CORPORATE PROFILE

Digital Equipment Corporation designs, manufactures, sells and services computers and associated peripheral equipment, and related software and supplies. The Company's products are used world-wide in a wide variety of applications and programs, including scientific research, computation, communications, education, data analysis, industrial control, timesharing, commercial data processing, word processing, health care, instrumentation, engineering and simulation.
Copyright® 1980 Digital Equipment Corporation.
All Rights Reserved.

Digital Equipment Corporation makes no representation that the inter-
connection of its products in the manner described herein will not
infringe on existing or future patent rights, nor do the descriptions
contained herein imply the granting of license to make, use, or sell
equipment constructed in accordance with this description.

The information in this document is subject to change without notice
and should not be construed as a commitment by Digital Equipment
Corporation. Digital Equipment Corporation assumes no responsibili-
ity for any errors that may appear in this manual.

DEC, DECnet, DECSYSTEM-10, DECSYSTEM-20, DECtape
DECUS, DECwriter, DIBOL, Digital logo, IAS, MASSBUS, OMNIBUS
PDP, PDT, RSTS, RSX, SBI, UNIBUS, VAX, VMS, VT
are trademarks of
Digital Equipment Corporation

This handbook was designed, produced, and typeset
by DIGITAL's New Products Marketing
using an in-house text-processing system
operating on a DECSYSTEM-20.
## CONTENTS

### PART 1 INTRODUCTION

<table>
<thead>
<tr>
<th>Chapter 1</th>
<th>Introduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Introduction</td>
<td>1</td>
</tr>
<tr>
<td>Architecture Overview</td>
<td>5</td>
</tr>
<tr>
<td>Software Overview</td>
<td>6</td>
</tr>
<tr>
<td>Hardware Overview</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 2</th>
<th>Process Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Definition</td>
<td>21</td>
</tr>
<tr>
<td>Process Context</td>
<td>21</td>
</tr>
<tr>
<td>Asynchronous System Traps (AST)</td>
<td>25</td>
</tr>
<tr>
<td>Process Structure Interrupts</td>
<td>26</td>
</tr>
<tr>
<td>Process Structure Instructions</td>
<td>26</td>
</tr>
<tr>
<td>Usage Example</td>
<td>31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 3</th>
<th>Exceptions and Interrupts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>33</td>
</tr>
<tr>
<td>Interrupts</td>
<td>35</td>
</tr>
<tr>
<td>Serious System Failures</td>
<td>39</td>
</tr>
<tr>
<td>Arithmetic Exceptions</td>
<td>41</td>
</tr>
<tr>
<td>System Control Block</td>
<td>44</td>
</tr>
<tr>
<td>Stacks</td>
<td>48</td>
</tr>
<tr>
<td>Serialization of Exceptions and Interruptions</td>
<td>51</td>
</tr>
<tr>
<td>Initiate Exception or Interrupt</td>
<td>52</td>
</tr>
<tr>
<td>Return from Exception or Interrupt (REI)</td>
<td>55</td>
</tr>
<tr>
<td>Differences Between the VAX-11/750 and the VAX-11/780</td>
<td>57</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 4</th>
<th>Memory Management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>59</td>
</tr>
<tr>
<td>Virtual Address Space</td>
<td>60</td>
</tr>
<tr>
<td>Access Control</td>
<td>63</td>
</tr>
<tr>
<td>Address Translation</td>
<td>65</td>
</tr>
<tr>
<td>System Space Address Translation</td>
<td>67</td>
</tr>
<tr>
<td>Process Space Address Translation</td>
<td>69</td>
</tr>
<tr>
<td>Memory Management Control</td>
<td>74</td>
</tr>
<tr>
<td>Faults and Parameters</td>
<td>75</td>
</tr>
<tr>
<td>Privileged Services and Argument Validation</td>
<td>76</td>
</tr>
<tr>
<td>Size of System Page Table</td>
<td>77</td>
</tr>
<tr>
<td>Sharing</td>
<td>78</td>
</tr>
</tbody>
</table>
CHAPTER 13  RELIABILITY, AVAILABILITY, AND MAINTAINABILITY FEATURES
INTRODUCTION ................................................................. 223
DESCRIPTION OF RELIABILITY AND MAINTAINABILITY FEATURES ................................................ 226

PART 3 VAX-11/780

CHAPTER 14  CONSOLE SUBSYSTEM
INTRODUCTION ................................................................. 239
CONSOLE INTERFACE BOARD ........................................... 240
CONSOLE BUS STRUCTURE .............................................. 242
CONSOLE/VAX-11 INTERACTION ........................................ 243
READ-ONLY MEMORY (ROM) ............................................... 245
THE CONSOLE COMMAND LANGUAGE .................................. 245
CONSOLE ERROR MESSAGES .............................................. 249

CHAPTER 15  CENTRAL PROCESSOR
INTRODUCTION ................................................................. 255
HARDWARE ELEMENTS .................................................... 256
PROCESSOR OPERATION ................................................... 259
USER PROGRAMMING CONCEPTS ........................................ 263
USER PROGRAMMING ENVIRONMENT ................................... 267
SYSTEM PROGRAMMING CONCEPTS ..................................... 274
SYSTEM PROGRAMMING ENVIRONMENT ................................ 276

CHAPTER 16  SYNCHRONOUS BACKPLANE INTERCONNECT
INTRODUCTION ................................................................. 289
SBI STRUCTURE .............................................................. 290
SBI THROUGHPUT ........................................................... 309

CHAPTER 17  MAIN MEMORY SUBSYSTEM
INTRODUCTION ................................................................. 311
MEMORY CONTROLLER ..................................................... 311
BASIC MEMORY OPERATIONS ............................................ 313
INTERLOCK CYCLES ........................................................ 316
ERROR CHECKING AND CORRECTION (ECC) .......................... 317
MEMORY CONFIGURATION REGISTERS ................................. 317
MEMORY INTERLEAVING ................................................... 323
ROM BOOTSTRAP ........................................................... 323
BATTERY BACKUP .......................................................... 324
VAX is DIGITAL's family of 32-bit word length minicomputers. With the introduction of the newest family member, the VAX-11/750, the power of VAX systems is now available to a broader range of users and applications. VAX systems typically fill the needs of users where traditional minicomputers cannot meet the processing requirements of the application, but large dedicated mainframe systems cannot meet the economic justifications.

Both VAX (Virtual Address eXtension) processors are completely software compatible, sharing the same powerful VAX/VMS (Virtual Memory System) operating system. VAX systems are characterized by a large, flexible instruction set, 32-bit capability, byte addressability, stack orientation, and a highly efficient page-oriented memory management scheme. The recently enhanced VAX/VMS operating system is designed to complement the VAX hardware and, in fact, was designed in conjunction with the VAX architecture. VAX/VMS encompasses a highly sensitive scheduling algorithm, extensive record and file management capabilities, and virtual memory features achieved by an extremely efficient paging algorithm.

VAX systems are general purpose in nature, with the inherent capability to deal with a multitude of user environments. Designed to optimize throughput, either system enables enormous amounts of data to flow through it swiftly and unobstructedly. Data transfers are accomplished via a 32-bit high speed bus structure. This hardware mechanism ties the system components together by providing a common point of interface including the communications protocol. The bus structure interconnects the central processor, main memory, the UNIBUS subsystem, and a mass storage subsystem comprising a maximum of 4.3 billion bytes. VAX supports a 32-bit word architecture, thereby establishing a virtual address space of $2^{32}$ or 4.3 billion bytes for user application. The VAX instruction set consists of 248 instructions including general purpose, special function, commercial, and floating point.

The VAX/VMS operating system is flexible in supporting many user environments such as time-critical, interactive program development, and batch, either concurrently, independently, or in any combination.

The VAX handbook set is presented in three books:

- The VAX Architecture Handbook introduces VAX system architecture, addressing modes, and the native mode instruction set.
The VAX Software Handbook introduces the VAX/VMS virtual memory operating system, its operation, hardware interaction, data structures, features, and capabilities.

This book, VAX Hardware Handbook, introduces the VAX hardware elements, including the central processor units, intelligent console subsystems, MASSBUS and UNIBUS subsystems, main memory, and memory management.

Part 1 of this book describes the VAX family of computers in general, certain architectural characteristics, and functions independent of processor type. Part 2 contains information pertinent to the new VAX processor, the VAX-11/750. Part 3 describes the VAX-11/780 and Part 4 contains the appendices, glossary and index.
PART 1

INTRODUCTION
CHAPTER 1
INTRODUCTION

SYSTEM INTRODUCTION
The VAX series is DIGITAL's family of 32-bit minicomputer systems. The newest member of the family is the VAX-11/750. Like its companion processor, the VAX-11/780, the VAX-11/750 was designed to meet the needs of many users with large data bases and increased processing needs. VAX systems are high performance multiprogramming computer systems which implement a 32-bit architecture, efficient paging memory management, and a virtual memory operating system to provide excellent applications performance and essentially unlimited program address space. DIGITAL's VAX-11/780 has already set new standards for 32-bit computer systems in a wide variety of applications. The VAX-11/750 continues this technological leadership and incorporates many innovations designed to increase performance further and to reduce the overall cost of ownership. The first 32-bit minicomputer to be implemented primarily in custom bipolar LSI Schottky logic (designed entirely by DIGITAL engineers), the VAX-11/750 is a technological breakthrough in the 32-bit minicomputer market. In fact, thoughtful design and state-of-the-art technology combine to give the VAX-11/750 approximately 60% of the performance of DIGITAL's VAX-11/780 at less than 40% the price!

Both VAX systems are multiuser systems for both program development and application execution. For large projects which require a range of CPU family members, the VAX-11/750 is an ideal companion to the VAX-11/780. In addition, the VAX-11/750 provides an attractive alternative to mainframe upgrading in a timesharing environment when the number of users must increase. This capability to offload mainframes offers both faster response times and reduced cost of service.

The VAX-11/750 and the VAX-11/780 are priority-scheduled, event-driven systems: the assigned priority and activities of the processes in the system determine the level of service they receive. Time-critical jobs receive service according to their priority and ability to execute, while the system manages allocation of central processing unit (CPU) time and memory residency for normal executing processes. The following paragraphs describe some of the exciting features of the VAX series and illustrate how they reduce the cost of ownership while providing increased performance and ease of use.

1
Introduction

Application Performance — One reason VAX systems perform so impressively is that the hardware and software were designed to complement each other. Hardware implementation (DIGITAL-designed state-of-the-art gate array devices in the case of the VAX-11/750 central processing unit) combined with the VAX/VMS operating system, 32-bit addressing and 4 billion byte virtual memory, memory cache, address translation buffer, prefetch instruction buffer, and the powerful VAX instruction set, give VAX systems their impressive performance.

The systems use an optimized FORTRAN along with a powerful micro-coded instruction set to increase the speed of program and operating system functions dramatically. The majority of the context switching operation is performed in one instruction; and VAX’s polynomial evaluation operation can do in one instruction what takes 17 instructions to do on "competitive" 32-bit machines. The processor’s variable-length instruction set and variety of data types, including decimal, character string, and floating point, promote high bit efficiency.

This impressive CPU power and throughput, plus the high performance-to-cost ratio, make VAX systems ideal for interactive applications. Yet the high computational ability and large program size mean VAX can handle tough real-time applications as well. Furthermore, the VAX/VMS operating system provides extensive facilities for good batch performance—including job control, multistream, spooled input and output, operator control, conditional command branching, and accounting functions. A choice of options like additional physical memory, user control store, and additional peripheral equipment interfaces, allow even greater flexibility in configuring systems to optimize performance for specific applications.

Ease of Use — The VAX systems are user-oriented systems designed for easy operation. The DIGITAL Command Language (DCL) interface used by VAX/VMS is easy to learn and is suitable for both interactive and batch environments. Because all VAX family members implement a common architecture and run the same VAX/VMS virtual memory operating system, they are 100% software compatible. This means that users with applications software already developed for the VAX-11/780 can run the same code on the VAX-11/750. Because both VAX systems use the same instruction set, it also means that users need not learn a new series of instructions to take full advantage of the newer VAX-11/750’s capabilities. In addition, VAX-11/750 users can take advantage of the maturity of the VAX/VMS operating system, proven in thousands of VAX-11/780 installations. VAX/VMS provides extensive system management facilities, giving system managers and
operators the tools necessary to control the system configuration and the operations of system users for maximum efficiency. With the complete set of VAX/VMS development tools, file system features, and other software packages, VAX family computers are extremely easy to use and require far less debugging time. The VAX family processors also implement a PDP-11 Compatibility Mode which recognizes almost all PDP-11 instructions. This allows users to execute code written for the PDP-11 with few modifications. Users will also appreciate the fact that VAX uses a single operating system, with extensive HELP commands and complete multiuser security.

The VAX Console Subsystem also contributes to ease of use. A separate console terminal replaces the traditional toggle switches and lights; and a thoughtfully designed console command language lets the user perform operations such as EXAMINEs and DEPOSITs, or boot the system, using simple commands. This console terminal also provides a hard copy record for complete documentation of console transactions. Furthermore, switches on the front panel of the CPU can be set up to restart a currently executing program or reboot the system automatically, with no operator intervention, in the event of a power failure or system crash. This feature makes VAX easy to use, and contributes significantly to system availability for user applications.

Easy Installation and Maintenance — VAX systems are equally easy to install and maintain. A variety of system configurations is available so customers can purchase exactly what is required; however, the system is so tunable and adaptable that additional peripherals and options can be interfaced at any time. For example, an option like the user control store can easily be inserted into a CPU by Field Service. Options and power supplies may also be replaced in a few minutes, in case of hardware failures. In addition, customers may choose from a wide variety of peripherals and packaging options so that a VAX system may be configured to suit most requirements—whether the site is an office, a laboratory, or an industrial setting.

Once the system is installed, extensive reliability, availability, and maintainability features in both the hardware and the software ensure data integrity and increase system uptime. Such features as ECC (error-correcting code) memory, online error logging, and a complete range of online and stand-alone diagnostics verify system integrity and ensure that it continues to operate properly. Furthermore, a remote diagnosis option may be installed with the customer's permission to allow dial-up to the DIGITAL Diagnostic Center for remote diagnosis of hardware and software failures. This option can substan-
Introduction

tially lower monthly maintenance charges to the customer while increasing system uptime.

The console subsystem also increases system maintainability and availability. Should the power, hardware, or software fail, the VAX console front panel switches can be set up to restart the process which was executing at the time of the failure or to reboot the system automatically—even in an unattended environment. Users will appreciate the convenience of this automatic recovery feature. Furthermore, the integral mass storage medium, a TU58 tape cartridge on the VAX-11/750 and an RX01 floppy disk system on the VAX-11/780, can be used for loading microdiagnostics and software updates, thereby providing a medium which is convenient, reliable, and cost-effective.

Rapid Development — VAX systems are designed to facilitate rapid, low cost applications development. Implementation is expedited not only by the reliability of the hardware and maturity of the software, but also because VAX is an entire family of computer systems. With the VAX/VMS operating system (identical on all VAX systems), PDP-11 Compatatability Mode, and ANSI standard languages, users will find software development an easy task, and in many cases can use existing routines with little or no modification. Furthermore, DIGITAL's extensive educational services are available to train users and to assist in exploiting the enormous capabilities of VAX systems.

Sound Long-Term Investment — The many excellent features of the VAX series described above, and many other features which will be described in detail later in this book, make VAX a sound long-term investment. The new VAX-11/750 system is a reflection of DIGITAL's ongoing commitment to the VAX family of 32-bit minicomputers and further proof that the VAX family exemplifies the architecture of the 1980s. Furthermore, the wide range of systems possible with the VAX-11/750, and VAX-11/780, ensures that these systems can be tailored to individual application requirements and can be easily reconfigured if those needs should change in the future—an important consideration for customers involved in long-term projects and implementations. VAX systems have large physical memory (up to 2 Mb on the VAX-11/750 and 8 Mb on the VAX-11/780), expandable I/O structure (including UNIBUSs for standard peripherals, MASSBUSs for high speed peripherals, and a special, very high performance parallel interface for VAX-11/780 systems), and 4 billion byte virtual memory, allowing users ample room for growth within present or future applications. In addition, users can be confident that they will receive continuing DIGITAL support—on a worldwide basis—for Field Service, software development, and educational requirements.
Introduction

The following sections in this chapter will introduce the reader to a variety of VAX family architectural and software features, as well as to many of the important hardware features in the VAX implementation. Appendix A in the back of this book contains a table of commonly used VAX family mnemonics.

ARCHITECTURE OVERVIEW
All VAX family systems implement a common architecture; architecture is defined as the set of features that are visible to the programmer. The VAX family architecture provides a significant enhancement to the virtual addressing capability of the PDP-11 series consistent with small code size, easy utilization by high-level languages, and a high degree of compatibility with the PDP-11 family of minicomputers. While VAX is not strictly binary-compatible with the PDP-11 binary code, it does implement a compatibility mode which executes most PDP-11 instructions.

The VAX family architecture is characterized by a powerful and complete instruction set of 248 basic instructions (see Appendix B), a wide range of data types, an efficient set of addressing modes, full demand paging memory management, and a very large virtual address space of over 4 billion bytes. Arithmetic and logical operations can be performed on byte integers (8 bits), word integers (16 bits), and longword integers (32 bits); plus, some instructions can perform operations on 64-bit quadword integers. Additionally, the native mode instruction set includes floating point operations, character string manipulations, queue management, packed decimal arithmetic, and many instructions which improve the performance and memory utilization of the system. Applications also can be performed on variable-length bit fields—a feature that the VAX family processors have added to the PDP-11 operational capabilities.

Another significant feature of the VAX architecture is that instruction addressing is arbitrary. This means that there are no fixed formats, and no restrictions as to the location of an operand for a particular instruction or even the instruction itself. Therefore, operands and instructions can begin on any byte address, odd or even. The result of this flexibility is that high-level language compilers, such as FORTRAN, can generate code that is smaller in size, very efficient, and easy to manipulate in the compiler's data structures. This results in faster performance and lower memory utilization. The VAX/VMS operating system and the hardware work together as one unit to provide VAX with its multiuser, multiprogramming, virtual memory capabilities. For further information concerning VAX architecture, refer to the VAX Architecture Handbook.
SOFTWARE OVERVIEW
VAX/VMS is the general-purpose operating system for the VAX family processors. This software compatibility between family members means that users with VAX-11/780 systems can run existing software on the VAX-11/750 without having to make modifications. Furthermore, VAX/VMS provides a reliable, high performance environment for the concurrent execution of multiuser timesharing, batch, and time-critical applications. VAX/VMS provides:

- Virtual memory management for executing large programs
- Event-driven priority scheduling
- Shared memory, file, and interprocess communication data protection based on ownership and application groups
- Programmed system services for process and subprocess control and interprocess communication

VAX/VMS uses the VAX memory management features to provide swapping, paging, protection, and sharing of both code and data. Memory is allocated dynamically. Applications can control the amount of physical memory allocated to executing processes, to the protection of pages, and to swapping; furthermore, these controls can be added after the application is implemented.

CPU time and memory residency are scheduled on a pre-emptive priority basis. Thus, time-critical processes do not have to compete with lower priority processes for scheduling services. Scheduling rotates among processes of the same priority.

VAX/VMS includes system services to control processes and process execution, control time-critical response, control scheduling, and obtain information. Process control services allow the creation of subprocesses as well as independent detached processes. Processes can communicate and synchronize using mailboxes, shared areas of memory, or shared files. A group of processes can also communicate and synchronize, using multiple common-event flag clusters.

Memory access protection is provided both between and within processes. Each process has its own independent virtual address space which can be mapped to private pages or shared pages. VAX/VMS uses the four processor access modes to read- and/or write-protect individual pages within a process. Protection of files, shared pages of memory, and interprocess communication facilities such as mailboxes and event flags, is based on user identification codes individually assigned to accessors and data.

