CONVEX C-1 SUPERCOMPUTER SYSTEM OVERVIEW

CONVEX

CONVEX C-1 SUPERCOMPUTER SYSTEM OVERVIEW

Part Number: DSW-100

Revision Number: 1

First Printed
July, 1985

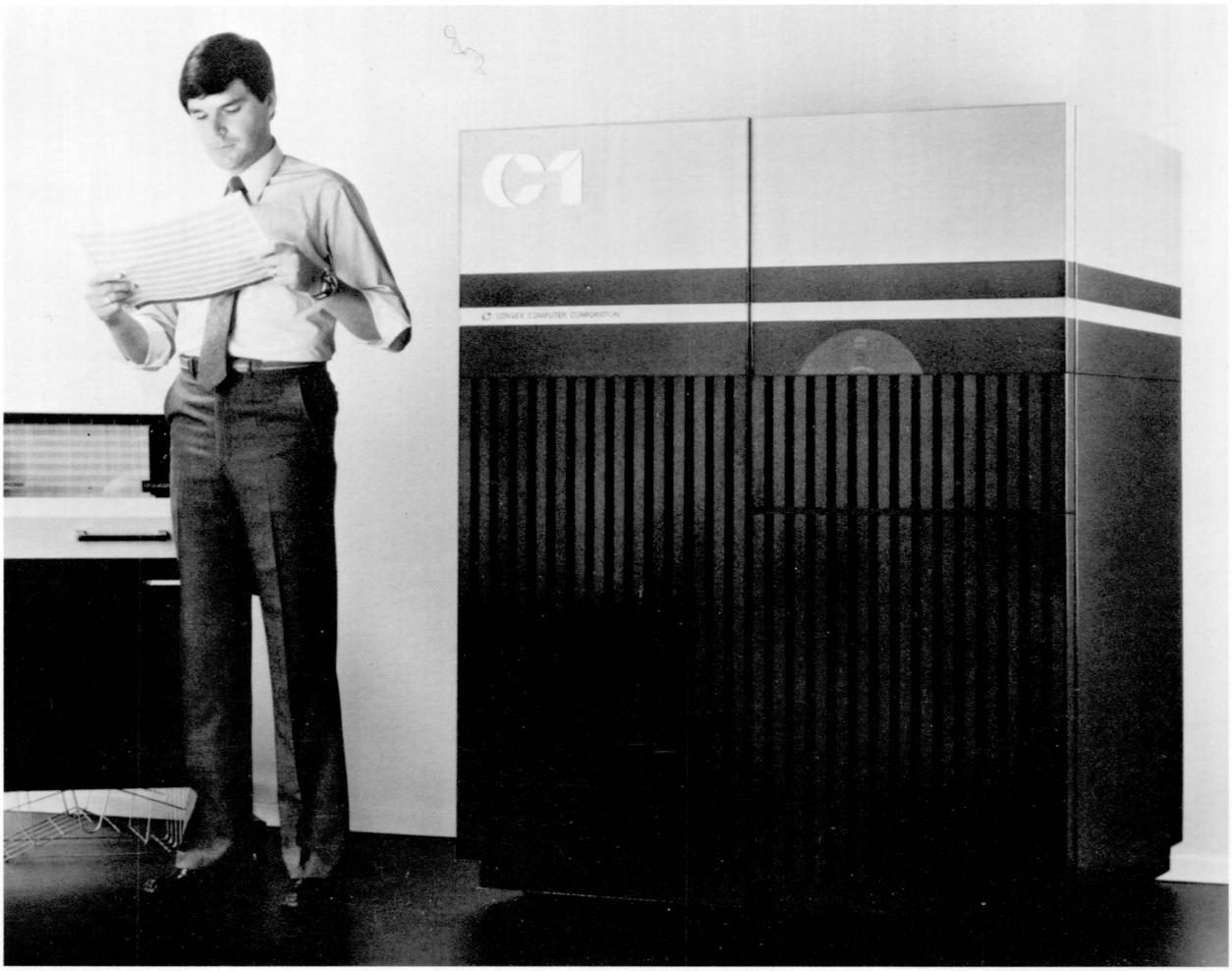

C1
CONVEX COMPUTER CORPORATION

TABLE OF CONTENTS

APPENDICES

LIST OF FIGURES

Preface

Manual Objectives and Intended Audience

This document is designed to be an overview of the features and capabilities of the CONVEX C-1 supercomputer. It is directed to a wide range of users, including non-programmers, applications programmers, system developers, and system administrators.

The System Overview can be read in a sequential or selective mode. Specifications on the C-1 are summarized in Appendix D.

Document Structure

Following is a chapter-by-chapter breakdown of the System Overview:

- Chapter 1, "Introduction", is a system introduction. It contains a comprehensive overview of the features of the C-1 supercomputer, hardware and software.
- Chapter 2, "CONVEX Architecture", is a detailed description of the individual components that make up the CPU, I/O subsystem, memory, vector processor, and service processor unit.
- Chapter 3, "Peripherals", presents an overview of the major peripherals that are available with the C-1.
- Chapter 4, "Software", describes the software features and functionality of CONVEX UNIX, C, Networking, and Fortran.
- Chapter 5, "Vectorization", a brief tutorial on vectorization.
- Chapter 6, "CONVEX Fortran Optimizations and Vectorizations", discusses the Machine-Independent Optimizations both local and global, vectorization, and the Machine-dependent Optimization, instruction scheduling, strength reduction, and so on.

Associated Documentation

The following reference manuals contain detailed descriptions of the C-1 supercomputer which CONVEX provides:

- *CONVEX Architecture Handbook* (DHW-005)
- *CONVEX UNIX Documentation* (DSW-001)
- *CONVEX Utilities Reference Manual* (DSW-005)
- *CONVEX C Documentation* (DSW-045)
- *CONVEX Fortran Documentation* (DSW-035)

1 Introduction

1.1 The CONVEX C-1 Affordable Supercomputer

CONVEX C-1: The affordable 64-bit supercomputer that sets a new standard for price/performance in scientific computing applications. By using a combination of widely accepted software and hardware standards, implemented with advanced technology, the CONVEX C-1 system offers the scientific user access to supercomputer performance at minicomputer prices.

A well-balanced computer system, the CONVEX C-1 blends high-speed scalar and vector processing with large real and virtual memory, high-performance input/output, and productivity-oriented system software.

1.2 System Features

- High-speed scientific computer with integrated vector processing
- 4 Gigabyte virtual address space
- Main memory expansion to 128 megabytes
- Multiple, high-performance cache memories
- RISC (Reduced Instruction Set Computer) architecture
- Highly-pipelined implementation
- High-performance I/O bus with 80 MBytes per second bandwidth
- Independent, high-performance intelligent I/O subsystems
- Extensive reliability, availability, and serviceability features.
- Compact 19-inch RETMA package, air-cooled.
- Rugged construction and low power consumption.
- Extensive use of high density CMOS VLSI gate arrays for low power and high reliability.

1.3 Comprehensive system software

- UNIX virtual memory operating system derived from 4.2 bsd, with real-time extensions.
- Globally optimizing and vectorizing Fortran '77 compiler
- C Compiler and native assembly language
- UNIX 4.2 utilities for editing, debugging, source code maintenance, macro processing, and program performance analysis
- Common run-time environment that permits Fortran, C and assembler to be

linked into a single executable

1.4 Extensive user and program capacity.

- Fast, consistent response in multi-user timesharing environment
- User program size up to 2 gigabytes.
- 5-level address space protection
- Input/Output Processors (IOP's) for concurrent I/O control.
- High-speed MULTIBUS subsystem with up to 150 MULTIBUS I/O controllers.

2 CONVEX Architecture

2.1 System Architecture Overview

The CONVEX C-1 system is a high performance system based on a proprietary bus-oriented architecture. The central processing unit (CPU) portion of the system utilizes a CRAY-like architecture. This architecture employs vector accumulators and multiple arithmetic units functioning at 100 nanoseconds for 64-bit operands and 50 nanoseconds for 32-bit operands to achieve peak processing rates exceeding 60 million operations per second(MOPS).

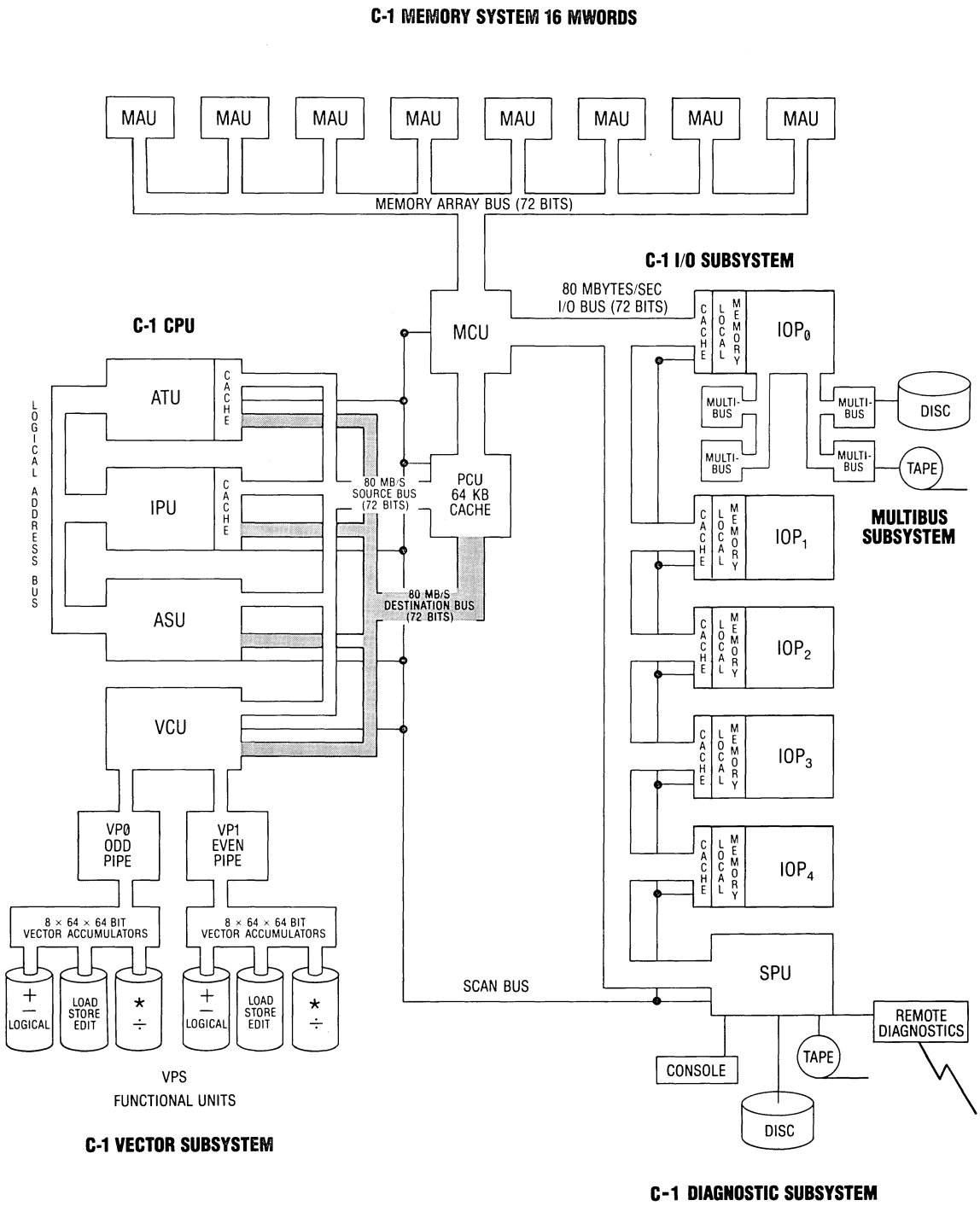
The C-1 central processor(sometimes referred to as Job Processor - JP) consists of multiple asynchronous units, operating concurrently in a pipelined fashion. These units are interconnected through multiple high-speed buses. The asynchronous nature of these units permits up to 8 operations to be executing at the same time. The inclusion of a highly intelligent I/O system permits the vast majority of I/O operations to be concurrently executed and controlled independent of the central processor. The off-loading of these functions makes the central processor much more productive.

The CONVEX C-1 is a tightly coupled asymmetric multiprocessor that comprises five functional subsystems:

1. Central Processing Unit Subsystem
2. Memory Subsystem
3. Input/Output Subsystem
4. Service Processor Subsystem
5. Electromechanical Subsystem

Refer to Figure 2-1. The service processor, input/output processor, and central processing unit subsystems make this a multiprocessor system. It is an asymmetric system because each processor performs a unique set of tasks. The system is tightly coupled because communication between the processors is through shared memory. UNIX is the main operating system. There are two smaller specialized operating systems called SPU UNIX and SPOKE that coexist with UNIX. Portions of UNIX are shared by all processors, and other specialized portions of UNIX are used only by specific processors.

Figure 2–1: CONVEX C-1 Functional Subsystems



2.2 Central Processing Unit Subsystem

The CPU subsystem performs the computational functions. These include arithmetic and logical operations on scalar and vector data.

The central processing unit (CPU) subsystem comprises five parts:

1. Physical Cache Unit (PCU)
2. Address Translation Unit (ATU)
3. Instruction Processor Unit (IPU)
4. Address and Scalar Unit (ASU)
5. Vector Processor System(VPS)

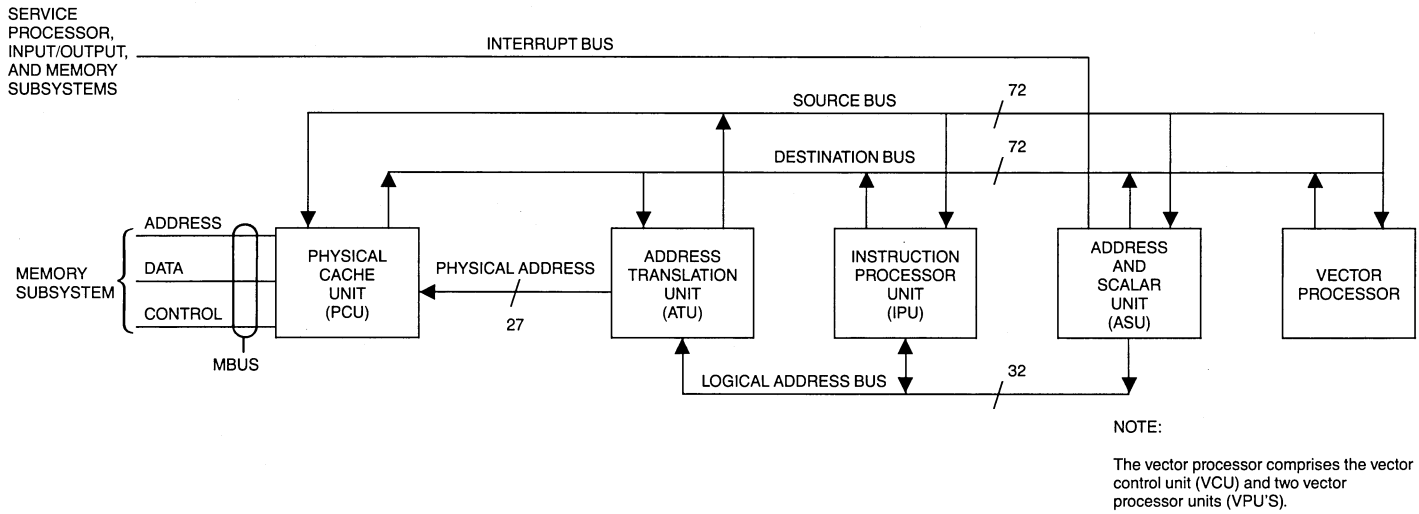
    a. Vector Control Unit (VCU)
    b. Vector Processor Unit (VPU) (two identical boards)

Refer to Figure 2-2. The source bus carries operands from the ATU to the PCU, IPU, ASU, VCU, and VPU's. The destination bus carries operands from the PCU, IPU, ASU, VCU, and VPU's to the ATU. Logical addresses move between the ATU, IPU, and ASU over the logical address (LA) bus. Physical addresses are sent from the ATU to the PCU through the back plane and sent from the PCU to the Memory Control Unit(MCU) through the MBUS in connectors on the foreplane.

2.2.1 Physical Cache Unit

The PCU interfaces the CPU to the MCU. The MCU deals with memory only in terms of 64-bit longwords, longword aligned. The CPU sees memory in terms of two independent 32-bit data paths, byte aligned. That is, the CPU may load or store two 32-bit words on every clock cycle, with both words coming from unrelated memory addresses, aligned on arbitrary byte boundaries. The PCU control logic maps each such request into the most efficient pattern of longword accesses and cycles the MCU to achieve them.

Figure 2–2: Central Processing Unit Subsystem



2.2.1.1 CPU/Memory Interface

The CPU/memory interface provides a two-level pipeline and a seven-level pipeline. Requests for data that are in the p-cache use the two-level pipeline. This is necessary because typical PCU operations require two clock cycles. The request is decoded and the p-cache is accessed in the first clock cycle, and data transfer occurs in the second clock cycle. The two-level pipeline permits the decoding of one request and the data transfer of the preceding request to be performed concurrently. Requests for data that are not in the p-cache use the seven-level pipeline. The seven-level pipeline can accept a request for data from main memory on every clock cycle.

The PCU receives store data and parity from the source bus, and it drives data and parity from the p-cache, main memory, or both, onto the destination bus.

The MCU and PCU are connected by the MBUS. Data and parity are bidirectional signals in the MBUS. The MBUS also contains address and control signals.

2.2.1.2 P-cache

The PCU also manages a pipeline of addresses and data that enable longword memory access to begin on every clock cycle. This depends on the addresses being aligned such that full interleave is possible. Because this is not always the case, and to provide lower latency to scalar operations, the PCU also contains the P-cache, a 64 Kbyte cache memory accessed with physical addresses. This cache provides dual 32-bit data ports, allowing two arbitrary cache locations to be read or written on each clock cycle. A write-back algorithm is used, so that data written to the cache is not written into main memory until the block is overwritten by a subsequent access or until an I/O operation attempts to access the old data in main memory.

2.2.1.3 P-cache Bypass

There are three cases encountered when loading or storing vectors from memory. The vectors can be contiguous in memory, can have constant stride, or can involve random memory accesses (gather/scatter). According to a study done at one of the national laboratories involved in scientific processing, typical programs contained code using contiguous elements 78% of the time, constant stride 20%, and random access 2%. Therefore a technique to enhance performance for the contiguous case would be very beneficial. This technique is the cache bypass.

The p-cache bypass prevents instruction fetches and sequential vectors from overwriting operands in the p-cache. The address translation unit (ATU) can designate an operand request as bypass mode. In this case, the operand will be transferred to or from main memory unless main memory has not been updated by the p-cache. In this case, the operand is transferred to or from the p-cache.

Physical memory accessing is fully pipelined. For contiguous memory locations the memory interleaving is efficiently utilized, thus the total time for loading or storing contiguous vectors via cache bypass is practically the same as the time for loading or storing using the p-cache. The benefit of this approach is that data already residing in the p-cache does not need to be flushed. Thus the purpose of the p-cache in storing multiple non-contiguous quantities is not compromised by the loading/storing of long contiguous vectors.

