in the logical operation, rather than the low 64 bits (the default).

Triadic/Memory Register Restriction Note: When a triadic arithmetic operation and a memory operation are `joined`, the *rLS* operand of the arithmetic operation must be identical to the *rLS* operand of the memory operation.

4.1.8 No-Op Operator

The untyped arithmetic no-op allows modifier side effects without specifying an operation. The no-op takes no arguments.

Formats:

`fnop{v,s}` (No arithmetic operation.)

4.2 DPEAC Memory Instructions

The following opcodes are supported for memory operations.

Format:

opcode memory-address, rD

Opcodes	Function
`{i,di,u,du,f,df}load{v,s}`	Load *Vls* from memory.
`{i,di,u,du,f,df}store{v,s}`	Store *Vls* to memory.

4.2.1 No-Op Operator

The following opcodes are supported for memory operations.

Format:

`memnop` *memory-address, rD*

Opcodes	Function
`memnop`	No memory operation.

The `memnop` operation allows the side effects of memory syntax (setting of stride defaults by stride markers, etc.) to happen without an actual memory operation.

4.3 DPEAC Instruction Modifiers

This section describes the statement modifiers that can be combined with arithmetic and memory operations to affect their assembly and/or execution. **Note:** Some of these modifiers (such as the last three) can be used on their own.

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

[no]pad[:*pad-size*] Default: **pad:4**

Vector Length Padding: Pads vector length of instruction to at least *pad-size*. Has no effect if vector length is already that size. Used to avoid instruction pipeline hazards. If not supplied, defaults to **pad:4**. The nopad variant is the same as **pad:0**. Pads between **0** and **4** are allowed, but have the same effect as **pad:4**.

maddr=*memory-operand* Default: None

Memory Operand Specifier: Used to supply a default memory operand for DPEAC statements that omit the memory instruction — this memory operand is used solely to determine VU selection.

{vmrotate, vmcurrent} Default: **vmrotate**

Status Bit Rotation Mode: Determines how status bits from vector operations are stored in the register **dp_vector_mask**. **vmrotate** "rotates" them in, **vmcurrent** inserts them in bit order. (See Figure 14.) Note: this modifier is allowed by the short format for conditional operations only. Otherwise, it can only be used in the mode set format.

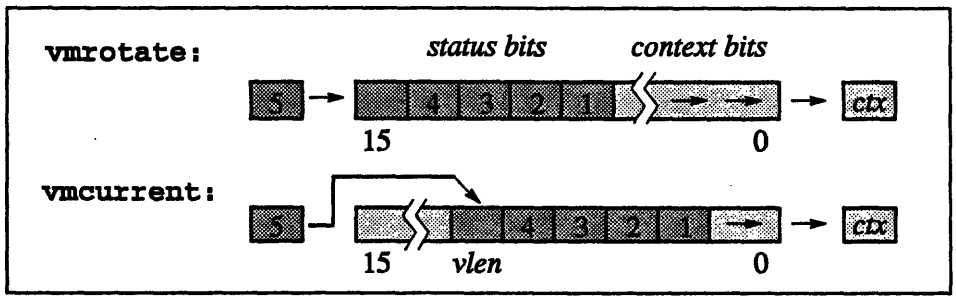


Figure 14. Bit-shifting modes of vector mask register.

`[no]align` Default: `noalign`

Doubleword Alignment Guarantee: Declares whether or not the memory operand is doubleword-aligned (even for singleword operations). If alignment is guaranteed, `dpas` can generate more efficient code. (**Note:** The default setting of this modifier can be reversed by providing the `-a` command line switch to `dpas`.)

4.3.2 Conditionalization Modifiers

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

`vmmode:[=]`*mode-keyword* Default: `vmmode:vmmode`

Conditionalization Mode: The `vmmode` modifier overrides the value of the `dp_vector_mask_mode` register, which affects whether arithmetic operations and/or memory operations are to be conditionalized. The permitted *mode-keyword* operands are:

Mode	Effect
`vmmode:vmmode`	Use current value of `dp_vector_mask_mode`.
`vmmode:always`	Do not use conditionalization in this instruction.
`vmmode:=always`	Set `dp_vector_mask_mode` for no conditionalization.
`vmmode:condmem`	Conditionalize loads and stores in this instruction.
`vmmode:=condmem`	Set `dp_vector_mask_mode` for conditionalization.
`vmmode:condalu`	Conditionalize arithmetic in this instruction.
`vmmode:=condalu`	Set `dp_vector_mask_mode` for conditional arithmetic.
`vmmode:=cond`	Set `dp_vector_mask_mode` for full conditionalization.

It is not legal to override `dp_vector_mask_mode` for full conditionalization. Thus, "`vmmode:cond`" is not allowed.

Usage Note: Scalar instructions are executed without conditionalization, so you may add `vmmode:always` to any scalar instruction in any format with no effect. Similarly, you may add `vmmode:vmmode` to any vector instuction in any format since it represents the default action taken by the hardware.

{vminvert, vmtrue} Default: vmtrue

Conditionalization Bit Sense: The vminvert and vmtrue modifiers control whether the conditionalization bits shifted out of the dp_vector_mask are inverted. If inverted, the sense of these bits is reversed; i.e., 0 selects a vector element, and 1 deselects it.

Modifier	Effect
vminvert	Invert sense of vector mask bits for conditionalization.
vmtrue	Do not invert sense of vector mask bits.

Note: This modifier is only allowed in the mode set statement format.

{vmold, vmnew, vmnop} Default: vmold

Vector Mask Copy Mode: The vmold, vmnew, and vmnop modifiers control the copying of the vector mask and vector mask buffer registers prior to instruction execution:

Modifier	Effect
vmold	Copy dp_vector_mask_buffer to dp_vector_mask.
vmnew	Copy dp_vector_mask to dp_vector_mask_buffer.
vmnop	No copy.

Note: This modifier is only allowed in the mode set statement format.

4.3.3 Special Modifiers (Mode Set Format Only)

[d]epc{v,s} (*vLS*[:unit]) =*rIA*[:*stride*] Default: None

Population Count: The [d]epc{v,s} modifier enables the population count feature. Specifically, the single- or double-precision register *vLS* (and subsequent registers at a unit stride) are read and the "1" bits in each are counted. The results, each a single-precision unsigned integer between 0 and either 32 (single-precision) or 64 (double-precision), are written to the register *Ria* (and subsequent single-precision registers at the specified *stride*, a *constant-expression* that defaults to the unit stride for the data type).

The [d]epc{v,s} modifier effectively replaces the normal memory operation in a DPEAC statement. The *Vls* register operand is used, so population counting cannot be combined with any memory operation. Population counting also cannot be used in conjunction with register or memory indirection or the

vmcount[s] or **[no]exchange** modifiers. The population count result is written before the operands are read for the arithmetic operation, so the **[d]epc{v,s}** modifier chain loads. The *vLS* operand is always strided with a unit (1 or 2 register) stride, so the **:unit** keyword is optional and has no effect other than to emphasize the unit striding.

Implementation Note: Currently, the **[d]epc{v,s}** modifier cannot be used in conjunction with a long-latency arithmetic operation, i.e., **[f,df]div**, **[f,df]sqrt**, **[f,df]inv**, or **[f,df]isqt**.

vmcount[s]=*reg*:*stride* Default: None

Accumulated Context Count: The **vmcount** modifier enables the VU chip's *accumulated context count* feature. The single-precision VU register *reg* (and subsequent registers at the given *stride*, a *constant-expression*) is loaded with the accumulated count of "1" bits in the vector mask at each step in the vector operation. This accumulation is inclusive; the count includes the bit that is shifted out of the vector mask register for each element. The scalar version, **vmcounts**, is intended for use with scalar operations. It is an error to use **vmcounts** with any vector operation.

For each element in the vector, the **vmcount** result is written before the operands are read for the arithmetic operation, so this modifier chain loads. This modifier cannot be used in conjunction with either register or memory indirection, nor with the **[d]epc{v,s}**, or **[no]exchange** modifiers.

[no]exchange Default: **noexchange**

VU On-Chip Data Swapping: Controls exchange of data between two VUs on the same chip. Specifying **exchange** causes arithmetic results on each VU to be written to the destination register(s) of the other VU. In conditionalized ALU operations, deselected elements are not written to the opposite VU. Selected elements *are* written, even if the corresponding element in the opposite VU is deselected.

The **[no]exchange** modifier is only used in the mode set format. However, it is incompatible with register stride indirection, memory stride indirection, and with the **[d]epc{v,s}**, and **vmcount[s]** modifiers.

Implementation Note: This modifier is implementation-dependent, and may not be available in the future. Also, the current implementation of exchanging does not allow chain loading into the arithmetic destination register.

4.4 DPEAC Accessor Instructions

These accessor instructions are always used as single statements, execute on the node microprocessor (the SPARC), and generally move data between the SPARC and the VU, or affect values stored in SPARC registers.

4.4.1 VU Register Accessor Instructions

Instruction(s)	Function(s)
dpwrt[d], dprd[d]	Write and read VU data registers.
dpset[d], dpget[d]	Write and read VU control registers.
dpchgbk	Convert address from one VU region to another.
dpchgsp	Convert between VU data and instruction spaces.
dpld[d], dpst[d]	Read and write VU parallel memory.
dpsync	Synchronize instruction pipelines of VUs.

These instructions move data between VU data registers and SPARC registers:

dpwrt[d] *VU-selector, as-src-reg, VU-dest-reg* [, {**sync, nosync**}]
dpwrt[d] *VU-selector, value, VU-dest-reg* [, {**sync, nosync**}]
dprd[d] *VU-selector, VU-src-reg, as-dest-reg* [, {**sync, nosync**}]

```
dpwrt   ALL_DPS,%i1,V0,sync
dpwrt   DPS_0_AND_1,29,V0,nosync
dprd    DP_3,V0,%i0
```

These instructions move data between VU control registers and SPARC registers. (See Section 2.5 for a list of predefined control register constants.)

dpset[d] *VU-selector, as-src-reg, ctl-reg-offset* [, **SUPERVISOR**]
dpset[d] *VU-selector, value, ctl-reg-offset* [, **SUPERVISOR**]
dpget[d][s] *VU-selector, ctl-reg-offset, as-dest-reg* [, **SUPERVISOR**]

```
dpset   DP_3,%i0,DP_VECTOR_MASK
dpset   ALL_DPS,0,DP_VECTOR_MASK
dpget   DPS_0_AND_1,DP_VECTOR_MASK,%i0
```

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

dpchgsp *srcreg, destreg*

```
dpchgsp      R5, R6
```

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

dpchgbk *srcreg, VU-selector, destreg*

```
dpchgbk      R5, DPS_0, R6
```

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

dpld[d] *as-mem-operand, as-dest-register*
dpst[d] *as-src-register, as-mem-operand*

```
dpld    [%i0], %i1
dpst    %i1, [%i0]
```

This instruction generates code to prevent the preceding and following instructions from overlapping in the instruction pipeline of the VUs (see Appendix C):

dpsync

```
faddv V0,V1,V2
dpsync
fmulv V1,V2,V3
```

4.4.2 VU Trap Instructions

These instructions generate traps and provide direct SPARC access to the **dp_vector_mask** register:

Opcode	Function
trap	Generate trap unconditionally.
etrap	Generate trap if **set_enb** bit in the **dp_interrupt_cause_green** register is set.

4.4.3 Vector Mask Instructions

Opcode	Function
`ldvm` *rS1*	Move *rS1* to `dp_vector_mask`.
`stvm` *rS1*	Move `dp_vector_mask` to *rS1*.

```
ldvm V1
stvm V1
```

4.4.4 SPARC Accessor Instructions

These instructions assemble into SPARC-only code, and do not affect the VUs:

Instruction	Function
`dpentry`	Creates a callable DPEAC routine.
`dpretn`	Returns from DPEAC routine.
`load[d]`	Loads an IU register with a constant value.
`dpunset`	Signals that one or both reserved registers may have been overwritten.
`dpregs`	Overrides SPARC default register usage.

`dpentry` *name, argwords, localbytes*

```
_ROUTINENAME, 0, 0
```

The `dpentry` instruction creates a callable DPEAC routine. *name* is an *as-symbol*, the name of the routine. (Don't forget the leading underscore when naming routines to be called from C.)

argwords is the number of stack words reserved for arguments (in excess of 6) to subroutines. (Doubleword arguments count as two words.) If there are no subroutine calls (or none with more than 6 arguments), *argwords* is 0.

localbytes is the number of bytes beyond the standard frame size (`MINFRAME`, i.e., 92) to be allocated on the stack frame for local temporaries. (These are located at the top of the frame and referenced by negative offsets from `%fp`.)

The `dpentry` instruction implies a "`dpregs =,=,` " — that is, the Default Maddr Base and Instruction Extension Pointer registers are initialized.

dpretn

```
dpretn
```

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

```
ret
restore
```

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

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

```
load 1066, %i0
```

This instruction loads a SPARC IU register with a constant value, automatically generating the SPARC instructions needed to load the value. **loadd** loads an aligned pair of registers with a doubleword value.

dpunset

```
dpunset
```

The **dpunset** "instruction" informs the **dpas** assembler that one or both of the reserved registers (`%l6` and `%l7`) may have been overwritten. If succeeding code requires the original values of these registers, **dpas** inserts instructions to reinitialize them.

dpregs

```
dpregs    %l6,%l7,%g2
dpregs    %l6=,-,%g2+
```

The **dpregs** "instruction" modifies the default SPARC registers used by **dpas** for code construction. The syntax is:

dpregs *MaddrReg, InstExtPtrReg, TempRegs*

Argument	Register Usage
MaddrReg	Default Maddr Base Register, used for executing DPEAC statements lacking a memory address. Default value is `DPV_STACK_INSTR_PORT_ALL`.
InstExtPtrReg	Instruction Extension Pointer Register, used in non-doubleword-aligned memory references; contains 0xC0000176, a value used to compute a pointer to the `dp_instruction_ext` register.
TempRegs	An even-numbered SPARC register; declares that both the specified register and its immediate successor can be used as temporaries to execute VU instructions.

Each of these arguments can have any one of the following forms:

Argument	Meaning
as-register	Use this register and mark it as uninitialized.
as-register=	Use this register and generate code to initialize it.
=	Generate code to initialize the current register.
-	Tell **dpas** to not use any register for this feature.
{*as-register*}+	Tell **dpas** to use the register, but not alter its value.
(blank)	Do not change the current setting (NOP).

The default setting at the beginning of **dpas** assembly is:

```
dpregs %l6,%l7,%g2  ! declare regs, but don't initialize
```

If "-" syntax is used to turn off Maddr Register usage, subsequent VU instructions that don't specify a memory address will signal an error. If the Instruction Extension Pointer Register is turned off, **dpas** will generate code two cycles slower wherever a long format instruction performs a memory operation on a possibly non-doubleword-aligned memory address (one for which neither the the **align** modifier nor the **-a** switch to **dpas** were given).

The "-" (disable) and "=" (initialize) markers can only be applied to the *MaddrReg* and *InstExtPtrReg* operands.

Declaring a register but not immediately setting it up (i.e., specifying a register name, but not using the "=" syntax), causes **dpas** to mark that register as uninitialized. This causes initialization code to be inserted later in assembly when the value of the register is needed again. This can be used when a register may have been overwritten to declare that **dpas** should not assume its contents are valid.

Chapter 5

The CDPEAC Instruction Set

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

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

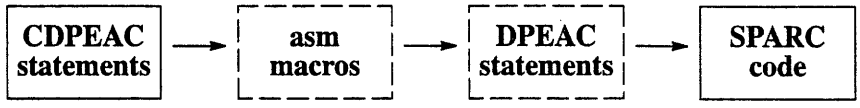


Figure 15. Process of translation used for CDPEAC code.