A complete program development environment is offered. In addition to the native assembly language, it offers optional high-level program-
Introduction

Programming languages commonly used in developing both scientific and commercial applications: FORTRAN, COBOL, BLISS, PASCAL, PL/1 and BASIC. It provides the tools necessary to write, assemble or compile, and link programs, as well as to build libraries of source, object, and image modules.

VAX/VMS data management includes a file system that provides volume structuring and protection, and record management services that provide device-independent access to VAX peripherals.

The VAX/VMS on-disk structure provides a multiple-level hierarchy of named directories and subdirectories. Files can extend across multiple volumes and can be as large as the volume set on which they reside. Volumes are mounted to identify them to the system. VAX/VMS also supports multivolume ANSI format magnetic tape files with transparent volume switching.

The VAX/VMS record management input/output system (RMS) provides device-independent access to disks, tapes, unit record equipment, terminals, and mailboxes. RMS allows users and application programs to create, access, and maintain data files with efficiency and economy. Under RMS, records are regarded by the user program as logical data units that are structured and accessed in accordance with application requirements.

RMS provides sequential record access to sequential file organizations, and sequential, random, or combined record access to relative file organizations.

For further information concerning VAX software and the VAX/VMS operating system, refer to the VAX Software Handbook or the VAX Technical Summary.

HARDWARE OVERVIEW

VAX computer systems consist of the central processing unit, the Console subsystem, the main memory subsystem, and the input/output (I/O) subsystems—the UNIBUS, the MASSBUS, and the DR780 (VAX-11/780 only) subsystems. The VAX-11/750 hardware configuration is illustrated in Figure 1-1. The VAX-11/780 hardware configuration is illustrated in Figure 1-2.

VAX Central Processing Unit (CPU)

The VAX processor is a 32-bit high-speed microprogrammed computer that executes instructions in native mode, and nonprivileged PDP-11 instructions in compatibility mode.

The processor can directly address four billion bytes of virtual address space, and provides a complete and powerful instruction set that in-
Figures 1-1 and 1-2 illustrate the hardware configurations of the VAX-11/750 and VAX-11/780 systems, respectively.

1. Native Instruction Set — The VAX instructions are an extension of the PDP-11 instruction set. Instructions can be grouped into classes based on their functions and uses:

   1. Instructions to manipulate arithmetic and logical data types.
Introduction

Figure 1-3 illustrates the elements of the VAX central processing unit.

![VAX CPU Diagram]

Figure 1-3   Central Processing Unit (CPU)

These include integer, floating point, packed decimal, character string, and bit field instructions.

The data type identifies how many stored bits are to be treated as a unit and how the unit is to be interpreted. Data types that may be used are:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Represented As</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Byte (8 bits), word (16 bits), longword (32 bits), quadword (64 bits)</td>
</tr>
<tr>
<td>Floating point</td>
<td>4-byte F_floating, 8-byte D_floating, 8-byte G_floating, 16-byte H_floating</td>
</tr>
<tr>
<td>Packed decimal</td>
<td>String of bytes (up to 31 decimal digits, 2 digits per byte)</td>
</tr>
<tr>
<td>Character string</td>
<td>String of bytes interpreted as character codes; a numeric string is a character string of codes for decimal numbers (up to 64 Kb)</td>
</tr>
<tr>
<td>Bits and bit-fields</td>
<td>Field length is arbitrary and is defined by the programmer (0 to 32 bits in length)</td>
</tr>
</tbody>
</table>

Integer, floating point, packed decimal, and character data are stored starting on an arbitrary byte boundary. Bit and bit field data do not necessarily start on a byte boundary. A collection of data structures can be packed together to use less storage space.
2. Instructions to manipulate special kinds of data. These include queue manipulation instructions (i.e., those that insert and remove queue entries), address manipulation instructions, and user-programmed general register load and save instructions. These instructions are used extensively by the VAX/VMS operating system.

3. Instructions to control basic program flow. These include BRANCH, JUMP, and CASE instructions, subroutine CALL instructions, and procedure CALL instructions.

4. Instructions to perform special operating system functions quickly. These include process control instructions (such as two special context switching instructions which allow process context variables to be loaded and saved using only one instruction for each operation), and the FIND FIRST instruction which (among other uses) allows the operating system to locate the highest priority executable process. These instructions contribute to rapid and efficient rescheduling.

5. Instructions provided specifically for high-level language constructs. During the design of the VAX family architecture, special attention was given to implementing frequently-used, high-level language constructs as single VAX instructions. These instructions contribute to decreased program size and increased execution speed. Some of the constructs which have become single VAX instructions include:

- The FORTRAN-computed GOTO statement (translates into the VAX CASE instruction).
- The loop construct (e.g., add, compare, and branch translates into the VAX ACB instruction).
- An extensive CALL facility (which aligns the stack on a long-word boundary, saves user-specified registers, and cleans up the stack on return); the CALL facility is used compatibly among all native mode languages and operating system services.

VAX instructions and data are variable in length. They need not be aligned on longword boundaries in physical memory, but may begin at any odd or even byte address. Therefore, instructions not requiring arguments use only one byte, while other instructions may take two, three, or up to 54 bytes depending on the number of arguments and their addressing modes. The advantage of byte alignment is that instruction streams and data structures can be stored in much less physical memory.
Introduction

The VAX processors offer several addressing modes. Seven of these use the general registers to identify the operand location and operate similarly to the PDP-11 addressing modes. The names of the modes are:

- Register
- Register deferred
- Autoincrement
- Autoincrement deferred
- Autodecrement
- Displacement (similar to the PDP-11 index mode)
- Displacement deferred (similar to the PDP-11 index deferred mode)

The two additional addressing modes implemented by VAX family processors are:

- Indexed
- Literal

Because the VAX instruction set is so flexible, fewer instructions and less storage than on non-VAX processors are required to perform most functions. The result is more compact and efficient programs, faster program execution, faster context switching, more precise and faster math functions, and improved compiler-generated code.

General Registers and Stacks — The VAX family CPUs provide sixteen 32-bit general registers which can be used for temporary storage, or as accumulators, index registers, and base registers. Registers R0 through R11 can be used as general purpose registers and the remaining four have special significance depending on the instruction being executed: Register 12 (the Argument Pointer); Register 13 (the Frame Pointer); Register 14 (the Stack Pointer); and Register 15 (the Program Counter).

Stacks are associated with the processor's execution state. The processor may be in a process context (in one of four modes—kernel, executive, supervisor, or user) or in the system-wide interrupt service context. A stack pointer is associated with each of these states. Whenever the processor changes from one state to another, Register 14 (the Stack Pointer) is updated accordingly.

Caches — The VAX family CPUs provide three cache systems—the memory cache, an address translation buffer, and an instruction buffer.

The memory cache (typically 90% hit rate on the VAX-11/750 and 95% on the VAX-11/780) provides the central processor with high-speed data access by storing frequently referenced addresses, data and in-
Instruction items. The memory cache reduces a typical main memory cycle time to half what it would be without the cache feature.

The instruction buffer is an 8-byte buffer that enables the CPU to fetch and decode the next instruction while the current instruction completes execution. The instruction buffer in combination with the parallel data paths (which can perform integer arithmetic, shifting, and memory access at the same time) significantly enhances VAX's performance because the CPU does not have to wait for the next instruction.

VAX systems provide an address translation buffer that eliminates extra memory accesses during virtual-to-physical address translations. The address translation buffer contains frequently used virtual-to-physical address translations: 128 page table entry translations in the VAX-11/780 and 512 in that of the VAX-11/750.

Clocks — VAX CPUs have two clocks—a programmable real-time clock used by system diagnostics and by the VAX/VMS operating system for accounting and scheduling, and a time-of-year clock, which insures the correct time of day and date.

Memory Management — The VAX memory management hardware enables the VAX/VMS operating system to provide a flexible and efficient virtual memory programming environment. Hardware memory management, with the operating system, provides both paging (with user control) and swapping.

In addition, memory management provides four hierarchical modes: kernel, executive, supervisor, and user, with read/write access control for each mode.

The memory management hardware facilitates program and data sharing, and allows larger program size and increased performance.

The Console Subsystem
The Console subsystem configuration makes the VAX series extremely approachable systems. Four elements combine to give the user access to the system's capabilities: a set of machine status switches and lights on the CPU front panel, a console terminal, a console command language, and an integral mass storage device. Simple console commands, entered through the console terminal, replace the traditional toggle switches and provide operational control (i.e., bootstrapping, initialization, self-testing, examining/depositing data in memory, etc.). When it is not being used to communicate console command language instructions to the CPU, the same terminal functions as a VAX system terminal and is available to authorized users for normal
system operations. This dual-purpose console terminal results in significant hardware savings.

The VAX Console subsystem and the console command language also facilitate the loading of diagnostics and software updates from the mass storage device: a TU58 tape cartridge on the VAX-11/750 and an RX01 floppy disk system on the VAX-11/780.

With the customer's cooperation, the Console subsystem may also be equipped with a Remote Diagnosis Module (RDM) which allows VAX to be connected to a host computer at the DIGITAL Diagnostic Center for fault detection or preventive maintenance procedures. This remote diagnosis option can significantly lower maintenance costs as well as contribute to system availability and maintainability.

The VAX-11/750 Console subsystem is illustrated in Figure 1-4. The VAX-11/780 Console subsystem is illustrated in Figure 1-5.

![Diagram of VAX-11/750 Console Subsystem](image1)

Figure 1-4  VAX-11/750 Console Subsystem

![Diagram of VAX-11/780 Console Subsystem](image2)

Figure 1-5  VAX-11/780 Console Subsystem
Introduction

The Main Memory Subsystem

VAX-11/750 Systems — The main memory subsystem consists of ECC MOS memory, which is connected to the system via the memory controller, as illustrated in Figure 1-6.

![Memory Controller Diagram](image)

Figure 1-6  VAX-11/750 Main Memory Subsystem

MOS memory may be added in increments of 256 Kb to a maximum of 2 Mb.

The VAX-11/750 physical memory is built using 16K MOS RAM chips. It is organized in 32-bit longwords, plus a 7-bit ECC (error-correcting code). This ECC allows the correction of all single-bit errors, and the detection of all double-bit errors, to insure data integrity.

VAX-11/780 Systems — The VAX-11/780 main memory subsystem has the same features as the VAX-11/750 subsystem, plus increased performance and capacity options. These options include the capability to access up to 4 Mb of physical memory, which can be increased to a maximum array size of 8 Mb by means of a second controller. Two memory controllers with equal amounts of memory can be interleaved to improve I/O throughput.

A multiport memory option is also available on VAX-11/780 systems permitting shared physical memory by up to four VAX-11/780 systems. Two multiport options can be added to a system, each supporting up to 2 Mb of shared memory. This increases the VAX-11/780's maximum physical memory configuration to 12 Mb.

Figure 1-7 illustrates the VAX-11/780 main memory subsystem.

![Memory Controller Diagram](image)

Figure 1-7  VAX-11/780 Main Memory Subsystem
The Input/Output Subsytems

VAX-11/750 Systems — The VAX-11/750's I/O subsystems consist of UNIBUS and MASSBUS devices connected to the system through special buffered interfaces called adapters. As illustrated in Figure 1-8, each VAX-11/750 system has one UNIBUS adapter for standard peripherals and can have up to three MASSBUS adapters for high speed peripherals.

![I/O Adapters Diagram](image)

Figure 1-8 VAX-11/750 I/O Subsystems

VAX-11/780 Systems — The VAX-11/780's I/O subsystems also include UNIBUS and MASSBUS adapters. Each system's configuration includes one UNIBUS adapter as standard equipment with the capability to add up to three more as options. Up to four MASSBUS adapters can also be included.

The VAX-11/780 offers as an option a very high performance, 32-bit, general purpose parallel interface which allows user devices to be connected directly to the system's Synchronous Backplane Interconnect. Called the DR780, it is capable of transferring data to and from memory at speeds of up to 6.67 Mb per second.

Figure 1-9 illustrates the VAX-11/780 I/O subsystem.

![I/O Adapters Diagram](image)

Figure 1-9 VAX-11/780 I/O Subsystem

The UNIBUS — General purpose peripherals and UNIBUS-compatible customer-developed devices are connected to VAX systems via
Introduction

the UNIBUS. Since the memory deals in 24-bit addresses on the VAX-11/750 (16 Mb physical address space) and 30-bit addresses on the VAX-11/780 (1 Gb physical address space), 18-bit UNIBUS addresses must be translated to appropriate VAX memory addresses. This mapping function is performed by the UNIBUS adapter (a special interface between memory and the UNIBUS) which translates UNIBUS addresses and data, and interrupt requests to their memory interconnect equivalents, and vice versa.

The UNIBUS adapter performs priority arbitration among devices on the UNIBUS, a function handled by the central processor in PDP-11 systems. The address translation map permits contiguous disk transfers to and from noncontiguous pages of memory (these are called scatter/gather operations).

The UNIBUS adapter allows two kinds of data transfers; program interrupt and direct memory access (DMA). To make the most efficient use of the memory bandwidth, the UNIBUS adapter facilitates high-speed DMA transfers by providing buffered DMA data paths for up to 3 high-speed devices on the VAX-11/750 and up to 15 similar devices on the VAX-11/780. Each of these channels has a 32-bit buffer (plus byte parity) for holding two 16-bit transfers to or from UNIBUS devices. The result is that only one memory transfer (32 bits) is required for every two UNIBUS transfers. The maximum aggregate transfer rate through the buffered data paths is 1.5 Mb per second.

Any number of unbuffered DMA transfers are handled by one direct DMA data path. Every 8- or 16-bit transfer on the UNIBUS requires a 32-bit memory transfer (although only 16 bits are used). The maximum transfer rate through the direct data path is 1 Mb per second.

It should be noted that the UNIBUS adapter permits program interrupts, unbuffered data transfers and buffered data transfers to occur concurrently. The maximum aggregate throughput rate of the direct data path, plus the three buffered data paths, is 1.5 Mb per second.

The MASSBUS — High-performance mass storage devices, such as the RP series moving head disk drives, are connected to VAX systems using a MASSBUS adapter. The MASSBUS adapter is the interface between the MASSBUS and memory and performs all control, arbitration, and buffering functions. Address mapping is similar to that performed by the UNIBUS adapter.

Each MASSBUS adapter uses a 32-byte silo data buffer, which permits transfers at rates up to two million bytes/second to and from physical memory. The VAX-11/750 is capable of a maximum transfer rate of 5 Mb per second using all three MASSBUS adapters. The VAX-
Introduction

11/780's maximum rate is 8 Mb per second using four MASSBUS adapters and interleaved memory. As in the UNIBUS adapter, data are assembled in 32-bit longwords (plus byte parity) to make maximum efficient use of the memory bandwidth. I/O transfer rates listed here are maximum and will vary depending upon system configuration and workload.

The DR780 — The DR780 option is a synchronous 32-bit bidirectional path enabling connection between the SBI of the VAX-11/780 central processor and a user device via a bus called the DR32 Device Interconnect (DDI). Data transfer rates are program selectable from 0.156 to 6.67 Mb per second.

The DDI actually consists of two separate and independent interconnects: one for control signals and one for data. The control interconnect is an asynchronous 8-bit bidirectional path for transferring control information to and from the user device. A great deal of flexibility for the design of the user device is available since the width of the control interconnect (8 bits) enables the DR780 to address up to 256 individual registers. These registers can pass various kinds of information, such as buffer descriptors, commands, or status, to the user process.

The data interconnect is a synchronous 32-bit bidirectional path for data which is synchronized to a single clock. The user can choose to use the internal DR780 clock or can provide a clock within the user device.

The DR780 is fully supported by the VAX/VMS operating system with a simple, easy-to-use I/O driver, high-level language support routines, and extensive documentation.

Further information on the DR780 can be found in Chapter 20, Interconnects.

Optional Hardware Equipment — Prewired mounting space is available within VAX systems for mounting the following optional equipment:

1. High performance floating point accelerator (VAX-11/780). The FPA is an independent processor that works in parallel with the basic CPU to execute the standard floating point instruction set with substantial performance improvement. The FPA takes advantage of the CPU's instruction buffer to prefetch instructions and the memory cache to access main memory. Once the CPU has the required data, the FPA overrides the normal execution flow of the standard floating point microcode and forces use of its
own code. Then, while the FPA is executing, the CPU can be performing other operations in parallel (for example, fetching the third operand of a three-address instruction). The result is much greater throughput and decreased execution time.

2. **Additional memory.** Memory may be added in 256 Kb increments to a total of 2 Mb MOS memory with ECC on the VAX-11/750 and 4 Mb per controller on the VAX-11/780 (two controllers maximum). A multiport memory option is available on VAX-11/780 systems.

3. **Main memory battery backup.** Battery backup may be added to maintain the contents of memory (10 minutes minimum for 2 Mb of memory on VAX-11/750, or for 4 Mb memory on VAX-11/780).

4. **User Control Store (UCS).** Sophisticated users can add to the native mode instruction set by programming frequently used routines in microcode. The control store size is 10 Kb on the VAX-11/750 systems and 12 Kb on the VAX-11/780.

5. **Remote Diagnosis Option.** Allows connection to the DIGITAL Diagnostic Center (DDC) for hardware/software fault diagnosis, preventive maintenance diagnostics, etc. This option can substantially lower maintenance contract prices and eliminate downtime by pinpointing faults before a Field Service engineer is dispatched to correct the problem.
CHAPTER 2
PROCESS STRUCTURE

PROCESS DEFINITION
A process is the basic entity schedulable for execution by VAX family processors. A process consists of an address space and both a hardware and a software context. The hardware context of a process is defined by a Process Control Block (PCB) which contains images of the 14 general purpose registers, the Processor Status Longword (PSL), the Program Counter (PC), the four per-process stack pointers, the process virtual memory defined by the base and length registers (P0BR, P0LR, P1BR, and P1LR) and several minor control fields. In order for a process to execute, the majority of the PCB must be moved into internal registers. While a process is being executed, some of its hardware context is being updated in the internal registers. When a process is not being executed, its hardware context is stored in the process control block. Saving the contents of the privileged registers in the PCB of the currently executing process and then loading a new set of context in the privileged registers from another PCB is termed context switching. Context switching occurs as one process after another is scheduled for execution.

PROCESS CONTEXT

Process Control Block Base (PCBB)
The process control block for the currently executing process is pointed to by the content of the Process Control Block Base (PCBB) Register, an internal privileged register. The Process Control Block Base Register is illustrated in Figure 2-1.

![Figure 2-1](imageURL)

Process Control Block (PCB)
The process control block (PCB) contains all of the switchable process context collected into a compact form for ease of movement to and from the privileged internal registers. Although in any normal operating system there is additional software context for each process, the following description is limited to that portion of the PCB known to the hardware. The process control block is illustrated in Figure 2-2.
## Process Structure

### Figure 2-2  Process Control Block

A description of the process control block follows.