Those operations involving constant stride and random access often cannot take full advantage of memory interleaving and thus would not benefit from cache bypass. For the first constant stride or random access of particular operands, a cache load is required, but subsequent accesses to those operands are to the p-cache which operates at the maximum bandwidth of the memory system.

The result of this design is that the hardware automatically chooses the most efficient method for loading or storing vectors, totally transparent to the user.

2.2.1.4 Referenced Bits and Modified Bits

The referenced and modified bits indicate activity for each memory page that can be physically implemented in the CONVEX C-1. A table of referenced bits indicates which physical pages of memory have been accessed by the CPU subsystem. A referenced bit is set when the CPU subsystem accesses (reads or writes) the corresponding physical page of memory. A table of modified bits indicates which physical pages of memory have been written into by the CPU subsystem. A modified bit is set when the CPU subsystem writes into the corresponding physical page of memory. These tables are used by the operating

system to keep track of the least recently used pages of physical memory. Both tables must be initialized (set to all zeros) by the CPU subsystem when power is applied to the system.

2.2.2 Address Translation Unit

The Address Translation Unit(ATU) contains the circuitry required to translate logical addresses generated by the Address & Scalar Unit (ASU) or the Instruction Preprocessing Unit (IPU) into the physical addresses required by the Physical Cache Unit (PCU), thus the name Address Translation Unit. However, the address translation function is only one of many functions performed by the ATU. The primary ATU functions:

- Contain and control the primary data path switch for the CONVEX system.
- Provide arbitration and source/destination selection for all system busses.
- Align the data being transferred as required by the destination.
- Stages data as required to deliver complete operands to the destination.
- Maintain a logical cache of aligned data to accelerate memory accesses.
- Perform the majority of the memory address computations for the CONVEX system.
- Provide arbitration and control for the logical address bus.
- Perform indirect addressing for data accesses.
- Generate secondary logical addresses for non-aligned scalar accesses.
- Generate secondary logical addresses for all vector accesses.
- Perform logical to physical address translation for all memory accesses.
- Maintain a cache of page table entries to accelerate address translation.
- Perform all protection checks for data accesses to generate system faults.
- Contain control and queuing for up to nine concurrent memory accesses.

2.2.2.1 Data alignment

Operands in registers are aligned to the register. Operands in memory may begin on any byte boundary. The ATU provides the means to go from one alignment to the other during the data transfer to or from memory. Alignment is performed as data is moved from the destination bus to the source bus, and the ATU provides arbitration among the users of these busses.

2.2.3 Instruction Processor Unit

The Instruction Preprocessing Unit (IPU) is a major speed factor in the CONVEX C-1. All instructions are preprocessed and pipelined by the IPU in a manner such that the Address and Scalar Unit or the Vector Units received the decoded instructions and displacement fields each clock cycle.

The Instruction Preprocessing Unit (IPU) performs the following major functions:

- Dispatches instructions to the VCU and ASU with the proper fields and an entry point address for the ASU or VCU microcontroller.
- Cracks(decodes) instruction opcodes.

- Maintains the system Program Counter.
- Maintains an instruction cache.

The IPU saves machine time by executing some instructions such as branch or jump instead of dispatching them to the ASU or VCU.

2.2.3.1 Instruction Decode and Dispatch

The IPU is started by execution of a branch by the ASU. Subsequently, it fetches, decodes, and presents each instruction for dispatch. It always attempts to have ready the next instruction to be executed by either the ASU or VCU, or both. It will execute unconditional branches without intervention of the ASU. When a conditional branch is decoded, it will wait until the ASU signals whether to branch or not. Owing to a unique pre-decoding scheme, conditional branches are usually executed in a single clock cycle.

2.2.3.2 Instruction Pre-fetching and Cacheing

In order to minimize the amount of time the processor is waiting on instruction fetches, a cache is provided which contains up to 512 instructions. The IPU attempts to fetch instructions from this cache, which may be accessed without interfering with memory data operations. In addition, a pre-fetcher looks ahead of the current point of execution and pre-loads the cache if it is not already loaded.

2.2.4 Address and Scalar Unit

The Address and Scalar Unit (ASU) is a 64-bit microprogrammed processor within the CONVEX C-1 Computer. The ASU performs the following functions within the C-1 computer.

- Executes all address register manipulation instructions. The Operands for multiplication and division are sent to the VPU, which does the arithmetic operation.
- Executes all scalar register manipulation instructions. The Operands for multiplication, division, population counts, floating point adds and subtracts, and portions of float to fix and fix to float conversions are sent to the VPU for arithmetic operations.
- Generates operand address for address, scalar, and vector loads and stores.
- Executes all program control and priviledged instructions.
- Generates addresses and control the JP during context load and stores.
- Maintains the interval timer and elapsed time counter.
- Maintains the Program Status Word (PSW).

2.2.5 Vector Processor

The vector processor consists of one vector control unit (VCU) and two vector processor units (VPU's). The VCU provides all timing and control. The VPU's provide the vector accumulators and data paths, but are totally under control of the VCU. Refer to Figure 2-3.

2.2.5.1 Vector Control Unit

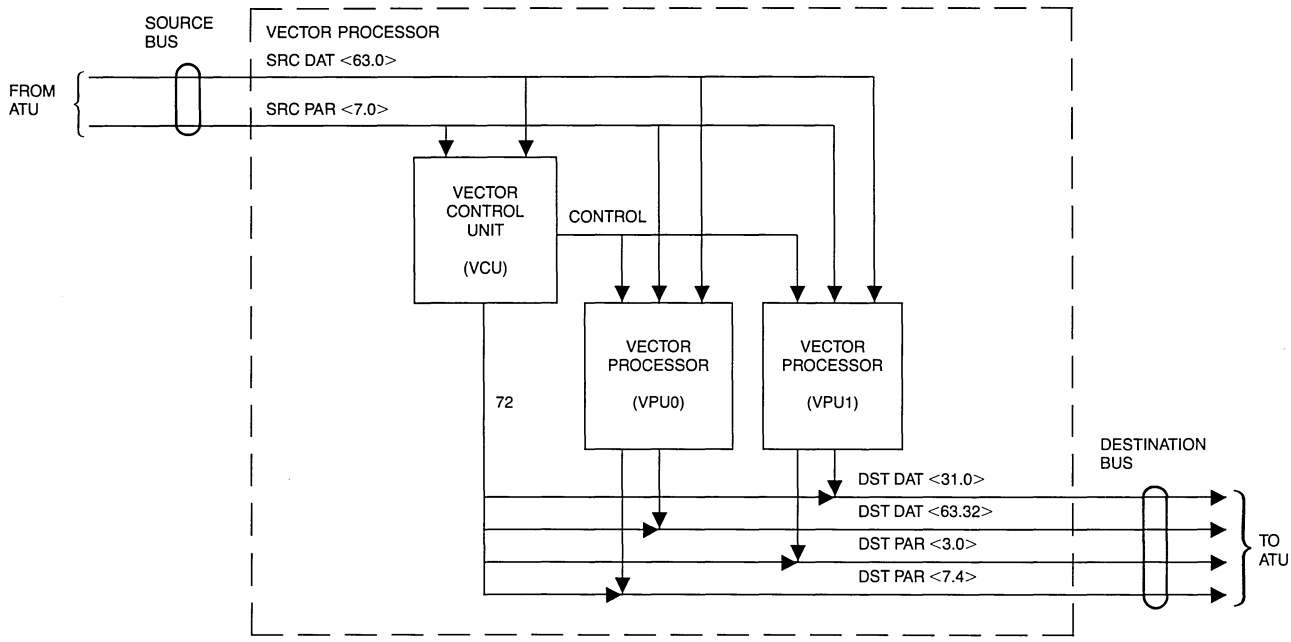
The Vector Control Unit is the master of the vector processor. It directs data to and from the Vector Processors. All vector processing and some scalar processing, ie, floating point operations, multiplication and division, are handled by the Vector Control Unit.

There are three independent microcoded controllers on the VCU:

- Load and Store Controller
- Multiplier and Divider Controller
- Adder and Logical Controller

All of these controllers are idle until started by the instruction dispatch control. The instruction dispatch receives vector register specification fields, an operand size code, a PSW hazard bit, and an entry address from the Instruction Processor Unit (IPU). Hazard conditions with the utilization of registers or the controllers are checked prior to issuing an instruction to one of the three microcoded controllers. The IPU is signaled with a ready signal when the registers and function controller specified by the opcode are not busy and the opcode is accepted. The instruction dispatch sends a microcode entry address, function unit opcode, and address counter claims to the appropriate controllers as the instruction is issued within the VCU.

Figure 2–3: Vector Processor



2.2.5.2 Vector Processor Unit

There are two identical VPU's in the CONVEX C-1. VPU0 contains the even data elements and VPU1 the odd elements of the eight vector accumulators. During 32-bit vector loads and stores, an even and an odd data element are moved on each 100 nanosecond clock cycle. In addition, both VPU's execute concurrently. This effectively doubles the data element processing rate for 32-bit operands relative to 64-bit.

The source bus provides the input data to the VPUs for vector load data, scalar floating point arithmetic operands, and VCU to VPU or VPU to VPU transfers. The output bus of VPU0 is wired to the upper 36 bits of the Destination Bus and VPU1 to the lower 36 bits of the bus. The ATU board comprehends this odd-even connection of VPU0 and VPU1 when vectors are transferred out of the vector registers. Long word (64 bit) vectors are transferred to and from the VPUs upper word first followed by the lower word. Longword vector stores proceed with the same order of vector elements being sourced by the VPUs.

2.2.5.3 Floating Point Operations

The ASU executes most scalar operations in a single 100 nanosecond clock cycle, however, it has no floating point capabilities. Hence, it sends scalar floating point operands to VPU0 for execution. For this reason, the VPU's receive the entire 64-bit source bus, since it allows both operands to be transferred at once during 32-bit operations.

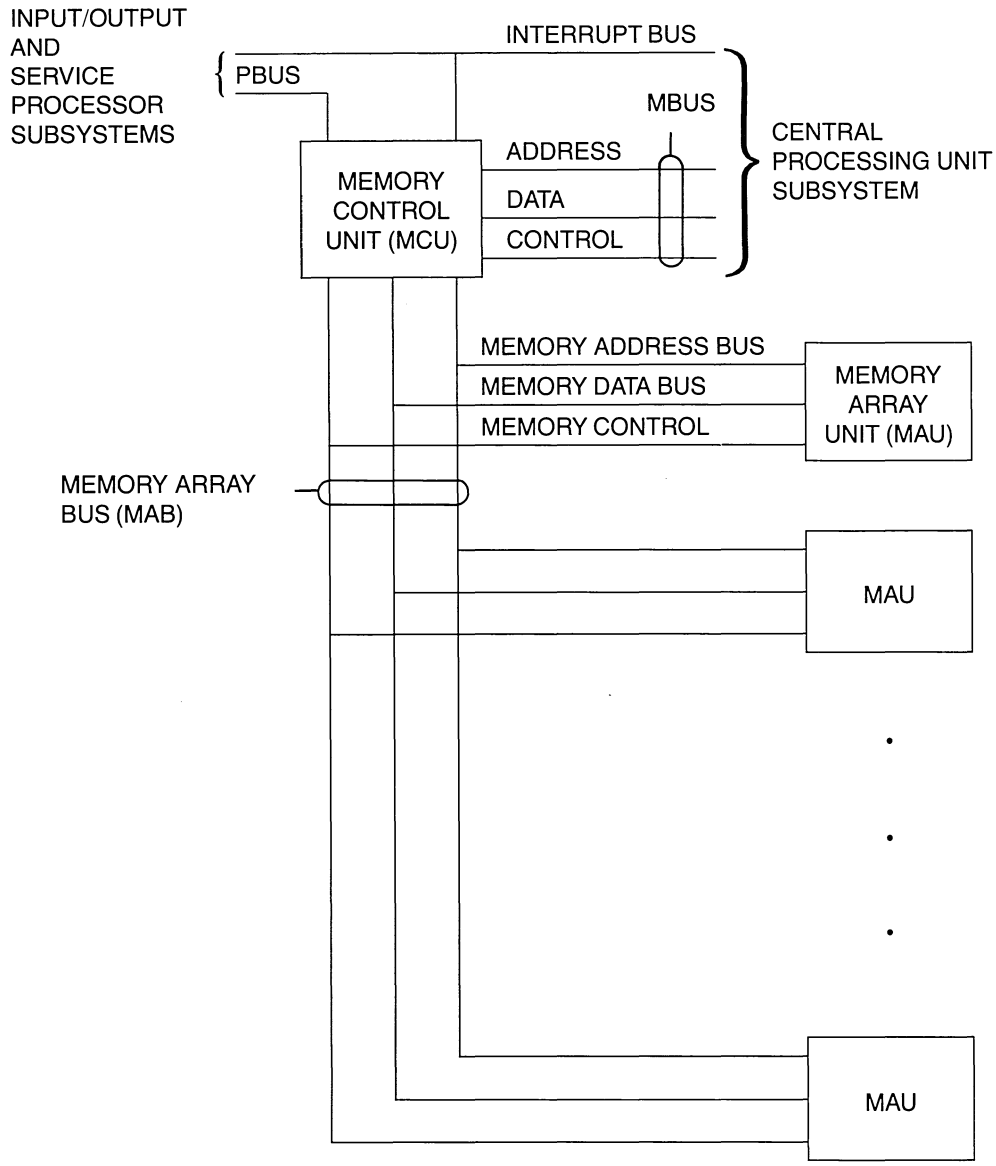
2.2.5.4 Chaining

The vector processor also provides for a technique whereby the output of one set of vector operations can be directed to the input for a different set of vector operations. This technique is called chaining and allows multiple operations to occur simultaneously. For example, one sequence of operations might call for the multiplication of two vectors of numbers, the addition of a third vector and the store of the result. The multiplication of the two vectors would be started; after the start-up delay of the multiplication unit the result from the first two operands would be available and would then be routed to the vector addition unit to be added to the third vector. The sum of these two operands would be given to the store functional unit for storage in the memory system. The resultant concurrency is made possible by the chaining of the three distinct operations.

2.3 Memory Subsystem

The memory subsystem provides the main memory for the CONVEX C-1. It includes one memory control unit (MCU) and from one to eight memory array units (MAU's) and connects to the CPU and I/O subsystems through the MBUS and PBUS paths respectively. Refer to Figure 2-4.

Figure 2–4: Memory Subsystem



NOTE:

From one to eight MAU's can be installed.

2.3.1 Memory Control Unit

The MCU provides all control functions and interfaces the two memory ports, the P-Bus and M-Bus, to main memory. It sequences the MAU's, both for data transfers and refresh operations, and maintains an error log for memory operations. It contains a configuration map which monitors memory accesses and traps accesses to non-existent or deconfigured memory. It also contains the arbiter for the system interrupt bus.

The MCU also contains a tag store which keeps a record of all data blocks which have been accelerated into the P-cache (PCU). If an I/O access is attempted to such a block, the PCU will ensure that only the most recent copy of the block is used for the I/O access.

2.3.2 P-Bus

The P-Bus provides a high-bandwidth, block oriented, path to main memory for the IOP's and SPU. Arbitration of the P-bus is performed by the MCU. A ring-grant algorithm which sequentially allocates the bus to requesters prevents any user from being starved. A set of latency timers assures that the bus is reallocated frequently during high usage.The P-Bus is optimized for burst transfers. Each transfer begins with the requester passing a control and address word, followed by continuous data transfer.

2.3.3 M-Bus

The M-Bus provides both high bandwidth and low-latency, but has a single user which eliminates the need to arbitrate usage or to propagate data on a multi-drop bus. It has parallel paths for address, control and data. It supports block transfers to and from the P-cache, but otherwise requires the PCU to supply one address for each data cycle.

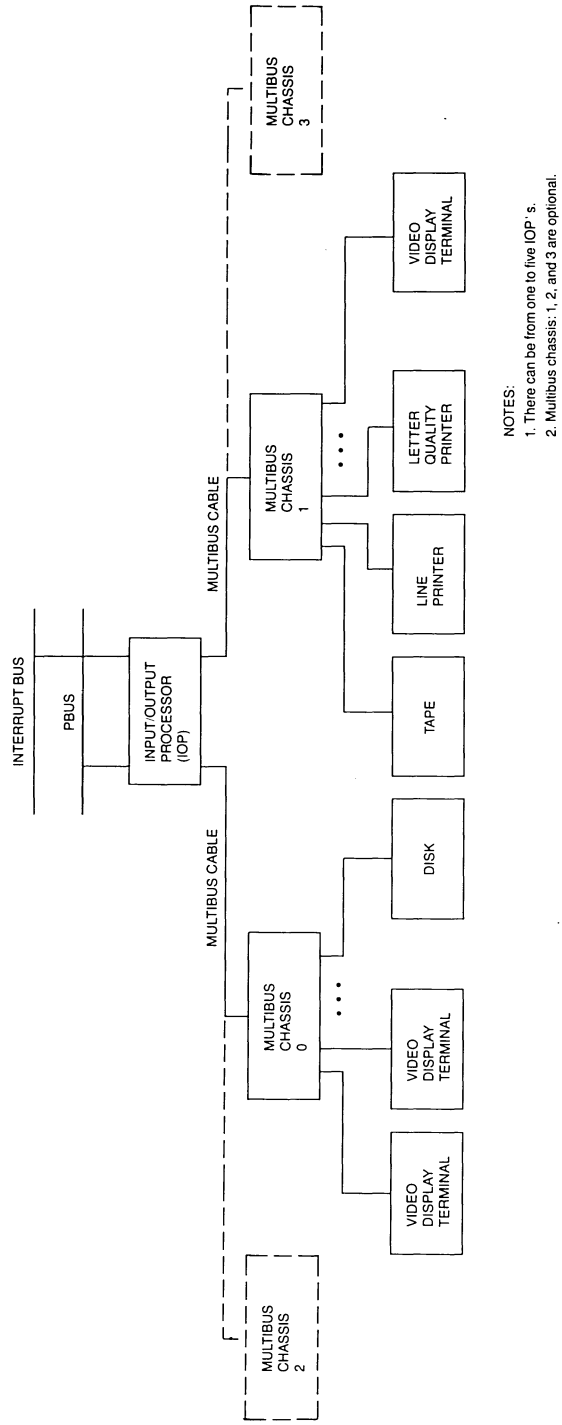
2.3.4 Memory Array Unit

The memory array unit (MAU) is populated with 256K DRAM devices, for a board capacity of 16 Mbytes. The 16 Mbyte board is organized as 2 Mwords by 8 bytes. Cycle time of the array is 400 nanoseconds, but four way interleaving based on the least significant two word address bits allows sequential accesses to be started every 100 ns.

2.4 Input/Output Subsystem

The C-1 system performs I/O via distributed intelligent I/O processors, which transfer data to and from system memory over the P-Bus, a CONVEX-proprietary 80 Mbyte/sec bus. One to five IOP's may be configured on a C-1. Refer to Figure 2-5.

Figure 2–5: Input/Output Subsystem



2.4.1 Input/Output Processor Unit

The C-1 I/O Processor (IOP) interfaces Multibus controllers to the P-Bus. It incorporates a 68000 microprocessor with 512 Kbytes of local memory and a high speed 32Kbyte cache. I/O device drivers execute on the 68000, under a resident executive called SPOKE. Data transfers may, under device driver control, be either to/from local IOP memory, or to/from main memory, buffered through the IOP's cache.