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

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

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

5.1 CDPEAC Code

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

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

5.1.1 VU Instructions

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

A VU instruction is either:

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

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

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

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

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

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

- or some combination of the above instruction types, made with the CDPEAC `joinn` operator:

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

5.1.2 VU Accessor Instructions

A CDPEAC *VU accessor instruction* is a CDPEAC instruction that doesn't correspond to a VU arithmetic or memory operation (that is, an instruction executed by the SPARC). Accessor instructions are typically utility operations such as reading and writing VU registers from the SPARC, directly reading and writing parallel memory locations, etc. Accessor instructions can be recognized by their "**dp**" prefix, i.e., **dpset**, **dpget**, etc.:

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

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

5.1.3 VU Special Instructions

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

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

5.1.4 The Join Macro

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

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

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

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

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

```
joinn ( arithmetic-inst, memory-inst, modifier-1, modifier-2 ... )
```

This order is recommended because while the memory operation and modifiers are usually executed and/or applied before the arithmetic operation, it is the arithmetic part of the instruction that is typically of greatest interest.

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

Chain Loading

When a join statement includes both memory and arithmetic instructions, the memory instruction executes first, and any value it obtains from memory can be used by the arithmetic instruction.

When a join statement refers to the same register in both the memory and arithmetic operations, and when the memory operation is a `load`, the loaded value from the memory operation is used in the arithmetic operation. This is called *chain loading*. In a vector operation, this can happen for each step in the vector operation.

There are some modifier operations (such as population counting), that can also chain load, and some modifier operations that *cannot* chain load. Section 6.7 lists the CDPEAC statement modifiers and indicates which can and can't chain load.

5.1.5 Instruction Suffixes

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

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

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

5.1.6 Argument Macros

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

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

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

5.2 CDPEAC Syntax

5.2.1 General Syntax

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

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

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

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

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

5.2.2 Vector Unit Data Registers

CDPEAC code refers to the 128 VU *data registers* by the following names:

`R0 - R127`	All 128 registers in sequential order.
`V0 - V15`	Vector regs (first in each vector, same as `R0`, `R8` ... `R120`).
`S0 - S15`	Scalar regs (single-precision), same as `R0 - R15`.
`S0 - S30` (even)	Scalar regs (double-precision), same as `R0 - R30` (even).

Restrictions: CDPEAC statements in immediate format use the `R0` and `R1` registers to store immediate arguments, so these registers should be avoided.

The VU data registers are grouped in banks of 8, called *vector* registers. The special register names `V0 - V15` are used to refer to the first data register in each vector. When a vector instruction requires an "aligned vector" argument, the argument must be one of the `Vnn` registers (or the equivalent `Rnn`).

V0	V1	V2	V3	V4	V5	V6	V7					V14	V15
R0	R8	R16	R24	R32	R40	R48	R56					R112	R120
R1	R9	R17	R25	R33	R41	R49	R57					R113	R121
R2	R10	R18	R26	R34	R42	R50	R58					R114	R122
R3	R11	R19	R27	R35	R43	R51	R59					R115	R123
R4	R12	R20	R28	R36	R44	R52	R60					R116	R124
R5	R13	R21	R29	R37	R45	R53	R61					R117	R125
R6	R14	R22	R30	R38	R46	R54	R62					R118	R126
R7	R15	R23	R31	R39	R47	R55	R63					R119	R127

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

A subset of these registers is designated as the *scalar registers*. These are `S0 - S15` (singleword), or the even registers from `S0 - S30` (doubleword). (The `Snn` names are equivalent to `Rnn`, and explicitly show use of scalar registers.) Scalar operations that use scalar registers assemble into efficient instructions.

5.2.3 Vector Unit Control Registers

There are symbols for all the VU control registers, as described in Section 2.5. However, these symbols are typically only used for accessor instructions such as `dpset` and `dpget`. CDPEAC `join` statement formats allow you to implicitly use and/or set the value of one or more control registers while executing a VU operation. See the mode set format in particular (Section 5.9) for examples.

5.2.4 Register Offset Macro

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

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

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

5.2.5 VU Register Stride Macros

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

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

In the above, *register* is any valid VU data register, and *stride* and *set-stride* are constant expressions in the range -128 to +128. Register striding is always in terms of the **R***nn* ordering, even when a **V***nn* register name is specified. A stride of zero causes the same register to be used at each step.

Important: The stride marker forms shown here are not valid for all statement formats — most statement formats restrict the types of stride markers that are allowed. In particular, the latter four forms are valid only for the *rS1* register argument of an arithmetic instruction.

rS1 Stride Restriction: When you apply a stride of 0 to the *rS1* argument of an arithmetic operation (for example, **dreg_u**(R0,0)), the *rS1* register must be one of the scalar registers **S0** through **S15**, or **S30** for double-precision.

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

5.2.6 VU Memory Striding

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

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

Instruction Suffix	Effect
mem-inst(*type*, *memop*, *reg*)	Use default stride on *memop*.
mem-inst`_u`(*type*, *memop*, *stride*, *reg*)	Use stride *stride* on *memop*.
mem-inst`_s`(*type*, *memop*, *stride*, *reg*)	Set default stride to *stride* and use it.
mem-inst`_u_s`(*type*, *memop*, *stride*, *set-stride*, *reg*)	Set default to *set-stride*, but use *stride* for this instruction.

For all the above suffix formats, the *stride* values that can be specified are restricted to `4` or `8` for singleword operations, and `8` or `16` for doubleword operations. When you write CDPEAC code by hand, you should make sure the default memory stride register `dp_stride_memory` is set to the stride you require (for example, 4 bytes for single-precision or 8 bytes for double-precision). You can use the CDPEAC accessor instruction `dpset` for this purpose; for example:

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

5.2.7 VU Selection in CDPEAC Statements

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

- the value of the *memory-argument* in the memory instruction.
- the value specified by the `maddr` modifier, if any.
- If neither of these is supplied, a default address that selects all the VUs. The default address used is `DPV_STACK_INST_PORT_ALL`.

Typically, you won't construct these memory addresses yourself; your high-level language compiler and/or the `dpcc` compiler generate these addresses for you.

5.2.8 VU Selection in CDPEAC Accessor Instructions

The VU(s) referenced by a CDPEAC accessor instruction are determined by the "VU selector" argument. This argument must be a valid VU selector, as described below.

A *VU selector* is an integer or symbolic constant that specifies one or more VUs to perform a given accessor instruction. The syntax is:

Syntax	Immediate Value
constant-expression	Use the specified selector constant (see table below).
C variable	Use value of specified variable.
*	Select all VUs.
**n*	Use both VUs on chip *n* (0=VU's 0&1, 1=VU's 2&3).

The *constant-expression* form can be either an integer VU selector value, a physical VU selector (an integer preceded by a "`$`"), or one of the symbols defined by the header file `cdpeac.h` for these values. (Use of predefined symbols is recommended.)

The legal VU selector values, and their corresponding symbols, are:

VU	VU Selector		Physical VU Selector	
Number(s)	Value	Symbol	Selector	Symbol
VU *n*	2**n*	`DP_`*n*	`$`*n*	`DP_PHYS_NUM_`*n*
ALL VUs	8	`ALL_DPS`	`$8`	`ALL_PHYS_NUM_DPS`
VUs 0 and 1	10	`DPS_0_AND_1`	`$9`	`DP_PHYS_NUM_0_AND_1`
VUs 2 and 3	12	`DPS_2_AND_3`	`$11`	`DP_PHYS_NUM_2_AND_3`

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

5.3 CDPEAC Instructions

5.3.1 Scalar and Vector Instructions

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

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

Vector instructions execute repeatedly for each of a series of arguments, and are distinguished by a "`v`" suffix on the instruction name. Vector instructions start with the specified register or memory address argument(s) and then step through succeeding locations determined by the *vector stride* and *vector length*:

- The *vector stride* determines the number of registers or memory addresses a vector instruction advances at each step. The default vector stride depends on the type of operation (memory or arithmetic).
- The *vector length* determines the number of registers or memory addresses affected by a vector instruction. The vector length defaults to the value of the VU register `dp_vector_length`, unless a different vector length is specified explicitly.

Note: If a CDPEAC `join` statement includes both a memory instruction and an arithmetic instruction, the two must agree in form: they must be either both scalar instructions or both vector instructions.

5.3.2 Register Arguments

The register arguments of CDPEAC arithmetic and memory instructions are indicated by the following symbols, indicating arbitrary VU registers:

rS1, rS2	First and second source registers.
rLS	Load/store (or third source) register.
rD	Destination register.
rIA	Indirect addressing (used in register indirect format).

When an instruction format requires vector (`v`*nn*) register arguments, the symbols *vS1*, *vS2*, *vLS*, *vD*, and *vIA* are used instead. Similarly, when scalar (`s`*nn*) register arguments are required, the symbols *sS1*, *sS2*, *sLS*, *sD*, and *sIA* are used.

5.3.3 Data Type Argument

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

`i`	Signed 32-bit integer	`u`	Unsigned 32-bit integer
`di`	Signed 64-bit integer	`du`	Unsigned 64-bit integer
`f`	Single (32-bit) float	`df`	Double (64-bit) float

5.3.4 Arithmetic Instructions

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

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

Note: In the statement format descriptions in Section 5.4, the arithmetic operation is always shown in triadic form. Dyadic and monadic forms are obtained simply by omitting the appropriate arguments (*rLS* and *rS2*).

(Appendix D describes the VU arithmetic instructions in detail, and describes the VU status bits that are affected by each instruction.)

Vector instructions have a default stride of 1 (singleword) or 2 (doubleword) for register arguments, unless the argument explicitly specifies a different stride (see Section 5.2.5.)

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

- For vector operations, *rS2* cannot be any of `R0` through `R7`, by any name (`S0`, `V0`, etc.).
- In scalar operations, *rS2* cannot be any of `R`*nn*, where *nn* is any multiple of `16` (for single-precision) or `32` (for double-precision).

Triadic/Memory Register Restriction Note: When a triadic arithmetic operation and a memory operation are `joined`, the *rLS* operand of the arithmetic operation must be identical to the *rLS* operand of the memory operation.

5.3.5 Memory Instructions

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

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

The *rLS* argument can be any VU register, but if a triadic arithmetic operation and a memory operation are combined, the *rLS* argument of both must be the same, and the memory operation can only be a `load`, not a `store`. The default stride for the *rLS* register is determined by the arithmetic operation, and the stride required by the memory operation must agree.

The *memory-argument* can be any memory address that selects one or more VUs, and it is specified by a C variable containing the address as an unsigned integer.

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

Singleword / Doubleword Performance Note: Doublewords are the natural word size for the VUs. Singleword operations require a read-modify-write step. Thus, singleword operations are less efficient than doubleword operations.

5.3.6 Modifiers

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

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

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

Conditionalization Modifiers (Mode Set Format Only):

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

Special Modifiers (Mode Set Format Only):

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

5.4 CDPEAC Statement Formats

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

A *short format statement* assembles into a single word (32-bit) operation. Short format statements execute faster than those in long format, but lack some of the features provided by the long format.

A *long format statement* assembles into a doubleword (64-bit) operation. Long format statements are slower is issue, but use the extra word to provide additional argument types and modifiers that are not permitted by the short format. Specifically, the long format comes in four varieties:

Immediate format allows an immediate argument in the arithmetic operation.

Register stride format allows register striding in the arithmetic operation.

Memory stride format allows address striding in the memory operation.

Mode set format provides access to a number of VU features, including register/memory indirection and overriding of many VU instruction defaults.

Each of the varieties of long format represents a modification of the short format. You can think of the short format as the backbone of features that all CDPEAC `join` statements are allowed to have, with each of the long formats representing some modification of or addition to those features.

Important! Because of the way that CDPEAC code is compiled and assembled, the modifications provided by each of the long formats cannot be combined. You can use only one of the long formats, or none of them (that is, use the short format) in a single `join` statement.

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

5.5 The Short Format

The *short statement format* is:

Vector Instuctions:

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

Scalar Instuctions:

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

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

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

The remaining register argument(s) must be aligned vector (`v`*nn*) registers for a vector operation, or scalar (`s`*nn*) registers for a scalar operation. Vector instructions always use unit striding, so stride markers are not allowed in short format (see register stride format, Section 5.7).

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

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

Only the following modifiers are permitted by the short format.

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

Note: `{vmcurrent, vmrotate}` are useful only for comparison operations, where the result of the comparison produces status bits that can be rotated.

Examples:

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

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

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

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

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

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

5.6 Immediate (Long) Format

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

Vector Instuctions:

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

Scalar Instuctions:

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

The *imm* argument is a 32-bit immediate value, either a C variable or a general expression. Immediate values are sign-extended in double integer arithmetic (zero-extended for double unsigned operations). For double-precision constants, only the upper 32 bits are included in the instruction. Thus, only floating-point numbers with 0's in the 32 least significant bits of their mantissas are allowed.

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

Examples:

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

5.7 Register Stride (Long) Format

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

Vector Instuctions:

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

Scalar Instuctions:

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

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

The *stride* macros on the *rS2*, *rLS*, and *rD* can be any of the register stride macros described in Section 5.2.5, except those that apply to *rS1* only. If a triadic arithmetic operation is used, the *rLS* stride must be the same for both the arithmetic and memory operations.

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

Examples:

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

5.8 Memory Stride (Long) Format

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

Vector Instuctions:

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

Scalar Instuctions:

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

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

Examples:

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

5.9 Mode Set (Long) Format

The *mode set format* is the most complex of the long formats. It allows you to do any or all of the following:

- Override and/or set the default vector length in `dp_vector_length`.
- Override the default conditionalization mode (`vmmode`).
- Override the default conditionalization sense (`vminvert`, `vmtrue`).
- Override the default vector mask copy mode (`vmold`, `vmnew`, `vmnop`).
- Use any of the modifiers permitted by the short format.

Mode set format also allows you to use one (and only one) of the following mutually incompatible extensions to the short format:

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

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

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

5.9.1 Mode Set Format Variants

The legal mode set variants are:

Vector Length Variant

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

This is the basic mode set variant, in which the only features used are those that are allowed in all mode set variants. In other words, this variant lets you specify an arbitrary vector length for a vector operation, and use general mode set modifiers like **vmnew**, **vminvert**, and **vmcurrent**. (The syntax for the *vec-len* specifier is described in Section 5.9.2.)

Examples:

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

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

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

rS1 Stride Variant

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

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

Examples:

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

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

Register Stride Indirection Variant

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

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

Examples:

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

Memory Stride Indirection Variant

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

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

Examples:

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

Population Count Variant

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

This variant allows you to specify the **epc{v,s}** modifier, which cannot be combined with a memory operation, or with any other mode set variant. (See Section 4.3.3.)

Examples:

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

Special Modifier Variant

joinn(*arith-inst*[_{**v,vs,vh,vhs**}]
(*type*, [*vlen*,] *rS1*, *vLS*, *vS2*, *vD*) ,
mem-inst[_{**v,vs,vh,vhs**}][_**u**]
(*type*, [*vlen*,] *mem-argument*, [*stride*,] *vLS*) ,
{[**no**]**exchange**, **vmcount**[**s**](*reg*)}
modifier ...)

This variant allows you to specify one (and only one) of the [**no**]**exchange** or **vmcount**[**s**] modifiers, which cannot be combined with any other mode set variant. (See Section 4.3.3.)

Examples:

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

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

Scalar Instruction Variant

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

5.9.2 Vector Length Instruction Suffixes

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

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

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

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

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

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

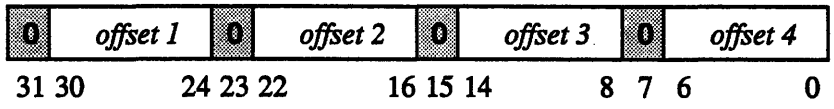
5.9.3 Register Stride Indirection

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

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

The *rIA* register argument contains offsets that are separately added to the *rS1* base register to obtain the actual `R`*nn* register containing the *rS1* stride. (**Note:** This offset addition is *not* cumulative.)

The register offsets are packed *four to a register* in the specified *rIA* register and in subsequent registers at the specified stride. Since offsets cannot exceed 127 (7 bits), the eighth bit of each offset byte must be zero:



Note: If a *stride* is not specified, then the "unit" stride is always 1 register for both single- and doubleword operations; one doubleword "register" corresponds to two singleword registers.