<table>
<thead>
<tr>
<th>Long word</th>
<th>Bits</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&lt;31:0&gt;</td>
<td>KSP</td>
<td>Kernel Stack Pointer. Contains the stack pointer to be used when the current access mode field in the Processor Status Longword (PSL) is zero and Interrupt Stack (IS) is zero.</td>
</tr>
<tr>
<td>Long word</td>
<td>Bits</td>
<td>Mnemonics and Description</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>-------</td>
<td>---------------------------</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>&lt;31:0&gt;</td>
<td>ESP</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Executive Stack Pointer. Contains the stack pointer to be used when the current access mode field in the PSL is one.</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>&lt;31:0&gt;</td>
<td>SSP</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Supervisor Stack Pointer. Contains the stack pointer to be used when the current access mode field in the PSL is two.</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>&lt;31:0&gt;</td>
<td>USP</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>User Stack Pointer. Contains the stack pointer to be used when the current access mode field in the PSL is three.</td>
<td></td>
</tr>
<tr>
<td>4-17</td>
<td>&lt;31:0&gt;</td>
<td>R0-R11, AP, FP</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>General registers R0-R11, Argument Pointer and Frame Pointer.</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>&lt;31:0&gt;</td>
<td>PC</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Program Counter.</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>&lt;31:0&gt;</td>
<td>PSL</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Processor Status Longword.</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>&lt;31:0&gt;</td>
<td>P0BR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Base register for page table describing process virtual addresses from zero to $2^{30} - 1$.</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>&lt;21:0&gt;</td>
<td>P0LR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Length register for page table located by P0BR. Describes effective length of page table.</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>&lt;23:22&gt;</td>
<td>MBZ</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Must be zero.</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>&lt;26:24&gt;</td>
<td>ASTLVL</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Contains access mode number established by software of the most privileged access mode for which an asynchronous system trap is pending. (ASTs will be discussed in the next section.) Controls the triggering of the AST delivery interrupt during REI (return from interrupt or exception) instructions.</td>
<td></td>
</tr>
</tbody>
</table>
### Process Structure

<table>
<thead>
<tr>
<th>Long word</th>
<th>Bits</th>
<th>Mnemonic Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASTLVL</td>
<td></td>
<td>Meaning</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>AST pending for access mode 0 (kernel)</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>AST pending for access mode 1 (executive)</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>AST pending for access mode 2 (supervisor)</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>AST pending for access mode 3 (user)</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>No pending AST</td>
</tr>
<tr>
<td>5-7</td>
<td></td>
<td>Reserved to DIGITAL</td>
</tr>
</tbody>
</table>

21  

\(<31:27>\) MBZ  Must be zero.

22  

\(<31:0>\) P1BR  Base register for page table describing process virtual addresses from \(2^{30}\) to \(2^{31} - 1\).

23  

\(<21:0>\) P1LR  Length register for page table located by P1BR. Describes effective length of page table.

23  

\(<30:22>\) MBZ  Must be zero.

23  

\(<31>\) PME  Performance Monitor Enable. Controls a signal visible to an external hardware performance monitor. This bit is set to identify those processes for which monitoring is desired and to permit their behavior to be observed without interference from other system activity.

Software symbols are defined for these locations by using the prefix PTX$L_ and the mnemonic shown above.

To alter its P0BR, P1BR, P0LR, P1LR, ASTLVL or PME, a process must be executing in kernel mode. It must first store the desired new value in the memory image of the PCB, then move the value to the appropriate privileged register. This protocol results from the fact that these are read-only fields (for the context switch instructions) in the process control block.

### Process Privileged Registers

The ASTLVL and PME fields of the PCB are contained in registers when the process is executing. In order to access them, two privileged
Process Structure

registers are provided. These are the AST Level Register and the Performance Monitor Enable Register (PME). The AST Level Register is illustrated in Figure 2-3.

![Figure 2-3 AST Level Register (Read/Write)]

An MTPR src, #ASTLVL with src<2:0> GEQU 5 results in a reserved operand fault. At bootstrap time, the content of ASTLVL is four. The Performance Monitor Enable Register is illustrated in Figure 2-4. At bootstrap time, PME is cleared.

![Figure 2-4 Performance Monitor Enable Register (Read/Write)]

ASYNCHRONOUS SYSTEM TRAPS (AST)
Asynchronous system traps are used to notify a process of events that are not synchronized with its execution and to initiate processing for asynchronous events with the least possible delay. The delay in delivery may be due to either process nonresidence or an access mode mismatch. The efficient handling of ASTs in the VAX family processors requires some hardware assistance to detect changes in access mode (current access mode in PSL). Each of the four execution access modes (kernel, executive, supervisor, and user) may receive ASTs; however, an AST for a less privileged access mode must not be permitted to interrupt execution in a more privileged access mode. Since transitions to a less privileged access mode occur only in the REI instruction, comparison of the current access mode field is made with a privileged register (ASTLVL) containing the most privileged access mode number for which an AST is pending. If the new access mode is greater than or equal to the pending ASTLVL, an IPL 2 interrupt is triggered to cause delivery of the pending AST.
Process Structure

General software flow for AST processing:

1. An event associated with an AST causes software to put an AST control block in the queue to the software PCB; software then sets the ASTLVL field in the hardware PCB to the most privileged access mode for which an AST is pending. If the target process is currently executing, the ASTLVL privileged register also has to be set.

2. When an REI instruction detects a transition to an access mode that can be interrupted by a pending AST, an IPL 2 interrupt is requested to cause delivery of the AST. Note that the REI instruction does not make pending AST checks while returning to a routine executing on the interrupt stack.

3. The IPL 2 interrupt service routine computes the correct new value for ASTLVL to prevent additional AST delivery interrupts while in kernel mode, and moves that value to the PCB and the ASTLVL register before lowering IPL and actually dispatching the AST. This interrupt service routine normally executes on the kernel stack in the context of the process receiving the AST.

4. At the conclusion of processing for an AST, the ASTLVL is recomputed and moved to the PCB and ASTLVL register by software.

PROCESS STRUCTURE INTERRUPTS
Two of the software interrupt priorities are reserved for process structure software: IPL 2 is for AST delivery interrupts and IPL 3 is for process scheduling interrupts. For details on these interrupts, please see the VAX Software Handbook.

PROCESS STRUCTURE INSTRUCTIONS
Process scheduling software executes on the interrupt stack (PSL<IS> set) in order to have a non-context-switched stack available for use. If the scheduler were running on a process's kernel stack, then any state information it had there would disappear when a new process is selected. Running on the interrupt stack can occur as the result of the interrupt origin of scheduling events; however, some synchronous scheduling requests such as a WAIT service may cause rescheduling without any interrupt occurrence. For this reason, the Save Process Context (SVPCTX) instruction can be executed while on either the kernel or interrupt stack and forces a transition to execution on the interrupt stack.

All of the process structure instructions are privileged and may only be executed in kernel mode. In the following description of the load and save process context instructions, these notation conventions are used:
### Notation Meaning

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tmp</code></td>
<td><code>tmp1</code> and <code>tmp2</code> are pseudo-registers which are not normally implemented in hardware</td>
</tr>
<tr>
<td><code>!</code></td>
<td>indicates comment statement</td>
</tr>
<tr>
<td><code>←</code></td>
<td>the back arrow is an assignment operator, i.e., the value indicated on the right is copied to the register or pseudo-register indicated on the left</td>
</tr>
<tr>
<td><code>()</code></td>
<td>this indicates the contents of the address specified by the included expression</td>
</tr>
<tr>
<td><code>&lt;N:M&gt;</code></td>
<td>this notation indicates the field consisting of bits N through M of the immediately preceding value</td>
</tr>
<tr>
<td><code>{}</code></td>
<td>used to indicate a composite action</td>
</tr>
<tr>
<td><code>[]</code></td>
<td>used to group terms for clarity, usually appear in logical expressions</td>
</tr>
</tbody>
</table>

---

### LDPCTX Load Process Context

**Purpose:** restore register and memory management context  

**Format:** Opcode

**Operation:**

```plaintext
if PSL<current mode> NEQU 0 then
  [privileged instruction fault];
  [invalidate per-process translation buffer entries];
  IPCB is located by physical address in PCBB
  KSP←(PCB);
  ESP←(PCB+4);
  SSP←(PCB+8);
  USP←(PCB+12);
  R0←(PCB+16);
  R1←(PCB+20);
  R2←(PCB+24);
  R3←(PCB+28);
  R4←(PCB+32);
  R5←(PCB+36);
  R6←(PCB+40);
  R7←(PCB+44);
  R8←(PCB+48);
  R9←(PCB+52);
  R10←(PCB+56);
  R11←(PCB+60);
  AP←(PCB+64);
```
Process Structure

FP←(PCB+68);
tmp1←(PCB+80);
if [tmp1<31:30> NEQU 2] OR [tmp1<1:0> NEQU 0] then
    {reserved operand abort};
P0BR←tmp1;
if (PCB+84)<31:27> NEQU 0 then {reserved operand abort};
if (PCB+84)<23:22> NEQU 0 then {reserved operand abort};
P0LR←(PCB+84)<21:0>;
if (PCB+84)<26:24> GEQU 5 then {reserved operand abort};
ASTLVL←(PCB+84)<26:24>;
tmp1←(PCB+88);
tmp2←tmp1 + 2^{23}
if [tmp2<31:30> NEQU 2] OR [tmp2<1:0> NEQU 0] then
    {reserved operand abort};
P1BR←tmp1;
if (PCB+92)<30:22> NEQU 0 then {reserved operand abort};
P1LR←(PCB+92)<21:0>;
PME←(PCB+92)<31>;
if PSL <IS> EQU 1 then
    begin
        ISP←SP;
        {interrupts off};
        PSL<IS>←0;
        SP←(PCB);    !get KSP
        {interrupts on};
    end;
-(SP)←(PCB+76);    !push PSL
-(SP)←(PCB+72);    !push PC

Condition Codes:  N←N;
Z←Z;
V←V;
C←C;

Exceptions: reserved operand privileged instruction

Opcodes: 06 LDPCTX Load Process Context
Description: The Process Control Block is specified by the privileged register Process Control Block Base. The general registers are loaded from the PCB. The memory management registers describing the process address space are also loaded and the process entries in the translation buffer are cleared. Execution is switched to the kernel stack. The PC and PSL are moved from the PCB to the stack, suitable for use by a subsequent REI instruction.

VAX keeps a copy of each of the per-process stack pointers in internal registers, and LDPCTX loads the internal registers from the PCB.

SVPCTX Save Process Context

Purpose: Save register context

Format: Opcode

Operation: if PSL<current mode> NEQU 0 then

{privileged instruction fault};

!PCB is located by the physical address in PCBB

(PCB)←KSP;

(PCB+4)←ESP;

(PCB+8)←SSP;

(PCB+12)←USP;

(PCB+16)←R0;

(PCB+20)←R1;

(PCB+24)←R2;

(PCB+28)←R3;

(PCB+32)←R4;

(PCB+36)←R5;

(PCB+40)←R6;

(PCB+44)←R7;

(PCB+48)←R8;

(PCB+52)←R9;

(PCB+56)←R10;

(PCB+60)←R11;

(PCB+64)←AP;

(PCB+68)←FP;

(PCB+72)←(SP)+;

!pop PC

(PCB+76)←(SP)+;

!pop PSL

If PSL<IS> EQLU 0 then
Process Structure

```
begin
PSL<IPL> ← MAXU(1, PSL<IPL>);
(PCB) ← SP;  !save KSP
KSP ← SP;
{interrupts off};
PSL<IS> ← 1;
SP ← ISP;
{interrupts on};
end;
```

**Condition**

N ← N;

**Codes:**

Z ← Z;
V ← V;
C ← C;

**Exceptions:**

reserved instruction

**Opcodes:**

07 SVPCTX Save Process Context

**Description:**

The Process Control Block is specified by the privileged register Process Control Block Base. The general registers are saved into the PCB. The PC and PSL currently on the top of the current stack are popped and stored in the PCB. If a SVPCTX instruction is executed when IS is clear, then IS is set, the interrupt stack pointer activated, and IPL is maximized with 1 because of the switch to the interrupt stack.

**Notes:**

1. The map, ASTLVL, and PME contents of the PCB are not saved because they are rarely changed. Thus, not writing them saves overhead.
2. The processor keeps a copy of each of the per-process stack pointers in internal registers, and SVPCTX stores the internal registers into the PCB.
3. Between the SVPCTX instruction that saves the state for one process and the LDPCTX that loads the state of another, the internal stack pointers may not be referenced by MFPR or MTPR instructions. This implies that interrupt service routines invoked at a priority higher than the lowest one used for context switching must not reference the process stack pointers.
**Process Structure**

**USAGE EXAMPLE**
The following example is intended to illustrate how the process structure instructions can be used to implement process dispatching software. It is assumed that this simple dispatcher is always entered via an interrupt.

```
; ENTERED VIA INTERRUPT
; IPL = 3

RESCHED: SVPCTX ; Save context in PCB

<set state to runnable>
<and place current PCB>
<on proper RUN queue>

<Remove head of highest>
<priority, nonempty,>
<RUN queue.>
MTPR @#PHYS PCB, PCBB ; Set physical PCB
; address in PCBB
LDPCTX ; Load context from PCB
; For new process
REI ; Place process in execution
```
CHAPTER 3
EXCEPTIONS AND INTERRUPTS

INTRODUCTION
At certain times during system operation, internal or external events may require the execution of pieces of software outside the explicit flow of control. The processor transfers control by forcing a change in the flow of control from that indicated in the currently executing process.

Some of these events are relevant to the currently executing process, and normally invoke software in the context of the current process. The notification of these events is termed an exception.

Other events are relevant to other processes, or to the system as a whole, and are serviced in a system-wide context. The notification process for these events is termed an interrupt, and the system-wide context is described as executing on the interrupt stack (IS). Some interrupts are so urgent that they require high priority service. To meet these needs, the processor has priority logic that grants interrupt service to the highest priority event at any point in time. The priority associated with an interrupt is termed its interrupt priority level (IPL).

The processor arbitrates interrupt requests according to priority. Only when the priority of an interrupt request is higher than the current IPL (bits<20:16> of the Processor Status Longword) does the processor raise the IPL and service the interrupt request. The interrupt service routine is entered at the IPL of the interrupt request and does not usually change the IPL set by the processor. Note that this is different from the PDP-11 where the interrupt vector specifies the IPL for the ISR.

Interrupt requests can come from devices, controllers, other processors (in customer-designed systems), or the processor itself. Software executing in kernel mode can raise and lower the priority of the processor by executing MTPR src, #IPL where src contains the new priority desired. However, in customer-designed multiprocessor systems, a processor cannot disable interrupts on other processors. Furthermore, the priority level of one processor does not affect the priority level of the other processors. Thus, in multiprocessor systems, interrupt priority levels cannot be used to synchronize access to shared resources. Even the various urgent interrupts including those exceptions that run at IPL 1F₁₆ do so on only one processor. Thus, special software action is required to stop other processors in a multiprocessor system.
Exceptions and Interrupts

Most service routines for software-generated exceptions execute at IPL 0. However, if a serious system failure occurs, the processor raises the IPL to the highest level (1F₁₆) to prevent interruption until the problem is corrected. Exception service routines are usually coded to avoid exceptions; however, nested exceptions may rarely occur in the case of an access control violation, reserved operand, or reserved addressing mode fault.

Processor Interrupt Priority Levels (IPLs)
The processor has 31 interrupt priority levels (IPLs), divided into 15 software levels (numbered 1 to F₁₆) and 16 hardware levels (1₀₁₆ to 1F₁₆). User applications, system calls, and system services all run at IPL 0, which may be thought of as process level. Higher numbered IPLs have higher priority; that is to say, any requests at an interrupt level higher than the processor's current IPL interrupt immediately, but requests at a lower or equal level are deferred.

Interrupt levels 1 through F₁₆ exist entirely for use by software. No device can request interrupts on those levels, but software can force an interrupt by executing MTPR src,#SIRR (Software Interrupt Request Register). Once a software interrupt request is made, it is cleared by hardware when the interrupt is taken.

Interrupt levels 1₀₁₆ to 17₁₆ are for use by devices and controllers, including UNIBUS devices. UNIBUS levels BR4 to BR7 correspond to VAX interrupt levels 1₄₁₆ to 1₇₁₆.

Interrupt levels 1₈₁₆ to 1F₁₆ are used by urgent conditions, including the interval clock, serious errors, and power fail.

Contrast Between Exceptions and Interrupts
Generally, exceptions and interrupts are very similar. When either is initiated, both the Processor Status Longword (PSL) and the Program Counter (PC) are pushed onto a stack. However, there are seven important differences:

1. An exception condition is caused by the execution of the current instruction, while an interrupt is caused by some activity in the computing system that usually is independent of the current instruction.

2. An exception condition usually is serviced in the context of the process that produced the exception condition, while an interrupt is serviced independently from the current process.

3. The IPL of the processor usually is not changed when the processor initiates an exception, while the IPL always is raised when an interrupt is serviced.
4. Exception service routines usually execute on a per-process stack, while interrupt service routines normally execute on a per-CPU stack. Machine check always executes on the ISP, however.

5. Enabled exceptions are initiated immediately, independent of the processor IPL. Interrupts, however, are delayed until the processor IPL drops below the IPL of the requesting interrupt.

6. Most exceptions cannot be disabled. However, if an exception-causing event occurs while that exception is disabled, no exception is initiated for that event, even when enabled subsequently. This includes overflow, which is the only exception whose occurrence is indicated by a condition code (V). If an interrupt condition occurs while that interrupt is disabled, or the processor is at the same or higher IPL, the condition eventually initiates an interrupt when the proper enabling conditions are met (if the condition is still present).

7. The previous mode field in the PSL is always set to kernel on an interrupt, but on an exception it indicates the mode in which the exception occurred.

**INTERRUPTS**

The processor services an interrupt request when the currently executing instruction is completed. The processor also services interrupt requests at well-defined points during the execution of long, iterative instructions such as the string instructions. For these instructions, to avoid saving additional instruction state in memory, interrupts are initiated when the instruction state can be completely contained in the registers, PSL, and PC.

The following events cause interrupts:

- Device completion (IPL 10-17\(_{16}\))
- Device error (IPL 10-17\(_{16}\))
- Device alert (IPL 10-17\(_{16}\))
- Device memory error (IPL 10-17\(_{16}\))
- Console terminal transmit and receive (IPL 14\(_{16}\))
- Console storage device (IPL 17\(_{16}\) for VAX-11/750 and IPL 14\(_{16}\) for VAX-11/780)
- Interval timer (IPL 18\(_{16}\))
- Recovered memory, bus or processor errors (the VAX-11/750 interrupts at IPL 1A\(_{16}\) for corrected memory reads; the VAX-11/780 at IPL 1B\(_{16}\), implementation specific)
- Unrecovered memory, bus or processor errors (the VAX-11/750 and VAX-11/780 interrupt at IPL 1D\(_{16}\) for write memory errors, implementation specific)
Exceptions and Interrupts

- Power fail (IPL 1E_{16})
- Software interrupt invoked by MTPR #SIRR (IPL 1 to F_{16})
- AST delivery when REI restores a PSL with IS clear and mode greater than or equal to ASTLVL (IPL 2_{16})

Each device controller has a separate set of interrupt vector locations in the system control block (SCB), thus eliminating the need for polling to determine which controller originated the interrupt. The vector address for each controller is fixed by hardware.

In order to reduce interrupt overhead, no memory mapping information is changed when an interrupt occurs. Thus, the instructions, data, and contents of the interrupt vector for an interrupt service routine must be in the system address space or present in every process at the same address.

Urgent Interrupts—Levels 18-1F_{16}
The processor provides eight priority levels for use by urgent conditions including serious errors (e.g., machine check) and power fail. Interrupts on these levels are initiated by the processor upon detection of certain conditions. Some of these conditions are not interrupts. For example, machine check is usually an exception but it runs at a high priority level on the interrupt stack.

Interrupt level 1E_{16} is for power failure. Interrupt level 1F_{16} is for those exceptions that must lock out all processing until handled. This includes the hardware and software "disasters" (machine check and kernel stack not valid). It also can be used to allow a kernel mode debugger to gain control on any condition.

Device Interrupts—Levels 10-17_{16}
The processor provides eight priority levels for use by peripheral devices. The VAX family IPLs 14_{16} through 17_{16} correspond to the UNIBUS levels BR4 to BR7. Any given implementation may or may not implement all levels of interrupts. On the VAX-11/750 only levels 14 through 17 are available for device interrupts.