The IOP provides two multibus ports, each of which may support up to two multibus cages, although highest performance is available when only one cage per port is used. Each port is capable of a 4 Mbyte/sec transfer rate.

2.4.2 Multibus Chassis

Each chassis contains one Multibus Control Unit (MBCU), and from one to eight Multibus device controllers. The MBCU buffers the cable interface and provides a multibus compatible protocol within the card cage. This allows industry standard multibus controllers to be used to interface I/O devices.

A second version of the Multibus Chassis is available that enables two MBCU controllers. This effectively provides two logical expansion chassis, for up to three device controllers each, in a single physical unit.

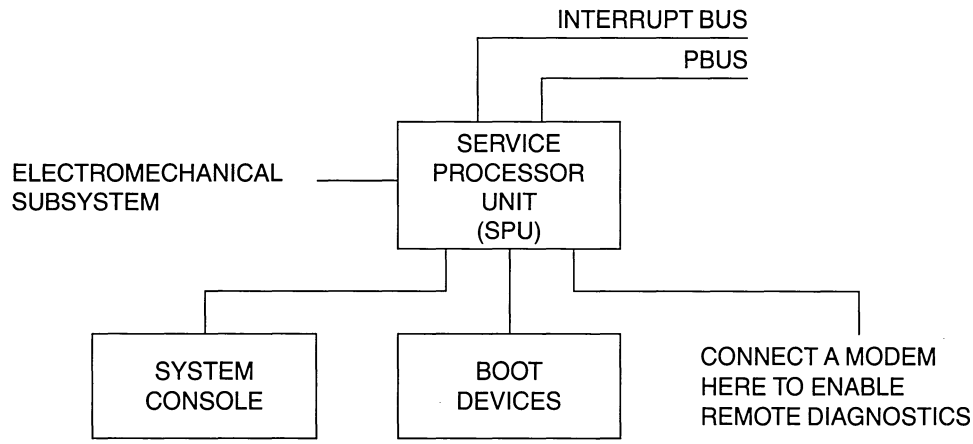
2.5 Service Processor Subsystem

Integral to the CONVEX C-1 is a Service Processor Unit (SPU). The is a microprocessor-based unit that controls the operation of diagnostic programs, as well as maintaining a log of detected and corrected errors. The SPU runs under its own diagnostic UNIX operating system, and contains a 20 MByte Winchester disk and cartridge tape peripheral devices as well as communications ports for the operator's console and for remote diagnostics. The disk maintains an on-line log of errors as well as code for the diagnostics. The cartridge tape, with a capacity of 20 MBytes, is used for software distribution for both system and diagnostic software and may be used as a distribution medium for user software. Refer to Figure 2-6.

2.5.1 Service Processor Unit

The Service Processor Unit (SPU) is a microcomputer that is connected to the PBUS and interrupt bus. The system console is connected to the SPU through one RS232 port. Another RS232 port is provided as a remote diagnostics port. Two boot devices -- the Winchester disk and cartridge tape drive -- are connected to the SPU. The Winchester disk is connected to the SPU through a Small Computer System Interface (SCSI). The cartridge tape drive is connected to a controller on the SPU.

Figure 2–6: Service Processor Subsystem



NOTE:
Boot devices include a Winchester disk and a cartridge tape drive.

2.5.1.1 SPU Functions

During normal operation, the SPU:

1. Provides an interface to the system console
2. Provides an interface to the boot devices
3. Provides all of the system clocks
4. Provides a battery backed-up real-time clock/calendar
5. Executes UNIX

During diagnostics, the SPU:

1. Provides a remote diagnostics port
2. Executes UNIX
3. Controls the system with:

    a. Scan rings(diagnostic bus)
    b. The PBUS

2.5.1.2 System Initialization

The SPU is the only system element which comes up running from a system reset. A system reset is generated automatically when the system is powered up, or may be generated manually by the reset switch when the keylock switch is not in the "secure" position. Following the reset, the SPU will begin execution from on-board EPROM. Depending on the switch settings, the SPU's next actions will vary from waiting for further input from the system console to commencing a boot of the full system.

Assuming the latter, the will load UNIX into SPU memory and begin execution of the CPU boot procedure, which runs under UNIX. This boot procedure will determine the size and configuration of main memory, and load this information into a protection map on the MCU. It will also generate a logical to physical page translation table in main memory, which allows the CPU to begin execution using logical addresses.

With memory initialized, the SPU will load CONVEX UNIX boot image from its hard disk into main memory, and then initialize the CPU scan rings, registers, and writable control stores. The IOP's must also be loaded from the hard disk with a SPOKE I/O executive image. Finally, the SPU turns the system clocks on, and the CPU and IOP's then proceed to bring up UNIX.

2.5.1.3 Interface to Electromechanical Subsystem

The SPU senses the environmental monitors and front panel switches via a cable connection to the System Monitor Board (SMB). These environmental monitors allow it to sense air temperature, airflow, and ac and dc voltages. In addition to being able to read the state of these monitors under program control, the SMB will interrupt the if it detects an out-of-range value. Certain conditions, in fact, will cause the system power to be shut down. The also controls the front panel lights via its connection to the SMB.

2.5.1.4 Battery Backed-up Clock

The battery backed-up clock on the provides the year, month, date, day, and time when the system is initialized. The battery backed-up clock is a clock/calendar hybrid module that contains a CMOS integrated circuit, a crystal, and a lithium battery. When the system is running, time is derived from the 60 Hz ac power line. When ac power is not applied to the system, the lithium battery powers the clock/calendar module.

2.5.2 System Console

The system console is physically connected to the SPU, and serves as the console during system boot or diagnostic execution. In addition, processes on the SPU and CPU logically connect it to the CPU, and during normal UNIX execution, it serves also as the UNIX system console.

2.5.3 Boot Devices

Two magnetic storage devices are attached to the SPU, a 20 Mbyte Winchester disk, and a 1/4 inch cartridge tape drive. The disk is required by UNIX, and additionally contains the load images used to boot the CPU and IOP's. It also contains diagnostic programs used to maintain the system.

The cartridge tape transports data to and from the SPU. If the disk is empty, as on a new system, or has been corrupted, a special program can format and load the disk from the cartridge tape. Additionally, the cartridge tape provides a low-cost mechanism to distribute software, both for the SPU and CPU.

2.5.4 Remote Console Connection

The remote console connection is nearly identical to the system console connection, except that it is hardware disabled when the keylock switch is not in the "remote" position on the front panel. It is intended to be connected through a modem to a dial-up phone line. This allows a remote service center to have the same access to the system as a local field engineer, when enabled by the customer.

2.6 Reliability/Availability/Serviceability

The SPU subsystem controls one of the most sophisticated diagnostic systems available. Each subsystem is interfaced to the SPU via a proprietary diagnostic scan bus. Once initiated, the diagnostic software scans in test patterns and scan out results without using any of the buses used during normal computations. This approach results in high resolution diagnostics thereby significantly reducing the mean time to repair. In addition, the provision of a remote diagnostics capability improves the efficiency of maintenance personnel, resulting in minimum downtime.

Additional advanced features include: extensive parity checking throughout the entire design, single-bit error correction and double-bit error detection for main memory accesses, alterable control store within the ASU and VPS, as well as diagnostic-readable serial numbers contained on each logic board. This last feature permits the diagnostic system to ensure that boards are up to the latest revision level and in the proper slots.

2.7 Compact, Rugged Packaging

The CONVEX C-1 system is housed in a standard 19 inch RETMA rack, 5 feet high with a 'foot print' of less than 7 square feet. Contained within this rack are: the CONVEX C-1 central processing unit, up to 128 Mbytes of main memory, up to 5 IOP's, the SPU with its cartridge tape and winchester disk, and one MULTIBUS card cage. A typical system configuration includes a second, adjacent rack, which could contain a tape drive and two

414MB (formatted) winchester disk drives, resulting in an extremely powerful system in a very compact package. The CONVEX C-1 is air cooled, with cooling provided by two fans drawing air through the front of the rack and exhausting through the rear.

All CONVEX processor boards are vertically mounted and attached to the backpanel via rugged 96-pin DIN connectors. To ensure rigidity, these boards are fastened to the card cage with a screw-type ejector and injector mechanism.

The components packaged in the processor cabinet consume approximately 3200 watts of input power in the standard configuration, and less than 4500 watts when fully configured with 128 MB of main memory. The system operates in a temperature range between +15 and +32 degrees centigrade.

3 Peripherals

The CONVEX C-1 offers a wide range of peripherals for input/output operations.

3.1 DKD-101 Disc Drive

The DKD-101 Disk Drive is a compact state-of-the-art moving head disk drive with a storage capacity of 474 MBytes (unformatted) in a very compact package. The DKD-101 disk drive is a 10.5-inch SMD Winchester, non-removable, sealed disk system offering high recording density, data transfer rate, superlative performance and high reliability. The DKD-101 is appropriate for large capacity, high speed, data storage for both on-line and large-scale data base applications.

The DKD-101 consists of a disk enclosure, four printed circuit boards, and a DC power supply unit. The disk enclosure is completely sealed and integrates six disks, Winchester-type contact start/stop heads, a rotary actuator, DC spindle motor, and IC read pre-amplifiers. The DKD-101 is rack-mountable in the standard CONVEX C-1 cabinet and interfaces to the CONVEX Disk Controller.

The DKD-101 includes the SMD Disk Controller. The controller interfaces to the CONVEX MULTIBUS as generated by the CONVEX C-1 IOP(Input Output Processor). It is a high performance DMA channel interface for maximum system throughput.

3.1.1 Storage Capacity

The DKD-101 has 842 cylinders with a 28,160 byte-unformatted track capacity, and their servo-controlled track-following system assures accurate head positioning on extremely high-density tracks of 880 tracks per inch. The DKD-001 offers the user a storage capacity of 474 megabytes on six surfaces.

3.1.2 Performance

The adoption of an advanced rotary actuator and a direct-drive DC spindle motor of 3,961 rotations per minute, as well as two heads per surface, allows for exceedingly high performance: 18 millisecond average positioning time (5 milliseconds for track-to-track, 35 milliseconds maximum), 7.5 millisecond average latency time, and 1.859 megabyte-per-second data transfer rate.

3.2 Magnetic Tape Drives

Today's ever increasing disk drive capacities in dramatically reduced packaging sizes are dictating the need for economic, flexible, and performance-oriented tape subsystems. The CONVEX MTD-001 and MTD-002 tape systems utilize state-of-the-art technology to meet the increased demands for full performance, general purpose tape subsystems. These systems also provide exceptional serviceability and maintainability through extensive resident micro-diagnostics.

3.2.1 MTD-001 Description

The MTD-001 tape subsystem is a dual-density (1600/6250 bpi), 50 ips, start/stop and GCR subsystem, complete with tape transport, formatter/controller, power supply, and resident microdiagnostics, all contained in a single self-contained package. The 24.5-inch high MTD-101 is mounted vertically in a standard 19-inch retma rack. Also included is a reliable mechanical tape buffering system that is capable of handling both high density Group Coded Recording (GCR) and convenient auto-threading.

3.2.2 MTD-001 Performance

The tape subsystem provides reliable start-stop performance. Disk/dump restore, interchange, archiving, batch processing, and journaling are important capabilities accomplished without special software modifications or repositioning. The MTD-001's average access time is only 5.6 milliseconds.

3.2.3 MTD-001 GCR Recording

GCR recording technology is an added plus for high capacity disk dump/restore applications. In 8K blocks, a 10.5-inch reel of magnetic tape will hold up to 134 megabytes of data. GCR also offers higher throughput. An average transfer rate of 233 kilobytes per second is typical using 8K data blocks. The instantaneous transfer rate is 312.5 kilobytes per second.

GCR also offers extensive read/write reliability with 1- and 2-track on-the-fly error correction; it will also indicate multi-track errors. The system verifies CRC, Aux CRC, and ECC characters during all GCR read and write operations.

3.2.4 MTD-002 High Performance Tape Subsystem

The MTD-002 is a high-performance, precision-vacuum, column-tape subsystem that meets all of the traditional high-performance start/stop digital tape requirements. These tape units operate at tape transport speeds of 125 inches per second. A choice of software or operator selectable recording densities is available featuring 800 bpi None-Return-Zero-Inverted (NRZI), 1600 bpi Phase Enabled (PE), and 6250 bpi Group Coded Recording (GCR) recording formats. The MTD-002 provides gentle but precise tape handling and reliable tape threading.

3.2.5 MTC-001 GCR Tape Controller

The CONVEX MTC-001 GCR Tape Controller is a MULTIBUS board which will interface either the MTD-001 or the MTD-002 tape subsystems to the CONVEX MULTIBUS as generated by the IOP. It is a high performance DMA channel with minimal intelligence, and depends on the IOP to perform I/O control block interpretation, error recovery and retry operations, and so forth.

It supports standard blocked tape formats, and also contains features to allow the tape subsystem to read and write gapless format tapes. It contains numerous diagnostic features to facilitate test and maintenance operations. The MTC-001 generates a full 24 bit DMA address to allow transfers to and from the IOP via the CONVEX MULTIBUS.

3.2.6 MTC-001 Controller Operation

MTC-001 has two levels of command buffers: the "pending" level and the "execution" level; status bits describe the state of both levels. The Input/Output Processor (IOP) writes commands to the pending level of the MTC-001 , one byte at a time. When the IOP loads a complete tape command into the pending level, a write to the start flag notifies the controller. This action also sets the command pending flag. As soon as the execution level is available, the controller copies the contents of the pending buffer to the execution level, clears the command pending flag, and sets the command executing flag. The IOP will then load the pending level with the next command.

This double buffering of tape commands allows the device driver to keep the tape as busy as possible. Rather than starting and stopping in the inter-block gaps, the tape will move through them at maximum velocity. However, if an unrecoverable error occurs during execution of one command, and another command is enqueued behind it, the controller will abort the second command, so that recovery procedures can be invoked.

The MTC-001 offers two modes of data transfer, and uses a bit in the MTC-001 control register to select the current mode. In normal blocked mode, inter-block gaps are expected during reads and generated during writes at the conclusion of each transfer. Blocked mode transfers will occur whenever the controller resets the chain mode bit.

In gapless mode, the controller does not expect nor generate inter-block gaps. Hardware support for double buffering allows the controller to transfer data continuously to or from the tape, generating interrupts to the IOP whenever data buffers are switched. Gapless transfers will occur when the controller sets the chain mode bit.

3.3 PRT-001 High Speed Printer

CONVEX Computer Corporation offers the PRT-101--a 600 lines per minute printer/plotter. In addition, the controller PRC-001 is also available as a separate product. The features of both products are detailed below.

3.3.1 Features

- 600 lines per minute dot matrix printer
- 96 character ASCII set of upper and lower case characters
- Full plotter capability
- Self-test feature with built-in diagnostics
- The PRC-001 provides dual printer support with a single controller
- Proven reliability and modular design
- Fully supported by CONVEX UNIX operating system and graphics software

3.3.2 Description

The PRT-101 dot matrix printer can overlap dots both horizontally and vertically. Characters are formed by electronic random access to PROM character sets, which place dots at appropriate positions on each horizontal line. As the paper advances, each successive line of appropriately placed dots overlaps the previous line to form characters which appear solid.

The PRT-101's hammer energy is optimized to print dots only, thereby maintaining uniform character density and quality, regardless of character size. This optimization allows for enhanced print quality, particularly noticeable on multi-part forms, and prevents character shadowing or ghosting from occurring.

The PRC-001 Dual Line Printer Controller will support two PRT-001 printers from one MULTIBUS slot. The controller accesses both printers over two independent high-speed Direct Memory Access (DMA) channels. Among the unique features are automatic printer selection and an on-board self test capability.

3.3.3 Plotting Capability

Full plotting capability stems from the inherent accuracy and ability of the PRT-101 line printer to place a single dot anywhere on the paper or print a solid sheet of overlapping dots. A single computer command puts the printer into the plot mode, enabling it to plot drawings, graphs, charts, and characters of any size or shape.

4 Software

4.1 CONVEX UNIX

The CONVEX UNIX operating system provides the highest level of programmer productivity and interactivity to the supercomputer user. Designed, optimized and enhanced for the C-1 supercomputer, CONVEX UNIX creates the ideal environment for the scientific user.

The C-1 architecture has many extensions specifically designed to support the UNIX environment of high interactivity among multiple users as well as computationally-intensive tasks. These extensions include hardware virtual memory support, memory protection (based on a ring structure), and full support for the multiple intelligent input/output subsystems of the C-1. Derived from 4.2 bsd, the CONVEX UNIX operating system provides the ideal environment for users of the CONVEX C-1 computer system. CONVEX UNIX is a virtual memory operating system that provides extensive functionality to span the range of applications in the scientific computing environment. By combining the excellent timesharing features that the UNIX operating system has with sophisticated batch processing capabilities, the CONVEX C-1 computer system is an excellent environment for interactive users as well as large production-oriented computing applications.

In addition to the facilities provided in CONVEX UNIX, the operating system also serves as the foundation for a variety of additional software for the management and sharing of information and additional program development tools.

4.1.1 Features

- Sophisticated virtual memory management enables programmers to concentrate on the application rather than on physical memory limitations by providing up to 4 Gigabytes of virtual memory.
- Full software support of the hierarchical hardware protection system, based on a ring structure, provides total program integrity.
- The CONVEX UNIX input/output is partitioned between the CPU and intelligent IOP subsystems, ensuring maximum throughput.
- User-level preemptive priority based task scheduling allows both interactive and batch applications to run efficiently at the same time.
- "Pipes" and filters are a powerful feature to redirect input and output from one process/command to another.
- Extensive set of cooperative software tools greatly improves the productivity of both programmers and users.

- The CONVEX UNIX file system is a hierarchical structure made up of directories and files to ensure optimum flexibility and ease of use.
- The kernel is the innermost layer of the operating system. It manages all resources of the computer system, including interrupts, memory management, process control, and I/O.
- Disk Striping for maximum I/O thruput.
- Fundamental to the development of CONVEX UNIX is the CONVEX C compiler. The CONVEX C compiler is an industry-standard compiler based on the C programming language developed at Bell Laboratories.
- Easy-to-use flexible command language through the C and Bourne shells allows for rapid program development.
- Subroutines written in C, Fortran, and assembly language can be linked together to make one program.
- The product is fully documented. Manuals include the *CONVEX UNIX Programmer's Manual, the CONVEX UNIX Tutorial Papers,* and the *CONVEX UNIX System Manager's Guide,* among others.

4.1.2 Description

CONVEX UNIX sets a new standard for efficiency and ease of use on supercomputers. The CONVEX UNIX operating system retains the features that have made UNIX one of the most popular operating environments in the world. In addition, CONVEX has enhanced this system to provide the additional features needed by scientific and engineering users.

CONVEX UNIX is a demand paging virtual memory operating system. In addition to supporting the full complement of up to 16 megawords (128 Mbytes) of physical memory, virtual memory support enables programmers to use more memory than is physically available. Programmers need not concern themselves with physical memory limitations; they can concentrate on applications program development. The virtual address space for the C-1 is 4 Gigabytes, evenly distributed between system and user space. Performance and security of the virtual memory system are further enhanced by the hardware paging system and support for the hierarchical memory protection system. These advanced features of CONVEX UNIX coupled with the hardware provide the user with a highly productive development environment that is optimized for total system performance.

