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.

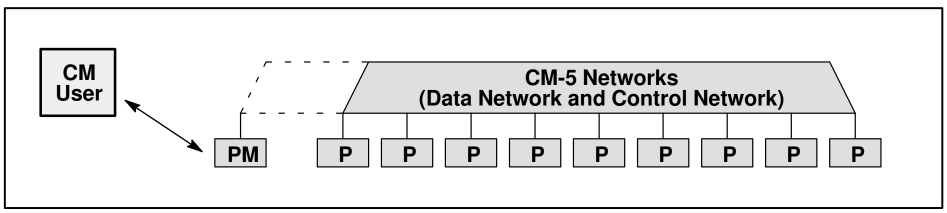


Figure 1. The CM-5 computing environment.

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.

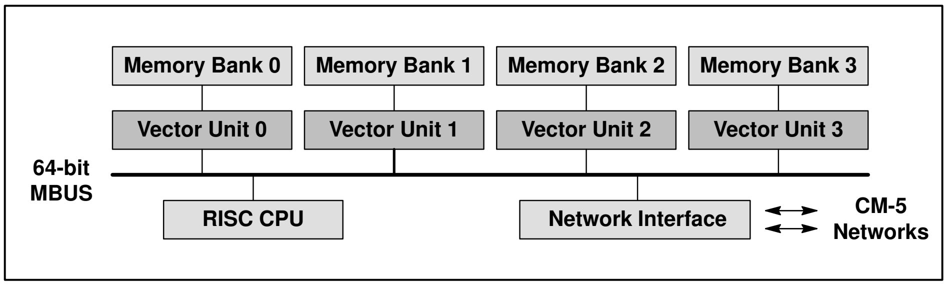


Figure 2. A typical CM-5 processing node.

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.

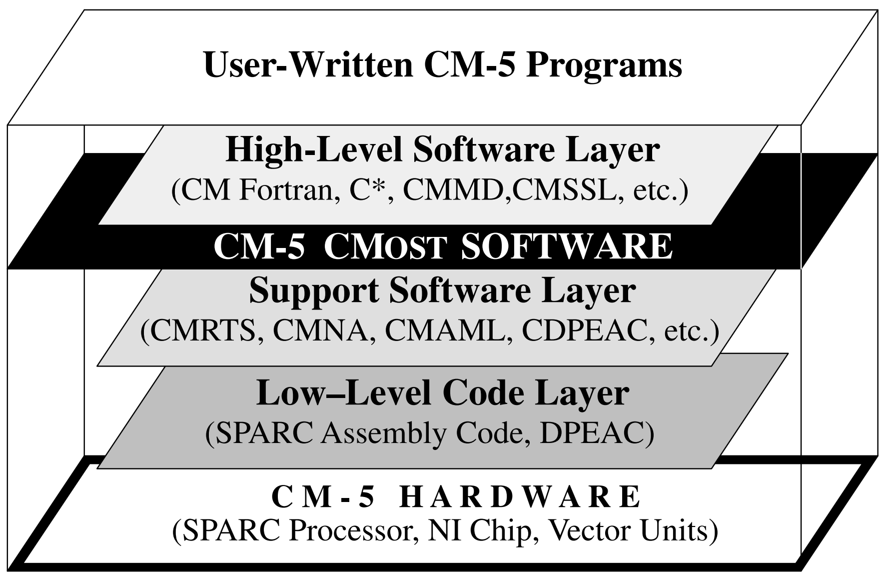


Figure 3. Structure of software layers on the CM-5.

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
```

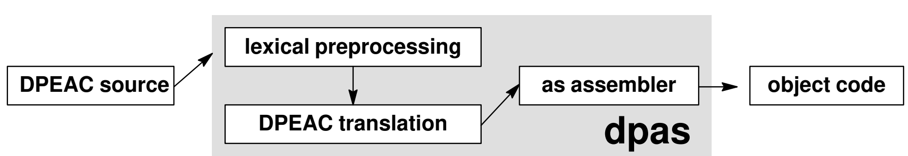
However, DPEAC instructions are not executed directly by the SPARC. Instead, they are assembled into singleword or doubleword values that can be written to the VUs to cause them to execute the appropriate arithmetic and/or memory operations.

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:

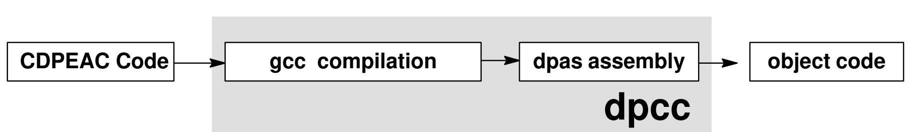
```
CDPEAC_routine(aloc,bloc,size)
unsigned aloc,bloc,size;
{ dpsetup();
  for ( ; size ; size -= 8 ); {
    loadv_u(f,aloc,4,V2);
    join2( loadv_u(f,bloc,4,V3), madav(f,V2,V2,V3) );
    storev_u(f,bloc,4,V3);
    aloc += (4*8); bloc += (4*8);
  }
}
```

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.