Software-Generated Interrupts—Levels 1-F_{16}
The processor provides 15 priority interrupt levels for use by software. For greater detail, see the VAX Software Handbook.

Software Interrupt Summary Register
The Software Interrupt Summary Register (SISR) is a privileged register which records pending software interrupts. The SISR contains 1s in the bit positions corresponding to levels on which software interrupts are pending. All such levels, of course, must be lower than the current
Exceptions and Interrupts

processor IPL, or the processor would have taken the requested interrupt. Figure 3-1 illustrates the Software Interrupt Summary Register.

<table>
<thead>
<tr>
<th>31</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MBZ</td>
<td>PENDING SOFTWARE INTERRUPTS</td>
<td>F, E, D, C, B, A, 9, 8, 7, 6, 5, 4, 3, 2, 1</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3-1  Software Interrupt Summary Register (Read/Write)

At bootstrap time, the contents of SISR are cleared. The mechanism for accessing it is:

MFPR #SISR,dst  Reads the Software Interrupt Summary Register.

MTPR src,#SISR  Loads it, but this is not the normal way of making software interrupt requests. It is useful for clearing the software interrupt system and for reloading it after a power failure, for example.

Software Interrupt Request Register
The Software Interrupt Request Register (SIRR) is a write-only 4-bit privileged register used for making software interrupt requests. Figure 3-2 illustrates the software interrupt request register.

<table>
<thead>
<tr>
<th>31</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>IGNORED</td>
<td>REQUEST</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3-2  Software Interrupt Request Register (Write-Only)

Executing MTPR src,#SIRR requests an interrupt at the level specified by src<3:0>. Once a software interrupt request is made, the corresponding bit in the SISR is set. The hardware then clears the bit in the SISR when the interrupt is taken. If src<3:0> is greater than the current IPL, the interrupt occurs before execution of the following instruction. If src<3:0> is less than or equal to the current IPL, the interrupt is deferred until the IPL is lowered to less than src<3:0>, with no higher interrupt level pending. The IPL is lowered by either REI or by MTPR x,#IPL. If src<3:0> is 0, no interrupt will occur or be requested.
Exceptions and Interrupts

No indication is given if there is already a request at the selected level. Therefore, the service routine must not assume a one-to-one correspondence of interrupts generated and requests made. A valid protocol for generating such a correspondence is:

1. The requester uses INSQUE, INSQHI or INSQTI to place a control block describing the request onto a queue for the service routine.
2. The requester uses MTPR src,#SIRR to request an interrupt at the appropriate level.
3. The service routine uses REMQUE, REMQHI or REMQTI to remove a control block from the queue of service requests. If REMQUE, REMQHI or REMQTI returns failure (nothing in the queue), the service routine exits with REI.
4. If REMQUE, REMQHI OR REMQTI returns success (an item was removed from the queue), the service routine performs the service and returns to step 3 to look for other requests.

Interrupt Priority Level Register

Writing to the IPLR with the MTPR instruction will load the processor priority field in the Processor Status Longword (PSL). That is, bits<20:16> of the PSL are loaded from IPLR<4:0>. Reading from IPLR with the MFPR instruction will read the processor priority field from the PSL. On writing IPLR, bits<31:5> are ignored, and on reading IPLR, bits<31:5> are returned zero. Figure 3-3 illustrates the Interrupt Priority Level Register.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
</tr>
</thead>
</table>
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    | PSL<20:16>

Figure 3-3  Interrupt Priority Level Register (Read/Write)

At bootstrap time, the IPL is initialized to $1F_{16}$.

Interrupt service routines must follow the discipline of not lowering the IPL below their initial level. If they do, an interrupt at an intermediate level could cause the stack nesting to be improper. This would result in REI faulting. Actually, a service routine could lower the IPL if it ensured that no intermediate levels could interrupt. However, this would result in unreliable code.

Interrupt Example

As an example, assume the processor is running in response to an interrupt at IPL 5 (all numbers in this example are in hexadecimal); it
then sets the IPL to 8, and posts software requests at IPL 3, IPL 7, and IPL 9. Subsequently, a device interrupt arrives at IPL 11. Finally the IPL is set back to IPL 5. The sequence of execution is shown in Table 3-1.

### Table 3-1 Interrupt Sequence Example

<table>
<thead>
<tr>
<th>On Event</th>
<th>State After Contents of IPL(hex)</th>
<th>Event SISR (hex)</th>
<th>IPL In PSL on stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>(initial)</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MTPR #8,#IPL</td>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MTPR #3,#SIRR</td>
<td>8</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>MTPR #7,#SIRR</td>
<td>8</td>
<td>88</td>
<td>0</td>
</tr>
<tr>
<td>MTPR #9, #SIRR interrupts to device interrupts to</td>
<td>9</td>
<td>88</td>
<td>8,0</td>
</tr>
<tr>
<td>device service routine REI</td>
<td>9</td>
<td>88</td>
<td>8,0</td>
</tr>
<tr>
<td>IPL 9 service routine REI</td>
<td>8</td>
<td>88</td>
<td>0</td>
</tr>
<tr>
<td>MTPR #5,#IPL changes IPL to 5 and the request for 7 is granted immediately</td>
<td>7</td>
<td>8</td>
<td>5,0</td>
</tr>
<tr>
<td>IPL 7 service routine REI</td>
<td>5</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>initial IPL 5 service routine REI back to IPL 0 and the request for 3 is granted immediately</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>IPL 3 service routine REI</td>
<td>0</td>
<td>0</td>
<td>—</td>
</tr>
</tbody>
</table>

**SERIOUS SYSTEM FAILURES**

Although serious system failures are exceptions, they are discussed here rather than in the VAX Architecture Handbook because they are not linked to user software, but rather are processed by privileged software.

**Kernel Stack Not Valid Abort**

Kernel stack not valid abort is an exception that indicates that the kernel stack was not valid while the processor was pushing information onto the stack during the initiation of an exception or interrupt. Usually this is an indication of stack overflow or another executive
software error. The attempted exception is transformed into an abort that uses the interrupt stack. No information other than the PSL and PC is pushed onto the interrupt stack. The IPL is raised to $1F_{16}$. Software may abort the process without aborting the system; however, because of the lost information, the process cannot be continued. If the kernel stack is not valid during the normal execution of an instruction (including CHMK or REI), the processor initiates the normal memory management fault. If the exception vector $<1:0>$ for kernel stack not valid is zero or three, the behavior of the processor is UNDEFINED.

**Interrupt Stack Not Valid Halt**

An interrupt stack not valid halt is an exception indicating that the interrupt stack was not valid or that a memory error occurred while the processor was pushing information onto the stack during the initiation of an exception or interrupt. No further interrupt requests are acknowledged on this processor. The processor leaves the PC, the PSL, and the reason for the halt in registers so that it is available to a debugger, the normal bootstrap routine or an optional watchdog bootstrap routine. A watchdog bootstrap can cause the processor to leave the Halted state.

**Machine Check Exception**

A machine check exception indicates that the processor detected an internal error in itself. Machine check exceptions can be caused by such bus errors as non-existent memory, cache parity, translation buffer parity, or by a control store parity error. Like other exceptions, this exception is taken independently of IPL. IPL is raised to $1F_{16}$. A length parameter, an error code, and the contents of several registers are pushed onto the stack as longwords. The processor specifies the length parameter by placing the number of bytes pushed as the last longword pushed. This count excludes the PC, PSL, and the length parameter itself. Software decides, on the basis of the information presented, whether to abort the current process if the machine check came from the process. Machine check includes uncorrected bus and memory errors, and any other processor-detected errors. Some processor errors cannot ensure the state of the machine at all. For such errors, the state will be preserved on a "best effort" basis. If the exception vector $<1:0>$ for machine check is zero or three, the behavior of the processor is UNDEFINED. Under these conditions, the VAX processor will halt. Figure 3-4 shows the format of the stack after machine check exceptions on the VAX-11/750.
ARITHMETIC EXCEPTIONS
This section describes exceptions occurring as the result of an arithmetic or conversion operation. These mutually exclusive exceptions all are assigned the same vector in the System Control Block. Each of them indicates that an exception occurred during the last instruction and that the instruction has been completed (in the case of a trap) or backed up (fault). A code unique to each exception type is then pushed on the stack as a longword. Figure 3-5 illustrates the stack after the occurrence of an arithmetic exception. Note that in the case of a fault, the PC of the next instruction will be the same as the instruction which caused the exception.

The specific type codes saved are described in Table 3-2.
### Table 3-2 Arithmetic Exception Type Codes

<table>
<thead>
<tr>
<th>Type Code (hex)</th>
<th>Exception Type</th>
<th>Software Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRAPS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Integer overflow</td>
<td>SRM$K_INT_OVF_T</td>
</tr>
<tr>
<td>2</td>
<td>Integer divide by zero</td>
<td>SRM$K_INT_DIV_T</td>
</tr>
<tr>
<td>3</td>
<td>Floating overflow (not used on VAX-11/750)</td>
<td>SRM$K_FLT_OVF_T</td>
</tr>
<tr>
<td>4</td>
<td>Floating/decimal divide by 0</td>
<td>SRM$K_FLT_DIV_T</td>
</tr>
<tr>
<td>5</td>
<td>Floating underflow (not used on VAX-11/750)</td>
<td>SRM$K_FLT_UND_T</td>
</tr>
<tr>
<td>6</td>
<td>Decimal overflow</td>
<td>SRM$K_DEC_OVF_T</td>
</tr>
<tr>
<td>7</td>
<td>Subscript range</td>
<td>SRM$K_SUB_RNG_T</td>
</tr>
<tr>
<td>FAULTS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Floating overflow</td>
<td>SRM$K_FLT_OVF_F</td>
</tr>
<tr>
<td>9</td>
<td>Floating divide by zero</td>
<td>SRM$K_FLT_DIV_F</td>
</tr>
<tr>
<td>A</td>
<td>Floating underflow</td>
<td>SRM$K_FLT_UND_F</td>
</tr>
</tbody>
</table>

### Integer Overflow Trap
An integer overflow trap is an exception indicating that the last instruction executed had an integer overflow which set the V condition code. This trap only occurs if the integer overflow enable bit (IV) in the PSW is set. The result stored is the low order part of the correct result. N and Z are set according to the stored result, and the type code pushed on the stack is a 1. Note that the instructions RET, REI, REMQUE, REMQHI, REMQTI, MOVTUC, and BISPSW, do not cause overflow even if they set V. Also note that the EMODx floating point instructions can cause integer overflow.

### Integer Divide by Zero Trap
An integer divide by zero trap is an exception indicating that the last instruction executed had an integer zero divisor. The result stored is
equal to the dividend, and condition code V is set. The type code pushed on the stack is 2.

Floating Overflow Trap
A floating overflow trap is an exception that indicates that the last instruction executed resulted in an exponent greater than 127 (unbiased) after normalization and rounding. The result stored contains a one in the sign and zeros in the exponent and fraction fields. This is a reserved operand, and will cause a reserved operand fault if used in a subsequent floating point instruction. The N and V condition code bits are set and Z and C are cleared. The type code pushed on the stack is 3. This trap is not implemented on VAX-11/750 systems.

Floating/Decimal String Divide by Zero Trap
A decimal string divide by zero trap is an exception indicating that the last instruction executed had a decimal string zero divisor. The destination and condition codes are UNPREDICTABLE. The zero divisor can be either +0 or −0.

The type code pushed on the stack is 4.

Floating Underflow Trap
A floating underflow trap is an exception that indicates that the last instruction executed resulted in an exponent less than −127 (unbiased) after normalization and rounding and that floating underflow was enabled (FU set). The result stored is zero. Except for POLYx, the N, V, and C condition codes are cleared and Z is set. In POLYx, the trap occurs on completion of the instruction, which may be many operations after the underflow. The condition codes are set on the final result in POLYx. The type code pushed on the stack is 5. This trap is not implemented on VAX-11/750 systems.

Decimal String Overflow Trap
A decimal string overflow trap is an exception indicating that the last instruction executed had a decimal string result too large for the destination string provided, and that decimal was enabled (DV set). The V condition code is always set. The type code pushed on the stack is 6.

Subscript Range Trap
A subscript range trap is an exception indicating that the last instruction was an INDEX instruction with a subscript operand that failed the range check. The value of the subscript operand is lower than the low operand or greater than the high operand. The result is stored in indexout, and the condition codes are set as if the subscript were within range. The type code pushed on the stack is 7.
Floating Overflow Fault
A floating overflow fault is an exception indicating that the last instruction executed resulted (after rounding and normalization) in an exponent greater than the largest representable exponent for the data type. The destination is unaffected and the saved condition codes are UNPREDICTABLE. The saved PC points to the instruction causing the fault. In the case of a POLY instruction, the instruction is suspended with FPD set (see VAX Architecture Handbook for details). The type code pushed on the stack is 8.

Floating Divide By Zero Fault
A floating divide by zero fault is an exception indicating that the last instruction executed had a floating zero divisor. The quotient operand is unaffected and the saved condition codes are UNPREDICTABLE. The saved PC points to the instruction causing the fault. The type code pushed on the stack is 9.

Floating Underflow Fault
A floating underflow fault is an exception indicating that the last instruction executed resulted (after normalization and rounding) in an exponent less than the smallest representable exponent for the data type. The destination operand is unaffected and the saved condition codes are UNPREDICTABLE. The saved PC points to the instruction causing the fault. The type code pushed on the stack is A.

The floating underflow fault does not appear on VAX-11/780 systems.

SYSTEM CONTROL BLOCK (SCB)
The System Control Block is a page containing the vectors by which exceptions and interrupts are dispatched to the appropriate service routines. On the VAX-11/750, the SCB has a second page that contains the addresses of interrupt service routines for UNIBUS devices.

System Control Block Base (SCBB)
The SCBB is a privileged register containing the physical address of the System Control Block, which must be page-aligned. Figure 3-6 illustrates the System Control Block Base Register.

![Figure 3-6 System Control Block Base Register (Read-Only)](image-url)
Exceptions and Interrupts

At bootstrap time, the contents of SCBB are UNPREDICTABLE. SCBB must specify a valid address in physical memory or the processor operation is UNDEFINED.

Vectors
A vector is a longword in the SCB that is examined by the processor when an exception or interrupt occurs, to determine how to service the event.

Separate vectors are defined for each interrupting device controller and each class of exception. The following paragraphs describe how the hardware interprets each vector.

The contents of bits <1:0> can be interpreted as:

0  Service this event on the kernel stack unless already running on the interrupt stack, in which case service on the interrupt stack. Behavior of the processor is UNDEFINED for a kernel stack not valid exception with this code.

1  Service this event on the interrupt stack. If this event is an exception, the IPL is raised to $1F_{16}$.

2  Service this event in user control store, passing bits <15:2> to the microcode there. If user control store does not exist or is not loaded, the operation is UNDEFINED, and the VAX processor halts.

3  Operation UNDEFINED. Reserved to DIGITAL.

For codes 0 and 1, bits <31:2> contain the virtual address of the service routine, which must begin on a longword boundary and will ordinarily be in the system space. CHMx is serviced on the stack selected by the new mode.

System Control Block (Exception and Interrupt Vectors)

<table>
<thead>
<tr>
<th>Vector (hexadecimal)</th>
<th>Name</th>
<th>Type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Unused</td>
<td></td>
<td>Reserved to DIGITAL. Length parameter and error specific data are pushed onto the stack, if possible. The VAX processor is sometimes restartable, if the cause of the machine check was a cache parity, translation buffer parity, or uncorrected data error. Vector &lt;1:0&gt; must be 1 for meaningful operation. IPL is raised to $1F_{16}$. The number of bytes of parameters is pushed onto the stack.</td>
</tr>
<tr>
<td>04</td>
<td>Machine Check</td>
<td>abort/ trap</td>
<td></td>
</tr>
</tbody>
</table>
### Exceptions and Interrupts

<table>
<thead>
<tr>
<th>Vector (hexadecimal)</th>
<th>Name</th>
<th>Type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>08</td>
<td>Kernel Stack Not Valid</td>
<td>abort</td>
<td>Vector &lt;1:0&gt; must be 1 for meaningful operation. IPL is raised to 1E_{16}. There are zero parameters.</td>
</tr>
<tr>
<td>0C</td>
<td>Power Fail</td>
<td>interrupt</td>
<td>IPL is raised to 1E_{16}. There are zero parameters.</td>
</tr>
<tr>
<td>10</td>
<td>Reserved/Privileged Instru-</td>
<td>fault</td>
<td>Opcodes reserved to DIGITAL and privileged instructions. There are zero parameters. On the VAX-11/750, attempting an REI with PSL&lt;FPD&gt; = 1 to an instruction that does not set FPD will also be reported here.</td>
</tr>
<tr>
<td>14</td>
<td>Customer Reserved Instruc-</td>
<td>fault</td>
<td>XFC instruction. There are zero parameters.</td>
</tr>
<tr>
<td>18</td>
<td>Reserved Operand</td>
<td>fault/abort</td>
<td>There are zero parameters.</td>
</tr>
<tr>
<td>1C</td>
<td>Reserved Addressing Mode</td>
<td>fault</td>
<td>There are zero parameters. For greater detail refer to the Architecture Handbook.</td>
</tr>
<tr>
<td>20</td>
<td>Access Control Violation</td>
<td>fault</td>
<td>The virtual address causing the fault is pushed onto the kernel stack. There are two parameters. See Chapter 4 for details.</td>
</tr>
<tr>
<td>24</td>
<td>Translation Not Valid</td>
<td>fault</td>
<td>The virtual address causing the fault is pushed onto stack. The two parameters are identical to those for access control violations. See Chapter 4 for details.</td>
</tr>
<tr>
<td>28</td>
<td>Trace Pending (TP)</td>
<td>fault</td>
<td>This vector is used for a trace fault. There are zero parameters. For greater detail refer to the Architecture Handbook.</td>
</tr>
<tr>
<td>2C</td>
<td>Breakpoint Instruc-</td>
<td>fault</td>
<td>This vector is used for a breakpoint fault. There are zero parameters.</td>
</tr>
<tr>
<td>30</td>
<td>Compatibility</td>
<td>fault/abort</td>
<td>A type code is pushed onto the stack. There is one parameter.</td>
</tr>
<tr>
<td>34</td>
<td>Arithmetic</td>
<td>trap/fault</td>
<td>A type code is pushed onto the stack. There is one parameter.</td>
</tr>
<tr>
<td>38-3C</td>
<td>Unused</td>
<td></td>
<td>Reserved to DIGITAL.</td>
</tr>
</tbody>
</table>
## Exceptions and Interrupts