The hardware architecture of the C-1 has been designed to address specifically total system performance and throughput. The C-1 is composed of a high performance CPU and from one to five Input/Output processors (IOP's), that make up the intelligent IOP subsystem. CONVEX UNIX takes full advantage of this underlying architecture with the kernel distributed between the CPU and the IOP's. Each IOP contains a real-time executive and peripheral device drivers and controls and manages all peripheral device functions: device addressing; interrupt handling; overlap seeks; data transfers, and so on. In other words, the CPU is not impeded handling mundane I/O. This effective use of distributed processing

means that more computational power is devoted to solving application problems quickly and efficiently. Further, asynchronous I/O is supported that enables a user program to issue an I/O request and to proceed with program execution.

CONVEX UNIX also manages multiple concurrent processes for efficient sharing of the computational power of the C-1 supercomputer system. User-level preemptive priority scheduling of processes allows users to manage the work-load on the system efficiently. This scheduling method allows interactive computer users to remain productive even when large batch processing jobs are also running on the system. The amount of computational time allocated to the batch jobs can be controlled. This capability, along with facilities for establishing multiple batch streams enables effective use of the CONVEX C-1 supercomputer.

4.1.3 CONVEX UNIX Operating Environment

The CONVEX UNIX operating environment avoids some of the traditional distinctions between system programs and user programs, between operations performed interactively from a terminal and operations under program control, and between files and peripheral devices. By avoiding many of these traditional distinctions, CONVEX UNIX provides a highly productive environment for users and programmers.

The UNIX command interpreters, called shells, can operate in both a command mode and in a program language mode. The most simple function of the command processors is to provide a method for users to issue commands to the system. In UNIX, a command has a very simple structure: a command name possibly followed by one or more parameters. In one sense, commands are like subprograms. However, unlike subprograms, the number and types of parameters may vary, so a command can handle a broader range of capabilities and options.

The shells also allow programmers to combine commands together in two ways. In addition to using commands interactively at a terminal, programmers can combine commands as part of a command-level procedure located in a file. These procedures are called shell scripts. Shell scripts appear to the user of the system as just another command. The shells also provide control flow statements that allow users to write programs that are composed of high-level commands.

Another method of combining commands together to form more powerful or more specific commands is piping. Pipes provide a simple, yet powerful, data-flow model for connecting the output of one command to the input of its successor. In this way, simple commands can be efficiently combined together with pipes to perform higher level operations.

Much of the power of the CONVEX UNIX operating system is derived from a large number of small programs that have been developed to aid in the day-to-day operation and use of the C-1 computer. With the features provided by the shells, programmers can use these small programs, called tools, independently or in combination with other tools. In fact, much of the success of UNIX is derived from this facility: users can combine tools and do not have to develop new programs for each specific application task.

4.1.4 CONVEX UNIX Utilities

CONVEX UNIX provides over 200 tools to help users code, test, debug, and control their programs. To aid in program development, CONVEX supplies several text and screen-oriented editors, including *vi*, for source code creation; a loader, and a revision control system--rcs--for source-code maintenance. As an additional product, CONVEX also supplies the CONVEX CONSULTANT--for profiling program execution and debugging. The CONVEX language translators--Fortran, the assembler, and C-- are fully integrated and supported by these program development tools.

4.1.5 The File System

CONVEX UNIX provides a file system for managing and controlling information. The file system consists of a single tree-structured name space for user and system software components. Unlike some systems, the file system makes no assumptions about file structure - that is left to the programs. The CONVEX UNIX file system provides a highly uniform name space by minimizing arbitrary distinctions and special cases. For example, directories within the file system are also files; tools that operate on files can also be used on directories when appropriate, and peripheral devices are in the file system name space. This uniformity and consistency of name space frees user programs from much of the concerns of dealing with all of the special cases found in many file systems.

4.1.6 Disk Striping

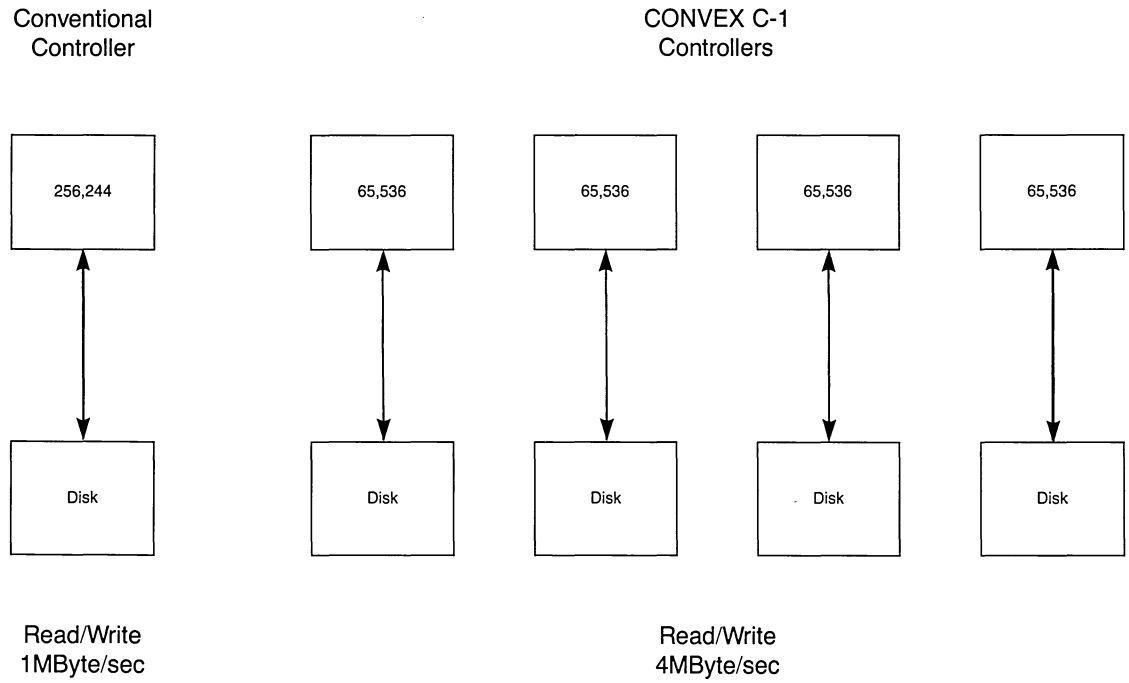
Conventional Unix file systems split each physical disk into one or more logical disk partitions, usually named *a* through *h*. The system manager has relatively little flexibility in defining a disk layout. In many cases, despite the best efforts of the system manager, the disk traffic is very unevenly distributed among the available disks. Carried to an extreme, uneven disk load distribution can cause disk throughput to become the limiting factor on system performance, even for jobs which might otherwise be CPU bound.

A less serious problem is that disk partitions are often of less than optimal size for a given user environment. The typical Unix system allows relatively little flexibility in disk partition layout.

These two problems may have a negative impact on the efficiency of the system. As a result, the productivity of the system's users may suffer.

The CONVEX Unix disk striping capability addresses these problems by allowing the system manager much greater flexibility in the layout of the disk system. A striped disk partition may span two or more conventional disk partitions, and may be split across multiple physical disk drives. The drives which contain these partitions may be on different Multibuses on the same IOP, or even on different IOPs. The splitting of a logical disc partition over several multiple discs, multibuses, and IOPs results in greater concurrency of I/O operations, and higher performance in most cases. (Refer to Figure 3-1.)

Figure 4–1: Conventional versus Striping



Striped file systems may take advantage of the increased disk basic block size feature of CONVEX Unix for additional performance gains. Disk partitions may be defined as normal disk partitions, striped disk partitions with nonstandard basic block sizes, or a combination of these.

Any program which runs under CONVEX Unix may take advantage of striped file systems without the need for special coding. No special source code modifications are required.

4.1.7 Advantages of Disk Striping

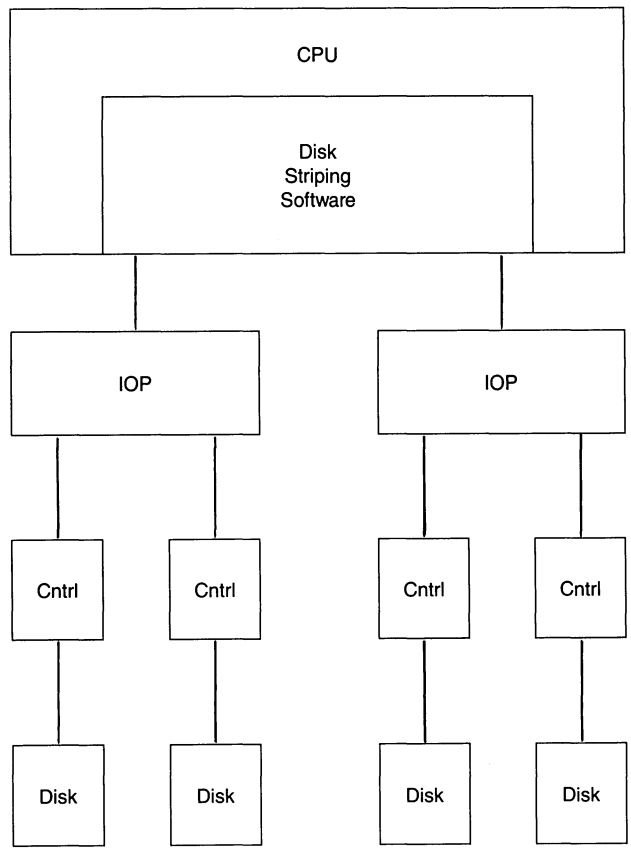
There are two primary advantages to using disk striping. The most important one is the capability to increase the performance of the disk subsystem. Although disk striping does not guarantee increased performance, it is possible to achieve a significant increase in overall disk throughput through the intelligent use of striping. It should be stressed that the actual difference in system throughput is highly dependent upon the particular system configuration and application.

The other major advantage of the use of disk striping is flexibility in disk layout. It is possible to create a single striped partition which holds more data than any single physical disk drive can hold. By combining two or more physical disk partitions (potentially of different sizes) you can create a striped disk partition which has the same size as the sum of the sizes of the conventional disk partitions. For example, with DKD-101 disks, you could combine two "a" partitions (~17 Mbytes each) and an "h" partition (~106 Mbytes) to create one striped partition which holds ~140 Mbytes.

4.1.8 Multiple IOP Support

Disc striping is completely transparent to the UNIX user and requires no source code changes. CONVEX UNIX will handle all the data repartitioning automatically. This includes full support for systems with multiple IOPs, where further concurrent I/O performance is possible.

Figure 4–2: Multiple IOPs



4.2 Asynchronous I/O

Further performance enhancements to the file system are provided by the asynchronous I/O feature. Asynchronous I/O permits concurrent disk and tape I/O so that the applications program does not have to wait for the completion of the I/O to continue executing. This is an important capability in many key applications areas such as geophysical processing and finite element analysis. This I/O structure fully utilizes the CONVEX parallel I/O structure based on independent and asynchronous Input Output processors(IOPs). The benefits to the include improved system throughput because I/O transfers overlap with the CPU and I/O processors.

4.3 The CONVEX C Programming Language

C is a general-purpose high-level programming language that has been used extensively for systems programming. Although it has been associated with the UNIX operating system, it provides many of the language features of other modern high-level languages. Subroutines written in C can be linked with both Fortran and assembly language routines to make one program. The CONVEX C compiler is an extended implementation of the C language originally defined at Bell Laboratories.

The data types and control structures of the C language are efficiently supported by the CONVEX architecture. One significant example of the implementation of systems programs in C is the CONVEX UNIX operating system. The operating system kernel and its associated utilities have been implemented in C.

The C language provides a variety of control-flow statements that allow for the development of well-structured, readable programs. Decisions can be made using the if statement with optional else clauses or with the switch statement which transfers control to one of several cases depending on the value of an expression. Looping constructs are supported by the for, while, and do statements. With the for and while statements, the loop termination condition is tested at the beginning of the loop. With the do statement, the loop terminates at the end.

CONVEX C directly supports integer data types of several different sizes. Integers can be declared as "short int" (16 bits), "int" and "long int" (32 bits), and "long long int" (64 bits). In addition, C supports a character data (8 bits) that can also be used for 8 bit integers. Floating point numbers are also supported in two different sizes: "float" (32 bits) and "double" (64 bits). There are also enumerated data types, structured data types, pointers, and arrays.

Unlike many other high-level languages, the C language does not define I/O operations. However, there are extensive facilities for performing I/O operations and accessing other operating system services with the standard CONVEX run-time libraries.

An additional utility program, called *lint* applies strict rules for checking many aspects of a program for correctness. In addition, the CONVEX C language is fully supported by the CONVEX symbolic debugger (csd).

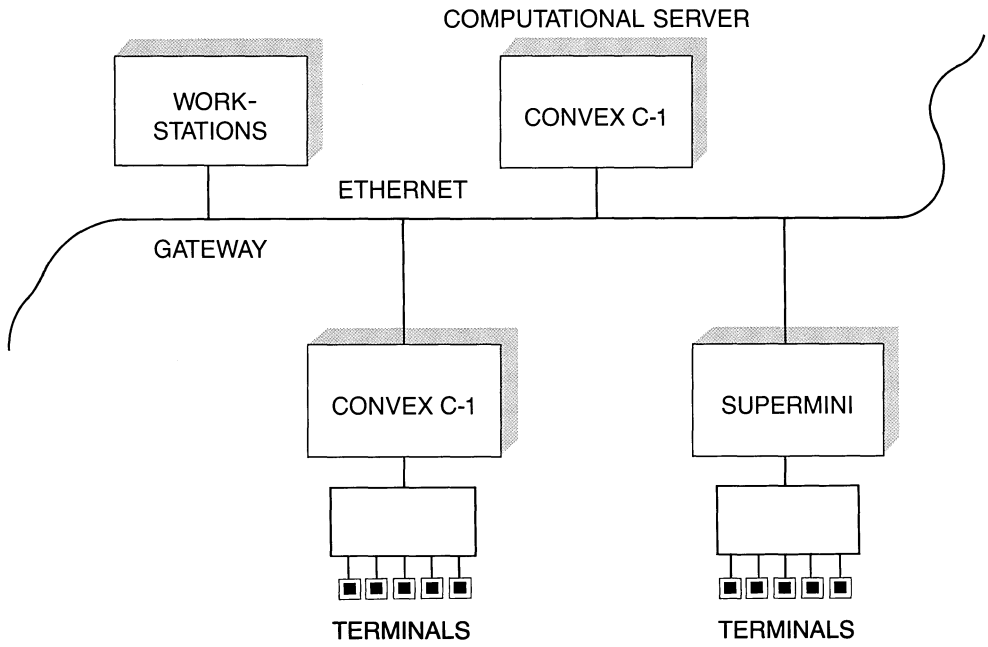
The CONVEX C provides an excellent language environment for the implementation of systems programs on the C-1 computer system.

4.4 Networking

Local area networks (LANs) have evolved rapidly in the last few years. The rapid development of LANs is driven by the need for efficient information exchange in a local computer environment. Since more computer systems are being installed, computer users both need and expect to have reasonable methods of exchanging information between these potentially heterogeneous computer systems.

The CONVEX network architecture has been specifically designed to address the growing need to share information. This integrated distributed supercomputer network will address the needs of the future as well as current requirements. This distributed networking system not only enables C-1 systems to communicate and share data; it also allows the C-1 to be a computational serving supercomputer in a network of workstations and minicomputers. Refer to Figure 4-3.

Figure 4–3: Networking



The CONVEX networking product, LAN-001, is a local area network based on the industry standard Ethernet. The communications protocol is IP/TCP(Internet Protocol/Transmission Control Protocol). IP/TCP provides an industry and DOD standard method of medium performance communication amongst multiple host computer systems. Communication amongst the hosts can involve file transfers, mail, virtual terminal services, remote execution, etc.

The CONVEX Product LAN-001 enables the user to do the following:

- Network a C-1 to other C-1s
- Network a C-1 to other systems supporting IP/TCP running UNIX 4.2
- Network to a VAX running VMS using the IP/TCP software

LAN-001 includes everything needed to connect a C-1 to an existing Ethernet network, specifically:

- Network Software Utilities
- Documentation
- Multibus Controller
- Transceiver
- Transceiver Cable

4.4.1 Network Commands/Utilities

Below is a brief description of some of the commands and utilities supported by LAN-001:

ftp - File Transfer Program. This utility allows a user to transfer files between host computer systems.

rcp - Remote Copy. Copies files from host to host.

rlogin - Remote Login. This connects a user terminal on the current host system to a remote host system.

rsh - Remote Shell. Connects to a specific host and executes the specific command.

TELNET - User interface to TELNET protocol. This is used to communicate with another host using the TELNET protocol.

TELNET and ftp enable networking with VAX/VMS, and appropriate IP/TCP software, systems. A C-1 terminal user can login to VMS using the TELNET utility command, or visa versa. Or the user can transfer a file using the *ftp* utility.

4.5 Fortran

CONVEX Computer Corporation's Fortran compiler is one of the most sophisticated compilers available. Designed to ANSI '77 standards, the CONVEX Fortran compiler is a multipass, optimizing, vectorizing compiler that includes both language and run-time library (RTL) extensions. The compiler will process existing Fortran programs without modification to the source, to take advantage of the full performance potential of the C-1 supercomputer's architecture. As a result, CONVEX Fortran increases programmer productivity, and maximizes maximum software portability and speed of execution.

4.5.1 Features

- CONVEX Fortran is an ANSI standard Fortran '77 compiler with optional support for programs conforming to the previous standard ANSI X3.9-1966.

- VAX-11 Fortran source code compatibility allows existing VMS programs to be easily ported.

- CONVEX Fortran offers optimization and vectorization as major features. Classical global and local optimization and vectorization of object code mean significant improvement in performance. The optimizing compiler performs dataflow analysis on iterative sequential procedures to produce parallel executable code that fully utilizes the integrated vector processing capabilities of the CONVEX hardware. In essence, the CONVEX Fortran compiler makes the hardware run faster.

- After compilation, a vectorization summary is generated that provides the programmer with information relating to the vectorized Fortran.

- Three levels of optimization are available with the CONVEX Fortran compiler: levels 0, 1, and 2. Level 0 is a set of local optimizations that are performed where 'local' means a sequence of consecutive code with one entrance and one exit. Level 1 is global optimization which refers to optimizations performed over the entire program unit, in particular IF statements and loops. Finally, level 2 is vectorization optimizations which replace DO loops with equivalent intermediate vector code. All three levels of optimization are object code compatible, which means that a programmer can link and execute programs that have been compiled with any combination of optimization level.

- No special vector syntax is required to utilize this highly advanced integrated vector processing architecture and the automatic vectorizing compiler fully. In other words, the user writes in standard Fortran for maximum programmer productivity.

- The user can specify compiler directives that provide information to the compiler to override conditions that inhibit optimization or vectorization. For example, the NO_RECURRENCE directive instructs the compiler to vectorize the following DO loop which would not normally vectorize because of an apparent recurrence. These compiler directives result in enhanced

program execution efficiency.

- Vector intrinsics are provided with the compiler. These are intrinsic functions within a vectorized loop, which are faster than the corresponding scalar intrinsics in a loop.

- Fully integrated with the CONVEX UNIX environment, the shareable, reentrant compiler operates under the CONVEX UNIX operating system to take full advantage of the C-1's architecture and virtual memory system, and the wide range of UNIX utilities.