5.9.4 Memory Indirection

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

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

The specified single-precision VU *register* contains offsets that are separately added to the memory address to obtain each argument location. The addition is done in two's-complement, so negative offsets will work correctly. (**Note:** This offset addition is *not* cumulative.) The memory offsets are stored *one byte per register* in the specified *register* and subsequent registers at the specified *stride*.

Note: If a *stride* is not specified, then the "unit" stride is 1 single-precision register for single-precision memory operations, and 2 single-precision registers (1 double-precision register) for double-precision memory operations.

5.9.5 Mode Set Format Modifier

The following modifiers are permitted by the mode set format:

These modifiers are permitted by the short format:

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

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

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

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

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

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

Chapter 6

CDPEAC Instruction Set Reference

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

6.1 The CDPEAC Join Macro

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

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

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

6.2 CDPEAC Type Abbreviations

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

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

6.3 CDPEAC Argument Macros

Data Register Offsets:

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

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

Data Register Striding:

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

Data Register Indirection:

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

6.4 Instruction Suffixes

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

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

6.5 CDPEAC Arithmetic Instructions

6.5.1 Monadic (One-Source) Arithmetic Instructions

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

Formats:

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

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

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

Format:

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

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

6.5.2 Dyadic (Two-Source) Instructions

These operators perform an arithmetic operation on the *rS1* and *rS2* arguments, and store the result in the *rD* argument. (In immediate, `i`, format, the *rS2* argument is the immediate value.)

Formats:

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

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

6.5.3 Arithmetic Comparisons

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

Format:

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

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

6.5.4 Compare (Dyadic with rD constant)

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

Format:

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

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

6.5.5 Dyadic Mult-Op Operators

These operations perform a muliplication and an arithmetic (or logical) operation on the *rS1*, *rS2*, and *rD* arguments, and store the result in *rD*. (In immediate, `i`, format, the *rS2* argument is the immediate value.)

Format:

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

Note: In the opcode descriptions below, the optional [h] indicates that the high 64 bits of the multiplication are to be used in the logical operation, rather than the low 64 bits (the default).

Accumulative Operators

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

Inverted Operators

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

6.5.6 Convert Operation (Dyadic with Rs2 Constant)

These operations convert the `src` argument to the type indicated by the constant *code* argument, and store the result in the *rD* argument. The symbolic code constants listed below are defined by the `cdpeac.h` header file. (In immediate, `i`, format, the *rS1* argument is the immediate value.)

Format:

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

Opcode/Type	Code	Purpose
`cvt i[r]`	`CVTICD_F_I` (4)	Single float to single signed integer.
`cvt i[r]`	`CVTICD_F_U` (5)	Same, to unsigned integer.
`cvt i[r]`	`CVTICD_F_DI` (6)	Single float to double signed integer.
`cvt i[r]`	`CVTICD_F_DU` (7)	Same, to unsigned integer.
`cvt i[r]`	`CVTICD_DF_I` (12)	Double float to single signed integer.
`cvt i[r]`	`CVTICD_DF_U` (13)	Same, to unsigned integer.
`cvt i[r]`	`CVTICD_DF_DI` (14)	Double float to double signed integer.
`cvt i[r]`	`CVTICD_DF_DU` (14)	Same, to unsigned integer.
`cvt f`	`CVTFCD_F_DF` (3)	Single float to double float.
`cvt f`	`CVTFCD_DF_F` (9)	Double float to single float.
`cvt fi`	`CVTFICD_I_F` (1)	Single signed integer to single float.
`cvt fi`	`CVTFICD_U_F` (5)	Same, but from unsigned integer.
`cvt fi`	`CVTFICD_I_DF` (3)	Single signed integer to double float.
`cvt fi`	`CVTFICD_U_DF` (7)	Same, but from unsigned integer.
`cvt fi`	`CVTFICD_DI_F` (9)	Double signed integer to single float.
`cvt fi`	`CVTFICD_DU_F` (13)	Same, but from unsigned integer.
`cvt fi`	`CVTFICD_DI_DF` (11)	Double signed integer to double float.
`cvt fi`	`CVTFICD_DU_DF` (15)	Same, but from unsigned integer.

6.5.7 True Triadic (Three-Source) Operators

These operations perform a muliplication and an arithmetic (or logical) operation on the *rS1*, *rS2*, and *rLS* arguments, and store the result in *rD*. (In immediate, `i`, format, the *rS2* argument is the immediate value.)

Format:

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

Note: In the opcode descriptions below, the optional [h] indicates that the high 64 bits of the multiplication are to be used in the logical operation, rather than the low 64 bits (the default).

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

Triadic/Memory Register Restriction Note: When a triadic arithmetic operation and a memory operation are `joined`, the *rLS* operand of the arithmetic operation must be identical to the *rLS* operand of the memory operation.

6.5.8 No-Op Operator

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

Format:

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

6.6 CDPEAC Memory Instructions

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

Formats:

opcode{**s**,**v**} (*type*, *address*, *dreg*)
— use default memory stride.

opcode{**s**,**v**}**_u** (*type*, *address*, *stride*, *dreg*)
— use stride once.

opcode{**s**,**v**}**_s** (*type*, *address*, *stride*, *dreg*)
— use *stride* and store it as default.

opcode{**s**,**v**}**_u_s** (*type*, *address*, *stride*, *set_stride*, *dreg*)
— use *stride*, and store *set_stride* as default.

opcode{**s**,**v**}**_i** (*type*, *address*, *ireg*, *dreg*)
— memory stride indirection.

opcode{**s**,**v**}**_i** (*type*, *address*, **dreg_u** (*ireg*, *stride*) , *dreg*)
— memory indirection with *stride* on *ireg*.

opcode{**s**,**v**}**_**{**v**,**vs**,**vh**,**vhs**} (*type*, *vlen*, *address*, *dreg*)
— explicit vector length for CDPEAC statement.

opcode{**s**,**v**}**_**{**v**,**vs**,**vh**,**vhs**}**_i** (*type*, *vlen*, *address*, *ireg*, *dreg*)
— vector length and memory stride indirection.

opcode{**s**,**v**}**_**{**v**,**vs**,**vh**,**vhs**}**_u** (*type*, *vlen*, *address*, *cstride*, *dreg*)
— vector length and use-once *cstride*.

type = {**u**, **du**, **i**, **di**, **f**, **df**}

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

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

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

6.7 CDPEAC Statement Modifiers

This section describes the statement modifiers that can be `joined` with arithmetic and memory operations to affect their assembly and/or execution. **Note:** Some of these modifiers (such as the last three) can be used on their own.

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

`nopad`, `pad`(*pad-size*) Default: **`pad(4)`**

Vector Length Padding: Pads vector length of instruction to at least *pad-size*. Has no effect if vector length is already that size. Used to avoid instruction pipeline hazards. If not supplied, defaults to **`pad:4`**. The **`nopad`** variant is the same as **`pad:0`**. Pads between `0` and `4` are allowed, but have the same effect as **`pad:4`**.

`maddr` (*memory-address*) Default: None

Memory Operand Specifier: Used to supply a default memory operand for DPEAC statements that omit the memory instruction — this memory operand is used solely to determine VU selection.

{`vmrotate`, `vmcurrent`} Default: **`vmrotate`**

Status Bit Rotation Mode: Determines how status bits from vector operations are stored in the register **`dp_vector_mask`**. **`vmrotate`** "rotates" them in, **`vmcurrent`** inserts them in bit order. (See Figure 17.) **Note:** this modifier is allowed by the short format for conditional operations only. Otherwise, it can only be used in the mode set format.

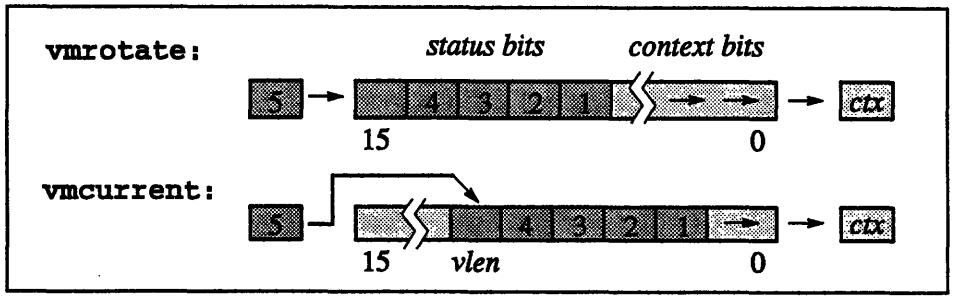


Figure 17. Bit-shifting modes of vector mask register.

[no]align Default: **noalign**

Doubleword Alignment Guarantee: Declares whether or not the memory operand is doubleword-aligned (even for singleword operations). If alignment is guaranteed, **dpas** can generate more efficient code. (**Note:** The default setting of this modifier can be reversed by providing the **-a** command line switch to **dpas**.)

6.7.2 Conditionalization Modifiers

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

vmmode[_s] (*mode-keyword*) Default: **vmmode(vmmode)**

Conditionalization Mode: The **vmmode** modifier overrides the value of the **dp_vector_mask_mode** register, which affects whether arithmetic operations and/or memory operations are to be conditionalized. The permitted *mode-keyword* operands are:

Mode	Effect
vmmode(vmmode)	Use current value of **dp_vector_mask_mode**.
vmmode(always)	Do not use conditionalization in this instruction.
vmmode_s(always)	Set **dp_vector_mask_mode** for no conditionalization.
vmmode(condmem)	Conditionalize loads and stores in this instruction.
vmmode_s(condmem)	Set **dp_vector_mask_mode** for conditionalization.
vmmode(condalu)	Conditionalize arithmetic in this instruction.
vmmode_s(condalu)	Set **dp_vector_mask_mode** for condit. arithmetic.
vmmode_s(cond)	Set **dp_vector_mask_mode** for full conditionalization.

It is not legal to override **dp_vector_mask_mode** for full conditionalization. Thus, "**vmmode(cond)**" is not allowed.

Usage Note: Scalar instructions are executed without conditionalization, so you may add **vmmode(always)** to any scalar instruction in any format with no effect. Similarly, you may add **vmmode(vmmode)** to any vector instuction in any format since it represents the default action taken by the hardware.

{**vminvert**, **vmtrue**} Default: **vmtrue**

Conditionalization Bit Sense: The **vminvert** and **vmtrue** modifiers control whether the conditionalization bits shifted out of the **dp_vector_mask** are inverted. If inverted, the sense of these bits is reversed; i.e., 0 selects a vector element, and 1 deselects it.

Modifier	Effect
vminvert	Invert sense of vector mask bits for conditionalization.
vmtrue	Do not invert sense of vector mask bits.

Note: This modifier is only allowed in the mode set statement format.

{**vmold**, **vmnew**, **vmnop**} Default: **vmold**

Vector Mask Copy Mode: The **vmold**, **vmnew**, and **vmnop** modifiers control the copying of the vector mask and vector mask buffer registers prior to instruction execution:

Modifier	Effect
vmold	Copy **dp_vector_mask_buffer** to **dp_vector_mask**.
vmnew	Copy **dp_vector_mask** to **dp_vector_mask_buffer**.
vmnop	No copy.

Note: This modifier is allowed only in the mode set statement format.

6.7.3 Special Modifiers (Mode Set Format Only)

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

Population Count: The **epc{v,s}** modifier enables the population count feature. Specifically, the single- or double-precision register *vLS* (and subsequent registers at a unit stride) are read and the "1" bits in each are counted. The results, each a single-precision unsigned integer between 0 and either 32 (single-precision) or 64 (double-precision), are written to the register *rIA* (and subsequent single-precision registers at the specified *stride*, a *constant-expression* that defaults to the unit stride for the data type).

The **epc{v,s}** modifier effectively replaces the normal memory operation in a DPEAC statement. The *Vls* register operand is used, so population counting cannot be combined with any memory operation. Population counting also cannot be used in conjunction with register or memory indirection or the

vmcount[s] or **[no]exchange** modifiers. The population count result is written before the operands are read for the arithmetic operation, so the **epc{v,s}** modifier chain loads. The *vLS* operand is always strided with a unit (1 or 2 register) stride, so the **:unit** keyword is optional and has no effect other than to emphasize the unit striding.

Implementation Note: Currently, the **epc{v,s}** modifier cannot be used in conjunction with a long-latency arithmetic operation, i.e., **[f,df]div**, **[f,df]sqrt**, **[f,df]inv**, or **[f,df]isqt**.

vmcount[s] (*reg*) Default: None

Accumulated Context Count: The **vmcount** modifier enables the VU chip's *accumulated context count* feature. The single-precision VU register *reg* (and subsequent registers at the given *stride*, a *constant-expression*) is loaded with the accumulated count of "1" bits in the vector mask at each step in the vector operation. This accumulation is inclusive; the count includes the bit that is shifted out of the vector mask register for each element. The scalar version, **vmcounts**, is intended for use with scalar operations. It is an error to use **vmcounts** with any vector operation.

For each element in the vector, the **vmcount** result is written before the operands are read for the arithmetic operation, so this modifier chain loads. This modifier cannot be used in conjunction with either register or memory indirection, nor with the **epc{v,s}**, or **[no]exchange** modifiers.

[no]exchange Default: **noexchange**

VU On-Chip Data Swapping: Controls exchange of data between two VUs on the same chip. Specifying **exchange** causes arithmetic results on each VU to be written to the destination register(s) of the other VU. In conditionalized ALU operations, deselected elements are not written to the opposite VU. Selected elements *are* written, even if the corresponding element in the opposite VU is deselected.

The **[no]exchange** modifier is used only in the mode set format. However, it is incompatible with register stride indirection, memory stride indirection, and with the **epc{v,s}**, and **vmcount[s]** modifiers.

Implementation Note: This modifier is implementation-dependent, and may not be available in the future. Also, the current implementation of exchanging does not allow chain loading into the arithmetic destination register.

6.8 CDPEAC Accessor Instructions

These accessor instructions are always used as single statements, execute on the node microprocessor (the SPARC), and generally move data between the SPARC and the VU, or affect values stored in SPARC registers.

6.8.1 VU Register Accessor Instructions

Instruction(s)	Function(s)
dpwrt[d], dprd[d]	Write and read VU data registers.
dpset[d], dpget[d]	Write and read VU control registers.
dpchgbk	Convert address from one VU region to another.
dpchgsp	Convert between VU data and instruction spaces.
dpld[d], dpst[d]	Read and write VU parallel memory.
dpsync	Synchronize instruction pipelines of VUs.

These instructions move data between VU data registers and SPARC registers:

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

```
dpwrt_sync(i,ALL_DPS,%i1,V0)
dpwrt_nosync(i,DPS_0_AND_1,29,V0)
dprd(i,DP_3,V0,%i0)
```

These instructions move data between VU control registers and SPARC registers. (See Section 2.5 for a list of predefined control register constants.)

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

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

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

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

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

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

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

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

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

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

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

This instruction generates code to prevent the preceding and following instructions from overlapping in the instruction pipeline of the VUs. (See Appendix C.)

dpsync()

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

6.9 CDPEAC Special Instructions

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

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

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

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

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

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

ldvm (dreg)
stvm (dreg)

```
ldvm    V1
stvm    V1
```

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

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

Chapter 7

Using DPEAC/CDPEAC in Programs

The most common use of DPEAC and/or CDPEAC in a CM-5 program is for writing highly efficient subroutines that are called from a program written in a high-level language. This chapter presents a simple example of just such a subroutine, shows how it can be written in either DPEAC or CDPEAC, and demonstrates how to call it from a CM Fortran program.

7.1 Example: An Arithmetic Subroutine

The subroutine described in this chapter calculates a specific arithmetic formula,

$$d = \frac{b^2 + c}{\sqrt{3.69a + 25.0b}}$$

elementwise across a set of four array arguments, `a`, `b`, `c`, and `d`. Each of these variables represents an element of a high-level array that is passed into the DPEAC or CDPEAC subroutine. The high-level program that calls this subroutine handles allocation of the arrays and subsequent processing of the results produced by the subroutine.