<table>
<thead>
<tr>
<th>Vector (hexadecimal)</th>
<th>Name</th>
<th>Type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>CHMK</td>
<td>trap</td>
<td>The operand word is sign-extended and pushed onto the kernel stack. Vector &lt;1:0&gt; MBZ. There is one parameter.</td>
</tr>
<tr>
<td>44</td>
<td>CHME</td>
<td>trap</td>
<td>The operand word is sign-extended and pushed onto the executive stack. Vector &lt;1:0&gt; MBZ. There is one parameter.</td>
</tr>
<tr>
<td>48</td>
<td>CHMS</td>
<td>trap</td>
<td>The operand word is sign-extended and pushed onto the supervisor stack. Vector &lt;1:0&gt; MBZ. There is one parameter.</td>
</tr>
<tr>
<td>4C</td>
<td>CHMU</td>
<td>trap</td>
<td>The operand word is sign-extended and pushed onto the user stack. Vector &lt;1:0&gt; MBZ. There is one parameter.</td>
</tr>
<tr>
<td>50</td>
<td>SBI SILO Compare</td>
<td>interrupt</td>
<td>IPL is 19&lt;sub&gt;16&lt;/sub&gt;. VAX-11/780 only.</td>
</tr>
<tr>
<td>54</td>
<td>Corrected Memory</td>
<td>interrupt</td>
<td>IPL is 1A&lt;sub&gt;16&lt;/sub&gt;. Also used for Read Data Substitute on VAX-11/780.</td>
</tr>
<tr>
<td>5C</td>
<td>Read Data</td>
<td>interrupt</td>
<td>IPL is 1B&lt;sub&gt;16&lt;/sub&gt;. VAX-11/780 only.</td>
</tr>
<tr>
<td>60</td>
<td>Memory Write</td>
<td>interrupt</td>
<td>IPL is 1D&lt;sub&gt;16&lt;/sub&gt;.</td>
</tr>
<tr>
<td>64-80</td>
<td>Unused</td>
<td></td>
<td>Reserved to DIGITAL.</td>
</tr>
<tr>
<td>84</td>
<td>Software Level 1</td>
<td>interrupt</td>
<td>There are zero parameters.</td>
</tr>
<tr>
<td>88</td>
<td>Software Level 2</td>
<td>interrupt</td>
<td>Ordinarily used for AST delivery. There are zero parameters.</td>
</tr>
<tr>
<td>8C-BC</td>
<td>Software Level 3</td>
<td>interrupt</td>
<td>There are zero parameters.</td>
</tr>
<tr>
<td>90-BC</td>
<td>Software Levels 4-</td>
<td>interrupt</td>
<td>There are zero parameters.</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C0</td>
<td>Interval Timer</td>
<td>interrupt</td>
<td>IPL is 18&lt;sub&gt;16&lt;/sub&gt;. There are zero parameters.</td>
</tr>
<tr>
<td>C4-DC</td>
<td>Unused</td>
<td></td>
<td>Reserved to DIGITAL.</td>
</tr>
<tr>
<td>E0-EC</td>
<td>Unused</td>
<td></td>
<td>Reserved to CSS/ customers.</td>
</tr>
<tr>
<td>Vector (hexadecimal)</td>
<td>Name</td>
<td>Type</td>
<td>Notes</td>
</tr>
<tr>
<td>----------------------</td>
<td>------</td>
<td>------</td>
<td>-------</td>
</tr>
<tr>
<td>F0</td>
<td>Console Storage Device (TU58) Transmit</td>
<td>interrupt</td>
<td>IPL is $17_{16}$, VAX-11/750 only. There are zero parameters.</td>
</tr>
<tr>
<td>F4</td>
<td>Console Storage Device (TU58) Transmit</td>
<td>interrupt</td>
<td>IPL is $17_{16}$, VAX-11/750 only. There are zero parameters.</td>
</tr>
<tr>
<td>F8</td>
<td>Console Terminal Receive</td>
<td>interrupt</td>
<td>IPL is $14_{16}$, There are zero parameters.</td>
</tr>
<tr>
<td>FC</td>
<td>Console Terminal Receive</td>
<td>interrupt</td>
<td>IPL is $14_{16}$, There are zero parameters.</td>
</tr>
<tr>
<td>100-3FC</td>
<td>Device Vectors</td>
<td></td>
<td>In the VAX-11/780 processor, only interrupt priority levels 14 to 17 are available to a NEXUS external to the CPU, and there is a limit of 16 such NEXUSs. A NEXUS is a connection on the SBI, which is the internal interconnection structure. The NEXUS vectors are assigned as follows: 100-13C IPL 14_{16} NEXUS 0-15 140-17C IPL 15_{16} NEXUS 0-15 180-1BC IPL 16_{16} NEXUS 0-15 1C0-1FC IPL 17_{16} NEXUS 0-15 In the VAX-11/750 processor, UNIBUS devices interrupt the processor directly. The vector is determined by adding 200_{16} to the vector supplied by the device. Only SCB vectors in the range 200 to 3FC are allowed. Interrupt priority levels 14 to 17 correspond to UNIBUS BR4 to BR7.</td>
</tr>
</tbody>
</table>

**STACKS**

At any time, the processor is either in a process context with the Interrupt Stack (IS) = 0, in one of four modes (kernel, executive, supervisor, user), or in the system-wide interrupt service context (IS = 1) that operates with kernel privileges. There is a stack pointer (SP) associated with each of these five states, and any time the processor changes states, the SP (R14) is stored in the process context stack pointer for the old state and loaded from that for the new state. The
process context stack pointers (KSP = kernel, ESP = executive, SSP = supervisor, USP = user) are allocated in the hardware PCB. In addition, VAX systems keep copies of the four per-process stack pointers in privileged registers. These registers are accessed during stack switch operations. The stack pointers in the hardware PCB are only referenced at context switch time by SVPCTX and LDPCTX.

Stack Residency
The USER, SUPER, and EXEC stacks need not be resident in main memory. The kernel can bring in or allocate process stack pages as address translation not valid faults occur. However, the kernel stack for the current process and the interrupt stack (which is process-independent) must be resident and accessible. Translation not valid and access control violation faults occurring on references to either of these stacks constitute serious system failures, from which recovery is impossible.

If either of these faults occurs on a reference to the kernel stack, the processor aborts the current sequence and initiates a kernel stack not valid abort on hardware level 1F₁₆. If either of these faults occurs on a reference to the interrupt stack, the processor halts.

The kernel stack for processes other than the current one need not be resident, but it must be resident before the software's process dispatcher selects a process to run. Furthermore, any mechanism using translation not valid or access control violation faults to gather process statistics must exercise care not to invalidate kernel stack pages.

Stack Alignment
Except on CALLx instructions, the hardware makes no attempt to align the stacks. For best performance, the software should align the stack on a longword boundary and allocate the stack in longword increments. The following instructions are recommended for pushing bytes and words onto the stack and popping them off in order to keep it longword-aligned:

- Convert byte to long (CVTBL)
- Convert long to byte (CVTLB)
- Convert long to word (CVTLW)
- Convert word to long (CVTWL)
- Move zero-extended byte to long (MOVZBL)
- Move zero-extended word to long (MOVZWL)

Stack Status Bits
The interrupt stack bit (IS) and current mode bits in the privileged
Exceptions and Interrupts

Processor Status Longword (PSL) specify which of the five stack pointers is currently in use as follows:

<table>
<thead>
<tr>
<th>IS</th>
<th>Mode</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>ISP</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>KSP</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>ESP</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>SSP</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>USP</td>
</tr>
</tbody>
</table>

The processor does not allow the current mode to be nonzero when IS = 1. This is achieved by clearing the mode bits when taking an interrupt or exception, and by causing a reserved operand fault if REI attempts to load a PSL in which both IS and mode are nonzero.

The stack to be used for an interrupt or exception is selected by the current PSL<IS> and bits <1:0> of the vector for the event as follows:

```
VECTOR<1:0>
00 01
  0 KSP ISP
PSL<IS>
  1 ISP ISP
```

Figure 3-7  Stack Selection

Values 10 and 11 of the vector <1:0> are for user control store and HALT, respectively.

Accessing Stack Registers
The processor implements five privileged registers to allow access to each stack pointer. These registers always access the specified pointer, even for the current mode. Because the per-process stack pointers are implemented as internal registers, the MTPR and MFPR of these registers do not access the hardware PCB. The register numbers were chosen to be the same as PSL<26:24>. The previous stack pointer is the same as PSL<23:22> unless PSL<IS> is set. At bootstrap time, the contents of all stack pointers are UNPREDICTABLE. Figure 3-8 illustrates the process stack pointer.
SERIALIZATION OF EXCEPTIONS AND INTERRUPTS
The sequence in which recognition of simultaneously occurring interrupts and exceptions takes place is:
1. Machine check exception.
2. Arithmetic exceptions.
3. Console halt or higher priority interrupt. (The order in which console halt and interrupt recognition occur is not dictated by the VAX architecture. Some future VAX machines may not take these in the same order as the VAX-11/750 or VAX-11/780, which take console halts before interrupts.)
4. Trace fault (only one per instruction).
5. Start instruction execution or restart suspended instruction.

Suspended Instructions
The VAX architecture allows certain instructions to be suspended at well-defined intermediate points in their execution in order to take memory management faults, console halts, or interrupts. In this case, the hardware uses PSL<TP> and PSL<T> to ensure that no additional trace faults occur when the suspended instruction is resumed.

As an example, an instruction starts with T = 1, it gets an arithmetic trap and an interrupt request is recognized. The following sequence occurs:
1. The instruction finishes, storing all its results.
2. The overflow trap sequence is initiated, pushing the PC and PSL (with TP = 1), loading a new PC from the vector, and creating a new PSL.
3. The interrupt sequence is initiated, pushing the PC and PSL appropriate to the trap service routine, loading a new PC from the vector, and creating a new PSL.

4. If a higher priority interrupt is noticed, the first instruction of the interrupt service routine is not executed. Instead, the PC and PSL appropriate to that routine are saved as part of the new interrupt initiation. The original interrupt service routine will then be executed when the higher priority routine terminates via REI.

5. The interrupt service routine runs and exits with REI.

6. The trap service routine runs and exits with REI, which finds a PSL having TP = 1.

7. The trace occurs, again pushing PC and PSL, but this time with TP = 0.

8. Trace service routine runs and exits with REI.

INITIATE EXCEPTION OR INTERRUPT
The following pseudo-ALGOL describes the sequence of events initiated by exceptions and interrupts. If a higher priority interrupt condition occurs after the start of this sequence, the interrupt will not be taken until the sequence completes.

```
vector ← SCB[vector];  !get correct vector
if vector<1:0> EQLU 3 then {UNDEFINED};
if vector<1:0> EQLU 0 and {machine-check or
   kernel-stack-not-valid} then {UNDEFINED};
if vector<1:0> NEQU 0 and {CHMx} then {UNDEFINED};
if vector<1:0> EQLU 2 then
   begin
      if {writable control store exists and is loaded}
         then {enter writable control store}
         else {UNDEFINED};
   end;
if PSL<IS> EQLU 0 then  !switch stacks
   begin
      PSL<current_mode>–SP ← SP; !save old SP
      if vector<1:0> EQLU 1 then
         SP ← ISP;
      else
         SP ← new_mode_SP; !kernel_SP unless CHMx
      end;
      –(SP) ← PSL;     !on a fault or abort, the saved
         !condition codes are UNPREDICTABLE
      –(SP) ← PC;      !as backed up, if necessary
```
Exceptions and Interrupts

{push parameters if any};
PSL<CM,TP,FPD,DV,FU,IV,T,N,Z,V,C> ← 0;  //clean out PSL
if {interrupt} then
  PSL<previous_mode> ← 0;  //kernel mode
else
  PSL<previous_mode> ← PSL<current_mode>;
PSL<current_mode> ← new_mode;  //kernel_mode unless CHMx
if {interrupt} then  //set new IPL
  PSL<IPL> ← new_IPL
else
  if vector<1:0> EQLU 1 then
    PSL<IPL> ← 31;  //11F (hex)
if vector<1:0> EQLU 1 then PSL<IS> ← 1;  //otherwise keep old IS
PC ← vector<31:2> ' 0<1:0>;  //longword-aligned
{enable interrupts};

Condition Codes

N ← 0;
(if vector <1:0> Z ← 0;
code is 0 or 1): V ← 0;
C ← 0;

Exceptions: interrupt stack not valid
kernel stack not valid

Description: A longword vector in the system control block, indexed by the exception or the interrupt being processed, determines the handling. If the processor is not executing on the interrupt stack, the current stack pointer is saved and the new stack pointer is fetched. The old PSL is pushed onto the new stack. The PC is backed up (unless this is an interrupt between instructions, a trace pending fault, or a trap) and is pushed onto the new stack. The PSL is initialized to a specific state. The IPL is changed if this is an interrupt or if it is an exception with vector <1:0> code 1. Any parameters are pushed. Except for interrupts, the previous mode in the new PSL is set to the old value of the current mode. Finally, the PC is changed to point to the longword indicated by the vector <31:2>.

Notes:
1. Interrupts are disabled during this sequence.
2. If the vector <1:0> code is invalid, the behavior is UNDEFINED.
3. On a fault, abort or interrupt, the saved condition codes are UNPREDICTABLE. On an abort, fault or interrupt that sets FPD (first part done), the general registers except the PC, SP and (unless modified by the instruction) FP are UNPREDICTABLE, unless the instruction description specifies a setting. On a kernel stack not valid abort, both the SP and FP are UNPREDICTABLE. In this case, UNPREDICTABLE means unspecified; upon REI the instruction behavior and results are predictable. This implies that processes stopped with FPD set cannot be resumed on processors of a different type or engineering change level.

4. If the processor gets an access control violation or a translation not valid condition while attempting to push information onto the kernel stack, a kernel stack not valid abort is initiated and the IPL is changed to $1F_{16}$. The additional information, if any, associated with the original exception is lost. However, the PSL and PC are pushed onto the interrupt stack with the same values as would have been pushed onto the kernel stack.

5. If the processor gets an access control violation or a translation not valid condition while attempting to push information onto the interrupt stack, the processor is halted and only the states of the ISP, PC, and PSL are definitely correct for subsequent analysis. The PSL and PC have the values that would have been pushed onto the interrupt stack.

6. As usual for faults, if the processor gets an access violation or translation not valid fault during the execution of a CHMx instruction, it saves the PC, PSL, and leaves the SP as it was at the beginning of the instruction except for any pushes onto the kernel stack.

7. The value of the PSL<TP> that is saved on the stack is as follows:

   - fault: clear
   - trace: clear
   - interrupt: from PSL<TP> (if after traps, before trace) clear (otherwise)
   - abort: from PSL<TP>
   - trap: from PSL<TP>
   - CHMx: from PSL<TP>
   - BPT, XFC: clear
   - reserved: clear
   - instruction: clear
Exceptions and Interrupts

8. The value of the PC that is saved on the stack points to the following:
   - fault: instruction faulting
   - trace: next instruction to execute
   - interrupt: instruction interrupted or next instruction to execute
   - abort: instruction aborting or detecting kernel stack not valid (not ensured on machine check)
   - trap: next instruction to execute
   - CHMx: next instruction to execute
   - BPT, XFC: BPT, XFC instruction
   - reserved instruction: first byte of the reserved instruction

9. The noninterrupt stack pointers are fetched and stored in privileged internal processor registers. LDPCTX loads these registers from the hardware PCB and SVPCTX saves these registers into the hardware PCB. MFPR and MTPR access only the privileged registers.

RETURN FROM EXCEPTION OR INTERRUPT (REI)
The following pseudo-ALGOL describes the sequence of events following the servicing of an exception or interrupt. Control is returned to the instruction being executed at the time of the exception or interrupt.

tmp1 ← (SP) +;  // Pick up saved PC
[tmp2<current_mode> LSSU <current_mode>]} or
[tmp2<IS> EQLU 1 and PSL<IS>EQLU 0] or
[tmp2<IS> EQLU 1 and
[tmp2<current_mode> NEQU 0] or
[tmp2<IS> EQLU 1 and tmp2<IPL> EQLU 0] or
[tmp2<IPL> GRTU 0 and
[tmp2<current_mode> NEQU 0] or
[tmp2<previous_mode> LSSU
[tmp2<current_mode>]} or
[tmp2<IPL> GTRU PSL<IPL>]} or
[tmp2<PSL_MBZ> NEQU 0] then
[reserved operand fault];
Exceptions and Interrupts

if (tmp2<CM> EQLU 1) and (tmp2<FPD,IS,DV,FU,IV> NEQU 0) or (tmp2<current_mode> NEQU 3) then [reserved operand fault];
[disallow interrupts];
if PSL<IS> EQLU 1 then ISP ← SP  !save old stack pointer
else PSL<current_mode>_SP ← SP;
if PSL<TP> EQLU 1 then tmp2<TP> ← 1;

!TP←TP or stack TP
PC ← tmp1;
PSL ← tmp2;
if PSL<IS> EQLU 0 then
    begin
        SP ← PSL<current_mode>_SP;  !switch stack
        if PSL<current_mode> GEQU ASTLVL

!check for AST delivery
    then [request interrupt at IPL 2];
end;
[allow interrupts];

**Condition Codes:**

- N ← saved PSL<3>
- Z ← saved PSL<2>
- V ← saved PSL<1>
- C ← saved PSL<0>

**Exceptions:** reserved operand

**Description:** A longword is popped from the current stack and held in a temporary PC. A second longword is popped from the current stack and held in a temporary PSL. Validity of the popped PSL is checked. The current stack pointer is saved and a new stack pointer is selected according to the new PSL current_mode and IS fields. The level of the highest privilege AST is checked against the current access mode to see whether a pending AST can be delivered. Execution resumes with the instruction being executed at the time of the exception or interrupt. Any instruction lookahead in the processor is reinitialized.

**Notes:**

1. The exception or interrupt service routine is responsible for re-
Exceptions and Interrupts

storing any registers saved and removing any parameters from the stack.

2. As usual for faults, any access violation or translation not valid conditions on the stack pops restore the stack pointer and fault.

3. The noninterrupt stack pointers are fetched and stored in privileged internal processor registers. LDPCTX loads these registers from the hardware PCB and SVPCTX saves these registers into the hardware PCB. MFPR and MTPR access only the privileged registers.

Differences Between the VAX-11/750 and the VAX-11/780
The information in this difference list will only be useful to system programmers who are not using VAX/VMS but are writing their own system software. In addition to a different format for the machine check logout stack, the VAX-11/750 and VAX-11/780 differ in the following specific ways:

VAX-11/750

The integral TU58 tape cartridge drive generates a console storage device interrupt at IPL 17. Vectors are F0\textsubscript{16} for Console Storage Device Receive, and F4\textsubscript{16} for Console Storage Device Transmit.

Floating overflow and floating underflow generate faults. The benefit of this feature is that continued operation is possible in some circumstances.

The System Control Block (SCB) implements additional pages which contain the addresses of interrupt service routines for UNIBUS devices.

VAX-11/780

The integral floppy disk generates a device interrupt at IPL 14. Vectors are F8\textsubscript{16} for receive and FC\textsubscript{16} for transmit.

Floating overflow and floating underflow generate traps.

SCB is a single page because interrupt service routines are not directly vectored.
CHAPTER 4
MEMORY MANAGEMENT

FEATURES

Memory management hardware

Four hierarchical privilege modes

Translation buffer

Map enable register

BENEFITS

Translates virtual addresses to physical addresses to allow large data structures to be run on smaller memory configurations.

Permits efficient sharing of instructions and data.

Protect multiple processes resident in memory and improve software reliability by controlling memory access.

Stores frequently used address translations so that fewer memory references are required when repeatedly accessing the same pages.

Permits memory management to be disabled for diagnosing hardware and software faults.

INTRODUCTION

Memory management describes the hardware and software that control the allocation and use of physical memory for the VAX family of processors. In a typical multiprogramming system, several processes may reside in physical memory at the same time. Therefore, to ensure that one process will not affect other processes or the operating system, memory protection is provided. To further improve software reliability, four hierarchical (privilege) modes are provided to control memory access. They are, from most to least privileged, kernel, executive, supervisor, and user. Protection is specified at the individual page level, where a page may be inaccessible, read-only, or read/write for each of the four access modes. Any location accessible to a less privileged mode is also accessible to all more privileged modes. Furthermore, for each access mode, any location that is writable is also readable.

While an image is being executed by the CPU, virtual addresses are generated. However, before these addresses can be used to access
instructions and data, they must be translated into physical addresses. Memory management software maintains tables of mapping information (page tables) that keep track of where each 512-byte virtual page is located in physical memory. The CPU memory management hardware uses this mapping information when it translates virtual addresses to physical addresses.