- The CONVEX Fortran compiler has been designed to anticipate Fortran 8x, the proposed ANSI standard for a vector Fortran syntax, when it is introduced.

- CONVEX Fortran extensions include a wide range of data types with automatic vectorization across all numeric data types.

- The CONVEX CONSULTANT program development tools, which include a run-time performance analyzer and the interactive high level symbolic debugger, csd, are fully integrated with the CONVEX compiler to simplify program development.

4.5.2 Optimization and vectorization

The CONVEX Fortran compiler's optimizations help to produce code that results in enhanced performance levels. Optimization involves the careful manipulation of operations in the source programs being compiled--essentially, data flow analysis is performed on iterative sequential procedures to produce parallel executable code. The result is an object program that can run more efficiently.

The CONVEX Fortran compiler uses both machine independent and machine dependent optimization techniques, where:

- Machine-independent optimization techniques are performed at the source program level and do not depend on the object language to be produced.

- Machine-dependent optimization techniques improve the efficiency of the object code by making the best use of the machine language and the underlying architecture of the machine, including the integrated vector processor.

To achieve the highest performance possible, the compiler must perform many sophisticated scalar and vector optimizations.

The compiler performs typical scalar optimizations, like constant propagation and folding, common subexpression elimination, strength reduction, dead code elimination, code motion,

precomputed subroutine argument packets, and many others.

After scalar optimizations are performed for all code, vectorization is performed. The compiler invokes a vectorizer, which replaces DO loops with vector code. Assembly language that makes use of vector registers and instructions is later generated from the intermediate code by the compiler's code generator.

Vectorization is an optimization that identifies DO loops and then performs a dependency analysis to determine what can be vectorized.

Among the vectorization optimizations performed by the CONVEX compiler are:

- Vectorization across all nested loops
- Loop interchange
- IF statements within DO loops
- Recognition of reduction operations (e.g. SUM, PRODUCT, AND, OR, EXCLUSIVE OR, MAX, MIN, POPULATION COUNT)
- Recognition of scatter/gather (vector of indices)
- Recognition of vector edits (MASK, MERGE, and COMPRESS)
- Partial vectorization of statements (partitions a loop into two or more loops, one which is vectorizable and one which is not)
- Strip mining (vector operations into 128 element groupings)
- Vector register optimizations
- Vectorization for logical, integer, floating point (single and double) and complex data types

Further machine-dependent optimizations are used to enhance the object code produced by the compiler. Typical examples include register allocations to maximize the number of registers in order to achieve more parallelism, and strength reduction operations on instruction-level operations.

One of the final optimizations performed is instruction scheduling. This machine dependent optimization determines an order of instructions which effectively uses the asymmetric processing architecture and pipelining of the C-1 system. Instruction scheduling results in a high level of execution concurrency and chaining of vector operations. A user's application program can automatically achieve peak C-1 performance in excess of 60 million operations per second.

4.5.3 Data Types

Standard and extended data types allow for total flexibility. In addition to the standard data types of LOGICAL*4, REAL*4, REAL*8, INTEGER*4, COMPLEX*8, and CHARACTER, seven new data types are provided:

- LOGICAL*1
- LOGICAL*2
- LOGICAL*8
- INTEGER*1, (BYTE)
- INTEGER*2
- INTEGER*8
- COMPLEX*16

4.5.4 CONVEX Consultant

The run-time performance analyzer and interactive source level (csd), are made available through the CONVEX CONSULTANT. To help the user enhance the performance of Fortran applications on the C-1, the CONVEX CONSULTANT provides an execution profile/analyzer as a tuning aid. Typical profile information for a subroutine includes the amount of CPU time required for each subroutine, the percent of overall program execution assigned to subroutines, the number of times each subroutine is called and from where. The CONSULTANT provides a complete run-time diagnosis of the application program execution.

The CONVEX CONSULTANT also includes an interactive screen oriented source level debugger. The CONVEX source level debugger (csd), provides a rich repertoire of commands that enable such capabilities as variable or source line trace, support for the breakpoint instruction, symbolic access to program variables, single step, and edit capability.

4.5.5 Portability and Support

ANSI '77 CONVEX Fortran with VAX/VMS extensions allows users to port existing Fortran applications programs with minimal effort. Through the many features provided and by compliance to industry standards, the CONVEX Fortran compiler assures maximum portability of programs written on other computer systems.

5 Vectorization

5.1 What is Vectorization ?

Vector processing is the system's ability to perform operations on arrays of data simultaneously. In contrast, the scalar processing used by most computers deals with only one operation and one element at a time. In order to perform vector operations, a computer must have the necessary hardware and software: an integrated vector processing unit and a vectorizing compiler.

Basically, vector processing takes advantage of those programs which have operations called loops and allows the CONVEX FORTRAN compiler to run vectorization optimizations on these loops and to compile the user program for translation into assembly language.

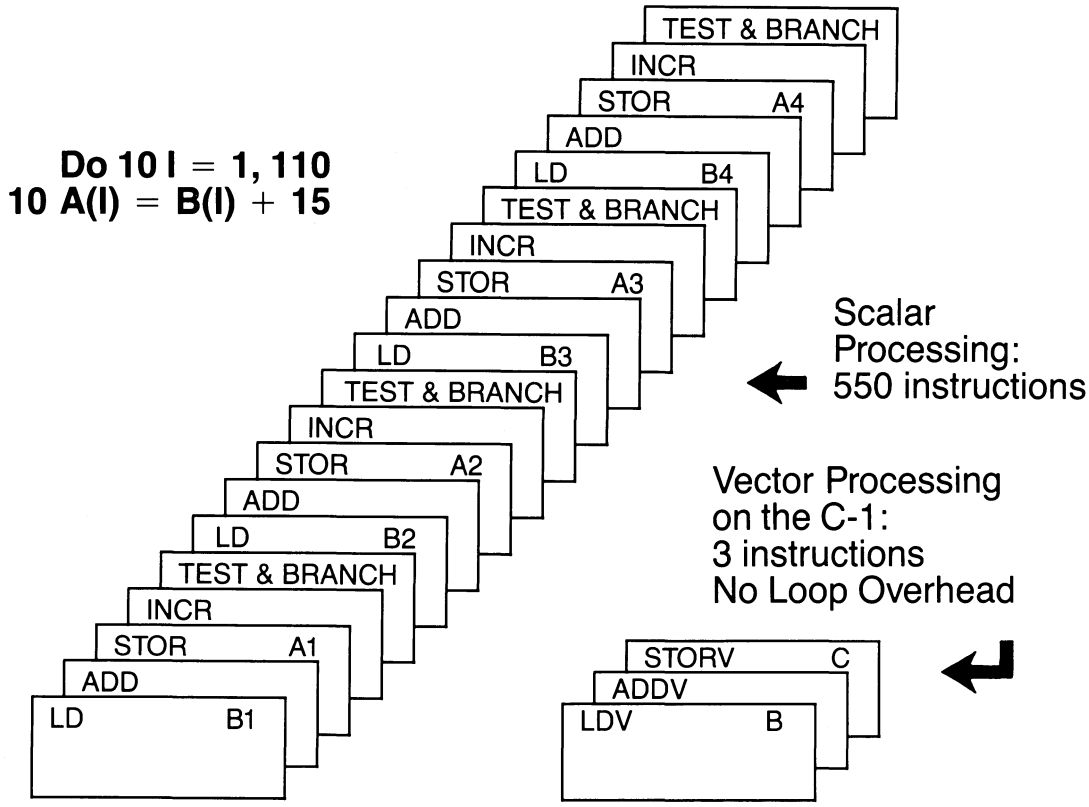
Program loops occur frequently within scientific and technical application programs. A simple loop example is shown below that adds 15 to each element of array B and stores the sum in array A. The loop induction variable I is used as the index into array A and B, and ranges from 1 to 110 in increments of 1, using contiguous memory locations:

```
      DO 10 I = 1, 110
10    A(I) = B(I) + 15
```

In traditional scalar computer systems, the machine instructions produced by the compiler for this example is a **sequential** issue of instructions that loads individual elements of array B, then adds 15, then stores the result into array A. The loop induction variable I is incremented by 1, and each step in the loop is then tested for completion. If any part of the loop is incomplete for any given instruction, the compiler "branches" back into the loop to compile the omission. Each of the five steps of this loop--load, add, store, increment, test and branch--are repeated over and over on each element of array B until 1 equals 110. In this method of processing arrays, the single elements of arrays A and B and the constant 15 are called scalars, and, hence, scalar processing is the mathematical and logical operations applied to scalars.

In contrast, vector processing is the technique whereby the instructions produced by the compiler for the same loop include one instruction that loads ALL of the elements of array B at once, one that adds 15 to each of the elements of array B, and a final single instruction that stores the result into array A. In this method of processing arrays, A and B are called vectors, and, hence, vector processing is the mathematical and logical operations applied to vectors. See Figure 5-1.

Figure 5–1: Vectorization Example



5.2 What are the advantages of vector processing ?

The biggest single advantage of vector processing versus scalar processing is increased performance. Increased performance occurs because it allows a pipelined memory system to run at full bandwidth and it eliminates the overhead associated with the incrementing and testing of the loop induction variable I. Consequently the loop is reduced to a very simple instruction sequence, one load, one add, and a single store instruction. The benefits to performance are enormous -- an order of magnitude increase in performance is realized.

5.3 How does this benefit the C-1 ?

The C-1 supercomputer is an integrated scalar and vector processor and has **both** instructions and register sets for both scalar and vector data. As not all problems lend themselves to vectorization, the capability to process scalar data quickly and efficiently is also important. Plus the CONVEX Fortran compiler is an automatic vectorizing compiler that performs dataflow analysis on iterative sequential procedures to produce the parallel

executable code that fully utilizes the architectural benefits of the integrated vector processor of the C-1.

Clearly the full performance potential of the C-1 can only be reached by running vector code. Achieving peak performance is dependent on vector length, vector stride and percent vectorization as explained below.

5.3.1 Vector Length

For those applications that have a vector length of 3 or less the performance will only equal that of scalar. As the vector length increases so will the performance. An example of this is the Los Alamos program BMK8a1. This program measures the MFLOPS(Million Foating Point Operations per Second) rate for certain expressions using several different contiguous vector lengths. For the following expression:

V = S*V+S*V (V=Vector, S=Scalar)

the C-1 MFLOPS rate is 3.66 for vector length of 10. This same expression with vector length of 100 results in an MFLOPS rate of 13.00. The CRAY-1S results are 16.59 and 53.14 MFLOPS for vectors of the same length. However, the same program ran on the VAX/780 returns an MFLOPS rate of 0.34 and 0.35. It can be concluded that integrated scalar and vector systems return greater performance relative to scalar systems as the vector length increases.

5.3.2 Vector Stride

Vector stride is the offset between consecutive vector elements in memory that is used as the array index; it is the number of memory locations between consecutive vector elements. A stride of 1 is often called contiguous or unity (no gaps between elements). When the stride is a value greater than one, it is constant, for example:

```
DO 10 I = 1, N      - defaults to contiguous stride of 1

DO 10 I = 1, N, 2   - constant stride of 2
```

A random stride, often called scatter/gather, is where the stride is neither constant nor contiguous, for example:

```
      DO 10 I = 1, 100
10    A(I) = B(INDEX(I)+K)
```

where INDEX is an integer array set up by a random number generator and K is an arbitary constant.

Where Fortran DO loops have a contiguous or unity(no gaps), the C-1 operates at maximum speed. If the stride is either constant or random, the C-1's performance is slightly less.

5.3.3 Percent Vectorized

There is no industry definition for "percent vectorized". Essentially, the term refers to the percent of code which the system can vectorize, so it is a reference to the vector processing time over the scalar processing time for any given program. For example:

- Los Alamos Program BMK4a1 executes on the C-1 in 14.9 secs without vectorization and 5.5 sec with vectorization, giving a 63% percentage vectorization, or a 2.96 times faster rate.
- Another example is Los Alamos program BMK14 which executes in 209 secs in scalar mode and 20.0 secs when vectorized, 10.53 times scalar.

5.4 What Are the Vectorization Optimizations Performed by the CONVEX Compiler?

The CONVEX FORTRAN compiler is an intelligent, sophisticated compiler capable of automatic vectorization across all data types. Many of the vectorized operations it runs are simply not available on conventional systems. The key features--transparent at the user level-- include:

- Vectorization across all nested loops within a program
- Loop interchange (Performed in nested DO loops to ensure that contiguous stride comes first)
- IF statements within DO loops
- Recognition of reduction operations (SUM, PRODUCT, AND, OR, EXCLUSIVE OR, MAX, MIN, POPULATION COUNT)
- Recognition of scatter/gather (vector of indices)
- Partial vectorization of statements (where a loop is partitioned into two or more loops, one of which is vectorizable and one which is not)
- Strip mining (vector operations are divided into 128 element groupings automatically)
- Vector register optimizations
- Vectorization for logical, integer, floating point (single and double) and complex data types

5.5 Automatic Vectorization

Vectorization on the C-1 is automatic and transparent to the user. The following example shows the compiler results based on the Dongarra Vector Program and 24-loop Lawrence Livermore programs:

Figure 5–2: Automatic Vectorization

COMPILER PERFORMANCE
DONGARRA'S VECTOR PROGRAM

Loop Label	1	2	3	4	10, 11	20, 21	30, 31	40	50	60	70	50, 51	80	100	110	120, 121	130, 131	140	150	160, 161	170, 171
CRAY CFT 1.15					X		X					X									
FUJITSU 77/VP							P	X				P									X
HITACHI Fort 77/hap							P	X													
Cyber 20s 200 1/7/85	X	X	X	X	X	X	X		X	X	X	X		X		X	X	X	X	X	X
CONVEX							P					P									

X - Loop Not Vectorized
P - Loop Partially Vectorized

COMPILER PERFORMANCE
24 LOOP LIVERMORE LOOPS

KERNEL	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	TOTAL
CRAY-1	V		V			V	V	V	V		V							V			V	V		10
FUJITSU	V	V	V	V		V	V	V	V	V		V	P	P	V				V		V	V	P		17
CYBER-205		V		V			V		V			V						V			V	V			8
CONVEX C1		V	V	V	V		V	V	V	V	V		V	P	P	V		V			V	V	P		17

V - Fully Vectorized
P - Partially Vectorized

5.5.1 Examples

5.5.1.1 Vectorization Across All Nested Loops:

Nests of DO loops are vectorized by first distributing the outermost loop, then vectorizing each of the resulting loops or loop nests. The example below shows a nest of DO loops which the compiler has to confront:

```
      DO 20 I=1,N
      B(I,1)=0
      DO 10 J=1,M
      A(I)=A(I)+B(I,J)*c(I,J)
10    CONTINUE
      D(I)=E(I)+A(I)
20    CONTINUE
```

The compiler distributes the outer loop, then generates intermediate code equivalents to complete the compilation of the code. Note that the compiler builds three loops, introducing labels (20a, 20b, and 20c) to represent the three intermediate code equivalents:

```
      DO 20a I=1,N
      B(I,1)=0
20a   CONTINUE

      DO 20b I=1,N
      DO 10  J=1,M
      A(I)=A(I)+B(I,J)*C(I,J)
10    CONTINUE
20b   CONTINUE

      DO 20c I=1,M
      D(I)=E(I)+A(I)
20c   CONTINUE
```

5.5.1.2 Loop Interchange

The 20a and 20c loops and the 10 loop of the foregoing example are all innermost loops and can be vectorized directly. However, to yield additional performance improvement, the vectorizer performs the 'loop interchange' optimization on the middle nest of loop 20b and loop 10, replacing it with the following nest:

```
      DO 10 J=1,M
      DO 20b I=1,N
      A(I)=A(I)+B(I,J)*C(I,J)
20b   CONTINUE
10    CONTINUE
```

The point here is that when the vector code is generated for 20b loop, elements of B and C are accessed contiguously as they are loaded into vector registers. This provided a substantial

performance improvement over the noncontiguous access that results if the interchange is not performed.

5.5.1.3 IF Statements Within DO Loops

As these examples have shown, the user rarely needs to exercise great care in writing a loop, assuming that the loop really is vectorizable. DO loops containing nested IF statements and nonlinear subscripts (subscripts whose values on succeeding iterations of a loop do not form arithmetic progressions) can be vectorized. The compiler's ability to process nested IF statements is illustrated below:

```
      DO 10 I=1,10
      A(I)=B(KK(I)+C(I*I)
      IF(A(I).LT.0)THEN
        IF (A(I).GT.-100)A(I)=0
      ELSE
        A(I)=SQRT(A(I))
      ENDIF
10    CONTINUE
```

Many more examples for each of these features are provided in Chapter 4 of the *CONVEX FORTRAN User's Guide.*

5.6 How Can I Restructure My Code to Ensure Maximum Performance?

There are innumerable ways to enhance and optimize programs to take advantage of the vectorizing capabilities of the C-1. The first example below shows a call routine to a scalar function inside a DO loop, and the different ways in which a scalar processor and a vector processor handle the same equation. The first equation shows the scalar processing approach to the problem:

```
    •
    •
    •
DO i=1,1000000
  call dist(x(i),y(i),hyp(i))
ENDDO
    •
    •
    •
subroutine dist (x,y,hyp)
hyp = sqrt(x**2 + y**2)
RETURN
END
    •
    •
    •
```

This first example generates scalar code as the vectorizer cannot vectorize subroutine calls.

Here is the same problem, this time with the code restructured to take advantage of the vector processing capabilities of the C-1.

```
    •
    •
    •
call vdist(1000000,x,y,hyp)

    •
    •
    •
subroutine vdist(n,x,y,hyp)
dimension x(n),y(n),hyp(n)
DO i=1,n
  hyp(i) = sqrt(x(i)**2 + y(i)**2)
ENDDO
RETURN
END

    •
    •
    •
```

This example of generates vector instructions and calls a vector version of the same function, but passes the length of the vectors as an additional parameter, moving the loop **inside** the function.

A second example, a typical finite element problem, is shown below, with three parameters. The C-1 compiler can vectorize both of these equations, but the second executes faster than the first for the following reasons. In the first equation, the stride length is constant; hence, as described in the section on Vector Stride above, the C-1's performance is slightly less because constant stride compilations perform loads and stores **via** the cache memory, impeding the speed:

```
DO 32 K = 1,NK
DO 32 J = 1,NJ
DO 32 I = 2,NI

      DW1(I) = W(1,I,J,K)-W(1,I-1,J,K)
      DW2(I) = W(2,I,J,K)-W(2,I-1,J,K)
      DW3(I) = W(3,I,J,K)-W(3,I-1,J,K)

32    CONTINUE
```

When the code is restructured to change the stride length from constant to contiguous, the program runs in half the time:

```
DO 32 K = 1,NK
DO 32 J = 1,NJ
DO 32 I = 2,NI

      DW1(I) = W(I,J,K,1)-W(I-1,J,K,1)
      DW2(I) = W(I,J,K,2)-W(I-1,J,K,2)
      DW3(I) = W(I,J,K,3)-W(I-1,J,K,3)

32    CONTINUE
```

The performance improvement is remarkable--50%--and is all controlled at the programmer level.

5.7 Is Vector Code on the C-1 Transportable?

Vector code on the C-1 is fully transportable, because CONVEX FORTRAN supports standard FORTRAN and runs on any machine. The C-1's vector code does not require the machine dependent code changes needed to utilize the many experimental parallel or multi-processor systems being developed.