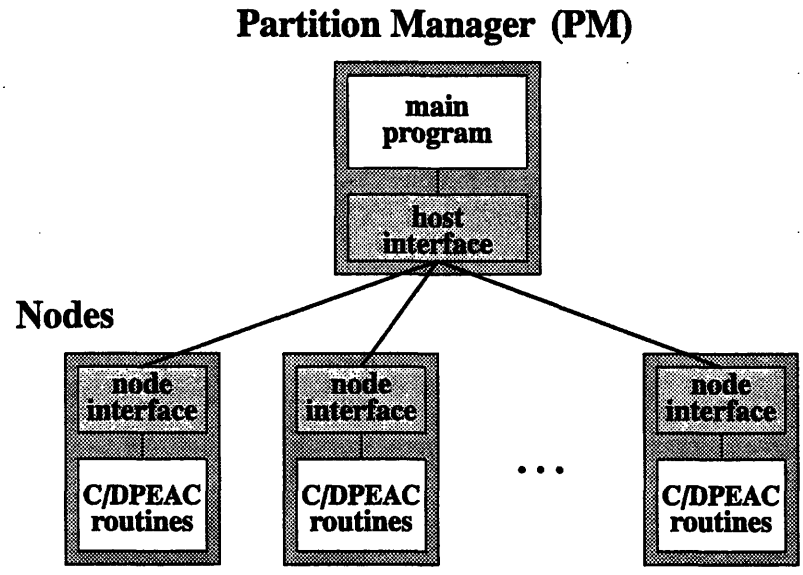
Note: You do *not* have to structure your programs as shown in this chapter to make use of the CM-5's vector units. The CM Fortran and C* compilers automatically define DPEAC routines in the process of compiling standard CM programs, and thus implicitly use the vector units whenever they are needed. The methods shown in this chapter allow you to duplicate the compiler's work for specific routines that you choose to write by hand.

7.2 Low-Level Program Structure

A CM-5 program that includes user-written DPEAC or CDPEAC routines has four main parts, each of which is contained in a separate source code file:

- The *DPEAC or CDPEAC subroutines*, which execute in identical copies on each of the processing nodes.
- The *node interface* functions, one for each DPEAC or CDPEAC routine, which define which node routines can be called from the PM.
- The *host interface* functions, on the PM, which broadcast a call to the node interface functions on all the nodes.
- The *main program*, written in a high-level language (such as CM Fortran), which calls the host interface functions to invoke the node subroutines.

The overall program structure is as shown below:



The host and node interface files describe the relationship between a specific set of function calls made on the PM, and a specific set of functions that are defined on the nodes. The interface files provide the "glue" that allows these function calls and definitions to compile and link correctly.

7.2.1 Program Files

Thus, a program with DPEAC or CDPEAC subroutines has four component source code files:

- A *main program* file, written in a high-level language.
- A *host interface* file, containing the definitions of all host interface functions called in the main program.
- A *node interface* file, containing the definitions for all node interface functions called in the host interface file.
- A *subroutine code* file, containing the definitions of all DPEAC/CDPEAC routines called from the node interface functions.

In addition, there is typically a *makefile* that is used to build the program via the UNIX **make** utility.

Source File Naming Conventions

The tools used to compile and link a program with DPEAC/CDPEAC routines impose the following restrictions on the program files:

- The host interface file must be written in C, and its name must end with the suffix "**.c**".
- The node interface file must also be written in C, but its name must end with the suffix "**.pe**". When the program is compiled, this file is run through a filter that produces a "**.c**" file for compilation.
- The subroutine code file must be written in DPEAC or CDPEAC, and must have the suffix "**.pe.dp**" (for a DPEAC routine file) or "**.pe.cdp**" (for a CDPEAC routine file).

It is a convention of the compilers and linkers used on the CM-5 that all object files containing code to be executed on the nodes must have the suffix "**.pe.o**". The suffix restrictions described above ensure that all object files produced in the compilation process will have the correct object file suffixes for the linker.

7.2.2 Host/Node Interface Naming Conventions

The tools used to compile and link a program with DPEAC/CDPEAC routines also impose the following restrictions on function names used in the program:

- The *host interface function* for each routine can have any legal name in the main program, but it must be defined in the host interface file with the same name, *in all lower case*, and with a trailing underscore "_" added.

  (For example, in the sample program below, the host interface function is called `NODECALC` in the CM Fortran source file, and `nodecalc_` in the host interface file.)

- The *node interface function* for each routine can have any legal function name in the host interface file, but its definition in the node interface file must have the same name with the prefix "`CMPE_`" attached.

  (In the sample program, the node interface function is called `nodecalc` in the host interface file, and `CMPE_nodecalc` in the node interface file.)

- The *DPEAC (or CDPEAC) subroutine* can have any legal function name in the node interface file, but its definition in the subroutine file must have the same name (and, in a DPEAC subroutine file, must have a leading underscore "_" character added).

  (In the sample program, the subroutine name used is `nodecalc` in the node interface file, `nodecalc` in the CDPEAC subroutine file, and `_nodecalc` in the DPEAC subroutine file.)

For the Curious: These special prefix and case requirements make it easy for the compiler and linker to determine which host and node interface functions correspond to which routines in the main program and the DPEAC code file.

The "`CMPE_`" prefix for the names of node functions callable from the host has the additional purpose of making it unlikely that a random function name used in the main program would happen to match a function name defined in the host/node interface.

7.3 Passing Arrays into DPEAC and CDPEAC Routines

The arguments of a DPEAC or CDPEAC subroutine depend on the manner in which the entire program is executing on the CM. For example, in programs that manipulate parallel arrays (such as the sample program in this chapter), the DPEAC or CDPEAC routine on each node handles the array subgrid that is stored in that node's memory.

In this case, the arguments to the DPEAC or CDPEAC subroutine are typically the memory addresses of the array subgrids located in the node's memory. The subroutines on each node handle the subgrids stored on that node, in such a way that every element of each array argument is handled by some node in the CM.

(There are other ways to pass data into DPEAC or CDPEAC routines. For example, you can use OS routines to allocate parallel memory yourself — Appendix H describes how to do this. You can then pass the addresses of these parallel memory regions into DPEAC or CDPEAC subroutines. However, this method of argument passing is not discussed further in this chapter.)

In CM Fortran, arrays are not referenced by the address of the array data itself, but instead by a pointer to a data structure known as an *array descriptor*. This descriptor contains, among other things, the address of the start of the array and the number of elements in the array.

Array descriptors are stored on the *partition manager*. The array location in the descriptor is a memory address in *node* memory. Thus, part of the job of the host interface function is to get the array location from the descriptors of any array arguments, and pass these memory addresses on to the node interface function.

The contents of an array descriptor can be extracted by calls to special accessor functions that are part of the CM Run-Time System (CMRTS). For example:

```
CMCOM_cm_address_t CMRT_desc_get_cm_location
(arr_desc);
int CMRT_desc_get_subgrid_size(arr_desc);
      CMRT_desc_t arr_desc;
```

CMRT_desc_get_cm_location returns the starting address (in node memory) of the array described by **arr_desc**.

CMRT_desc_get_subgrid_size returns the number of elements of the array that are stored in the memory of each VU (the "subgrid size" of the array). This value is required by the DPEAC routine, which must determine how many memory locations to operate on.

7.4 Sample Program Source Files

The sample program described below consists of five files:

- A main program written in CM Fortran: `main.fcm`
- A host interface file: `host.c`
- A node interface file: `interface.pe`
- A DPEAC subroutine file: `dpeac_code.pe.dp`
- A CDPEAC subroutine file: `cdpeac_code.pe.cdp`

The DPEAC and CDPEAC subroutine files contain the same routine, written appropriately for each of the two instruction sets.

The program is designed so that it can be compiled with either the DPEAC or the CDPEAC subroutine file; the main program and host/node interface files are identical in both cases. A sample makefile is also provided; this makefile can be used to compile the program with either (or both) of the subroutine files.

7.5 The Main CM Fortran Program (main.fcm)

The CM Fortran program `main.fcm` does three things:

- It initializes its arrays by assigning

```
a = 3.0
b = random numbers between 0.0 and 1.0
c = 19.0
d = 0.0
```

- It evaluates the formula `d = (b*b+c)/sqrt(3.69*a+25.0*b)` twice: first, by a ordinary CM Fortran expression (which is internally compiled into DPEAC code by the CM Fortran compiler); second, by a call to the host interface function `NODECALC`.

- The program then prints out the argument arrays and the computed results, for each of the two methods, to demonstrate that the two methods do in fact return the same values.

The **main.fcm** program is as follows:

```
      program main
      parameter (length=32)
      real a(length), b(length), c(length)
      real dh(length), dn(length)
      a=3.0
      b=0.0
      call CMF_RANDOM(b)
      c=19.0
c Host calculation
      dh=0.0
      dh=(b*b+c)/sqrt(3.69*a + 25.0*b)
c Node calculation
      dn=0.0
      call NODECALC(a,b,c,dn)
c Display results for comparison
      print *,''
      print *,'Computing d=(b*b+c)/sqrt(3.69*a + 25.0*b):'
      print *,' Item ',' A= ',' B= ',' C= ',' Host ',' 
Node '
      do 10 i=1,length
         print 900, i, a(i),b(i),c(i),dh(i),dn(i)
 10   continue
      print *,''
      stop
 900  format (i6,f6.2,f6.2,f6.2,f6.2,f6.2)
      end
```

7.6 The Host Interface File (host.c)

The host interface file contains a single function, **nodecalc_**, which does three things:

- It calls **CMRT_desc_get_cm_location** once for each of the array arguments to get the actual node memory locations of the arrays.
- Because all the array arguments must have the same size and shape, the host interface function calls **CMRT_desc_get_subgrid_size** just once to get the subgrid size of the array arguments.
- Finally, the host interface function makes a call to the corresponding **nodecalc** function to execute the DPEAC routine.

The **host.c** host interface file is as follows:

```
#include<cm/cmrt.h>
void nodecalc_ (a,b,c,d)
      CMRT_desc_t a,b,c,d;
{
  CMCOM_cm_address_t aloc,bloc,cloc,dloc;
  int size;

  /* get memory location for each array */
  aloc=CMRT_desc_get_cm_location(a);
  bloc=CMRT_desc_get_cm_location(b);
  cloc=CMRT_desc_get_cm_location(c);
  dloc=CMRT_desc_get_cm_location(d);

  /* subgrid size is same for all arrays */
  size=CMRT_desc_get_subgrid_size(a);

  /* call node interface function */
  nodecalc(aloc,bloc,cloc,dloc,size);
}
```

7.7 The Node Interface File (interface.pe)

The node interface file contains one node interface function, **CMPE_node**. **CMPE_node** takes the array addresses and subgrid size provided by the host function and passes them directly to the DPEAC (or CDPEAC) subroutine.

The **interface.pe** node interface file is as follows:

```
void CMPE_nodecalc(aloc,bloc,cloc,dloc,size)
      unsigned aloc,bloc,cloc,dloc,size;
{
  CMPE_nodecalc(aloc,bloc,cloc,dloc,size);
}
```

Note: This file is passed through a filter, **mkpestubs**, which converts it into an appropriate C code file. (This filtering step is handled internally by the makefile.)

7.8 The DPEAC Subroutine File (dpeac_code.dp)

This file contains the DPEAC version of the arithmetic subroutine:

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

      dpentry _CMPE_nodecalc,0,0 ! Entry point

! By convention, function args are in SPARC "input"
! registers, %i0, %i1, etc.

! Symbolic names for registers:
! Note that "%" prefix is used explicitly in code
! to make SPARC/VU register distinction clear.
# define A i0
# define B i1
# define C i2
# define D i3
# define Size i4

! By default, CM Fortran sets vector length to 8,
! and vector mask mode to "always". The following
! is insurance; when it is not needed, it is simply redundant

      set_vector_length_and_vmmode 8, always

#define VECTOR_LENGTH 8

! Formula being evaluated is:
! d=(b*b+c)/sqrt(3.69*a + 25.0*b)

Loop:
      floadv  [%B]:4, V2  ! (Short format, memory stride)
                  !  Load subgrid slice of B into V2,
                  !  striding by 4 bytes for each of
                  !  the 8 vector elements.
      add %B,(4*8),%B ! (SPARC instruction)
                  !  Bump array pointer B to next slice
                  !  in subgrid (4 bytes * 8 elements)
      floadv  [%C]:4, V3; \
      fmadav  V2,V2,V3! (Short format, chain-loading)
                  !  Load subgrid slice of C into V3,
                  !  striding by 4 for 8 elements,
                  !  and mult-add V3=(B*B)+C in same
                  !  operation.
      add %C,(4*8),%C ! (SPARC instruction)
                  !  Bump array pointer B to next slice
                  !  in subgrid (4 bytes * 8 elements)
```

```
floadv  [%A]:4, V4; \
fmulv   V4, 0r3.69, V5 \
            ! (Immediate format, chain-loading)
            !  Load subgrid slice of A into V4,
            !  striding by 4 for 8 elements,
            !  and multiply V5=(3.69*V4) in same
            !  operation.
add %A,(4*8),%A ! (SPARC instruction)
            !  Bump array pointer A to next slice
            !  in subgrid (4 bytes * 8 elements)
fmadav  V2, 0r25.0, V5  ! (Immediate format)
            !  Mult-add V5=(25.0*B)
fisqtv  V5, V5      ! (Short format)
            !  Calculate V5=1/SQRT(V5)
fmulv   V5, V3, V5  ! (Short format)
            !  Multiply V5=(V3*V5)
fstorev [%D]:4, V5  ! (Short format, memory stride)
            !  Store result in D subgrid slice
            !  striding 4 bytes for 8 elements.
addcc   %Size,-VECTOR_LENGTH,%Size \
            ! (SPARC instruction)
            !  Subtract vector length (8) from
            !  subgrid size argument to see if
            !  there are subgrid slices left
bne Loop    ! (SPARC instruction)
            !  If result is non-zero,
            !  go back and do next subgrid slice.
add %D,(4*8),%D ! (SPARC instruction, DELAY SLOT)
            !  Bump array pointer D to next slice
            !  in subgrid (4 bytes * 8 elements)

dpretn      ! (DPEAC Accessor Instruction)
            !  Return from DPEAC subroutine
```

A few notes on the structure of this program:

- Note that the **floadv** and **fstorev** instructions explicitly specify the memory stride as 4. An alternative to this would be to set the value of the **dp_stride_memory** register to 4.
- CM Fortran sets the following VU control registers to these defaults:

dp_vector_length	8
dp_stride_rs1	0
dp_stride_memory	0
dp_vector_mask_mode	always

dp_vector_mask_direction	right
dp_alu_mode_fast	fast, not IEEE

Nevertheless, it is always a good precaution to set these registers to the values you require within your DPEAC code routines, to avoid unnecessary surprises should these defaults change.

- Note that the array size is assumed to be a multiple of 8. Since the vector length is set to 8, there is no remainder, or "tail" of leftover elements. To handle a more general case, any "tail" of remaining values would need to be processed in a separate section of code, by resetting the vector length to the tail length, and repeating the calculation just once for the tail values.

7.9 The CDPEAC Subroutine File (cdpeac_code.cdp)

This file contains the CDPEAC version of the arithmetic subroutine:

```
#include <cm/cdpeac.h>
/* CDPEAC sample program.
   Formula being evaluated is:
   d=(b*b+c)/sqrt(3.69*a + 25.0*b) */
CMPE_nodecalc(aloc,bloc,cloc,dloc,size)
     unsigned aloc,bloc,cloc,dloc,size;
{
  /* Initialize SPARC registers for CDPEAC */
  dpsetup();

  /* By default, CM Fortran sets vector length to 8,
     and vector mask mode to "always". The following
     is insurance; when it is not needed, it is simply
     redundant */
  set_vector_length_and_vmmode(8,ALWAYS);