Therefore, memory management is the scheme that provides both memory protection and memory mapping mechanisms for VAX processors. The memory management scheme has been designed and implemented to achieve the following goals:

- Provide a large address space for instructions and data
- Allow data structures up to one billion bytes
- Provide convenient and efficient sharing of instructions and data
- Contribute to software reliability

A virtual memory system is used to provide a large address space, while allowing programs to run on smaller memory size hardware configurations. Programs are executed in an execution environment termed a process. The software operating system uses the mechanisms described in this chapter to provide each process with a potential 4 billion byte virtual address space.

The virtual address space is divided into two equal sized address spaces, the process address space and the system address space. The system address space is the same for all processes. The operating system is in the lower half of the system address space and is implemented as a series of callable procedures. Thus all of the system code is available to all other system and user codes using a simple CALL. The upper half of the system space is reserved for future use. The process address space is separate for each process. However, several processes may have access to the same page, thus providing controlled sharing.

VIRTUAL ADDRESS SPACE

The address space seen by the programmer is a linear array of over 4 billion \(2^{32}\) bytes. This results from the fact that a virtual address is 32 bits in length. The virtual address space consists of 512-byte units called pages. The page is the basic unit of both relocation and protection.

This virtual address space is too large to be contained in any presently available main memory. Therefore memory management provides the mapping mechanism to map the active part of the virtual address space to the available physical address space. Memory management also provides page protection between processes. The operating
system controls the memory management tables that map virtual addresses into physical memory addresses. The inactive but used parts of the virtual address space are mapped onto external storage media via the operating system. Figure 4-1 illustrates the virtual address space.

![Virtual Address Space Diagram](image)

**Figure 4-1 Virtual Address Space**

**Process Space**  
The lower half of virtual address space is termed *process space*. Each process has a separate address translation map for per-process space, so the per-process spaces of all processes are completely noncontiguous. The address map for per-process space is context-switched when the process running on the system is changed.

This process space is further divided by bit <30> into two regions termed the P0 and P1 regions of the process virtual address space. These regions will be described in greater detail later in this chapter.

**System Space**  
The upper half of virtual address space is termed *system space*. All processes use the same address translation map for system space, so
system space is shared among all processes. The address map for system space is not context-switched.

Page Protection
Independent of its location in the virtual address space, a page may be protected according to its use. Thus, even though all of the system space is shared, in that the program may generate any address, the program may be prevented from modifying, or even accessing portions of it. A program may also be prevented from accessing or modifying portions of per-process space.

For example, in system space, scheduling queues are highly protected, whereas library routines may be executable by code of any privilege. Similarly per-process accounting information may be in per-process space, but highly protected, while normal user code in per-process space is executable at low privilege.

Virtual Address
In order to reference each instruction and operand in memory, the processor generates a 32-bit virtual address as illustrated in Figure 4-2.

![Virtual Address Diagram](image)

**Figure 4-2  Virtual Address**

**Bit: 31:9  Name:** Virtual Page Number  
**Function:** The virtual page number field specifies the virtual page to be referenced. There are 8,388,608 pages of 512 bytes each in the virtual address space.

When bit 31 is one, the address is in the system space. When bit 31 is zero, the address is in the per-process space.

Within the per-process space, bit 30 distinguishes between the
Memory Management

program and control regions. When bit 30 is one, the control region is referenced, and when it is zero, the program region is referenced.

**Bit:** 8:0  **Name:** Byte  
**Function:** The byte number field specifies the byte address within the page. A page contains 512 bytes.

**Virtual Address Space Layout**  
Access to each of the three regions (P0, P1, System) is controlled by a length register (P0LR, P1LR, SLR). Within the limits set by the length registers, the access is controlled by a page table that specifies the validity, access requirements, and location of each page in the region.

**ACCESS CONTROL**  
Access control is the function of validating whether a particular type of memory access is to be allowed to a particular page. Every page has associated with it a protection code that specifies for each mode whether or not read or write references are allowed. Additionally, each address is checked to make certain that it lies within the P0, P1, or system region.

**Mode**  
There are four hierarchically ordered modes in the processor. The modes in the order of most to least privileged are:

0  Kernel. Used by the kernel of the operating system for page management, scheduling, and I/O drivers.

1  Executive. Used for many of the operating system service calls including the record management system.

2  Supervisor. Used for such services as command interpretation.

3  User. Used for user level code, utilities, compilers, debuggers, etc.

The mode at which the processor is currently running is stored in the current mode field of the Processor Status Longword (PSL).

**Protection Code**  
Associated with each page is a protection code (located within the page table entry for that page) that describes the accessibility of the page for each mode. The protection codes available allow choice of protection for each access level within the following limits:

1. Each level's access can be read/write, read only, or no access.
2. If any level has read access then all more privileged levels also have read access.
3. If any level has write access then all more privileged levels also have write access.
Memory Management

This results in 15 possibilities. The protection code is encoded in a 4-bit field in the Page Table Entry described in Table 4-1.

Table 4-1  Protection Codes

<table>
<thead>
<tr>
<th>CODE</th>
<th>MNEMONIC</th>
<th>DECIMAL</th>
<th>BINARY</th>
<th>K</th>
<th>E</th>
<th>S</th>
<th>U</th>
<th>COMMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NA</td>
<td>0000</td>
<td>0001</td>
<td>NO ACCESS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>UNPREDICTABLE</td>
<td>0001</td>
<td>UNPREDICTABLE</td>
<td>RESERVED</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>KW</td>
<td>0010</td>
<td>RW</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>AWAYS ACCESS</td>
</tr>
<tr>
<td>3</td>
<td>KR</td>
<td>0011</td>
<td>R</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>ACCESS</td>
</tr>
<tr>
<td>4</td>
<td>UW</td>
<td>0100</td>
<td>RW</td>
<td>RW</td>
<td>RW</td>
<td>RW</td>
<td>RW</td>
<td>ALL ACCESS</td>
</tr>
<tr>
<td>5</td>
<td>EW</td>
<td>0101</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>ACCESS</td>
</tr>
<tr>
<td>6</td>
<td>ERKW</td>
<td>0110</td>
<td>RW</td>
<td>R</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>ACCESS</td>
</tr>
<tr>
<td>7</td>
<td>ER</td>
<td>0111</td>
<td>R</td>
<td>R</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>ACCESS</td>
</tr>
<tr>
<td>8</td>
<td>SW</td>
<td>1000</td>
<td>RW</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
<td>-</td>
<td>ACCESS</td>
</tr>
<tr>
<td>9</td>
<td>SREW</td>
<td>1001</td>
<td>RW</td>
<td>RW</td>
<td>R</td>
<td>-</td>
<td>-</td>
<td>ACCESS</td>
</tr>
<tr>
<td>10</td>
<td>SRK</td>
<td>1010</td>
<td>RW</td>
<td>R</td>
<td>R</td>
<td>-</td>
<td>-</td>
<td>ACCESS</td>
</tr>
<tr>
<td>11</td>
<td>SR</td>
<td>1011</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>-</td>
<td>-</td>
<td>ACCESS</td>
</tr>
<tr>
<td>12</td>
<td>URSW</td>
<td>1100</td>
<td>RW</td>
<td>RW</td>
<td>RW</td>
<td>-</td>
<td>-</td>
<td>ACCESS</td>
</tr>
<tr>
<td>13</td>
<td>UREW</td>
<td>1101</td>
<td>RW</td>
<td>RW</td>
<td>R</td>
<td>-</td>
<td>-</td>
<td>ACCESS</td>
</tr>
<tr>
<td>14</td>
<td>URKW</td>
<td>1110</td>
<td>RW</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>ACCESS</td>
</tr>
<tr>
<td>15</td>
<td>UR</td>
<td>1111</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>ACCESS</td>
</tr>
</tbody>
</table>

- = no access    K = Kernel
R = read only    E = Executive
RW = read/write  S = Supervisor
U = User

Software symbols are defined using PTE$K$ as a prefix to the mnemonics listed in Table 4-1.

**NOTE**
For all occurrences within this chapter, the parenthesis indicate contents of, the angle brackets indicate referenced bits, and the apostrophe indicates concatenation. The exclamation point indicates comments.

**Length Violation**
Every virtual address is constrained to lie within one of the valid addressing regions (P0, P1, or System). The algorithm for making these checks is a simple limit check. The formal notation for this check is:
case VAddr <31:30>
    set
    (0): !P0 region
        if ZEXT (VAddr<29:9>) GEQU P0LR
            then (length violation);
    (1): !P1 region
        if ZEXT (VAddr<29:9>) LSSU P1LR
            then (length violation);
    (2): !S region
        if ZEXT (VAddr<29:9>) GEQU SLR
            then (length violation);
    (3): !reserved region
        (length violation);

Access Control Violation Fault
An access control fault occurs if the current mode of the PSL and the protection field(s) for the page(s) about to be accessed indicate that the access would be illegal. A fault of this type will occur if the address causes a length violation to occur.

ADDRESS TRANSLATION
The action of translating a virtual address to a physical address is governed by the setting of the Memory Mapping Enable (MME) bit. When MME is 0, the low order bits of the virtual address are the physical address and there is no page protection. The disabling of memory management is often useful to Field Service engineers in diagnosing hardware faults; however, this is not a normal user mode. This section describes the address translation process when MME is 1. A virtual to physical address translation is given in Appendix D of this book.

The address translation mechanism is presented with a virtual address, an intended access (read or write) and a mode against which to check that access. If the access is allowed and the address maps without faulting, the output of this routine is a physical address corresponding to the specified virtual address. The mode that is used is
normally the current mode field of the PSL, but per-process page
table entry references use kernel mode.

The intended access is read if the operation to be performed is a read.
The intended access is write if the operation to be performed is a write.
If, however, the operation to be performed is a modify (i.e. read fol-
lowed by write) the intended access for the read portion is specified as
a write.

Page Table Entry (PTE)
All virtual addresses are translated to physical addresses by means of
a page table entry (PTE). The page table entry is described in Figure 4-3.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PROT</td>
<td>M</td>
<td>OWN</td>
<td>0</td>
<td></td>
<td></td>
<td>PFN</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4-3 Page Table Entry

Bit: 31  Name: Valid bit
Function: Governs the validity of the M bit and PFN field. V=1 for
valid; V=0 for not valid.

Bit: 30:27 Name: Protection field
Function: This field is always valid and is used by the hardware even
when V=0.

Bit: 26  Name: Modify bit
Function: Set if page has already been recorded as modified. M=0 if
page has not been recorded as modified. Used by hardware only if V=
1. Hardware sets this bit on a valid, access-allowed memory access
associated with a modify or write access, and optionally on a PROBEW
or implied probe-write. If a write or modify reference crosses a page
boundary and one page faults, it is UNPREDICTABLE whether the page
table entry M bit for the other page is set before the fault. It is
UNPREDICTABLE whether the modification of a process PTE M bit
causes modification of the system PTE that maps that process page
table. (Note that the update of the M bit is not interlocked in customer-
designed multiprocessor systems.)

Bit: 25  Name: Zero bit, reserved to DIGITAL
Function: This bit is reserved to DIGITAL and must be zero. The
hardware does not necessarily test that this bit is zero because the
PTE is established only by privileged software.
Memory Management

Bit: 24:23  **Name:** Owner bits, reserved  
**Function:** Reserved for software use. VAX/VMS uses these bits as the access mode of the owner of the page (i.e., the mode allowed to alter the page); not examined or altered by hardware.

Bit: 22:21  **Name:** Software bits, reserved to DIGITAL  
**Function:** These bits are reserved for DIGITAL software. The operating system software uses some combinations of the software bits to implement its page management data structures and functions. Among the functions implemented this way are initialize-pages-with-zeros, copy-on-reference, page sharing, and transitions between active and swapped-out states. VAX/VMS encodes these functions in PTEs whose Valid bit, PTE<31>, is a 0 and processes them whenever a page fault occurs.

Bit: 20:0  **Name:** Page Frame Number (PFN)  
**Function:** The upper 21 bits of the physical address of the base of the page. Used by hardware only if \( V=1 \).

**NOTE**
On VAX-11/780 systems bits \(<20:0>\) of the page table entry are used to determine the page frame number. On VAX-11/750 systems bits \(<14:0>\) determine the page frame number. The unused bits are zero.

Software symbols are defined for the described fields using PTE$ as the prefix.

**Protection Check Before Valid Check**
The page table entry has been defined as having a valid bit that only controls the validity of the modify bit and page frame number field. The protection field is defined as always being valid and checked first. This is so that programs run in user mode do not PROBE all around in the system region faulting all the swappable pages.

**SYSTEM SPACE ADDRESS TRANSLATION**
A virtual address with \(<31:30>=2\) is an address in the system virtual address space as illustrated in Figure 4-4.

![Figure 4-4 System Space Address](#)

67
The system virtual address space is defined by the system page table (SPT), which is a vector of page table entries (PTEs). The physical base address of the SPT is contained in the System Base Register (SBR) described in Figure 4-5. The size of the SPT in longwords, i.e., the number of PTEs, is contained in the System Length Register (SLR) described in Figure 4-6. The PTE addressed by the SBR maps the first page of system space, i.e., virtual byte address 80000000₁₆.

![Figure 4-5 System Base Register](image)

**NOTE**

On VAX-11/780 systems bits $<29:2>$ of the system base register make up the physical longword address. On VAX-11/750 systems bits $<23:2>$ determine the physical longword address. The unused bits must be zero.

![Figure 4-6 System Length Register](image)

The virtual page number is bits $<29:9>$ of the virtual address. Thus, there could be as many as $2^{21}$ physical pages in the system region. (Typically the value is in the range of a few hundred to a few thousand system pages.) A 22-bit length field is required to express the values 0 through $2^{21}$ inclusive. At bootstrap time, the content of both registers are UNPREDICTABLE. The translation from system virtual address to physical address is illustrated in Figure 4-7.

Thus, for the VAX-11/780, the arithmetic necessary to generate a physical address from a system region virtual address is:

\[
\text{SYS-PA} = (\text{SBR} + \text{SVA} <29:9>*4) <20:0> '\text{SVA} <8:0> \text{ !System Region}
\]

On VAX-11/750 systems the equivalent arithmetic is:

\[
\text{SYS-PA} = (\text{SBR} + \text{SVA} <29:9>*4) <14:0> '\text{SVA} <8:0> \text{ !System Region}
\]

The result of the translation is a 30-bit, $<29:0>$, physical address of data on the VAX-11/780 or a 24-bit, $<23:0>$, physical address on the VAX-11/750.
PROCESS SPACE ADDRESS TRANSLATION

The process virtual address space is split into two separately mapped regions according to the setting of bit 30 in the process virtual address. If bit 30 is zero, the P0 region of the address space is selected and if bit 30 is one, the P1 region is selected.

The P0 region of the address space defines a contiguous area that begins at the smallest address (zero) in the process virtual space and grows in the direction of larger addresses. In contrast, the P1 region of the address space defines a contiguous area that begins at the largest address ($2^{31} - 1$) in the process virtual space and grows in the direction of smaller addresses.

Each region (P0 and P1) of the process virtual space is described by page tables. In contrast with the system page table which is addressed with a physical address, these two page tables are addressed with virtual addresses in the system region of the virtual address space. Therefore, for process space, the address of the PTE is a virtual address in system space, and the fetch of the PTE is simply a fetch of a longword using a system virtual address.
Memory Management

Process page tables are addressed in virtual rather than physical space because a physically addressed process page table that required more than a page of PTEs (i.e., that mapped more than 64 Kb of process virtual space) also would require physically contiguous pages. Such a requirement would make dynamic allocation of process page table space more complex.

A process space translation that causes a translation buffer miss will usually cause one memory reference for a PTE. If the virtual address of the page containing the process PTE is also missing from the translation buffer, a second memory reference is required.

When a process page table entry is fetched, a reference is made to system space. This reference is made as a kernel read. Thus the system page containing a process page table is either no access (protection code zero) or will be accessible (protection code nonzero). Similarly, a check is made against the System Page Table Length Register (SLR). Thus, the fetch of an entry from a process page table can result in access or length violation faults.

P0 Space
The P0 region of the address space is mapped by the P0 page table (P0PT) that is defined by the P0 Base Register (P0BR) and the P0 Length Register (P0LR). P0BR contains a virtual address in the system half of virtual address space which is the base address of the P0 page table. The P0 Base Register is illustrated in Figure 4-8. P0LR contains the size of the P0 page table in longwords, i.e., the number of page table entries. The P0 Length Register is illustrated in Figure 4-9. The page table entry addressed by the P0 Base Register maps the first page of the P0 region of the virtual address space, i.e., virtual byte address zero.

![Figure 4-8 P0 Base Register](image)

![Figure 4-9 P0 Length Register](image)
Memory Management

The virtual page number is bits <29:9> of the virtual address. Thus, there could be as many as $2^{21}$ pages in the P0 region. A 22-bit length field is required to express the values 0 through $2^{21}$ inclusive. P0LR<26:24> are ignored on MTPR and read back zero on MFPR. At bootstrap time, the contents of both registers are UNPREDICTABLE. An attempt to load P0BR with a value less than $2^{31}$ results in a reserved operand fault. The translation from P0 Virtual Address to Physical Address is illustrated in Figure 4-10.

![Diagram of memory management process]

Figure 4-10  P0 Virtual to Physical Translation

Thus, for the VAX-11/780, the arithmetic necessary to generate a physical address from a P0 region virtual address is:

\[
PVA-PTE = (PVA<29:9>*4) + P0BR \quad !P0 \text{ Region}
\]

\[
PTE-PA = (SBR+PVA-PTE<29:9>*4)<20:0>'PVA-PTE<8:0>
\]

\[
PROC-PA = (PTE-PA)<20:0>'PVA<8:0>
\]
Memory Management

On the VAX-11/750 the equivalent arithmetic is:

\[
\begin{align*}
PVA & = (PVA <29:9> \times 4) + P0BR & \text{IP0 Region} \\
PTE-PA & = (SBR + PVA-PTE <29:0> \times 4) <14:0> \cdot PVA-PTE <8:0> \\
PROC-PA & = (PTE-PA) <14:0> \cdot PVA <8:0>
\end{align*}
\]

The result of the translation is a 30-bit, \(<29:0>\), physical address of data on the VAX-11/780 or a 24-bit, \(<23:0>\), physical address on the VAX-11/750.

**P1 Space**

The P1 region of the address space is mapped by the P1 page table (P1PT) that is defined by the P1 Base Register (P1BR) and the P1 Length Register (P1LR). Because P1 space grows from higher to lower addresses and because a consistent hardware interpretation of the base and length registers was desired, P1BR and P1LR describe the portion of P1 space that is not accessible. P1BR contains a virtual address of what would be the PTE for the first page P1—virtual byte address \(40000000_{16}\). The P1 Base Register is illustrated in Figure 4-11. P1LR contains the number of nonexistent PTEs. The P1 Length Register is illustrated in Figure 4-12.

![Figure 4-11  P1 Base Register (Read/Write)](image)

It should be remembered that the address in P1BR is not necessarily an address in system space, but all addresses of PTEs must be in system space.

![Figure 4-12  P1 Length Register (Read/Write)](image)

P1LR \(<31>\) is ignored on MTPR and reads back zero on MFPR. At bootstrap time, the contents of both registers are UNPREDICTABLE.
An attempt to load P1BR with a value less than $2^{31} - 2^{23}$ ($7F800000_{16}$) results in a reserved operand fault. The translation from P1 virtual address to physical address is illustrated in Figure 4-13.