5.7.1 What affect does I/O have on the system performance ?

Clearly I/O performance will never achieve the same performance that CPU intensive vector execution will for several reasons:

- I/O instructions are not processed by the vector processing subsystem.
- Parts of the device drivers execute on board the IOPs which will offload the

CPU but instruction execution times are slower than the C-1 scalar speed.

- Heavy I/O programs will be limited by the physical speed of the device.

A heavy I/O load can cause a performance bottleneck on any system. Typically, this does not degrade CPU user time but can increase elapse(wall clock) time. The final result for heavy I/O bound programs could be excessive elapse/wall clock time.

If you can reconstruct the program to reduce the number of calls to the Fortran read and write routines, this would reduce the system time and result in improved wall clock times. Obviously, this is similar to scalar versus vector performance and will be very application dependent.

Some examples of recommended I/O structures:

- Use unformatted I/O wherever possible
- One array per implied-DO

```
      READ(10) (A(I), I = 1, 10),
     *         (B(I), I = 1, 10)
```

is better than:

```
      READ(10) (A(I), B(I), I = 1, 10)
```

- Use of implied-DO loops is better than enclosing read or write statements in DO loops. For instance:

```
      READ (10) (A(I), I = 1,N)
      DO 10 I = 1,N
10    B(I) = A(I)
```

is better than:

```
      DO 10 I = 1, N
            READ (10) A(I)
            B(I) = A(I)
10    CONTINUE
```

5.7.2 Summary

- The effective speed of the C-1 supercomputer is workload dependent. It is necessary to know the workload well in order to predict its performance.
- The slowest characteristic processing speed will affect the effective speed

critically unless the fraction of workload associated with that speed is negligibly small. The most effective way to speed up a machine is to increase this speed or to decrease the fraction of work associated with it.

6 CONVEX FORTRAN Optimizations and Vectorizations

6.1 Overview

The CONVEX FORTRAN compiler optimizations help to produce code that results in enhanced performance levels. Optimization involves carefully manipulating operations in the source programs being compiled; the result is an object program that can run more efficiently. Of course, when you compile programs using the optimization switch (*-Ol*), the compilation time increases as more optimizations are performed. Also, use of the optimization option affects the symbolic debugging capabilities of the debugger, *csd*.

The CONVEX FORTRAN compiler uses both machine-independent and machine-dependent optimization techniques, where:

- Machine-independent optimization techniques are performed at the source program level and do not depend on the object language to be produced.
- Machine-dependent optimization techniques improve the efficiency of the object code by making the best use of the machine language features.

6.2 Machine-Independent Optimization

6.2.1 Local Optimization

A set of optimizations is always performed where 'local' means a sequence of consecutive code with one entrance and one exit.

6.2.1.1 Assignment Substitution

Assignment substitution is the process by which the compiler removes redundant loads and stores. It involves substituting a preassigned value of a variable to all succeeding uses of the variable. For example:

```
x=y+c
 ...
  x
 ...
  x
x=y(z)
```

Effectively, the *y+c* replaces all uses of *x* up to the next assignment to *x*. As a result, the compiler can eliminate the loads on *x* if *x* can be retained in a register. Not only does this optimization save space and time, but it also opens up opportunities for

other optimizations, e.g., constant folding and redundant subexpression elimination.

6.2.1.2 Redundant-Assignment Elimination

This optimization involves removing redundant assignments to the same variable. An assignment to a variable can be followed by another assignment to the same variable, wiping out the result of the first assignment. Clearly, there is no need for the program to perform the first assignment; therefore, the compiler eliminates it.

6.2.1.3 Redundant-Use Elimination

This optimization collapses all uses of a variable between two assigns into one use; it is a simplistic form of redundant-subexpression elimination. As a result, the compiler can eliminate loads provided it can retain the variable in a register.

6.2.1.4 Redundant-Subexpression Elimination

Redundant-subexpression elimination involves removing repeated evaluations of equivalent arithmetic, logical, and relational operations, which are recognized as common subexpressions. When the compiler detects a common subexpression, it removes all but one of the common subexpressions from the program and replaces it with the one retained in a register. In the following example the first *y+c* replaces the second occurrence but not the third:

```
y+c
 ...
y+c
 ...
y=...
 ...
y+c
```

6.2.1.5 Constant Propagation and Folding

Propagating constants in a program means that when you assign a constant to a variable, everywhere the variable occurs later, the compiler replaces it with the constant. So, if you assign x = 5, wherever *x* occurs later, constant propagation replaces it with the constant 5.

In constant folding, when the compiler comes across an operation on constants, like y = 5 + 7, it replaces the operation with its value (here, 12). The compiler may assign the new value to *y*, so that *y* can now be propagated. Constant propagation and folding are optimizations which save both time and memory, as illustrated in the following example:

Original Program	Equivalent Transformed Program
i = 5 j = 0 ... j = j+2 ... k=k+i*j	i=5 ... j=2 ... k=k+10

Compile-time type conversions involve the compiler's performing type conversions on mixed-mode expressions. The conversion of constants and folded constants is performed as part of the constant propagation and folding optimization. For example, if the program contains the expression x = 1, the compiler converts the 1 to REAL data type; the effect is as if x = 1.0 had been written.

If during constant folding an integer overflow occurs, you get a user error message ("Integer constant truncation"). If floating-point underflow occurs, the folded result is zero. If floating-point overflow occurs, a user error message displays ("Real constant either too large or too small"). The user cannot override these compiler actions. It is necessary to modify the source, replacing the operator or constants with the correct constant.

6.2.2 Global Optimization

Global optimization refers to optimizations performed over the entire program unit, in particular, IF statements and loops.

6.2.2.1 Constant Propagation and Folding

Global constant propagation and folding is similar to local constant propagation and folding, except the folded constant is propagated across the program unit, provided the definition reaches the use.

Example:

Original Program		Equivalent Transformed Program	
	PROGRAM GFOLD1		PROGRAM GFOLD1
	INTEGER A,B,C		INTEGER A,B,C
	A=5		A=5
	B=15		B=15
	READ *,I		READ *,I
	IF (I) 10,10,15		IF (I) 10,10,15
10	A=6	10	A=6
	C=A		C=6
	GOTO 20		GOTO 20
15	C=A+B	15	C=20
	GOTO 25		GOTO 25
20	B=A+C	20	B=12
	GOTO 30		GOTO 30
25	B=A+8+C	25	B=33
30	PRINT *,A,B,C	30	PRINT *,A,B,C
	END		END

6.2.2.2 Dead-Code Elimination

As a result of constant propagation and folding, the arithmetic or logical expression of an IF statement may be folded to .TRUE. or .FALSE.. The alternative that is unreachable will be eliminated.

A classic example of the use of dead-code elimination is in conditional compilation. Code that is to be conditionally compiled is enclosed by an IF statement that tests a variable whose value is set to .TRUE. (compile enclosed code) or .FALSE. (do not compile enclosed code) via an assignment or PARAMETER statement, or data initialization.

6.2.2.3 Redundant-Assignment Elimination

Redundant-assignment elimination involves removing assignment statements (definitions) that are never used. Label assignments (i.e., ASSIGN statements) and assignments to a dummy parameter, name of a function, common variable, or local variable of a subprogram are never eliminated. Also, if the right side of an assignment statement involves a function call, the assignment is not eliminated, because the function could have side effects.

Example:

Original Program	Equivalent Transformed Program

program grael
common a,b,c,x,y,z
...
x=y*z
if (a.gt.0) then
...
* x not used
a=x*y+ ...
else
...
* x not used
x=a-b*c
end if
...
* a,x not used
end
	program grael
	common a,b,c,x,y,z
	...
	if(a.gt.0) then
	...
	else
	...
	end if
	...
	end

6.2.2.4 Redundant-Subexpression Elimination

Like local redundant-subexpression elimination, global redundant-subexpression elimination involves the removal of common subexpressions. However, instead of retaining the value of one common subexpression in a register, the value is assigned to a compiler-generated temporary; all other occurrences are replaced by use of this temporary.

Example 1:

Original Program	Equivalent Transformed Program

program gcsel
...
read *,c
if (k .lt. l) then
a=b+(c*4)/(-(j*b)+sqrt(c))
else
e=e-(b+(c*4)/(-(j*b)+sqrt(c)))
end if
f=b+(c*4)/(-(j*b)+sqrt(c))
...
end
	program gcsel
	...
	read *,c
	t1=b+(c*4)/(-(j*b)+sqrt(c))
	if (k .lt. l) then
	a=t1
	else
	e=e-(t1)
	end if
	f=t1
	...
	end

The common subexpression is moved only if it is safe, which means that the subexpression would always be evaluated.

Example 2:

Original Program	Equivalent Transformed Program

program gcse2
...
read *,c
a=b+(c*4)/(-(j*b)+sqrt(c))
if (k .lt. 1) then
l=5
else
l=6
end if
f=e-(b+(c*4)/(-(j*b)+sqrt(c)))
...
end
	program gcse2
	...
	read *,c
	t1=b+(c*4)/(-(j*b)+sqrt(c))
	a=t1
	if (k .lt. 1) then
	l=5
	else
	l=6
	end if
	f=e-t1
	...
	end

6.2.2.5 Code Motion

Code motion involves taking invariant computations in a loop and moving them before the loop. An invariant computation is one that yields the same result independent of the number of times through the loop. The computation can be a subexpression or assignment. For safety reasons, no code motion is performed on an invariant expression whose evaluation point does not lie on a path to all loop exits.

Example 1:

Original Program	Equivalent Transformed Program
program cm1 common a,b,e dimension ar(10) ... read *,c do i=1,10 a=b+(c*4)/(-(e*b)+sqrt(c)) ar(i)=a+b*c enddo ... end	program cm1 common a,b,e dimension ar(10) ... read *,c a=b+(c*4)/(-(e*b)+sqrt(c)) t1=a+b*c do i=1,10 ar(i)=t1 enddo ... end

Example 2:

Original Program	Equivalent Transformed Program

subroutine cm2
common a,b,c(10)
do i=1,10
  a=b
  c(i)=0
enddo
  ...
end
	subroutine cm2
	common a,b,c(10)
	a=b
	do i=1,10
	  c(i)=0
	enddo
	  ...
	end

6.2.2.6 Strength Reduction

Strength reduction involves replacing an operator whose operands are either a loop induction variable or a loop constant with an operator that executes faster. A loop induction variable is one whose value is changed within the loop linearly, i.e., incremented by a constant amount. A loop constant is a constant or variable that is loop invariant, i.e., whose value is not changed within the loop. Thus, for a loop on *i* containing *j=j+i*, *j* is not an induction variable. A typical operator subject to strength reduction is multiplication, in particular those multiplications involved in the address calculation of subscripted variables.

Strength reduction of *i*r* is not performed. The reduced operations are not numerically equivalent due to the imprecision of floating point for large numbers. For safety reasons, no strength reduction is performed on an expression whose evaluation point does not lie on a path to all loop exits.

Example:

Original Program	Equivalent Transformed Program

program sr1
i=1 ! i is a loop induction variable
10 x=i*c ! c is loop invariant
  ...
i=i+2
if (i .le. 100) goto 10
  ...
end
	program sr1
	i=1
	t1=i*c
	t2=2*c
	10 x=t1
	  ...
	t1=t1+t2
	i=i+2
	if(i .le. 100) goto 10
	  ...
	end

If i is dead on exit from the loop, i.e., is not used before being assigned, and there are no other uses of i in the loop except in the incrementation and test, the incrementation can be eliminated and the test replaced by a test on the induced induction variable—the induced temporary. This optimization is known as linear-

function test replacement. After linear-function test replacement, the equivalent transformed program is:

```
        program sr1
        i=1
        t1=i*c
        t2=2*c
        t3=100*c
10      x=t1
          ...
        t1=t1+t2
        if (t1 .le. t3) goto 10
          ...
        end
```

6.2.3 Vectorization

When you specify *-O2* as the optimization level, the compiler invokes a vectorizer, which replaces DO loops with vector intermediate code. Assembly code that makes use of vector registers and instructions is later generated from the intermediate code by the compiler's code generator.

The vectorizer vectorizes innermost DO loops directly. For example, vector code is generated for the following loop:

```
        DO 10 I = 1,100
        A(I) = B(I)+C(I)
10      CONTINUE
```

Instead of generating a loop to repeatedly load elements of *B* and *C*, add them, store into *A, and advance I,* vector code is generated to load 100 elements of *B* into a vector register, load 100 elements of *C* into another vector register, add them, and store the 100 resulting elements from the result vector register in *A*.

The algorithms that the CONVEX vectorizer uses are very general. You rarely need to exercise great care in writing a loop, assuming that it really is vectorizable. DO loops containing nested IF statements and nonlinear subscripts (subscripts whose values on succeeding iterations of a loop do not form arithmetic progressions) can be vectorized. For example, the following loop will be fully vectorized:

```
      DO 10 I = 1, 10
      A(I) = B(KK(I)+C(I*I)
      IF (A(I).LT.0)THEN
         IF (A(I).GT.-100) A(I) = 0
      ELSE
         A(I) = SQRT(A(I))
      ENDIF
10    CONTINUE
```

The vectorizer does have some limitations, however, which are described in "Limitations on the Vectorizer" and "Recurrence".

6.2.3.1 Strip Mining

The vector registers of the C-1 hold up to 128 elements. When the number of iterations of a vectorizable loop exceeds (or could exceed) 128 elements, the vectorizer 'strip mines' the loop before vectorizing it. Strip mining replaces the loop with two loops, the innermost of which has an iteration count that never exceeds 128; e.g., strip mining makes the following conversion:

Original Loop	Converted Loop
DO 10 I = 1,N A(I) = B(I)+C(I) 10 CONTINUE	I = 1 DO 10a LV = N, 0, -128 DO 10b IV = I, I + MIN(128,LV)-1 10b A(IV) = B(IV) + C(IV) I = I + 128 10a CONTINUE

where *LV* is a variable introduced by the compiler to count the number of elements remaining to be processed, and the *10b* loop on *IV* represents a vector operation.

6.2.3.2 Loop Distribution

Nests of DO loops are vectorized by first distributing the outermost loop, then vectorizing each of the resulting loops or loop nests; e.g., consider the following nest of DO loops:

```
      DO 20 I = 1,N
      B(I,1) = 0
      DO 10 J = 1,M
      A(I) = A(I)+B(I,J)*C(I,J)
10    CONTINUE
      D(I) = E(I)+A(I)
20    CONTINUE
```

Distribution of the outer loop yields intermediate code equivalent to the following three loops:

```
      DO 20a I = 1,N
      B(I,1)=0
20a   CONTINUE

      DO 20b I = 1,N
      DO 10  J = 1,M
      A(I) = A(I) + B(I,J) * C(I,J)
10    CONTINUE
20b   CONTINUE

      DO 20c I = 1,M
      D(I) = E(I) + A(I)
20c   CONTINUE
```

where *20a*, *20b*, and *20c* represent labels introduced by the compiler.

6.2.3.3 Loop Interchange

The *20a* and *20c* loops and the *10* loop of the foregoing example are all innermost loops and can be vectorized directly. However, to yield additional performance improvement, the vectorizer performs the 'loop interchange' optimization on the middle nest of loop *20b* and loop 10, replacing it with the following nest:

```
      DO 10 J = 1,M
      DO 20b I = 1,N
      A(I) = A(I) + B(I,J) * C(I,J)
20b   CONTINUE
10    CONTINUE
```

The vectorization summary is:

> The loop on line 1.1 of file (*DO I*) is interchanged to be the innermost loop of the nest.
>
> The loop on line 1.1 of file (*DO I*) is fully vectorized.

When the vector code is then generated for the *20b* loop, elements of *B and C* are accessed contiguously as they are loaded into vector registers. This provides a substantial performance improvement over the noncontiguous access that results if the interchange is not performed, but the *10* loop vectorized with the *20b* loop on the outside.

6.2.3.4 Vectorization Summary

If a vectorizable loop is not vectorized, you may be able to use the summary information to guide you in inserting the appropriate compiler directive before the DO loop to ensure that it will be vectorized. You can use the *error* utility to merge the vectorization diagnostics (along with other diagnostic messages) with the source file.

6.2.3.5 Limitations on the Vectorizer

The vectorizer has certain limitations of which you need to be aware:

1. Only DO loops and nests of DO loops are processed. Hand-written loops are ignored, i.e., loops constructed out of IF and GOTO statements.

2. Loops containing character data or operations cannot be vectorized.

3. Loops containing arithmetic IF statements with three different targets, computed or assigned GOTO statements, function or subroutine calls, I/O statements, or that have more than one exit cannot be vectorized.

4. If an 'outer' loop contains a nested loop with an induction variable whose start value or step value varies with iterations, the outer loop will not be vectorized. An induction variable is a variable that is incremented or decremented by the same amount on each iteration of the loop, typically, the DO loop control variable.

5. Loops containing references to variables or arrays equivalenced to other variables or arrays used in the program unit are not vectorized.

6.2.3.6 Recurrence

Please note that, in addition to these limitations, a loop may not be vectorized or may be only partially vectorized if a recurrence (real or apparent) is present. A recurrence is present when an assignment stores a value that is used during a subsequent iteration to compute the value on the right side of the same assignment. For example:

```
      DO 10 I = 2,100
      A(I) = A(I-1)+1
10    CONTINUE
```

Here, on the first iteration *A(2) = A(1)+1*, and on the second iteration *A(3) = A(2) + 1*, using the value of *A(2)* computed on the first iteration. Such a computation is inherently serial, and, hence, cannot be vectorized.

More generally, vectorization is inhibited if two array references are so related that (1) neither could validly be placed first in vectorized code, or (2) the compiler is unable to deduce which to place first. Situations like these are also referred to (somewhat loosely) as recurrences.

For example, the following loop cannot be vectorized is the sign of N is unknown:

```
      DO 10 I = 1,100
      A(I+N) = 1
10
```

```
A(I) = 0
```

If *N* were +1, the value of *A(2)* on termination of the loop would be 0, implying that the assignment to *A(I)* would have to follow the assignment to *A(I+N)*. However, if *N* were -1, the value of *A(2)* on termination would be 1, implying that the assignment to *A(I+N)* would have to follow the assignment to *A(I)*.

The previous example illustrates the most common reason for the compiler failing to vectorize a vectorizable loop—the addition of a loop constant quantity of unknown sign to a subscript. Another frequent cause of apparent recurrences is the use of array references in subscripts. For example,

```
      DO 10 I = 1,100
      A(J(I)) = A(J(I)) + 1
10    CONTINUE
```

This loop is almost certainly vectorizable, but the complier cannot ignore the possibility that elements of the *J* array may be repeated. Therefore, the assignment to *A(J(I))* could produce a value that would be used in computation of its right side on a subsequent iteration, and the compiler must assume that the references to *A(J(I))* are in a recurrence.

The NO_RECURRENCES directive (see Appendix D) can be used to vectorize loops where vectorization would otherwise be prevented by apparent recurrences.

The compiler will vectorize one important special class of recurrences: reductions. In general, a reduction has the form:

$X = X\ op\ Y$

where:

X is a scalar variable (or scalar relative to the loop in question).