  /* Loop over each 8-element subgrid slice */
  for ( ; size ; size -= 8 );
  {
    loadv_u(f,bloc,4,V2); /* (Short format, memory stride)
                             Load subgrid slice of B into V2,
                              striding by 4 bytes for each of
                              the 8 vector elements. */
     bloc += (4*8);      /* Bump array pointer of B to new slice
                            in subgrid (4 bytes * 8 elements) */
```

```
    join2(          /* (Join of memory and ALU operations) */
       loadv_u(f,cloc,4,V3), /* (Short format, chain-loading)
                             Load subgrid slice of C into V3,
                             striding by 4 bytes for each of
                             the 8 vector elements. */
      madav(f,V2,V2,V3)    /*  Mult-add V3=(B*B)+C
                            in same operation. */
      );         /* (End of join2 macro) */

    cloc += (4*8);   /* Bump array pointer of C to next slice
                        in subgrid (4 bytes * 8 elements) */

    join2(          /* (Join of memory and ALU operations) */
       loadv_u(f,aloc,4,V4),
                         /* (Immediate format, chain-loading)
                            Load subgrid slice of A into V4,
                            striding by 4 for 8 elements. */
      mulvi(f,V4,3.69,V5)     /* multiply V5=(3.69*V4)
                              in same operation. */
      );         /* (End of join2 macro) */

     aloc += (4*8);      /* Bump array pointer of A to new slice
                            in subgrid (4 bytes * 8 elements)*/

    madavi(f,V2,25.0,V5);/* (Immediate format)
                            Mult-add V5=(25.0*B) */

     isqtv(f,V5,V5);     /* (Short format)
                            Calculate V5=1/SQRT(V5) */

     mulv(f,V5,V3,V5);       /* (Short format)
                             Multiply V5=(V3*V5) */

    storev_u(f,dloc,4,V5);   /* (Short format, memory stride)
                             Store result in D subgrid slice
                             striding 4 bytes for 8 elements. */

     dloc += (4*8);      /* Bump array pointer of D to new slice
                            in subgrid (4 bytes * 8 elements)*/
  }
  /* Clean up VU control registers -- NOTE: not always needed */
  dpcleanup()
}
```

7.10 Makefile for the Sample Program (Makefile)

Below is a sample **Makefile** that can be used with the UNIX utility program **make** to compile and link the sample program described above.

For those who have not used **make** before, all you have to do is place the five code files plus this **Makefile** into the same directory, set that directory as the current one (i.e., **cd** to it in UNIX), and then type **make** to build the program. (You will want to be logged on to a CM-5 partition manager when you do this, so that you will have access to the appropriate compilers and libraries.)

Note: When compiling this program with **make**, you can select either of the two subroutine code files by providing an appropriate argument. For example:

make dpeac	builds the DPEAC version of the program (**run_dp**).
make cpdeac	builds the CDPEAC version (**run_cdp**).

By default, this **Makefile** builds both executable versions of the program.

Once you have used **make** to build the executable files (**run_dp** and/or **run_cdp**), you can run the program by typing the appropriate executable file name. (Again, you'll want to be logged on to a CM-5 partition manager.)

The **Makefile** shown here performs a number of different operations to bring the pieces of the sample program together:

- The main CM Fortran program is compiled by **cmf** to produce two object files, one for the PM (**main.o**) and one for the nodes (**main.pe.o**).
- The host interface program is compiled by **cc**, producing an object code file (**host.o**) for the PM.
- The node interface program is compiled by **dpcc** and then passed through a stubs filter (**mkpestubs**) to produce a PM object file (**pe-call.o**).
- The appropriate subroutine file(s) are assembled by **dpas**, producing node object files (**dpeac_code.o** and/or **cdpeac_code.o**).
- Finally, the various object code files are linked together (again by **cmf**) to produce the executable file(s) (**run_dp** and/or **run_cdp**).

The **Makefile** also includes a number of "suffix rule" definitions, which describe how the various code source files are compiled.

The **Makefile** is as follows:

```
# ------------------------------------------
# Makefile to assemble C/DPEAC example programs
# By William R. Swanson, 5/5/93
# ------------------------------------------

# === Setup Definitions ===
# Don't display commands while building program
.SILENT:

# Alias macros that are used to clarify Make syntax
SOURCE_FILE   = $<
OBJECT_FILE   = $@

# debugging: set to -g to compile for debuggers
DEBUG     =

# === Target File Names ===
# Names of final executable files
DPEAC_EXECUTABLE  = run_dp
CDPEAC_EXECUTABLE = run_cdp

# Names of source and object files
MAIN      = main
HOST_INTF = host
NODE_INTF = interface
DPEAC_CODE= dpeac_code
CDPEAC_CODE   = cdpeac_code

# Object file sets
HOST_OBJS  = $(MAIN).o $(HOST_INTF).o $(NODE_INTF).o
DPEAC_NODE_OBJS     = $(MAIN).pe.o $(DPEAC_CODE).pe.o
CDPEAC_NODE_OBJS = $(MAIN).pe.o $(CDPEAC_CODE).pe.o
DPEAC_OBJS = $(HOST_OBJS) $(DPEAC_NODE_OBJS)
CDPEAC_OBJS     = $(HOST_OBJS) $(CDPEAC_NODE_OBJS)

# === Top-level Rules ===

# By default, trigger build of both executables
default: dpeac cdpeac

# To rebuild, do a clean and then trigger both builds
scratch: clean default

# Trigger build of just dpeac executable
dpeac: $(DPEAC_EXECUTABLE)

# Trigger build of just cdpeac executable
cdpeac: $(CDPEAC_EXECUTABLE)
```

```
# === Cleanup Rules ===

cleanobj:
      echo "Removing old objects, stub files, etc."
      rm -f *.o *.s $(NODE_INTF).c
      rm -f *# *#[0-9]* *% *~

clean: cleanobj
      echo "Removing old executable files"
      rm -f $(DPEAC_EXECUTABLE) $(CDPEAC_EXECUTABLE)

# === Program-specific Build Rules ===

# CMF driver is used to handle linking:
LINKER    = $(CMF)
LINKFLAGS = $(CMFFLAGS)

# Main linking step that builds the executable program:
$(DPEAC_EXECUTABLE): $(DPEAC_OBJS)
      echo "Linking ($(LINKER)) $(DPEAC_OBJS)"
      echo "to make executable file \"$(DPEAC_EXECUTABLE)\""
      $(LINKER) $(LINKFLAGS) $(DPEAC_OBJS) -o $(DPEAC_EXECUTABLE)

# Main linking step that builds the executable program:
$(CDPEAC_EXECUTABLE): $(CDPEAC_OBJS)
      echo "Linking ($(LINKER)) $(CDPEAC_OBJS)"
      echo "to make executable file \"$(CDPEAC_EXECUTABLE)\""
      $(LINKER) $(LINKFLAGS) $(CDPEAC_OBJS) -o
$(CDPEAC_EXECUTABLE)

# Host stubs obj file is produced from node interface file
interface.o: interface.c

# All other compilation steps are handled by suffix rules

# === Suffix Rules ===
# Add CMF and DPEAC suffixes to SUFFIX variable:
SUFFIXES += .fcm .dp .cdp .pe

# Clear out default suffix-list and install new list:
.SUFFIXES:
.SUFFIXES: $(SUFFIXES)

# To compile a C file, run it through $(CC)
CC        = cc
CFLAGS    = -DCM5_DASH -O $(DEBUG)
.c.o:
      echo "Compiling ($(CC)) $(SOURCE_FILE) into $(OBJECT_FILE)"
      $(CC) $(CFLAGS) -c $(OBJECT_FILE) $(SOURCE_FILE)
```

```
# To compile a CMF file, run it through $(CMF)
# Note: This step produces _two_ object files: .o and .pe.o
CMF       = cmf2.0
CMFFLAGS  = -cm5 -vu -O -Zcmld -Bstatic $(DEBUG)
NOLINK    = -c
LINK      =
.fcm.o:
      echo "Compiling ($(CMF)) $(SOURCE_FILE) into
$(OBJECT_FILE) and $(OBJECT_FILE:.o=.pe.o)"
      $(CMF) $(CMFFLAGS) $(NOLINK) $(SOURCE_FILE)

# To assemble a DPEAC file, run it through $(DPAS)
DPAS      = /usr/bin/dpas
DPFLAGS       = -t
.dp.o:
      echo "Assembling ($(DPAS)) $(SOURCE_FILE) into
$(OBJECT_FILE)"
      $(DPAS) $(DPFLAGS) -o $(OBJECT_FILE) $(SOURCE_FILE)

# To assemble a CDPEAC file, run it through $(DPCC)
# This produces one object file: .o
DPCC      = /usr/bin/dpcc
DPCCFLAGS =
.cdp.o:
      echo "Assembling ($(DPCC)) $(SOURCE_FILE) into
$(OBJECT_FILE)"
      $(DPCC) $(DPCCFLAGS) -o $(OBJECT_FILE) $(SOURCE_FILE)

# To process a DPEAC node interface file, run it through
$(MKSTUB)
MKSTUB    = /usr/bin/mkpestubs
MKSTUBFLAGS   = -n
.pe.c:
      echo "Processing ($(MKSTUB)) $(SOURCE_FILE) into
$(OBJECT_FILE)"
      echo '#include <cm/cmcom_types.h>' > $(OBJECT_FILE)
      $(MKSTUB) $(SOURCE_FILE) $(MKSTUBFLAGS) >> $(OBJECT_FILE)
```

7.11 Sample Run of the Program

Here is a UNIX session in which the sample program is built and run. (The following assumes that you have logged on to the partition manager of a CM-5, and are currently in a directory containing the five code files and the makefile.)

```
%: make clean
Removing old objects, stub files, etc.
Removing old executable files

%: ls
Makefile              host.c              out
cdpeac_code.pe.cdp    interface.pe
dpeac_code.pe.dp      main.fcm

%: make
Compiling (cmf2.0) main.fcm into main.o and main.pe.o
cmf [CM5 VecUnit 2.0 Beta 2]
Compiling main.fcm
Compiling (cc) host.c into host.o
Processing (/usr/bin/mkpestubs) interface.pe into \
      interface.c
Compiling (cc) interface.c into interface.o
Assembling (/usr/bin/dpas) dpeac_code.pe.dp into \
      dpeac_code.pe.o
Linking (cmf2.0) main.o host.o interface.o \
      main.pe.o dpeac_code.pe.o \
      to make executable file "run_dp"
cmf [CM5 VecUnit 2.0 Beta 2]
Linking...done.
Assembling (/usr/bin/dpcc) cdpeac_code.pe.cdp into \
      cdpeac_code.pe.o
Linking (cmf2.0) main.o host.o interface.o \
      main.pe.o cdpeac_code.pe.o \
      to make executable file "run_cdp"
cmf [CM5 VecUnit 2.0 Beta 2]
Linking...done.
24.1u 16.5s 2:22 28% 0+676k 21+652io 177pf+0w
```

```
%: run_dp

Computing d=(b*b+c)/sqrt(3.69*a + 25.0*b):
 Item    A=      B=      C=     Host   Node
    1   3.00   0.77  19.00   3.56   3.56
    2   3.00   0.77  19.00   3.55   3.55
    3   3.00   0.67  19.00   3.68   3.68
    4   3.00   0.59  19.00   3.81   3.81
    5   3.00   0.19  19.00   4.78   4.78
    6   3.00   0.44  19.00   4.09   4.09
    7   3.00   0.20  19.00   4.73   4.73
      . . .  < other values omitted >
   30   3.00   0.88  19.00   3.44   3.44
   31   3.00   0.99  19.00   3.34   3.34
   32   3.00   0.39  19.00   4.21   4.21

 FORTRAN STOP

%: run_cdp
Computing d=(b*b+c)/sqrt(3.69*a + 25.0*b):
 Item    A=      B=      C=     Host   Node
    1   3.00   0.06  19.00   5.34   5.34
    2   3.00   0.88  19.00   3.43   3.43
    3   3.00   0.24  19.00   4.60   4.60
    4   3.00   0.25  19.00   4.60   4.60
    5   3.00   0.54  19.00   3.90   3.90
    6   3.00   0.04  19.00   5.48   5.48
    7   3.00   0.91  19.00   3.41   3.41
      . . .  < other values omitted >
   30   3.00   0.50  19.00   3.97   3.97
   31   3.00   0.41  19.00   4.16   4.16
   32   3.00   0.06  19.00   5.38   5.38