![Diagram](image)

**Figure 4-13** P1 Virtual To Physical Translation

Thus, on the VAX-11/780, the arithmetic necessary to generate a physical address from a P1 region virtual address is:

\[
PVA\text{-}PTE = (PVA < 29:9 > \times 4) + P0BR \quad !P1\text{ Region} \\
PTE\text{-}PA = (SBR + PVA\text{-}PTE < 29:9 > \times 4) < 20:0 > 'PVA\text{-}PTE < 8:0 > \\
PROC\text{-}PA = (PTE\text{-}PA) < 20:0 > 'PVA < 8:0 >
\]
Memory Management

On the VAX-11/750 the equivalent arithmetic is:

\[
PVA-PTE= (PVA<29:9>*4)+P0BR \quad \text{!P1 Region} \\
PTE-PA= (SBR+PVA-PTE<29:0>*4)<14:0>'PVA-PTE<8:0> \\
PROC-PA= (PTE-PA)<14:0>'PVA<8:0>
\]

The result of the translation is a 30-bit, <29:0>, physical address of data on the VAX-11/780 or a 24-bit, <23:0>, physical address on the VAX-11/750.

MEMORY MANAGEMENT CONTROL
There are three additional privileged registers used to control the memory management hardware. One register is used to enable and disable memory management, the other two are used to clear the hardware's address translation buffer when a page table entry is changed.

Memory Management Enable
The Map Enable Register, MAPEN, contains the value of zero or one according to whether memory management is disabled or enabled respectively. At bootstrap time, this register is initialized to zero. The map enable register is a privileged register and is illustrated in Figure 4-14.

![Figure 4-14 Map Enable Register (Read/Write)]

When memory management is disabled, virtual addresses are mapped to physical addresses by ignoring their high order bits. All accesses are allowed in all modes and no modify bit is maintained.

Translation Buffer
In order to save actual memory references when repeatedly referencing pages, the hardware includes a mechanism to remember successful virtual address translations and page statuses. Such a mechanism is termed a translation buffer.

Whenever the process context is loaded with LDPCTX, the translation buffer is automatically updated (i.e., the process virtual address translations are invalidated). However, whenever a page table entry for the system or current process region is changed other than to set the page table entry V bit, the software must notify the translation buffer of this by moving an address within the corresponding page into the
Memory Management

Translation Buffer Invalidate Single Register (TBIS). The TBIS Register is illustrated in Figure 4-15.

Whenever the location or size of the system map is changed (SBR, SLR) the entire translation buffer must be cleared by moving 0 into the Translation Buffer Invalidate All Register (TBIA). Therefore, before enabling memory management at processor initialization time, or any other time, the entire translation buffer must be cleared by moving 0 into TBIA with the MTPR instruction. The TBIA Register is illustrated in Figure 4-16.

![Virtual Address](image1)

Figure 4-15 Translation Buffer Invalidate Single (Write-only)

![MBZ](image2)

Figure 4-16 Translation Buffer Invalidate All (Write-only)

FAULTS AND PARAMETERS

There are two types of faults associated with memory mapping and protection. A translation not valid fault is taken when a read or write reference is attempted through an invalid PTE (PTE<31>=0). An access control violation fault is taken when the protection field of the PTE indicates that the intended access to the page for the specified mode would be illegal. Note that these two faults have distinct vectors in the system control block. If both access control violation and translation not valid faults could occur, then the access control violation fault takes precedence. An access control violation fault is also taken if the virtual address referenced is beyond the end of the associated page table. Such a length violation is essentially the same as referencing a PTE that specifies no access in its protection field. To avoid having the fault software redo the length check, a length violation indication is stored in the Fault Parameter Word described in Figure 4-17.

The same parameters are stored for both types of faults. The first parameter pushed on the kernel stack after the PSL and PC is the initial virtual address that caused the fault. A process space reference

75
can result in a system space virtual reference for the PTE. If the PTE reference faults, the virtual address that is saved is the process virtual address. In addition, a bit is stored in the fault parameter word indicating that the fault occurred in the PTE reference.

The second parameter pushed on the kernel stack contains the following information:

**Bit: 2**  **Name:** Write or Modify Intent  
**Function:** Set to one to indicate that the program’s intended access was a write or modify. This bit is zero if the program’s intended access was a read.

**Bit: 1**  **Name:** PTE Reference  
**Function:** Set to one to indicate that the fault occurred during the reference to the process page table associated with the virtual address. This can be set on either length or protection faults.

**Bit: 0**  **Name:** Length Violation  
**Function:** Set to one to indicate that an access control violation was the result of a length violation rather than a protection violation. This bit is always zero for a translation not valid fault.

**PRIVILEGED SERVICES AND ARGUMENT VALIDATION**

**Change Modes**  
There are four instructions provided to allow a program to change the mode at which it is running to a more privileged mode and transfer control to a service dispatcher for the new mode. (Refer to the VAX Architecture Handbook for greater detail.)

- CHMK  Change mode to kernel
- CHME  Change mode to executive
- CHMS  Change mode to supervisor
- CHMU  Change mode to user

These instructions provide the only mechanism for less privileged code to call more privileged code. When the mode transition takes place the previous mode is saved in the previous mode field of the
PSL, thus allowing the more privileged code to determine the privilege of its caller.

Validating Address Arguments
Two instructions are provided to allow privileged services to check addresses passed as parameters. To avoid protection holes in the system, a service routine must always validate that its less privileged caller could have directly referenced the addresses passed as parameters. This validation is done with the PROBE instructions. For detail on PROBE instructions, see the VAX Architecture Handbook.

Notes on the PROBE Instructions
1. The valid bit of the page table entry mapping the probed address is ignored.
2. A length violation gives a status of not-accessible.
3. On the probe of a process virtual address, if the valid bit of the system page table entry is clear, then a translation not valid fault occurs. This allows for the demand paging of the process page tables.
4. On the probe of a process virtual address, if the protection field of the system page table entry indicates no access, then a status of not-accessible is given. Thus, a single no access page table entry in the system map is equivalent to 128 no access page table entries in the process map.
5. It is UNPREDICTABLE whether the modify bit of the examined page table entry is set by a PROBEW.

SIZE OF SYSTEM PAGE TABLE
During the system design stage, the following question was raised: would a physically based, physically contiguous system page table require a large amount of memory to handle a reasonable number of very big processes? The following section addresses this issue.

To examine the size of the system page table, note first that one page of the SPT maps 64 Kb of system virtual address space. The system virtual address space contains the following mapped quantities:
1. Operating system code and data (excluding memory management data): 64 to 96 Kb, 1 to 1.5 pages.
2. Memory management data for physical page management: four to six longwords per physical page of memory. One longword of page table maps one page of memory management data which handles 24 physical pages of memory. One page of page table handles 3K physical pages or 1.5 Mb of physical memory.
3. Shared code—Allowing 16 Kb for each of the items, the total is 96 Kb or 1.5 pages of system page table.
   - command interpreter
   - debugger
   - record manager
   - run-time library

4. Process page tables. One longword of SPT maps one page of process page table which in turn maps 64 Kb of process virtual address space. Sixteen longwords of SPT map 1 Mb of process virtual address space. One page of SPT maps 8 Mb. A very straightforward balance set management design that reserved a fixed (SYSGEN) number of balance set slots each with a fixed (also SYSGEN) maximum virtual address space would use only two pages of SPT to allow 16 processes of up to 1 Mb each in the balance set.

The foregoing analysis shows that a 6-page SPT handles a very reasonable system, and increasing the 1 Mb process virtual space to 4 Mb and 16 processes in the balance set adds only 6 more pages of SPT for a total of 12. A smaller system with 256 Kb of memory and eight balance set processes, each 512 Kb maximum size, would need about three pages of SPT.

**SHARING**

To discuss sharing, it is useful to assume the concept of a section in the operating system. A section is a collection of pages that have some relationship to each other. Though units as small as pages may indeed be shared, sections are the usual unit of sharing.

**Shared Sections in Process Space**

Sharing in the process half of the virtual address space requires that the page table fragments for the sections being shared be replicated in the process page table(s). Clearly this introduces multiple PTEs for the same physical page. This is a problem traditionally avoided by one or more levels of indirection, i.e., the PTE points to the shared PTE that points to the page. We can avoid introducing this level of indirection in the hardware by observing the following software rules:

1. A share count is maintained for each shared page in memory and in effect counts the number of direct pointers to that page.

2. When a process releases a page from its working set and it is a shared page as indicated in the working set data base, the private PTE must be changed to point to the shared PTE for the page, and the private copy of the modify bit must be OR'ed into the shared
Memory Management

PTE. Then the share count is decremented and if the count is now zero, the page is released and the shared PTE is updated to reflect that. Note that the process's working set database allows it to find its private PTE and the physical page database points to the shared PTE.

3. When a process gets an invalid page fault, one of the possible states of the invalid PTE is that it points to a shared PTE. Of course that PTE might say that the page was not resident and required a page read. Whether or not the read was necessary, the shared PTE is eventually copied to the private PTE and the share count of the page is incremented.

4. Note that throwing a process out of the balance set is the equivalent of releasing all its pages.

5. The use of the indirect page pointer as a software-only mechanism is adequate for this form of sharing. It should be noted that it is very difficult to change the PFN of a page in memory when it is actively being shared. That would require a scan of the page tables for all the processes in the balance set.

Shared Sections in System Space

When a process is using a shared section in the system region of the address space, it is referencing a single shared page table. Since it is possible for a process to simply reference such a shared section without ever having declared its intention to do that, the operating system must be prepared to handle reference faults. A straightforward design for this kind of sharing is:

1. Have programs explicitly declare their intention to use each shared system section. This can be done statically at compile or link time or dynamically at run time.

2. Have the balance set manager swap in and lock down the entire section when the process intending to use it is swapped in.

3. The balance set manager maintains share counts on the section and only discards its pages when no process in the balance set wants it.

4. If a process faults such a page because it failed to declare its intention to use the section, then that is a programming error.

Another approach for shared system sections allows a process to reference pages of the section with no prior declaration of its intent to use them. Such pages would be demand paged within a pool of pages reserved for that purpose. That pool would keep a list of the pages in use and a fault for a new one would cause one in the pool to be
Memory Management

replaced. This would use the same sort of working set management that is used for the process address space but it would be global across processes.
CHAPTER 5
SYSTEM ARCHITECTURAL CHARACTERISTICS
AND PRIVILEGED INSTRUCTIONS

INTRODUCTION
The system structure common to all members of the VAX family of 32-bit minicomputers is the VAX system architecture. This common architecture defines a consistent functional behavior seen by a programmer regardless of which VAX family processor is in use. Areas of interaction that, from the programmer's viewpoint, are independent of the processor type include: data sharing and synchronization, restartability, interrupts and errors, and I/O structure. Of these, data sharing is most visible to the programmer.

The VAX architecture provides three types of instructions that enable user mode software to obtain operating system services without jeopardizing the integrity of the system. These privileged instructions include the Change Mode instructions, PROBE instructions, and the Return from Exception or Interrupt instruction. With these instructions, the user can gain access to system services offered in kernel, executive or supervisor modes, and examine the contents of processor registers. Context switching and controlled return operations are also available.

The latter part of this chapter describes the privileged instructions more fully.

DATA SHARING AND SYNCHRONIZATION
Data (or instructions) may be shared among various entities including programs, processors and I/O devices. Entities sharing data may do so explicitly by referencing the same datum or implicitly by referencing different items within the same physical memory location.

In the VAX family architecture, implicit sharing is transparent to the programmer. The memory system is implemented such that the basis of access for independent modification is the byte. Note that this does not imply a maximum reference size of one byte but only that independent modifying accesses to adjacent bytes produce the same results regardless of the order of execution. For example, locations 0 and 1 contain the values 5 and 6 respectively. One process executes INCB 0 and another executes INCB 1. Then, regardless of the order of execution (including effectively simultaneous execution) the final contents must be 6 and 7.
Access to explicitly shared data that may be written must be synchronized. Before accessing shared writable data, the programmer must acquire control of the data structure. Seven instructions are provided to permit interlocked access to a control variable. BBSSI and BBCCI instructions use hardware-provided primitive operations to make a read and a subsequent write reference to a single bit within a single byte in an interlocked sequence. The ADAWI instruction uses a hardware-provided primitive operation to make a read and a subsequent write operation to a single aligned word in an interlocked sequence to allow counters to be maintained without other interlocks. The INSQHI, INSQTI, REMQHI, and REMQTI instructions use a primitive operation provided by hardware to make a series of aligned longword reads and writes in an interlocked method to allow queues to be maintained without other interlocks. Use of the hardware primitives guarantees that no read operation within the synchronizing part of these instructions can occur between the synchronized reads and the writes of these instructions. The instructions are implemented so that no faults will cause the data structure to be locked for an extended period. On the processor, only interlocking instructions are locked out by the interlock.

In order to use some UNIBUS peripheral devices, processors insure that all instructions making byte- or word-sized modifying references (.mb and .mw) use the DATIP-DATO/DATOB functions when the operand physical address selects a UNIBUS device. This constraint does not apply to longword, quadword, field, floating, double or string operations. This constraint also does not apply to instructions precluded from I/O space references.

In a customer-designed shared memory multiprocessor configuration, changing any of the address mapping information for system space requires that all processors execute a MTPR xxx,#TBIS.

CACHE
Some hardware implementations—including VAX processors—have a mechanism to reduce access time by making local copies of recently used memory contents. This mechanism is termed a cache. In VAX family processors, the cache is implemented in such a way that its existence is transparent to software (except for timing and error reporting/control). In particular, the following are true:

1. Program writes to memory, followed by a peripheral output transfer, output the updated value.
2. A peripheral input transfer followed by a program reading of memory reads the input value.
3. A write or modify followed by a HALT on one processor, followed by a read or modify on another processor, reads the updated value. (Note that this only applies to customer-designed multiprocessor systems.)

4. A write or modify followed by a power failure, followed by restoration of power, followed by a read or modify, reads the updated value. This only occurs if the duration of the power failure does not exceed the maximum nonvolatile period of the main memory or if the contents of memory were protected by optional battery backup.

5. In customer-designed multiprocessor systems, access to variables shared between processors is interlocked by software executing BBSSI, BBCCI, ADAWI, INSQHI, INSQTI, REMQHI, or REMQTI instructions. In particular, the writer must execute an interlocking instruction after the write to release the interlock and the reader must execute a successful matching interlock instruction before the read.

6. Accesses to I/O registers are not cached.

RESTARTABILITY
The VAX architecture requires that all instructions be restartable after a fault or interrupt that terminated execution before the instruction was completed. Generally, this means that modified registers are restored to the value they had at the start of execution. For some complex or iterative instructions, intermediate results are stored in the general registers. In the latter case, memory contents may have been altered, but the former case requires that no operand be written unless the instruction can be completed. For most instructions with only a single modified or written operand, this implies special processing only when a multibyte operand spans a protection boundary, making it necessary to test accessibility of both parts of the operand.

Instructions which store intermediate results in the general registers must not compromise system integrity. They must ensure that any addresses stored or used are virtual addresses, subject to protection checking. Furthermore, they ensure that any state information stored or used does not result in a noninterruptable or nonterminating sequence.

Instruction operands that are peripheral device registers having access side effects may produce UNPREDICTABLE results due to instruction restarting after faults (including page faults) or interrupts. To ensure no interrupts, the programmer must avoid operand specifier addressing modes 9, 11, 13, and 15, and these modes indexed. (Symbolically, these are @(Rn)+, @B↑D(Rn), @W↑D(Rn), and @L↑D(Rn),
and these indexed.) The hardware may allow interrupts for these
modes in order to minimize interrupt latency. For the instructions list-
ed in Appendix H of this handbook, the hardware ensures that no
other interrupts will occur after the first I/O space access.

Since these instructions are not interruptable after I/O space accesses
(except for the addressing modes above), their execution will extend
the interrupt latency. The programmer should make some effort to
keep them short by minimizing the number of memory references.
Use R0 through R13 instead, for example.

Memory modificatons produced as a byproduct of instruction execu-
tion (e.g., memory access statistics) are specifically excluded from the
constraint that memory may not be altered until the instruction can be
completed.

Instructions that abort are constrained only to insure memory protec-
tion (e.g., registers can be changed).

**INTERRUPTS**

Underlying the VAX architectural concept of an interrupt is the notion
that an interrupt request is a static condition, not a transient event, and
can be sampled by a processor at appropriate times. Further, if the
need for an interrupt disappears before a processor has honored an
interrupt request, the interrupt request can be removed without
consequence.

So that software can operate deterministically, any instruction chang-
ing the processor priority (IPL) such that a pending interrupt is en-
abled, allows the interrupt to occur before executing the next instruc-
tion that would have been executed had the interrupt not been pend-
ing. For example, an interrupt service routine associated with a
specific IPL is forbidden from lowering IPL below its associated IPL
value.

Similarly, instructions that generate requests at the software interrupt
levels allow the interrupt to occur, if processor priority permits, before
executing the apparently subsequent instruction.

**ERRORS**

Processor errors, if not inconsistent with instruction completion, cre-
ate high-priority interrupt requests. Otherwise, they terminate instruc-
tion execution with a fault, trap or abort.

Error notification interrupts may be delayed from the apparent com-
pletion of the instruction in execution at the time of the error; but if
enabled, the interrupt is requested before processor context is
switched.
I/O STRUCTURE
The VAX family I/O architecture is very similar to the PDP-11 structure. Peripheral device control/status and data registers appear at locations in the physical address space, and can therefore be manipulated by normal memory reference instructions. This I/O space occupies the upper megabyte. Use of general instructions permits all the virtual address mapping and protection mechanisms described in Chapter 4, Memory Management, to be used when referencing I/O registers. Note that no references to the 1-megabyte I/O space on VAX systems are cached.

Because VAX systems have an integral UNIBUS, an area of the I/O physical address space, $2^{18}$ bytes in length, maps through to the UNIBUS addresses. This area is referred to as the UNIBUS space.

Constraints on I/O Registers
I/O registers always meet the following rules and constraints:

1. All registers are aligned on natural boundaries, and the physical address of an I/O register will always be an integral multiple of the register size in bytes (which must be a power of 2).

2. References using a length attribute other than the length of the register and/or an unaligned reference may produce UNPREDICTABLE results. For example, a byte reference to a word-length register will not necessarily respond by supplying or modifying the byte addressed.

3. In peripheral devices, error and status bits that may be asynchronously set by the device are usually cleared by software writing a one to these bits, and are not affected by writing a zero. This is to prevent clearing bits that may be asynchronously set between reading and writing a register.

4. Only byte and word references of a read-modify-write (.mb or .mw) type in UNIBUS I/O spaces are guaranteed to interlock correctly. References in the I/O space other than in UNIBUS spaces are UNDEFINED with respect to interlocking. This includes the BBSSI and BBCCI instructions.

5. String, quad, octa, F_floating, D_floating, G_floating, H_floating, and field references in the I/O space result in UNDEFINED behavior.

The information on VAX architectural characteristics will assist sophisticated users to modify or enhance the capabilities of VAX systems. This architectural information should also be of help to designers who are configuring their own multiprocessor systems. For additional details, please see the VAX Architecture Handbook.
PRIVILEGED INSTRUCTIONS
The processor provides three types of instructions that enable user mode software to obtain operating system services without jeopardizing the integrity of the system. They include:

- The Change Mode instructions
- The PROBE instructions
- The Return from Exception or Interrupt instruction

User mode software can obtain privileged services by calling operating system service procedures with a standard CALL instruction. The operating system's service dispatcher issues an appropriate Change Mode instruction before actually entering the procedure. Change Mode allows access mode transitions to take place from one mode to the same or more privileged mode only. When the mode transition takes place, the previous mode is saved in the Previous Mode field of the Processor Status Longword, allowing the more privileged code to determine the privilege of its caller.

A Change Mode instruction is simply a special trap instruction that can be thought of as an operating system service call instruction. User mode software can explicitly issue Change Mode instructions, but since the operating system receives the trap, nonprivileged users cannot write any code to execute in any of the privileged access modes. User mode software can include a condition handler for Change Mode to User traps, however, and this instruction is useful for providing general-purpose services for user mode software. The system manager ultimately grants the privilege to write any code that handles Change Mode traps to more privileged access modes.