Y is any expression not involving *X*, *X* is not assigned or used elsewhere in the loop, and *op* is one of the operators +, -, *, .AND., .OR., .EQV., .NEQV., or one of the intrinsic functions for maximum or minimum. For example, the following loop computes the sum of the first 100 elements of the array *A* with a sum reduction:

```
      SUM = 0
      DO 10 I = 1,100
      SUM = SUM + A(I)
10    CONTINUE
```

The vectorizer sometimes inserts vector temporaries to enable a loop with a recurrence to be partially vectorized. For example, the following loop cannot be vectorized as is:

```
      DO 10 I = 1,N
      A(I) = A(I-1) + B(I)*C(I)
```

```
10  CONTINUE
```

The vectorizer recognizes, however, that the multiplication *B(I)*C(I)* can be vectorized. To do so, it introduces a temporary array (here represented by *T(I)*) and splits the loop into two loops:

```
    DO 10a I = 1,N
    T(I) = B(I)* C(I)
10a CONTINUE
    DO 10b I = 1,N
    A(I) = A(I-1)+T(I)
10b CONTINUE
```

The first loop is then vectorized and a sequential loop is generated for the second loop.

Temporaries are allocated on the stack at time of entry to the loop being partially vectorized and deallocated on exit. Consequently, the space requirements for temporary storage are rarely excessive.

6.2.3.7 Examples

1. Use of a scalar temporary does not inhibit vectorization. The following loop will be fully vectorized.

```
    DO 10 I = 1,N
    X = A(I)+B(I)
    A(I) = C(I)
    Z(I) = X+Y(I)
10  CONTINUE
```

2. A loop containing only a scalar reduction will be fully vectorized. For example,

```
K = 1
DO J = 1,10
K = K+1
END DO
```

is fully vectorized. The loop is replaced by the equivalent code

```
K = 11.
```

6.3 Machine-Dependent Optimization

Machine-dependent optimizations are used to enhance the object code produced by the compiler.

6.3.1 Instruction Scheduling

Instruction scheduling determines an order of instructions which effectively uses the function units on the computer. You have no control over this scheduling. The compiler rearranges the instructions in the program to achieve a high level of concurrent operation. In debug mode, instruction scheduling is performed only within (not between) statements, so that *csd* can correlate instructions with the lines in the original program.

Instruction scheduling on the CONVEX family of supercomputers is a significant optimization. It schedules instructions across numbers of statements instead of in one statement only, often achieving substantial performance improvements. For example,

```
a=b+c
d=e-f
```

How Code from a Naive Compiler Would Look		How Code from CONVEX FORTRAN Looks	
ld.w	b,s0	ld.w	b,s0
ld.w	c,s1	ld.w	c,s1
add.s	s0,s1	ld.w	e,s2
st.w	s1,a	ld.w	f,s3
ld.w	e,s0	add.s	s0,s1
ld.w	f,s1	sub.s	s3,s2
sub.s	s1,s0	st.w	s1,a
st.w	s0,d	st.w	s2,d

In the left example, the subtraction cannot execute until the addition is completed. In the right example, these two operations proceed concurrently (almost).

6.3.2 Span-Dependent Instructions

The compiler generates span-dependent pseudo-instructions to take advantage of the assembler's ability to generate shorter instructions when possible. The assembler defines "span-dependent pseudo-instructions" for conditional and unconditional branches. For example:

is a pseudo-instruction that the assembler converts to a one-word branch (*br*) instruction or a two- or three-word jump (*jmp*) instruction. The shorter instructions require less code space and execute faster, but can only be chosen when *L1* is "close" enough to the instruction, i.e., within about 256 bytes for a branch instruction and about 32,768 bytes for a two-word jump instruction.

6.3.3 Branch Optimization

Many compilers generate branch instructions that branch to the next sequential instruction. Such branches are generated internally by the CONVEX FORTRAN compiler but are removed by 'branch optimization' before the assembly code output is produced.

6.3.4 Register Allocations

Register allocations are an optimization over which you have no control. The register allocations scheme in the CONVEX compiler is different from other machines because of the machine's unique architecture. Most machines try to minimize the number of registers allocated for a given expression; the CONVEX compiler attempts to maximize the number in order to achieve more parallelism.

The only time you need to be aware of this optimization is when you are invoking one of your assembly language routines. The compiler assumes on any call that all of the registers are destroyed; as a result, it saves and restores any that are active. The point is, you do not have to be concerned about what is in the various registers when coding an assembly language routine.

6.3.5 Strength Reduction and the Code Generator

The code generator performs certain strength-reduction operations on instruction-level operations. For example, instead of performing multiplies by a power of 2, the code generator transforms them to shifts.

6.4 Argument Lists

Argument lists (or packets) are a compile-time structure used to pass arguments to subroutines. In certain instances, the compiler builds argument packets at compile time (which is a faster process); at other times, it pushes the arguments physically onto the stack. For information on the stack pointer, see Chapter 5, "FORTRAN Calling Procedures." Those argument packets that can be built at compile time include those which contain constants, variables, expressions, or array elements with constant subscripts. Argument lists containing dummy arguments, subprogram names, and array elements with nonconstant subscripts cannot be built at compile time. In such instances, argument lists must be completed at run-time.

6.5 Using Vector Intrinsic Functions

The calling sequence for vector instrinsics is different from that for other intrinsics and for user subprograms. The user cannot specify the calling of a vector intrinsic. If an intrinsic function is in a vectorized loop, it is called a vector intrinsic; otherwise, it is called a scalar. In vectorized loops, vector intrinsics are faster than the corresponding scalar intrinsics in a loop.

6.6 Memory Allocation Scheme

The memory allocation scheme allows you to assign to a variable in a subprogram, exit the routine, and upon re-entering the subprogram access the value last assigned in the previous call, i.e., values of local variables in a subprogram are retained across calls. All local variables in subprograms are "saved" variables.

6.7 Tree-Height Reduction

Tree-height reduction is best explained by example; consider the following expression:

a+b+c+d+e+f+g+h

You can evaluate this expression in many ways, two of which are shown below:

Method 1 (Unbalanced parenthesization)	**Method 2** (Balanced parenthesization)
(a+(b+(c+(d+(e+(f+(g+h))))))	(((a+b)+(c+d))+((e+f)+(g+h)))

Method 1 requires *g+h* to be evaluated first. The result of that calculation is then used to compute *f+(g+h)* and so on. None of the additions can proceed simultaneously, because each must wait for the result of the addition to its right.

Method 2 allows four additions to execute in parallel: *(a+b)*, *(c+d)*, *(e+f)*, and *(g+h)* can be computed simultaneously, because none of these additions requires the results from any other addition. Furthermore, when the results from these additions become available, the additions *(a+b)+(c+d) and (e+f)+(g+h) can also execute in parallel.*

In general, the time required to evaluate a particular parenthesization is roughly proportional to the "depth" of the expression, i.e., the deepest nesting level of parentheses. This is 6 for the first method, but only 3 for the second.

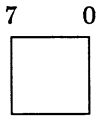
When the compiler has a choice, as in the previous example, of what order in which to evaluate expressions, it chooses the order that yields the least depth and therefore the highest degree of parallelism. (Internally, the compiler represents expressions as "trees," the height of which corresponds to the depth of the expression, and hence the name of the optimization.) This may result in a different numerical result for floating-point operators due to rounding off in the least-significant bits. However, note that FORTRAN requires that the compiler never override an order of evaluation made explicit by the user's own use of parentheses. For example, if you write *a+(b+(c+(d+(e+(f+(g+h))))))*, the compiler generates code to evaluate first *g+h*, then *f+(g+h)*, etc. To get faster code, omit the parentheses.

APPENDIX A
FORTRAN Data Representations

This appendix describes the 12 different data types supported by CONVEX FORTRAN and shows how each is stored in memory. The numbers on top of the figures are the bit ordering; the numbers on the bottom are the byte ordering.

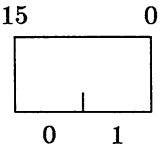
A.1 Logical Representation

LOGICAL*1

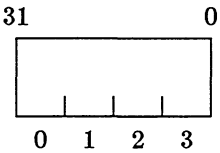


.FALSE. = all 0's
.TRUE. = all 1's

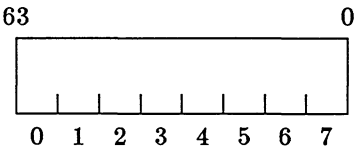
LOGICAL*2



LOGICAL*4



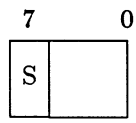
LOGICAL*8



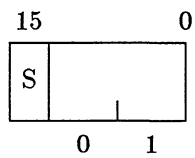
The leftmost byte (8 bits) is always stored FIRST in memory.

A.2 Integer Representation

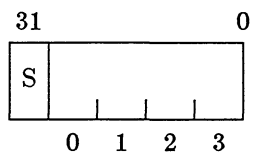
INTEGER*1
BYTE



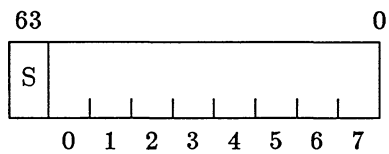
INTEGER*2



INTEGER*4



INTEGER*8



where:

S — Represents the sign bit, which is 0 for nonnegative integers and 1 for negative integers.

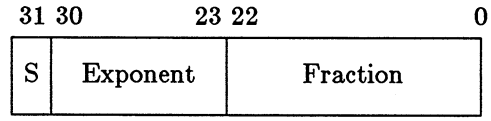
INTEGER — Data types use the two's complement numbering system, which means that a negative number -i is represented as:

(-1) times 2 sup {n - 1} + sum from {i = 0} to {n - 2} 2 sup i

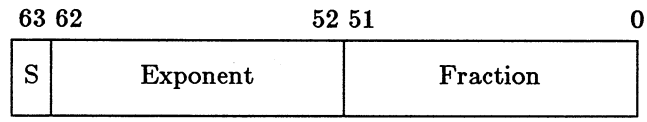
where n is the number of bits in the data type.

A.3 Real Representation

REAL*4



REAL*8



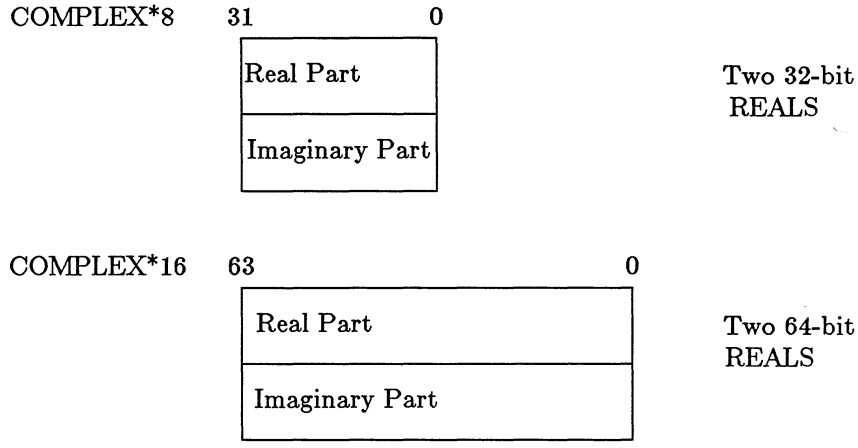
where:

S — Is the sign bit. A binary value of 0 denotes positive; a binary value of 1 denotes negative.

Exponent — Is a binary-biased exponent. The decimal value of the exponent is obtained by evaluating the unsigned binary value of bits <30..23> for REAL*4 and bits <62..52> for REAL*8. Next, 128 is subtracted from the REAL*4 value and 1024 from the REAL*8 value. These values are then used as a power of 2.

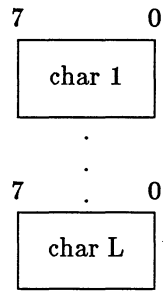
Fraction — Is a fractional value. For REAL*4, an implicit 1 bit is to the left of bit 22; the decimal point is to the left of the implicit 1 bit. For REAL*8, an implicit 1 bit is to the left of bit 51; the decimal point is to the left of the implicit 1 bit.

A.4 Complex Representation



A.5 Character Representation

A character string is stored internally as a sequence of bytes. A zero byte is appended to the end.



A character constant is limited to 4000 characters. Character strings formed at run-time may be of arbitrary length.

A.6 Hollerith Representation

A Hollerith constant is stored internally as a sequence of bytes and is limited to 4000 characters.

APPENDIX B
Compatibility with Fortran 66

B.1 Introduction

The CONVEX Fortran compiler adheres to the American National Standard Fortran-77, X3.9-1978, ISO 1539-1980(E), i.e., the default language interpretations are Fortran-77. However, the compiler can compile Fortran-66 programs. There are five incompatibilities between American National Standard Fortran-77 and Fortran-66, X3.9-1966.

- EXTERNAL statement
- DO loop minimum iteration count
- OPEN statement BLANK keyword default
- OPEN statement STATUS keyword default
- X format edit descriptor

Interpretation of the first two incompatibilities are controlled by the compiler, while the latter three are controlled by the run-time system. If your program uses the OPEN statement, and you want Fortran-66 interpretation rules at run-time, then either include a call to *ioinit* in your main program, or include the library I66 in your link via placing -lI66 in the *fc* command. The X format edit descriptor use must be modified.

To compile a Fortran-66 program, you can modify the program, transforming it into a Fortran-77 program, and/or use the -F66 flag.

1. Use the UNIX command *grep* to identify OPEN statements in which a STATUS keyword is to be added, EXTERNAL statements that must be changed to INTRINSIC statements, and FORMAT statements using the X edit descriptor.

2. Use the -F66 flag or OPTIONS statement to select Fortran-66 language interpretations. The -F66 flag allows for the interpretation of EXTERNAL statements, DO loop minimum iteration counts, and BLANK and STATUS keyword defaults in OPEN. It does not affect the X format edit descriptor. To avoid including the -F66 flag in the *fc* command each time, use the *alias* command:

   alias fc fc -F66

   you can include this format in your .cshrc file, with the $ parameter representing specified files.

B.2 Language Incompatibilities

B.2.1 EXTERNAL Statement

In Fortran-66, the EXTERNAL statement specifies that a symbolic name is the name of either a user-defined external procedure or a Fortran-supplied function. In Fortran-77, two statements accomplish this function:

1. The INTRINSIC statement specifies that the procedure is a Fortran-supplied intrinsic procedure, like SQRT.

2. The EXTERNAL statement specifies that the procedure is user-supplied.

Because of the exact specification of these two procedures, you cannot modify the EXTERNAL statements in your program so that the same source program works with both Fortran-77 and Fortran-66. you must substitute an equivalent statement to encompass the changes:

Fortran 66	Fortran 77
EXTERNAL USER	EXTERNAL USER (no change)
EXTERNAL SQRT	INTRINSIC SQRT
EXTERNAL *SQRT	EXTERNAL SQRT (where SQRT is a user function not the intrinsic for the square root)

B.2.2 DO Loop Minimum Iteration Count

Whereas in Fortran-66 the body of a DO loop is always executed, in Fortran-77 the body of the DO loop is not executed if the end condition of the loop is already satisfied when the DO statement is executed. Therefore, if you wish to run a Fortran-66 program with the regular Fortran-77 compiler, you can either use the -F66 flag, or modify the program's DO statements to ensure a minimum loop count of one; e.g., in Fortran-77, the loop

```
DO 20 J=INIT,LAST
```

is not executed if INIT > LAST, but is executed once in Fortran-66. If this DO statement occurred in a Fortran-66 program, its equivalent Fortran-77 statement is:

```
DO 20 J=INIT,MAX(INIT,LAST)
```

B.2.3 OPEN Statement Keywords

While Fortran-66 does not contain an OPEN statement, it does allow for many implementations based on Fortran-66 which contain an OPEN statement. Both the BLANK and STATUS keywords in OPEN for Fortran-77 differ from the implementations which are utilized under Fortran-66.

B.2.3.1 OPEN Statement BLANK keyword

The BLANK keyword affects the treatment of blanks in numeric input fields read with the D, E, F, G, I, O, and Z field descriptors. In Fortran-77, the OPEN statement BLANK keyword defaults to BLANK='NULL' (which means that blanks in numeric fields are ignored). The Fortran-66 interpretation of blanks in numeric input fields is equivalent to BLANK='ZERO'.

When a logical unit is opened without an explicit OPEN statement, CONVEX Fortran and Fortran-66 both provide a default equivalent to BLANK='ZERO'.

The use of BLANK='NULL' causes embedded and trailing blanks to be ignored and the value converted as if the nonblank characters were right-justified in the field. However, the use of BLANK='ZERO' causes embedded and trailing blanks to be treated as zeros.

If your program treats blanks in numeric input fields as zeros, and you do not want to use -lI66 or ioinit, include BLANK='ZERO' in the OPEN statement.

B.2.3.2 OPEN Statement STATUS Keyword

Whereas the OPEN statement STATUS keyword in Fortran-77 specifies the initial status of the file ('OLD', 'NEW', 'SCRATCH', or 'UNKNOWN'), and its default is 'UNKNOWN', in Fortran-66, where 'STATUS' is called 'TYPE', the default value is 'NEW'.

If your program assumes the default for 'TYPE' is 'NEW', and you do not want to use -lI66 or ioinit, then put 'STATUS' = 'NEW' in the OPEN statement.

B.2.3.3 X Format Edit Descriptor

Whereas the Fortran-66 implementation of the X format edit descriptor writes blanks to and may extend the output record, the Fortran-77 version does not modify character positions that are skipped and does not, as a result, affect the length of the output record. you can achieve the Fortran-77 effect, by changing nX to n(' ').

APPENDIX C
Incompatibilities Between CONVEX and VAX Fortran

CONVEX Fortran does not support certain VAX Fortran extensions:

- INCLUDE statement (The #include preprocessor provides the same function, however.)
- NAMELIST statement
- REAL*16 data type
- %DESCR
- Byte ordering with respect to characters and parameter passing is not VAX compatible.
- Logical values differ in the two compilers. On CONVEX, a logical is true if it is not 0. On VAX, it is true if the low-order bit is 1.
- Numerical differences exist owing to the accuracy of our floating-point representation and the rounding method used. See the *CONVEX Architecture Handbook,* Appendix D, for further information.
- The 'cc...c' form of octal constants is typeless, like the 'cc...c'O form, not of type integer.
- Hollerith constants can be used where a character value is expected.

The product does not support certain VAX Fortran I/O extensions:

- DELETE, UNLOCK statements
- NAMELIST-directed I/O
- Variable format expressions
- Indexed I/O (key-indexed files)
- File sharing
- DEFINEFILE statement

- OPEN keywords (PRINT and SUBMIT values for DISPOSE; USEROPEN; INITIALSIZE; EXTENDSIZE; BLOCKSIZE; BUFFERCOUNT; NOSPANBLOCKS; SHARED; and ORGANIZATION).