 FORTRAN STOP
```

Appendixes

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`8`0*mmmmmm*	Instruction memory space.
N`0`0*mmmmmm*	Data memory space.
N`4`0000*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

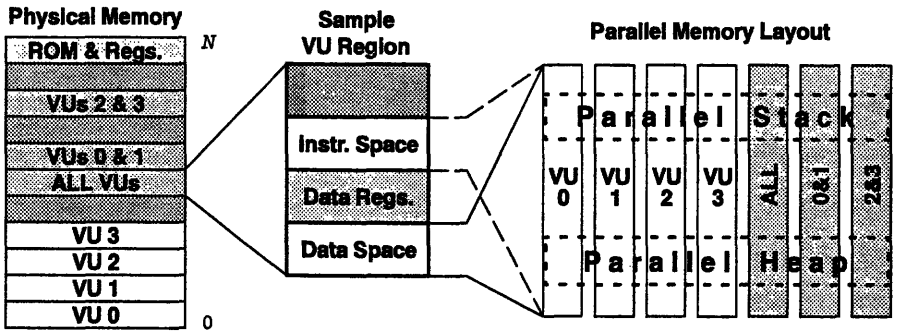
The data space and instruction space of a VU memory region in fact refer to the same piece of VU memory. A single memory location can thus be accessed in two ways: by an *instruction space* address, which triggers a VU operation, or by a *data space* address, which does not.

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

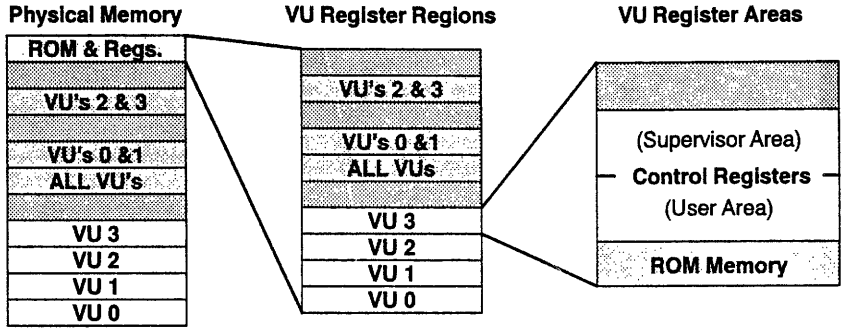
The memory region referred to by the data and instruction areas includes two regions: *parallel stack* and *parallel heap*. These occupy "stripes" of memory across the memory regions of all possible VU groupings:



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

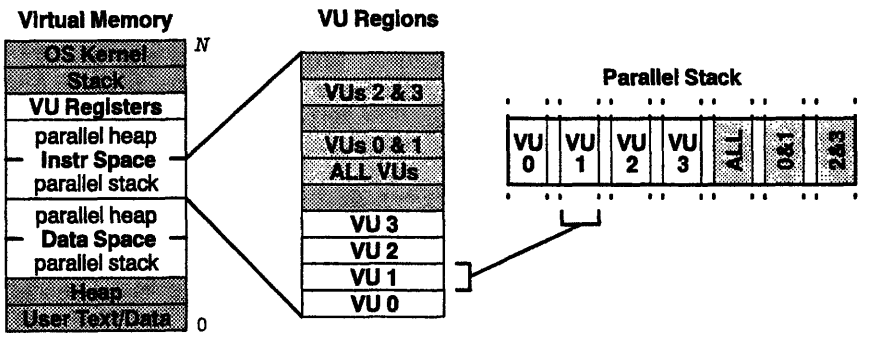
The virtual memory mapping for each CM node is established by CMOST, the CM-5 operating system. The VU memory and register regions are mapped into virtual memory by function, rather than by VU:

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.

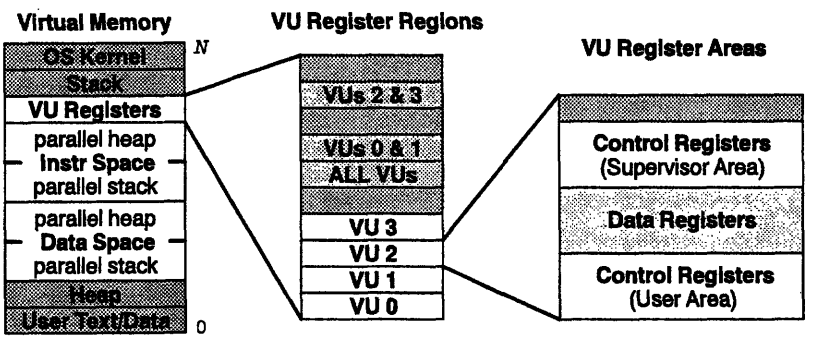
Each of the five virtual memory regions is divided into VU regions, with offsets as follows:

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:

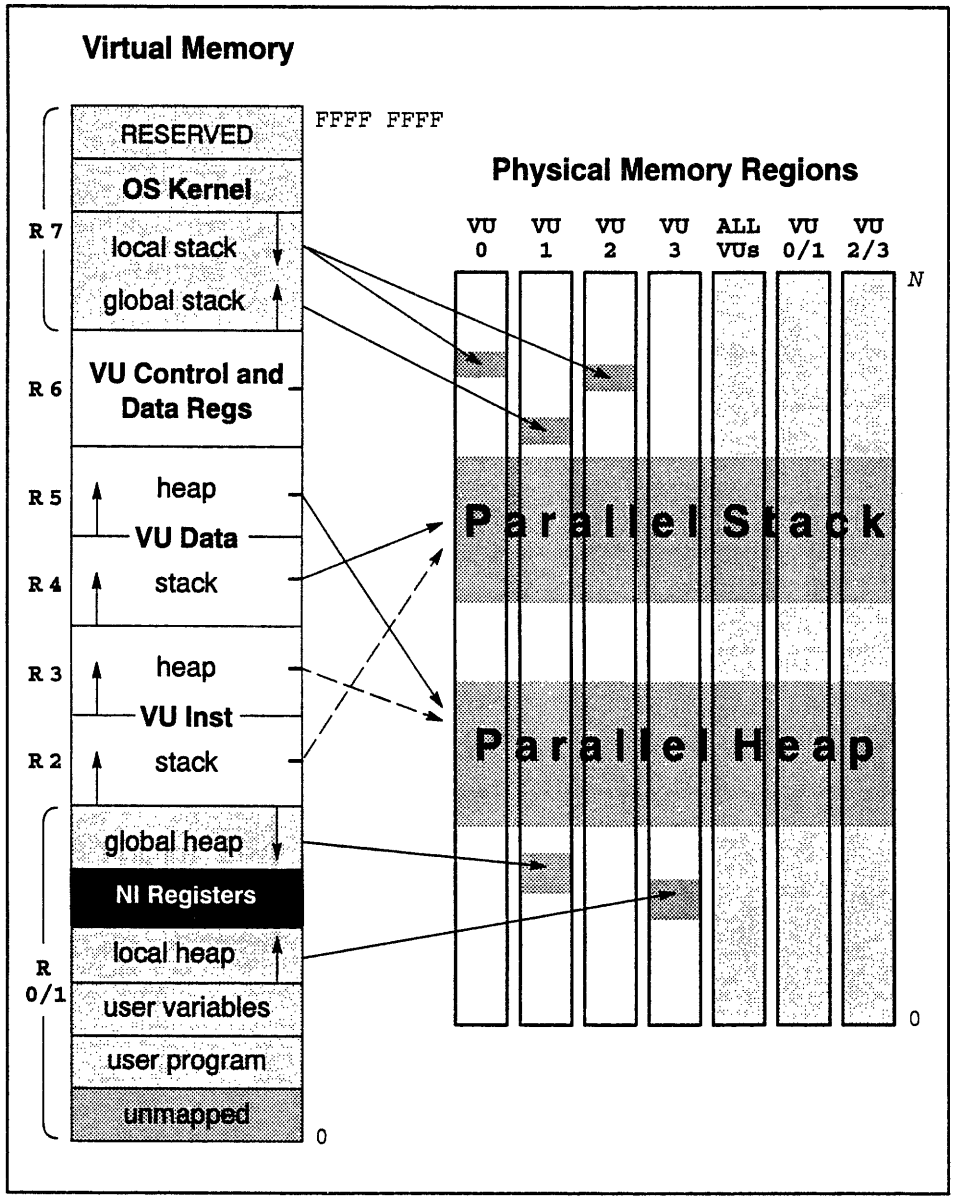
VU Region	Programming Constant Name	Address (hex)
VU 0	DPV_DATA_REGS_0	0xc0800000
VU 1	DPV_DATA_REGS_1	0xc4800000
VU 2	DPV_DATA_REGS_2	0xc8800000
VU 3	DPV_DATA_REGS_3	0xcc800000
All VUs	DPV_DATA_REGS_ALL	0xd0800000
VU 0/1	DPV_DATA_REGS_0_AND_1	0xd4800000
VU 2/3	DPV_DATA_REGS_2_AND_3	0xd8800000

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

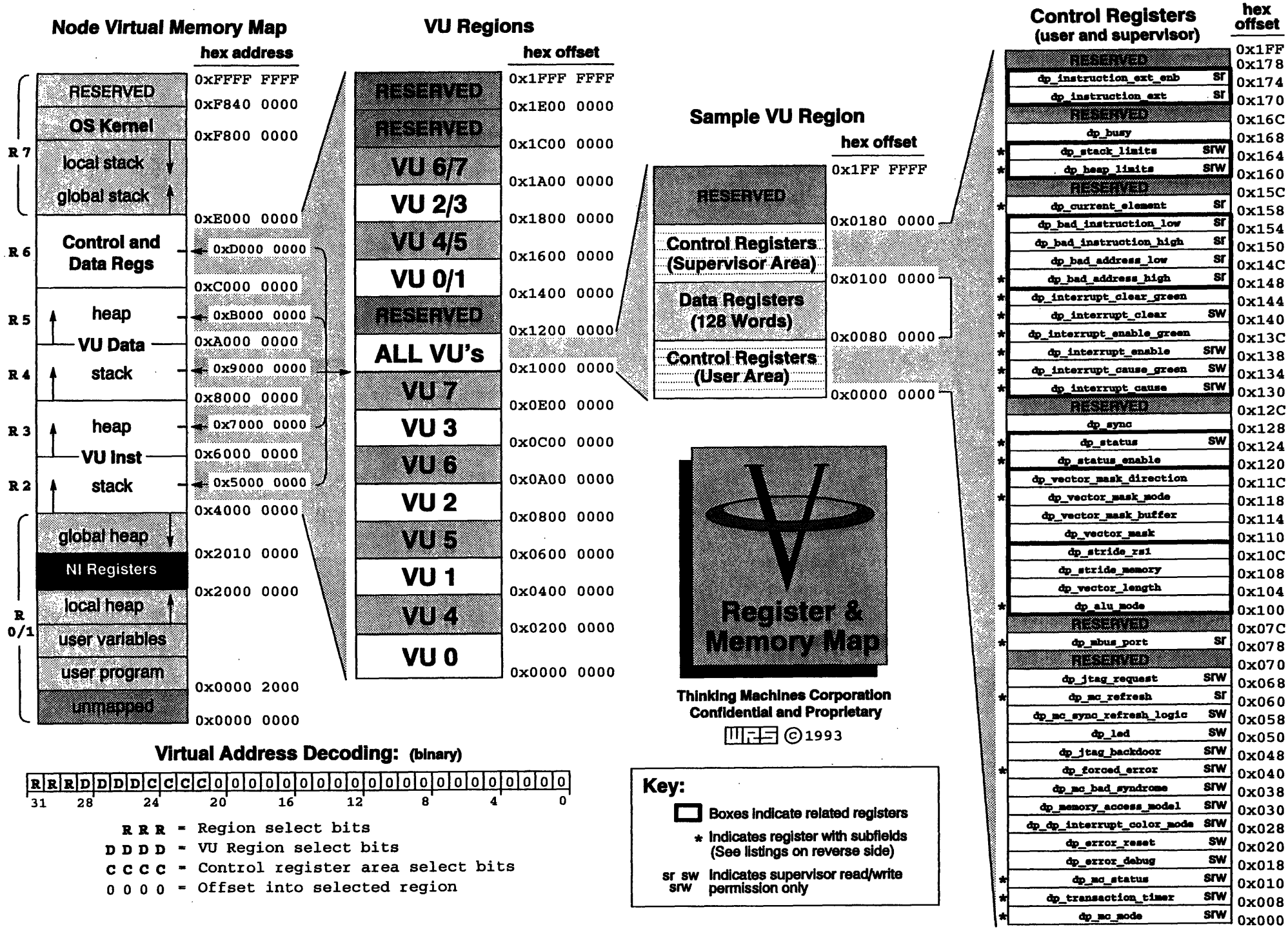
The diagram below summarizes the above description of the relationship between physical and virtual VU memory regions:



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.


Node Virtual Memory Map
hex address
RESERVED
OS Kernel
local stack
global stack
Control and Data Regs
heap
VU Data
stack
heap
VU Inst
stack
global heap
NI Registers
local heap
user variables
user program
unmapped
R 7
R 6
R 5
R 4
R 3
R 2
R 0/1
0xFFFF FFFF
0xF840 0000
0xF800 0000
0xE000 0000
0xD000 0000
0xC000 0000
0xB000 0000
0xA000 0000
0x9000 0000
0x8000 0000
0x7000 0000
0x6000 0000
0x5000 0000
0x4000 0000
0x2010 0000
0x2000 0000
0x0000 2000
0x0000 0000
VU Regions
hex offset
RESERVED
RESERVED
VU 6/7
VU 2/3
VU 4/5
VU 0/1
RESERVED
ALL VU's
VU 7
VU 3
VU 6
VU 2
VU 5
VU 1
VU 4
VU 0
0x1FFF FFFF
0x1E00 0000
0x1C00 0000
0x1A00 0000
0x1800 0000
0x1600 0000
0x1400 0000
0x1200 0000
0x1000 0000
0x0E00 0000
0x0C00 0000
0x0A00 0000
0x0800 0000
0x0600 0000
0x0400 0000
0x0200 0000
0x0000 0000
Sample VU Region
hex offset
RESERVED
Control Registers (Supervisor Area)
Data Registers (128 Words)
Control Registers (User Area)
0x1FF FFFF
0x0180 0000
0x0100 0000
0x0080 0000
0x0000 0000
Control Registers (user and supervisor)
hex offset
RESERVED
dp_instruction_ext_enb sr
dp_instruction_ext sr
RESERVED
dp_busy
* dp_stack_limits srw
* dp_heap_limits srw
RESERVED
* dp_current_element sr
dp_bad_instruction_low sr
dp_bad_instruction_high sr
dp_bad_address_low sr
* dp_bad_address_high sr
* dp_interrupt_clear_green
* dp_interrupt_clear sw
* dp_interrupt_enable_green
* dp_interrupt_enable srw
* dp_interrupt_cause_green sw
* dp_interrupt_cause srw
RESERVED
dp_sync
* dp_status sw
* dp_status_enable
dp_vector_mask_direction
* dp_vector_mask_mode
dp_vector_mask_buffer
dp_vector_mask
dp_stride_rs1
dp_stride_memory
dp_vector_length
* dp_alu_mode
RESERVED
* dp_mbus_port sr
RESERVED
dp_jtag_request srw
* dp_mc_refresh sr
dp_mc_sync_refresh_logic sw
dp_led sw
dp_jtag_backdoor srw
* dp_forced_error srw
dp_mc_bad_syndrome srw
dp_memory_access_model srw
dp_dp_interrupt_color_mode srw
dp_error_reset sw
dp_error_debug sw
* dp_mc_status srw
* dp_transaction_timer srw
* dp_mc_mode srw
0x1FF
0x178
0x174
0x170
0x16C
0x168
0x164
0x160
0x15C
0x158
0x154
0x150
0x14C
0x148
0x144
0x140
0x13C
0x138
0x134
0x130
0x12C
0x128
0x124
0x120
0x11C
0x118
0x114
0x110
0x10C
0x108
0x104
0x100
0x07C
0x078
0x070
0x068
0x060
0x058
0x050
0x048
0x040
0x038
0x030
0x028
0x020
0x018
0x010
0x008
0x000
Register & Memory Map
Thinking Machines Corporation
Confidential and Proprietary
©1993
Virtual Address Decoding: (binary)
R R R D D D D C C C C 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
31 28 24 20 16 12 8 4 0
R R R = Region select bits
D D D D = VU Region select bits
C C C C = Control register area select bits
0 0 0 0 = Offset into selected region
Key:
Boxes indicate related registers
* Indicates register with subfields (See listings on reverse side)