- CLOSE keywords (PRINT and SUBMIT values for STATUS)

- Record specifier 'r

APPENDIX D
Hardware Specifications

SPECIFICATIONS FOR C-1 COMPUTER SYSTEM	
Central Processing Unit	
Hardware supported data type: (scalar and vector, byte addressable) INTEGER *1, *2, *4, *8 LOGICAL *1, *2, *4, *8 REAL *4, *8 CHARACTER	
Processor cycle time:	
Major	100 nanoseconds
Minor	50 nanoseconds
Microinstruction Word Size: ASU	4096 x 80 (RAM)
Microinstruction Word Size: VPU	
Functional unit	
Add/Logical/Shift	4096 x 48 (RAM)
Mult/Divide	4096 x 48 (RAM)
Load/Store/Compress/Merge	4096 x 56 (RAM)
Virtual Address Space	4 Gigabytes
Maximum User Program	2 Gigabytes
Page Size	4096 bytes
Referenced/Modified Bits	In hardware, pair per pageframe
Hardware Logic Family	8000 Gate CMOS VLSI Gate Arrays, and high speed advanced Schottky logic
Memory System	
Maximum Memory Size	16 MWords (128 megabytes)
I/O to Main Memory Bandwidth	80 megabytes/second
CPU to memory Bandwidth	80 megabytes/second
Input/Output System	
I/O bandwidth per IOP	8 megabytes/second
Maximum Number of IOPs	5
Multibus Chassis per IOP	4
Multibus Bandwidth	4 megabytes/second

Environmental(processor cabinet only)	
Hardware Form Factor	
Central Processor Board	19.3 inch x 20 inch
MULTIBUS Board	6.75 inch x 12 inch
Power Consumption	
Standard	3200 Watts
Maximum	4500 Watts
Cooling	Forced air drawn from front, exhausted to the rear
AC Power Requirements	208 VAC, 60 HZ, 3 Phase, 15 amps
Operating Temperature Range	15-32 degrees centigrade
Voltage Tolerance	184-264 VAC, phase to phase
Frequency Tolerance	47-63 HZ
Cabinet Size (with industrial skins)	
Width	25.09 inches (19 inch RETMA rack)
Height	62.5 inches
Depth	39.5 inches
Cabinet Weight	500 pounds (estimated)
FCC Compliance	Class A, Subpart J of 155555
Software	
Operating System	CONVEX UNIX
Languages	Fortran, C
Networking	Local Area Networking - TCP/IP

DISC SPECIFICATION	
DKD-101 Functional Specifications	
Storage Capacity	
Unformatted	474 Megabytes
Formatted	414 Megabytes
Disks	6
Heads	20(2 per surface)
Bytes per Track	28,160
Tracks per Cylinder	20
Cylinders	842
Positioning Time	
Track to Track	5 milliseconds
Average	18 milliseconds
Maximum	35 milliseconds
Average Latency	7.5 milliseconds
Rotational Speed	3,961 rotations/minute +/- 2%
Recording Density	12,790 bits/inch
Track Density	880 tracks/inch
Data Transfer Rate	1.859 megabytes/second
DKD-101 Physical Specifications	
Dimensions	
Height	10.4 inches
Width	19.0 inches
Depth	26.0 inches
Weight	140 pounds
Power Requirements	120 VAC +/- 10%, 60HZ +/- 2HZ, 4.6A
Ambient Temperature	
Operating	50-140 Farenheight
Non-Operating	-40-140 Farenheight
Relative Humidity	
Operating	20 to 80%(non-condensation)
Non-operating	0.5 to 95% (non-condensation)

MTD-001, 002 SUBSYSTEM SPECIFICATIONS		
Model	**MTD-001**	**MTD-002**
PERFORMANCE		
Densities(bpi)	1600/6250	800/1600/6250
Speed(ips)	50	125
Rewind time 2400ft	2.5 mins	1 min
Transfer Rate		
800 bpi	-	100KB
1600 bpi	80KB	200KB
6250 bpi	312KB	768KB
Threading	Manual load/auto thread	Auto load/auto thread
Access Time-Nominal	5.6 Milliseconds	1.2 Milliseconds
PHYSICAL CHARACTERISTICS:		
Size - Height	24.5"	60"(free standing)
Depth	18.0"	24"
Width	19.0"	19.0"
Weight	114 lbs	320 lbs
POWER REQUIREMENTS:		
60 HZ voltage	120 VAC	110/120 VAC +10%-15%
Input current(max)	8 amps	20 amps
Power(nominal)	360 watts	1830 watts
ENVIRONMENT:		
Temperature(operation)	60-90 F	60-90 F
Humidity(operating)	20% - 80%	20% - 90%

PRT-101 SPECIFICATIONS	
FUNCTION	PRT-101
PRINTING RATE	600 lines per minute (LPM) of normal text 465 LPM for underlines or for lower case characters with descenders. 320 LPM for double high characters
PLOTTING RATE	33 1/3 inches per minute
PAPER FEED SPEED	16.5 ms 1/6 inch step 12.5 ms 1/8 inch step 16 inches per second slew
MATRIX	9 horizontal and 7 vertical 9 x 9 for lower case character with descenders
HORIZONTAL FORMAT	132 characters at 10 characters per inch
PLOTMODE	60 dots per inch horizontal 72 dots per inch vertical
DIMENSIONS	Height 16.5" Width 30" Depth 24.25" Weight 185 lbs.
POWER CONSUMPTION	200 Watts standby 450 Watts nominal 800 Watts max.
PRC-001 SPECIFICATIONS	
DIMENSIONS	6.75 inches x 12.00 inches One MULTIBUS slot
Parallel data transfer rate	1.6 MBytes per sec.

APPENDIX E
Glossary

Access Mode

Any of the five processor access modes in which software executes. On the CONVEX system, processor access modes are: (in order from most to least privileged and protected): kernel (mode 0), executive (mode 1), supervisor (mode 2), agent (mode 3) and user (mode 4). The operating system uses access modes to define protection levels for software executing in the context of a process.

Accumulator

A hardware register. This register contains the results of arithmetic and logical operations.

Address

User assigned number used by the operating system to identify a storage location.

Addressing mode

How the effective address of an instruction operand is calculated using the general registers.

Address space

Address space, either physical or virtual, available to a process.

Address Translation Fault

An exception that results from a PTE violation or a non-resident page.

Address Translation Unit

The address translation unit (ATU) translates logical addresses to physical addresses and stores them in a cache. Thus the ATU is an address cache which accelerates the generation of physical addresses.

Architecture

The conceptual structure and functional behavior of the system.

Argument Pointer

An address register specifically dedicated to point to the subroutine argument portion of a program. This program portion can either be in the stack or in part of logical memory pre-allocated by the compiler.

Arrays

An ordered structure of operands of the same data type. The structure of an array is defined as: length, rank or dimension, stride, and data type.

Base-level interrupt

An interrupt which occurs when the kernel stack is the process stack. A base-level interrupt is thus an interrupt which occurs when no other interrupts are pending or currently being processed.

Bit

A binary digit.

Bit complement

Exchanging 0's and 1's in the binary representation of a number (also known as 1's complement).

Bootstrap

The procedure by which a program is initiated the first time. Typically a bootstrap is performed when power is first applied to the processor.

Branch

A class of instructions used to transfer control of a program, specifically relative to the Program Counter.

Breakpoint

An instruction which aids in the debugging of a program. In particular, a breakpoint is a particular location in a program that one would desire to determine the various values of programmer-defined variables.

Byte (b)

A byte is a number of contiguous bits starting on an addressable byte boundary. In CONVEX machines, a byte is eight bits.

C

The systems programming language of the UNIX operating system.

Cache

(See Logical, Physical, Instruction).

Cache memory

A small, high-speed memory placed between main memory and the processor and transparent to the user. CONVEX computers contain many separate caches.

Cache purge

The act of invalidating or removing entries in a cache memory.

Central processing unit

The central processing unit is the portion of a CONVEX machine which recognizes and executes the instruction set.

Chaining

Chaining is the ability to overlap vector operations in the central processing unit. For instance, in the case of a vector load followed by a vector add, the add may be started as soon as the first operands are available rather than waiting for the load to complete.

Compiler

Software tool used to compile a high-level language (e.g., Fortran) into assembly code.

Context (processor)

The entire, current state of the machine associated with the executing process.

Data type

The way in which bits are grouped and interpreted. For processor instructions, the data type identifies the size of the operand and the significance of the bits in the operand.

Destination

The operand specified in an instruction which receives the result of the operation.

Displacement

A derived 32-bit value used to indicate the distance in bytes that the referenced datum is relative to some base value. This base value can either be 0 or the contents of an address register. Please note that 16-bit displacement values are sign extended to 32 bits.

Double (d)

A double precision floating point number, stored in 64 bits.

Exception

An exception is a hardware-detected event which disrupts the running of a program, process, or system.

Fault

An exception, which, while halting the instruction, leaves the registers and memory in a consistent state. The instruction can often resume its course when the cause of the fault is corrected.

Flag

A 1-bit operand that is generally used to indicate the results of an operation. The results are in the form true or false.

Floating point

A numerical representation. A floating point operand has a sign (positive or negative, an exponent, and a fraction). The fraction is a fractional representation. The exponent is the value used to produce a power of two scale factor that is subsequently used to multiply the fractions to produce an unsigned value.

FORTRAN

High-level software language mainly used for scientific applications.

Fraction

A part of a floating point number. The fraction is the unsigned fractional part that denotes the magnitude of the operand.

Frame

See Page Frame, Stack Frame

Function unit

A function unit is a part of the central processing unit (CPU) which performs a set of operations on quantities stored in registers.

Gate array

A structure that is used by the ring protection mechanism. The gate array defines the entry points from a lower privileged ring to a higher privileged ring.

Gather

Loading a vector register using another vector of indices instruction. See the ldvi instruction.

Guard bit

A bit to the right (the least significant bit positions) of a floating point fraction. The guard bit is used in intermediate calculations using floating point operands.

Halfword (h)

Two bytes (16 bits)

Icache

See Instruction Cache.

Immediates

Operands which are contained within the instruction stream.

Indexing

The process of adding a displacement to the contents of an address register.

Indirection

The process of obtaining the address of an operand by first referencing a word contained within memory.

Instruction

Used by the programmer to direct operations on the systems' register set and memory.

Instruction cache (ICACHE)

The I-Cache contains the most recently accessed instructions. The I-Cache accelerates the decoding of instructions. This permits the simultaneous decoding on one instruction with the execution of another instruction.

Interrupt

An occurrence other than an exception which changes the normal flow of instruction execution. An interrupt originates from hardware, such as an I/O device.

Interval timer

A privileged register. The interval timer is used to generate an interrupt based on the passage of a period of time.

Kernel

A part of the operating system that resides in ring 0. The kernel typically manages process creation and deletion, scheduling, and other high level, system wide features.

Linker

A software tool. The linker "links" together separate software modules into one monolithic module.

Loads

A class of instructions which move data from memory to a register.

Locality of reference

An attribute of a memory reference pattern. Locality of reference refers to the likelihood of an address of a memory reference being numerically close to a recent memory reference address, or the likelihood of a subsequent memory reference being identical to a previous memory reference within a given period of time.

Logical address

Logical address space is that space seen by the application programmer.

Logical cache

The logical cache is a cache that is accessed with logical addresses for fast retrieval of data. It resides in the central processing unit.

Logical memory

Logical or virtual memory is that memory seen by the programmer. The logical memory of a CONVEX computer is 4 Gigabytes.

Longword (l)

Eight bytes (64 bits), the largest integer data type directly supported by hardware in the CONVEX-1.

LSI (Language Specific Information)

The area in the stack that is created as part of a subroutine call. It is language dependent and may be zero.

Machine exceptions

Machine exceptions include fatal errors in the system which cannot be handled by the operating system. (See Exception).

Main memory

See Physical Memory.

Maskable interrupt

An interrupt that is masked out. That is, an interrupt that the operating system wishes, at this time, not to respond to.

Memory management

The hardware and software features which control page mapping and protection.

Microcode

A control program that resides within the central processing unit. Microcode also refers to firmware, providing the necessary control that maps assembly language instructions onto processor hardware.

Modified bit

A bit within the central processing unit. The modified bit records all valid write references to pageframes. The modified bit is used by the operating system for memory management.

Negate

An instruction which performs a 2's complement.

Normalization

The process of left shifting a fraction until the leading bit is a 1.

Opcode

The code or sequence of bits in an instruction which determines the operation to be performed.

Operand

A register or memory location referenced by an instruction.

Orthogonality

A characteristic that pertains to the relationship of instructions and the operands they manipulate. An instruction set is orthogonal if one can change one property without having to change other related properties.

Packets

A group of related items. A packet may refer to the subroutine arguments or to a group of bytes that is transmitted over a network.

Page

A page is the unit of logical memory controlled by the memory management algorithms. In CONVEX-1, a page is 4 K (4096) contiguous bytes.

Pagefault

An exception caused by a reference to a valid non-existent page.

Page Frame

A page frame is the unit of physical (main) memory in which pages are placed. Associated with each pageframe are referenced and modified bits to aid in memory management.

Page Table Entry (PTE)

An entry in a page table. A PTE is a word. A PTE contains various flags and fields that are used in the translation of logical to physical addresses. Address translation uses two levels of page table indexing. The first level page table is referenced using bits 28 through 22 of a logical address. This is called the Index.1 field. The second level page table is referenced using bits 21 through 12 of a logical address. This is called the Index.2 field.

Physical address

Hardware-identified address in physical (main) memory consisting of a page frame number and the number of a byte within the page.

Physical cache

The physical cache provides rapid access to recently used physical memory data items.

Pipelining

A technique used to construct high performance processors. Pipelining provides a means by which multiple operations occur concurrently.

Porting

Moving software from one type of machine to another.

Priority

An ordering of events. Priority is applied to protection levels as well as I/O interrupt levels.

Privileged instruction

An instruction used by the operating system or privileged systems programs. It must execute in ring 0, or an exception occurs.

Process

A process is the fundamental unit of program which is managed by the job scheduler.

Process exceptions

Process exceptions belong to the currently running process and may be handled with an exception handler in that process. The exception handler is in the current ring of execution. (See Exception).

Protection

A mechanism provided by hardware and software. Protection is used to ensure that one user is protected from another user or to ensure that a user does not perform an unsafe computation.

Processor

A word that contains control flags. The PSW is used to control and indicate the state of various computations and sequences within the processor.

Push

The act of storing an operand on the stack.

Queue

A data structure in which entries are made at one end and deletions at the other. Often referred to as first-in first-out or FIFO.

Quotient

The result of a division operation.

Read

A memory operation in which the contents of a memory location are accessed and passed to another part of the machine.

Recursion

An arithmetic operation that uses the output of a calculation as the input of the same calculation.

Reduced Instruction Set Computer (RISC)

An architectural concept that applies to the definition of the instruction set of a processor. A RISC instruction set is an orthogonal instruction set that is easy to decode in hardware and for which a compiler can generate highly optimized code.

Reductions

An arithmetic operation that performs a transformation on an array that produces a scalar result.

Register

A hardware entity that is used to contain addresses, operands, and status.

Reservation

Reservation is the process of managing the various function units in the central processing unit. A reservation table is used to record the current status and availability of the function units.

Reset

The process of establishing a known state in a machine register.

Rings

A ring is the unit of logical memory used for protection purposes. There are five rings in CONVEX machines: four for system level usage and one for users. The system rings (Ring0-Ring3) each correspond to one Segment of logical memory, while the user ring (Ring4) contains four segments.

Ring Maximization

Ring maximization is the mechanism used to enforce protection in the logical address space.

Round bit

One of the two guard bits used in the intermediate representation of a floating point number.

Rounding

The process of transforming the intermediate representation of a floating point number to the memory representation. Unbiased rounding uses the round, guard, and sticky bits to determine the exact nature of this transformation. Truncation (as used in converting floating point to fixed point integer) does not use the round, guard, or sticky bits.

Runtime

A software module. A runtime is a software module that is referenced as a procedure. A runtime represents a required function that is not directly supported by the hardware, but it is required by the software.

Scatter

Storing a vector register using another vector of indices. See the stvi instruction.

Segmented ALU

A logic design technique that permits multiple arithmetic operations of the same type to be pipelined.

Segment

The segment is the basic partition of the logical memory space. A segment is 512 megabytes.

Segment descriptor register

Each segment of virtual memory has a segment descriptor register associated with it. Each SDR contains information pertinent to the access and mapping of virtual addresses.

Shift

A class of instructions used to shift the contents of a register right or left.

Single (s)

A single precision floating point number stored in 32 bits.

Source

A register or memory location used as an input to a CONVEX instruction.

Spatial reference

An attribute of a memory reference pattern. Spatial reference pertains to the likelihood of a subsequent memory reference address being numerically close to a previous address.

Stack

A data structure in which the last item entered is the first to be removed. Also referred to as last-in first-out (LIFO). In particular, stacks are used by the Call and Return instructions.

Sticky bit

A bit used in the intermediate calculations of floating point operands. The sticky bit remembers if any binary 1's were shifted out during an alignment or partial product operation.

Stores

A class of instructions used to move the contents of registers to memory.

Subroutine

A software module. A subroutine is a frequently used program that is called from various places in a program.

System exceptions

System exceptions cannot be handled by the current process; they require intervention by the kernel executing in ring 0. (See Exception).

Trace of instruction execution

The process of tracking the execution of every instruction of a program.

Trap

An out of sequence branch due to the occurrence of an abnormal condition. Typically, this condition is a result of unexpected arithmetic results. (See Exception.)

Trojan Horse Pointer

The Trojan Horse Pointer is an addresss that is passed from one ring to another as part of a system call. In particular, this passed pointer references the more privileged ring as contrasted to the less privileged ring. This is unexpected and undesirable.

True Zero

A floating point number with zero sign bit, zero exponent, and zero fraction.

Unbiased rounding

The process of interpreting the round, guard, and sticky bits. Unbiased rounding, as contrasted to biased rounding, rounds to even in the event that the intermediate floating point result is exactly midway between two floating point representations.

UNIX

An operating system.

Unsigned

A value that is always positive.

Valid bit

A bit used in the control of caches. The valid bit is used to determine if a cache entry contains an entry that can be used.

Valid reference

A valid reference meets two requirements: first, the PTE must be valid (bit 31=1), and second, the type of access being made (Read, Write, or Execute) must be allowed by the appropriate protection bit (bits <3..1> of the PTE).

Vector

An array with one dimension.

Virtual address space

See Logical Address Space.

Word

Four bytes (32 bits)--the fundamental width of items in the CONVEX family of computers.

Working set

That portion of a user's program that is currently in physical memory. Typically the working set is much smaller that the user program.

Write

A memory operation in which a memory location is updated with new data.

Zero

In floating point number representations, zero is represented by a zero sign bit and zero exponent.

NOTES

NOTES

NOTES

NOTES

NOTES

CONVEX-1 System Overview

Reader's Forum

You are invited to submit your comments concerning the clarity and service of this manual. Constructive critical comments are most welcome and will help us continue in our efforts to generate quality customer documentation. (Please list the page number for questions, comments, and errors):

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

From:

Name_____ Title_____

Company_____ Date_____

Address_____

FOR ADDITIONAL INFORMATION/DOCUMENTATION:

Direct Mail Orders to:

CONVEX Computer Corporation
Marketing Department
701 Plano Road
Richardson, Texas 75081

SALES OFFICE DIRECTORY

CORPORATE HEADQUARTERS
701 N. Plano Road
Richardson, TX 75081
(214) 952-0200

SOUTHWESTERN AREA OFFICE
701 N. Plano Road
Richardson, TX 75081
(214) 952-0229

WESTERN AREA OFFICE
2099 Gateway Place
Suite 320
San Jose, CA 95110
(408) 275-0844

EASTERN AREA OFFICE
7474 Greenway Center Drive
Suite 450
Greenbelt, MD 20770
(301) 345-2400

CENTRAL AREA OFFICE
5051 Heritage Lane
Fenton, MI 48430
(313) 632-6359