sr sw srw Indicates supervisor read/write permission only

Register: dp_mc_mode

Field Name:	Pos:	Size:
dp_mc_mode_correct	0	1
dp_mc_mode_timing	1	2
dp_mc_mode_size	3	3
dp_mc_mode_psize	6	2
dp_mc_mode_pnum	8	2
dp_mc_mode_reserved	10	1
dp_mc_mode_refresh	11	1
dp_mc_mode_preset	12	13

Register: dp_transaction_timer

Field Name:	Pos:	Size:
dp_transaction_timer_preset	0	16
dp_transaction_timer_mode	16	1

Register: dp_mc_status

Field Name:	Pos:	Size:
dp_mc_status_reserved	0	3
dp_mc_status_scount	3	8

Register: dp_forced_error

Field Name:	Pos:	Size:
dp_forced_error_ecc	0	8
dp_forced_error_mode	8	1

Register: dp_mc_refresh

Field Name:	Pos:	Size:
dp_mc_refresh_timer	16	13

Register: dp_mbus_port

Field Name:	Pos:	Size:
dp_mbus_mvend	0	4
dp_mbus_mrev	4	4
dp_mbus_mdev	8	8
dp_mbus_tmc	16	16

Register: dp_alu_mode

Field Name:	Pos:	Size:
dp_alu_mode_round	0	2
dp_alu_mode_fast	2	1

Register: dp_stride_memory

Field Name:	Pos:	Size:
dp_stride_memory_value	2	25

Register: dp_vector_mask_mode

Field Name:	Pos:	Size:
dp_vector_mask_mode_mem_cond	0	8
dp_vector_mask_mode_alu_cond	8	1

Registers: (same bit positions, all flags)
dp_status, dp_status_enable

Field Name:	Pos:
dp_status_inexact	0
dp_status_divide_by_zero	1
dp_status_underflow	2
dp_status_overflow	3
dp_status_invalid_operation	4
dp_status_integer_overflow	5
dp_status_negative_unsigned	6
dp_status_denorm_input	7
dp_status_zero_result	8
dp_status_pos_result	9
dp_status_neg_result	10
dp_status_int_carry	11
dp_status_infinite_result	12
dp_status_nan_result	13
dp_status_denorm_result	14
dp_status_unordered	15
dp_status_under	16
dp_status_deno	17

Register: dp_bad_address_high

Field Name:	Pos:	Size:
dp_bad_address_high	0	4
dp_bad_address_transaction_type	4	4
dp_bad_address_transaction_size	8	3

Register: dp_current_element

Field Name:	Pos:	Size:
dp_id_lsb	0	1
dp_element	1	4

Registers: (same bit positions, all flags)
dp_interrupt_{op}

Field Name:	Pos:
dp_debug_trap	0
dp_invalid_register_address	1
dp_invalid_memory_address	2
dp_invalid_local_access	3
dp_register_write_collision	4
dp_invalid_opcode	5
dp_memory_sco	6
dp_memory_single	7
dp_internal_fault	8
dp_memory_error	9
dp_jtag_interrupt	10
dp_invalid_remote_access (no enable)	11
ni_cause/clear_dr_count_negative	12

MBUS Transaction Fields

Register: dp_bad_address_high

Field Name: dp_bad_address_high
4 m.s. bits of physical address

Field Name: dp_bad_address_type

value:	transaction type:
0	write
1	read
2	coherent invalidate
3	coherent read
4	coherent write and invalidate
5	coherent read and invalidate
6-15	reserved

Field Name: dp_bad_address_type

value:	transaction type:
0	byte
1	halfword
2	word
3	doubleword
4	16 bytes
5	32 bytes
6	64 bytes
7	128 bytes

Registers: (same bit positions, all flags)
dp_interrupt_{op}_green

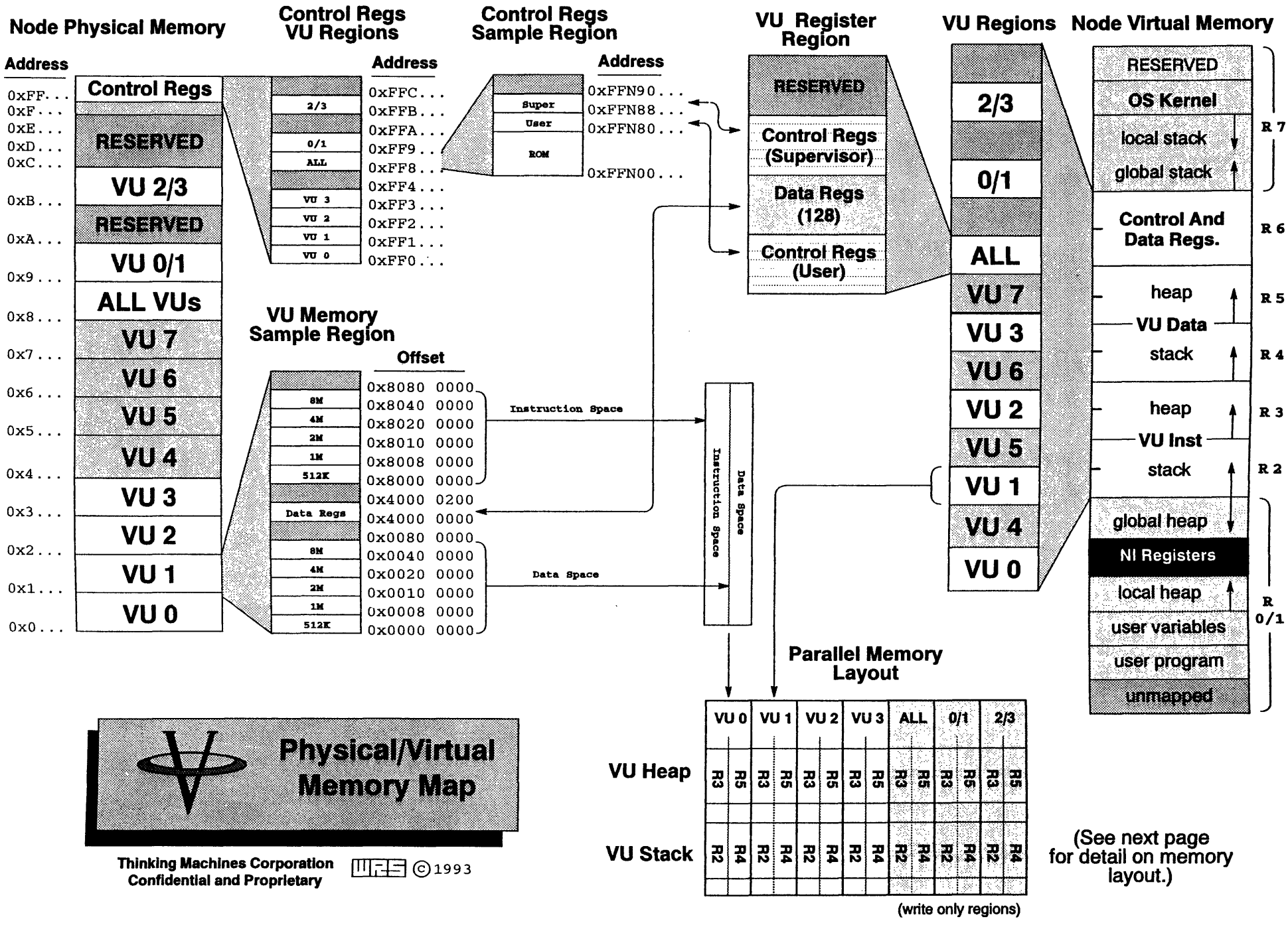
Field Name:	Pos:
dp_alu_inexact	0
dp_alu_divide_by_zero	1
dp_alu_underflow	2
dp_alu_overflow	3
dp_alu_invalid	4
dp_alu_integer_overflow	5
dp_alu_negative_unsigned	6
dp_alu_denorm_input	7
dp_set_enb	8

Register: dp_stack_limits

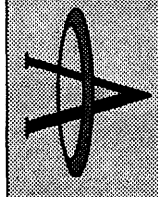
Field Name:	Pos:	Size:
dp_stack_limits_lower	0	15
dp_stack_limits_upper	16	15

Register: dp_heap_limits

Field Name:	Pos:	Size:
dp_heap_limits_lower	0	15
dp_heap_limits_upper	16	15


Node Physical Memory
Address
0xFF...
0xF...
0xE...
0xD...
0xC...
0xB...
0xA...
0x9...
0x8...
0x7...
0x6...
0x5...
0x4...
0x3...
0x2...
0x1...
0x0...
Control Regs
RESERVED
VU 2/3
RESERVED
VU 0/1
ALL VUs
VU 7
VU 6
VU 5
VU 4
VU 3
VU 2
VU 1
VU 0
Control Regs VU Regions
2/3
0/1
ALL
VU 3
VU 2
VU 1
VU 0
Address
0xFFC...
0xFFB...
0xFFA...
0xFF9...
0xFF8...
0xFF4...
0xFF3...
0xFF2...
0xFF1...
0xFF0...
Control Regs Sample Region
Super
User
ROM
Address
0xFFN90...
0xFFN88...
0xFFN80...
0xFFN00...
VU Register Region
RESERVED
Control Regs (Supervisor)
Data Regs (128)
Control Regs (User)
VU Regions
2/3
0/1
ALL
VU 7
VU 3
VU 6
VU 2
VU 5
VU 1
VU 4
VU 0
Node Virtual Memory
RESERVED
OS Kernel
local stack
global stack
R 7
Control And Data Regs.
R 6
heap
VU Data
stack
R 5
R 4
heap
VU Inst
stack
R 3
R 2
global heap
NI Registers
local heap
user variables
user program
unmapped
R 0/1
VU Memory Sample Region
Offset
8M
4M
2M
1M
512K
Data Regs
8M
4M
2M
1M
512K
0x8080 0000
0x8040 0000
0x8020 0000
0x8010 0000
0x8008 0000
0x8000 0000
0x4000 0200
0x4000 0000
0x0080 0000
0x0040 0000
0x0020 0000
0x0010 0000
0x0008 0000
0x0000 0000
Instruction Space
Data Space
Instruction Space
Data Space
Parallel Memory Layout
VU 0
VU 1
VU 2
VU 3
ALL
0/1
2/3
VU Heap
VU Stack
R3
R5
R2
R4
(write only regions)
Physical/Virtual Memory Map
Thinking Machines Corporation
Confidential and Proprietary
©1993
(See next page for detail on memory layout.)

Parallel Memory Addressing



(write only regions)

	VU Heap			VU Stack	
VU 2/3	0xA800 0000 DPV_HEAP_DATA_2_AND_3	R5	Data Space	R4	0x8800 0000 DPV_STACK_DATA_2_AND_3
	0x6800 0000 DPV_HEAP_INST_PORT_2_AND_3	R3	Instruction Space	R2	0x4800 0000 DPV_STACK_INST_PORT_2_AND_3
VU 0/1	0xA400 0000 DPV_HEAP_DATA_0_AND_1	R5	Data Space	R4	0x8400 0000 DPV_STACK_DATA_0_AND_1
	0x6400 0000 DPV_HEAP_INST_PORT_0_AND+_1	R3	Instruction Space	R2	0x4400 0000 DPV_STACK_INST_PORT_0_AND_1
ALL VUs	0xB000 0000 DPV_HEAP_DATA_ALL	R5	Data Space	R4	0x9000 0000 DPV_STACK_DATA_ALL
	0x7000 0000 DPV_HEAP_INST_PORT_ALL	R3	Instruction Space	R2	0x5000 0000 DPV_STACK_INST_PORT_ALL
VU 3	0xAC00 0000 DPV_HEAP_DATA_3	R5	Data Space	R4	0x8C00 0000 DPV_STACK_DATA_3
	0x6C00 0000 DPV_HEAP_INST_PORT_3	R3	Instruction Space	R2	0x4C00 0000 DPV_STACK_INST_PORT_3
VU 2	0xA800 0000 DPV_HEAP_DATA_2	R5	Data Space	R4	0x8800 0000 DPV_STACK_DATA_2
	0x6800 0000 DPV_HEAP_INST_PORT_2	R3	Instruction Space	R2	0x4800 0000 DPV_STACK_INST_PORT_2
VU 1	0xA400 0000 DPV_HEAP_DATA_1	R5	Data Space	R4	0x8400 0000 DPV_STACK_DATA_1
	0x6400 0000 DPV_HEAP_INST_PORT_1	R3	Instruction Space	R2	0x4400 0000 DPV_STACK_INST_PORT_1
VU 0	0xA000 0000 DPV_HEAP_DATA_0	R5	Data Space	R4	0x8000 0000 DPV_STACK_DATA_0
	0x6000 0000 DPV_HEAP_INST_PORT_0	R3	Instruction Space	R2	0x4000 0000 DPV_STACK_INST_PORT_0

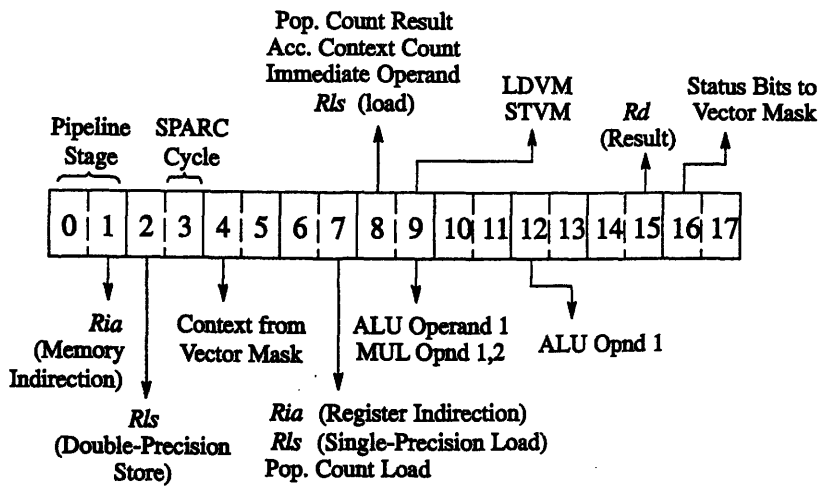
Appendix C

VU Pipeline

C.1 VU Instruction Pipeline

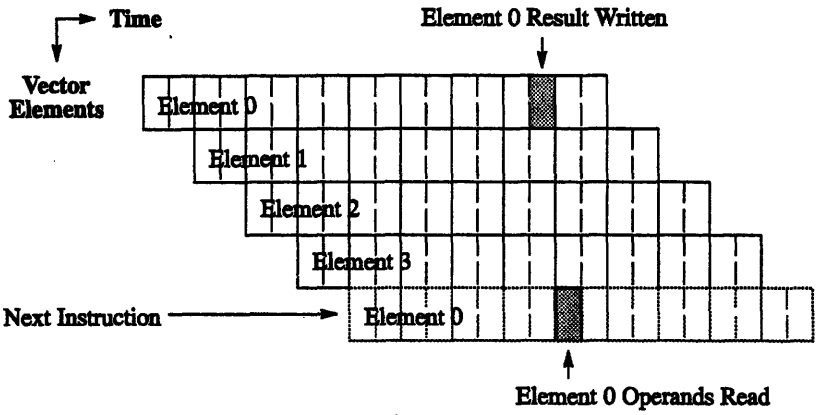
The VU accelerator chips execute vector instructions in a pipelined fashion, so that the operations on successive elements of vector operands can begin on successive clock cycles.

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 $2N*k$ Mbus (SPARC) cycles to execute, rather than the usual $2N$, 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 ORing 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 $2Nk$ 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

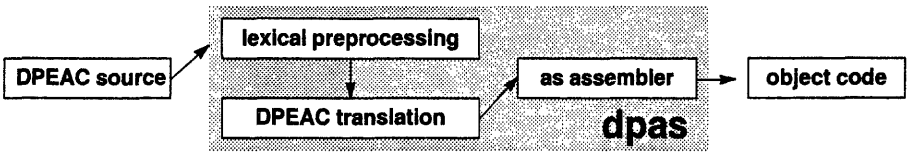
This operation, which is not a vector operation, forces a trap to occur. This trap will be a green interrupt identical to the trap caused by a masked status trap. Possible status outputs: None.

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 **#defined**.
#ifndef	*symbol*	— Assemble if *symbol* is not **#defined**.
#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 ifxxx	*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.

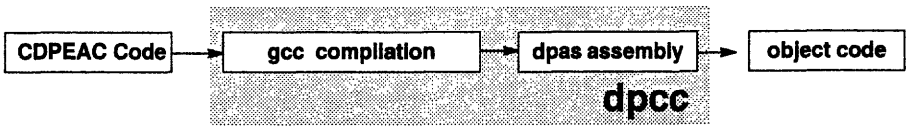
dpas pre-defines the symbol "**dpas**" to the string "**1**" before preprocessing a file; this symbol can be used to conditionally assemble code depending on whether or not **dpas** is being used as the assembler.

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.

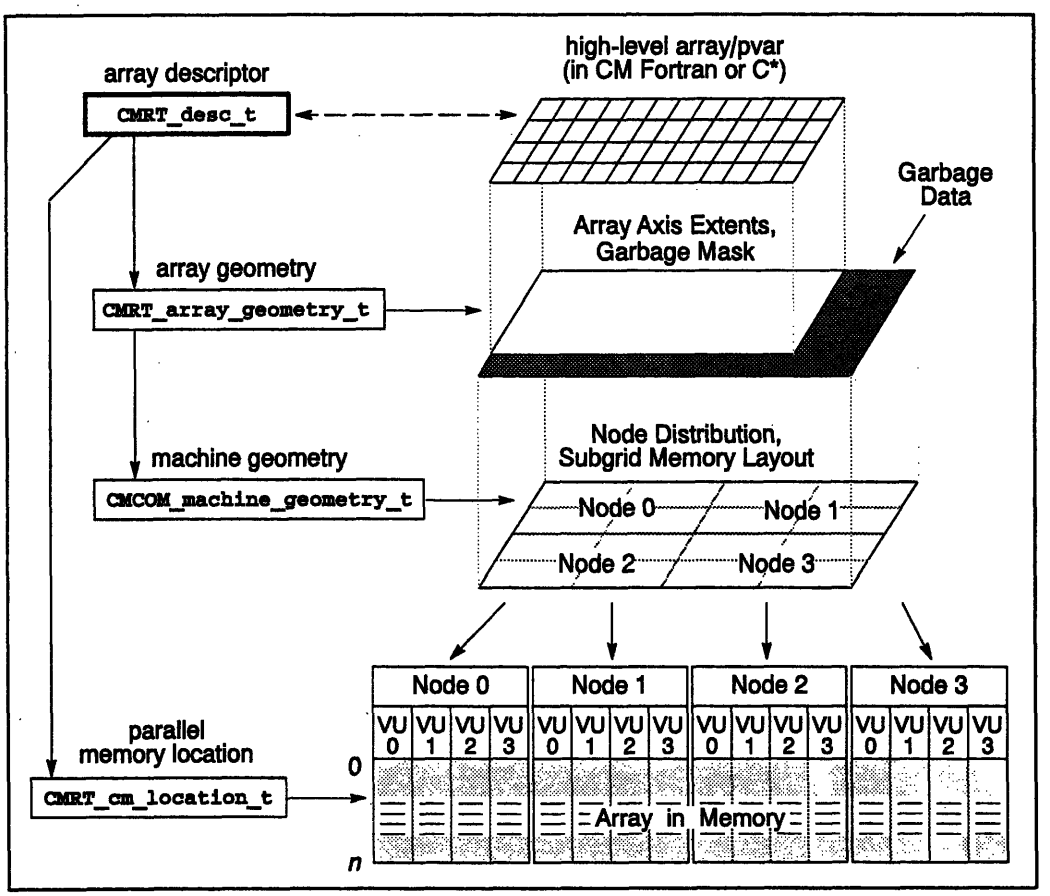


Figure 18. Internal Structure of a CM Array.

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.)

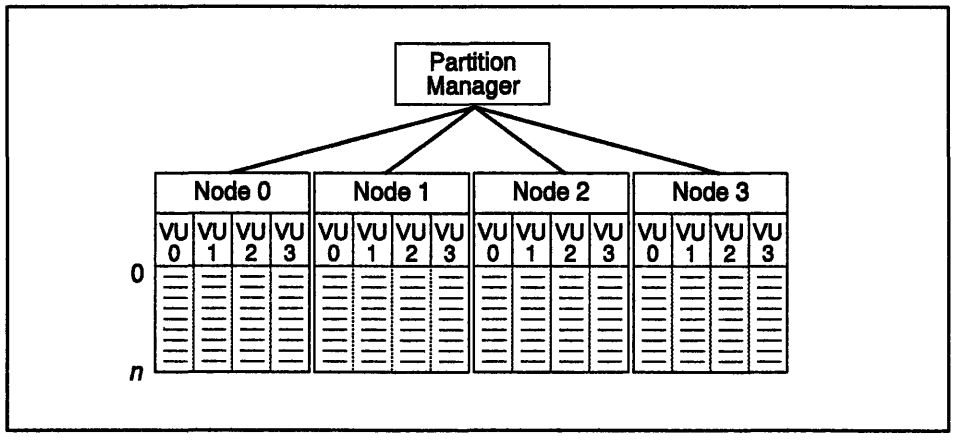
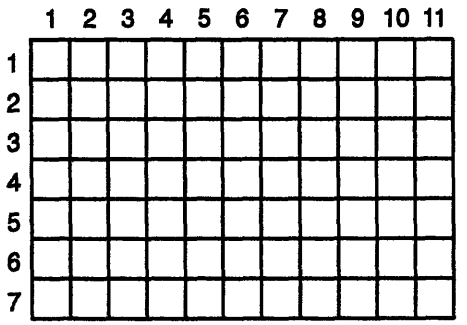


Figure 19. A hypothetical 4-node CM-5 system (with VUs).

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:



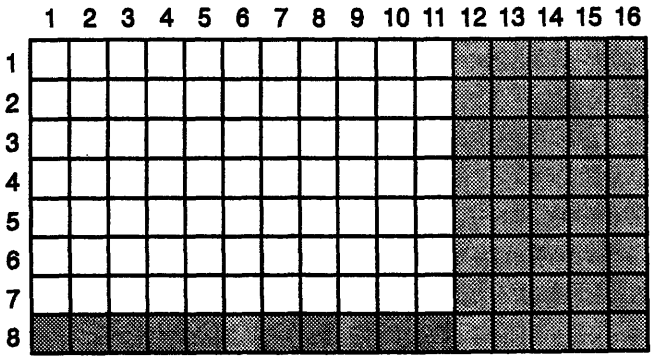
It's a two-dimensional array of 77 elements, each of which is a double-precision floating-point number. This is not evenly divisible across 16 VUs, so a small number of garbage elements will need to be added. In general, the garbage space of an array is added by extending the axes of the array, adding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

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

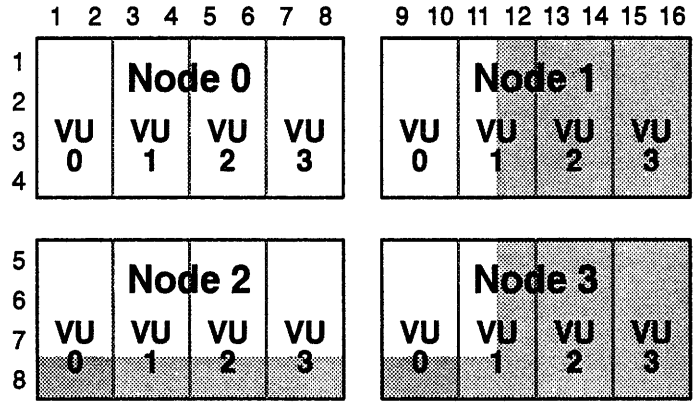
- Enough garbage space must be added so that the array can be divided into pieces of equal size and shape for each CM node.
- The part of the array assigned to each node must furthermore be divisible into 4 parts of equal size and shape, one for each of the node's VUs.
- Each node's portion of the array 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):

		subgrid axis 1			
		1		2	
subgrid axis 0	1	1	2	3	4
	2	5	6	7	8
	3	9	10	11	12
	4	13	14	15	16

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

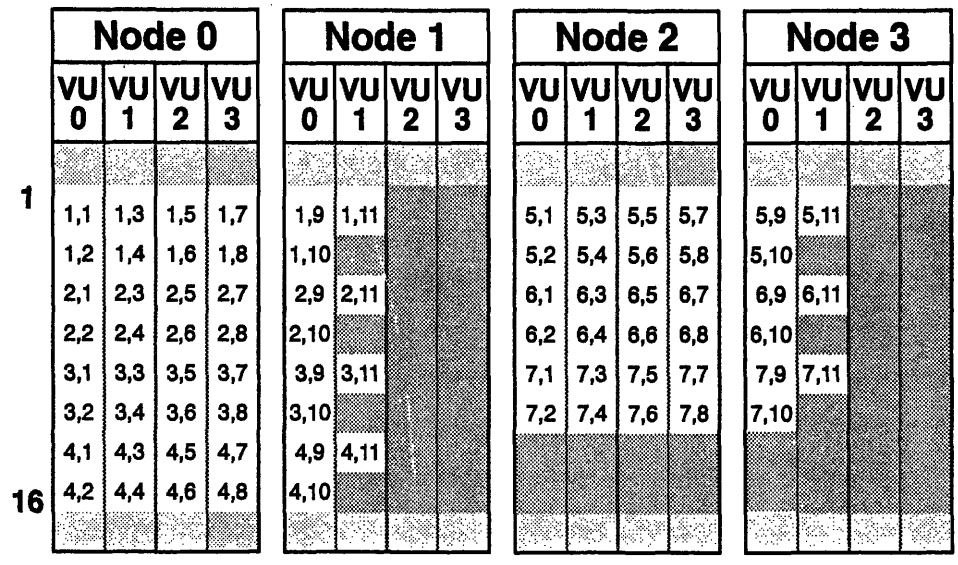


Figure 20. The actual memory distribution of a 7-by-11 CM Fortran array.

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).

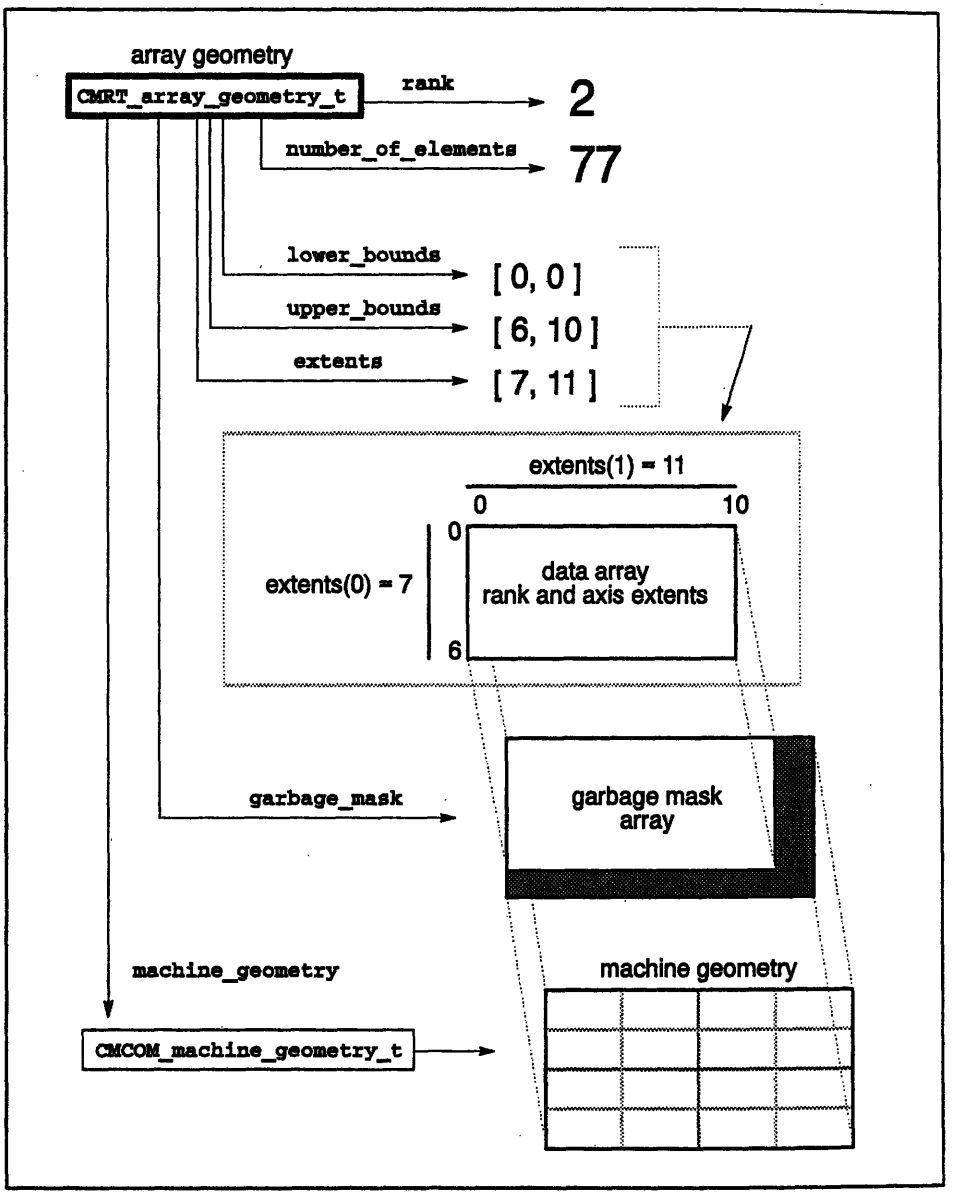


Figure 21. The CMRT_array_geometry data structure.

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).

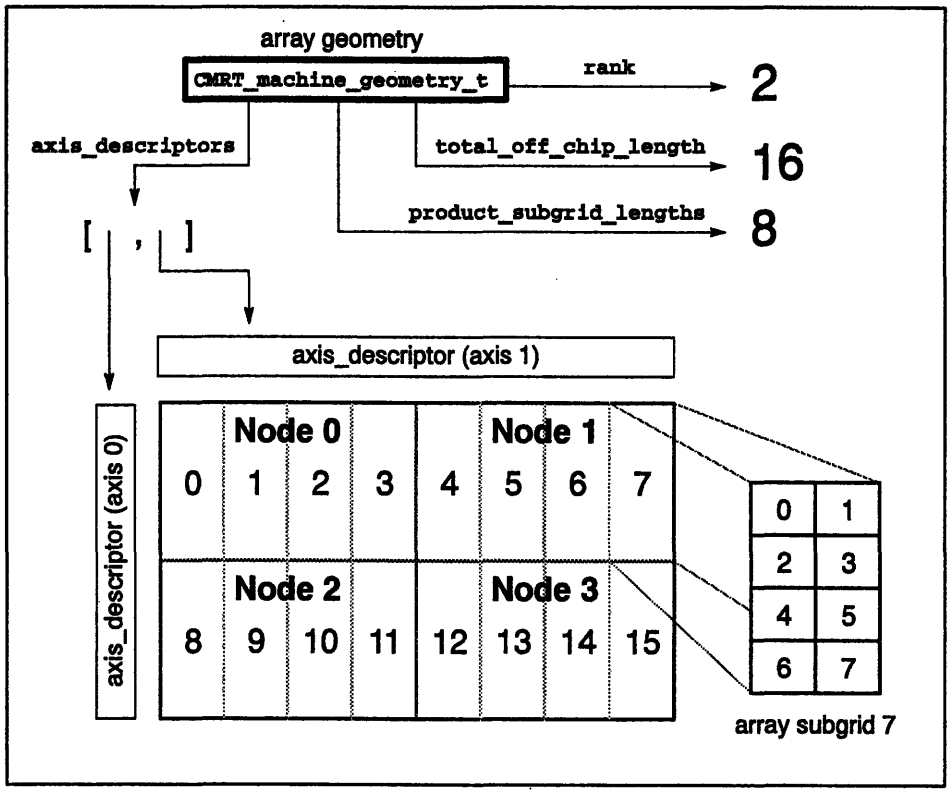


Figure 22. The CMCOM_machine_geometry data structure.

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.

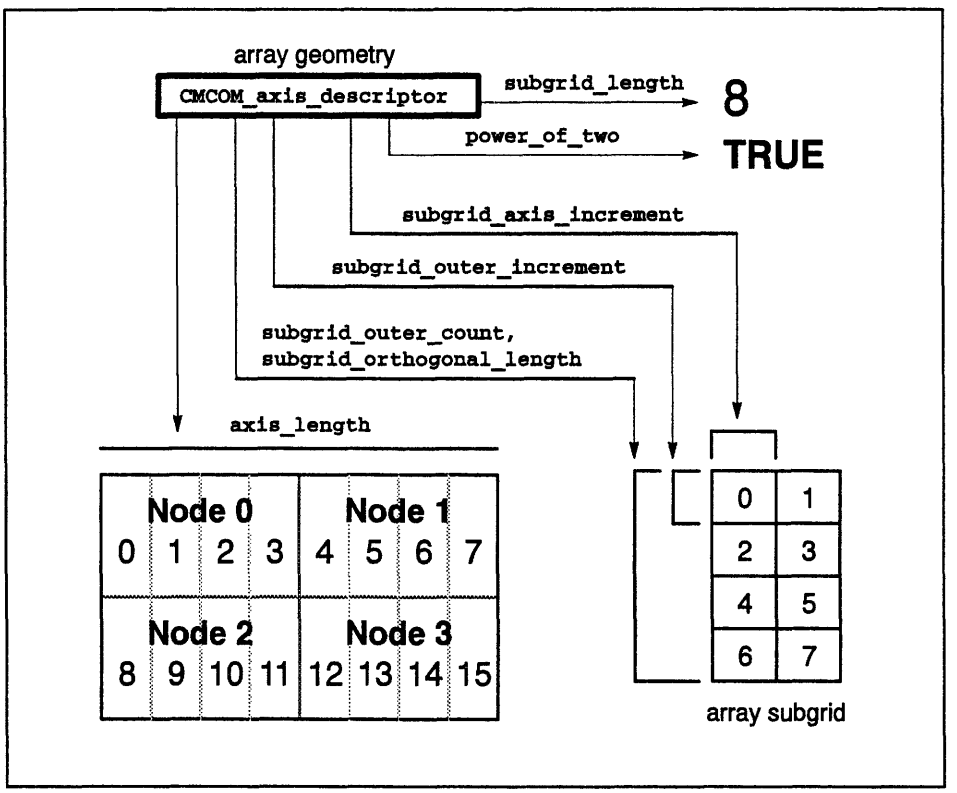


Figure 23. The CMCOM_axis_descriptor data structure.

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.

Index

Index

Symbols

A

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V