For service procedures written to execute in privileged access modes (kernel, executive, and supervisor), the processor provides address access privilege validation instructions. The PROBE instructions enable a procedure to check the read (PROBER) and write (PROBEW) access protection of pages in memory against the privileges of the caller who requested access to a particular location. This enables the operating system to provide services that execute in privileged modes to less privileged callers and still prevent the caller from accessing protected areas of memory.

The operating system's privileged service procedures and interrupt and exception service routines exit using the Return from Exception or Interrupt (REI) instruction. REI is the only way in which the privilege of the processor's access mode can be decreased. Like the procedure and subroutine return instructions, REI restores the Program Counter and the processor state to resume the process at the point where it was interrupted.
REI performs special services, however, that normal return instructions do not. For example, REI checks to see if any asynchronous system traps have been queued for the currently executing process while the interrupt or exception service routine was executing, and ensures that the process will receive them. Furthermore, REI checks to ensure that the mode to which it is returning control is the same as or less privileged than the mode in which the processor was executing when the exception or interrupt occurred. Thus REI is available to all software including user-written trap handling routines, but a program cannot increase its privilege by altering the processor state to be restored.

When the operating system schedules a context switching operation, the context switching procedure uses the Save Process Context (SVPCTX) and Load Process Context (LDPCTX) instructions to save the current process context and load another. The operating system's context switching procedure identifies the location of the hardware context to be loaded by updating an internal processor register.

Internal processor registers not only include those that identify the process currently executing, but also the memory management and other registers, such as the console and clock control registers. The Move to Processor Register (MTPR) and Move from Processor Register (MFPR) instructions are the only instructions that can explicitly access the internal processor registers. MTPR and MFPR are privileged instructions that can be issued only in kernel mode.

The extended function instruction provides a controlled mechanism for software to request services of nonstandard microcode in the writable control store or simulator software running in kernel mode. The request is controlled by the contents of the System Control Block.

Further information on each of the privileged instructions follows.

| CHANGE MODE |
| Purpose: request services of more privileged software |
| Format: opcode code.rw |
| Operation: if \{PSL<IS> EQLU 1\} then HALT;
  illegal from
  !Interrupt stack
  {switch stack pointer from current-mode to MINU (opcode-mode, PSL<current-mode>)} ;
  \(-\text{SP}\) \(\leftarrow\) PSL;
  !initiate CHMx |
System Architectural Characteristics and Privileged Instructions

- (SP) ← PC; exception
- (SP) ← SEXT (code);
PSL <CM, TP, FPD, DV, FU, IV, T, N, Z, V, C> ← 0;
!clean out PSL
PSL<previous-mode> ← PSL<current-mode>;
PSL<current-mode> ← MINU (opcode-mode, PSL<current-mode>);
!maximize
!privilege
PC ← {SCB vector for opcode-mode};

Condition:  N ← 0;
Codes:      Z ← 0;
            V ← 0;
            C ← 0;

Exceptions: halt

Opcodes:   BC      CHMK     Change Mode to Kernel
BD      CHME     Change Mode to Executive
BE      CHMS     Change Mode to Supervisor
BF      CHMU     Change Mode to User

Description: Change Mode Instructions allow processors to change their access mode in a controlled manner. The instruction only increases privilege (i.e., decreases the access mode).

A change in mode also results in a change of stack pointers; the old pointer is saved, the new pointer is loaded. The PSL, PC, and code passed by the instruction are pushed onto the stack of the new mode. The saved PC addresses the instruction following the CHMx instruction. The code is sign extended. After execution, the new stack's appearance is:

\[
\text{sign extended code } \quad \text{:(SP)}
\]

PC of next instruction

old PSL

The destination mode selected by the opcode is used to select a location from the System Control
Block. This location addresses the CHMx dispatcher for the specified mode.

Notes: By software convention, negative codes are reserved to CSS and customers.

Examples: CHMK #7 ;request the kernel mode ;service ;specified by code 7 CHME #4 ;request the executive ;mode service ;specified by code 4 CHMS #−2 ;request the supervisor ;mode service ;specified by customer ;code −2

**PROBE ACCESSIBILITY**

**Purpose:** verify that arguments can be accessed

**Format:** opcode mode.rb, len.rw, base.ab

**Operation:** probe-mode ← MAXU(mode<1:0>, PSL<previous-mode>); condition codes ← {accessibility of (base)} and {accessibility of (base + ZEXT(len)− 1)} using probe-mode;

**Condition Codes:**

- N ← 0;
- Z ← if both accessible then 0; else 1;
- V ← 0;
- C ← 0;

**Exceptions:** translation not valid

**Opcodes:**

- OC PROBER Probe Read Accessibility
- OD PROBEW Probe Write Accessibility

**Description:** The PROBE instruction checks the read or write accessibility of the first and last byte specified by the base address and the zero extended length. Note that the bytes in between are not checked. System software must check all pages between the two end bytes if they are to be accessed.

The protection is checked against the mode specified in bits <1:0> of the mode operand that is restricted (by maximization) from being more pri-
vileged than the previous access mode field of the 
PSL. Note that probing with a mode operand of 0 
is equivalent to probing the mode specified in 
PSL<previous-mode>.

Probing an address only returns the accessibility 
of the page(s) and has no affect on their residen-
cy. However, probing a process address may 
cause a page fault in the system address space 
on the per-process page tables.

Example:

MOVL 4(AP),R0 ;copy address of first 
;argument so 
;that it can't be 
;changed

PROBER #0,#4,R0 ;verify that the 
;longword 
;pointed 
;to by the first 
;argument 
;could be 
;read by the previous 
;access mode 
;Note that the 
;argument 
;list itself 
;must already have 
;been probed

MOVQ 8(AP),R0 ;copy length and 
;address 
;of buffer arguments 
;so that 
;they can't change

PROBEW $0,R0,R1 ;verify that the buffer 
described 
;by the second and 
third arguments 
;could be written by 
;the previous 
;access mode 
;Note that the 
;argument 
;list must 
;already have been 
;probed and that
RETURN FROM EXCEPTION OR INTERRUPT

Purpose: 
controlled return from exceptions or interrupts

Format: 
opcode

Operation: 
tmp1 ← (SP)+;   ! Pick up saved PC
tmp2 ← (SP)+;   ! and PSL
if {tmp2<CUR_MOD> LSSU PSL<CUR_MOD>} OR
   {tmp2<IS> EQLU 1 AND PSL<IS> EQLU 0} OR
   {tmp2<IS> EQLU 1 AND tmp2<CUR_MOD> NEQU 0} OR
   {tmp2<IS> EQLU 1 AND tmp2<IPL> EQLU 0} OR
   {tmp2<IPL> GTRU 0 AND tmp2<CUR_MOD> NEQU 0} OR
   {tmp2<PRV_MOD> LSSU tmp2<CUR_MOD>} OR
   {tmp2<IPL> GTRU PSL<IPL>} OR
   {tmp2<PSL_MBZ> NEQU 0} then {reserved operand fault};
if {tmp2<CM> EQLU} AND
   [{tmp2<FPD,IS,DV,FU,IV> NEQU 0} OR
   {tmp2<CUR_MOD> NEQU 3}] then {reserved operand fault};
if PSL<IS> EQLU 1 then ISP ← SP   !save old
   !stack
   pointer
   else PSL<CUR_MOD>_SP ← SP;
if PSL<TP> EQLU 1 then tmp2<TP> ← 1;
ITP ← TP or stack TP
PC ← tmp1;
PSL ← tmp2;
if PSL<IS> EQLU 0 then
   begin
   SP ← PSL<CUR_MOD>_SP;             !switch
   !stack
if PSL<CUR_MOD> GEQU ASTLVL
!check for AST delivery
    the {request interrupt at IPL 2};
end;

{check for software interrupts};

Condition
Codes:  N ← saved PSL<3>;
        Z ← saved PSL<2>;
        V ← saved PSL<1>;
        C ← saved PSL<0>;

Exceptions: reserved operand

Opcode:  02 REI Return from Exception or Interrupt

Description: A longword is popped from the current stack and
              held in a temporary PC. A second longword is
              popped from the current stack and held in a tem-
              porary PSL. Validity of the popped PSL is
              checked. The current stack pointer is saved and a
              new stack pointer is selected according to the
              new PSL CUR_MOD and IS fields. The level of the
              highest privilege AST is checked against the cur-
              rent mode to see whether a pending AST can be
              delivered. Execution resumes with the instruction
              being executed at the time of the execution or
              interrupt. Any instruction lookahead in the proc-
              essor is reinitialized.

Notes:

1. The exception or interrupt service routine is
   responsible for restoring any registers saved
   and removing any parameters from the stack.

2. As usual for faults, any Access Violation or
   Translation Not Valid conditions on the stack
   pops restore the stack pointer and fault.

3. The noninterrupt stack pointers may be
   fetched and stored either in privileged regis-
   ters or in their allocated slots in the PCB. On-
   ly LDPCTX and SVPCTX always fetch and
   store in the PCB. MFPR and MTPR always
   fetch and store the pointers whether in regist-
   ers or the PCB.
LOAD PROCESS CONTEXT
SAVE PROCESS CONTEXT

Purpose: save and restore register and memory management context

Format: opcode

Operation:
\[
\text{if } \text{PSL} < \text{current-mode} > \text{NEQU 0}
\text{then \{opcode reserved to DIGITAL fault\};}
\text{\{invalidate per-process translation buffer entries\};}
\text{\{LDPCTX\}}
\text{\{load process general registers from Process Control Block\};}
\text{\{load process map, ASTLVL, and PME from PCB\};}
\text{\{save PSL and PC on stack for subsequent REI\};}
\text{\{save process general registers into Process Control Block\};}
\text{\{remove PSL and PC from stack and save in PSB\};}
\text{\{switch to Interrupt Stack\};}
\]

Condition: \(N \leftarrow N;\)

Codes:
\(Z \leftarrow Z;\)
\(V \leftarrow V;\)
\(C \leftarrow C;\)

Exceptions: reserved operand
privileged instruction

Opcodes:
\(06 \text{ LDPCTX} \quad \text{Load Process Context}\)
\(07 \text{ SVPCTX} \quad \text{Save Process Context}\)

Description: The Process Control Block is specified by the internal processor register Process Control Block Base. The general registers are loaded from or saved to the PCB. In the case of LDPCTX, the memory management registers describing the process address space are also loaded and the process entries in the translation buffer are cleared. If SVPCTX is executed while running on the kernel stack, execution is switched to the interrupt stack. When LDPCTX is executed, execution is switched to the kernel stack. The PC and
PSL are moved between the PCB and the stack, suitable for use by a subsequent REI instruction.

MOVE FROM PRIVILEGED REGISTER
MOVE TO PRIVILEGED REGISTER

Purpose: provide access to the internal privileged registers

Format: opcode src.rl, regnumber.rl  MTPR
         opcode regnumber.rl, dst.wl    MFPR

Operation: if PSL<current-mode> NEQU kernel then
             [reserved instruction fault];
             PRS[regnumber] ← src;
             dst ← PRS[regnumber];
             !MTPR
             !MFPR

Condition: N ← dst LSS 0;
Codes:     Z ← dst EQL 0;
           V ← 0;
           C ← C;

Exceptions: reserved operand
            privileged instruction

Opcodes:   DA   MTPR  Move to Privileged Register
           DB   MFPR  Move from Privileged Register

Description: The specified register is loaded or stored. The regnumber operand is a longword that contains the privileged register number. Execution may have register-specific side effects.

Notes:
1. A reserved operand fault occurs if the privileged register does not exist or is read-only for MTPR or write-only for MFPR. It also occurs on some invalid operands to some registers.
2. A reserved instruction fault occurs if instruction execution is attempted in other than kernel mode.

The following table is a summary of the registers accessible in the privileged register space.
The "type" column indicates read-only (R), read/write (R/W), or write-only (W) characteristics.

"Scope" indicates whether a register is per-CPU or per-process. The implication is that, in general, registers labeled "CPU" are manipulated only through software via the MTPR and MFPR instructions. Per-process registers, on the other hand, are manipulated implicitly by context switch instructions. The "Init" column indicates that the register is ("yes") or is not ("no") set to some predefined value (note: not necessarily cleared) by a processor initialization command. A "—" indicates initialization is optional.

The number of a register, once assigned, will not change across implementations or within an implementation. Implementation-dependent registers are assigned distinct addresses for each implementation. Thus, any privileged register present on more than one implementation will perform the same function whenever implemented. All unsigned positive numbers are reserved to DIGITAL; all negative numbers (i.e., with bit 31 set) are reserved to CSS and customers.

Each register number has a symbol formed as PR$\_followed by the register's mnemonic.

### VAX Series Registers

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Mne-</th>
<th>Num-</th>
<th>Type</th>
<th>Scope</th>
<th>Init?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel Stack Pointer</td>
<td>KSP</td>
<td>0</td>
<td>R/W</td>
<td>PROC</td>
<td>—</td>
</tr>
<tr>
<td>Executive Stack Pointer</td>
<td>ESP</td>
<td>1</td>
<td>R/W</td>
<td>PROC</td>
<td>—</td>
</tr>
<tr>
<td>Supervisor Stack Pointer</td>
<td>SSP</td>
<td>2</td>
<td>R/W</td>
<td>PROC</td>
<td>—</td>
</tr>
<tr>
<td>User Stack Pointer</td>
<td>USP</td>
<td>3</td>
<td>R/W</td>
<td>PROC</td>
<td>—</td>
</tr>
<tr>
<td>Interrupt Stack Pointer</td>
<td>ISP</td>
<td>4</td>
<td>R/W</td>
<td>CPU</td>
<td>—</td>
</tr>
<tr>
<td>P0 Base Register</td>
<td>P0BR</td>
<td>8</td>
<td>R/W</td>
<td>PROC</td>
<td>—</td>
</tr>
<tr>
<td>P0 Length Register</td>
<td>P0LR</td>
<td>9</td>
<td>R/W</td>
<td>PROC</td>
<td>—</td>
</tr>
<tr>
<td>P1 Base Register</td>
<td>P1BR</td>
<td>0A</td>
<td>R/W</td>
<td>PROC</td>
<td>—</td>
</tr>
<tr>
<td>P1 Length Register</td>
<td>P1LR</td>
<td>0B</td>
<td>R/W</td>
<td>PROC</td>
<td>—</td>
</tr>
<tr>
<td>System Base Register</td>
<td>SBR</td>
<td>0C</td>
<td>R/W</td>
<td>CPU</td>
<td>—</td>
</tr>
<tr>
<td>System Length Register</td>
<td>SLR</td>
<td>0D</td>
<td>R/W</td>
<td>CPU</td>
<td>—</td>
</tr>
<tr>
<td>Process Control Block Base</td>
<td>PCBB</td>
<td>10</td>
<td>R/W</td>
<td>PROC</td>
<td>—</td>
</tr>
<tr>
<td>System Control Block Base</td>
<td>SCBB</td>
<td>11</td>
<td>R/W</td>
<td>CPU</td>
<td>—</td>
</tr>
<tr>
<td>Interrupt Priority Level</td>
<td>IPL</td>
<td>12</td>
<td>R/W</td>
<td>CPU</td>
<td>yes</td>
</tr>
<tr>
<td>AST Level</td>
<td>AST_LVL</td>
<td>13</td>
<td>R/W</td>
<td>PROC</td>
<td>yes</td>
</tr>
<tr>
<td>Software Interrupt Request</td>
<td>SIRR</td>
<td>14</td>
<td>W</td>
<td>CPU</td>
<td>—</td>
</tr>
<tr>
<td>Software Interrupt Summary</td>
<td>SISR</td>
<td>15</td>
<td>R/W</td>
<td>CPU</td>
<td>yes</td>
</tr>
<tr>
<td>Interval Clock Control</td>
<td>ICES</td>
<td>18</td>
<td>R/W</td>
<td>CPU</td>
<td>yes</td>
</tr>
<tr>
<td>Next Interval Count</td>
<td>NICR</td>
<td>19</td>
<td>W</td>
<td>CPU</td>
<td>—</td>
</tr>
</tbody>
</table>
### System Architectural Characteristics and Privileged Instructions

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Mnemonic</th>
<th>Number&lt;sub&gt;16&lt;/sub&gt;</th>
<th>Type</th>
<th>Scope</th>
<th>Init?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interval Count</td>
<td>ICR</td>
<td>1A</td>
<td>R</td>
<td>CPU</td>
<td>—</td>
</tr>
<tr>
<td>Time of Year</td>
<td>TODR</td>
<td>1B</td>
<td>R/W</td>
<td>CPU</td>
<td>no</td>
</tr>
<tr>
<td>Console Receiver C/S</td>
<td>RXCS</td>
<td>20</td>
<td>R/W</td>
<td>CPU</td>
<td>yes</td>
</tr>
<tr>
<td>Console Receiver D/B</td>
<td>RXDB</td>
<td>21</td>
<td>R</td>
<td>CPU</td>
<td>—</td>
</tr>
<tr>
<td>Console Transmit C/S</td>
<td>TXCS</td>
<td>22</td>
<td>R/W</td>
<td>CPU</td>
<td>yes</td>
</tr>
<tr>
<td>Console Transmit D/B</td>
<td>TXDB</td>
<td>23</td>
<td>W</td>
<td>CPU</td>
<td>—</td>
</tr>
<tr>
<td>Memory Management Enable</td>
<td>MAPEN</td>
<td>38</td>
<td>R/W</td>
<td>CPU</td>
<td>yes</td>
</tr>
<tr>
<td>Trans. Buf. Invalidate All</td>
<td>TBIA</td>
<td>39</td>
<td>W</td>
<td>CPU</td>
<td>—</td>
</tr>
<tr>
<td>Trans. Buf. Invalidate Single</td>
<td>TBIS</td>
<td>3A</td>
<td>W</td>
<td>CPU</td>
<td>—</td>
</tr>
<tr>
<td>Performance Monitor Enable</td>
<td>PMR</td>
<td>3D</td>
<td>R/W</td>
<td>PROC</td>
<td>yes</td>
</tr>
<tr>
<td>System Identification</td>
<td>SID</td>
<td>3E</td>
<td>R</td>
<td>CPU</td>
<td>no</td>
</tr>
</tbody>
</table>

### VAX-11/780 Specific Registers

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Mnemonic</th>
<th>Number&lt;sub&gt;16&lt;/sub&gt;</th>
<th>Type</th>
<th>Scope</th>
<th>Init?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accelerator Control/Status</td>
<td>ACCS</td>
<td>28</td>
<td>R/W</td>
<td>CPU</td>
<td>yes</td>
</tr>
<tr>
<td>Accelerator Maintenance</td>
<td>ACCR</td>
<td>29</td>
<td>R/W</td>
<td>CPU</td>
<td>no</td>
</tr>
<tr>
<td>WCS Address</td>
<td>WCSA</td>
<td>2C</td>
<td>R/W</td>
<td>CPU</td>
<td>no</td>
</tr>
<tr>
<td>WCS Data</td>
<td>WCSD</td>
<td>2D</td>
<td>R/W</td>
<td>CPU</td>
<td>yes</td>
</tr>
<tr>
<td>SBI Fault/Status</td>
<td>SBIFS</td>
<td>30</td>
<td>R/W</td>
<td>CPU</td>
<td>yes</td>
</tr>
<tr>
<td>SBI Silo</td>
<td>SBIS</td>
<td>31</td>
<td>R</td>
<td>CPU</td>
<td>no</td>
</tr>
<tr>
<td>SBI Silo Comparator</td>
<td>SBISC</td>
<td>32</td>
<td>R/W</td>
<td>CPU</td>
<td>yes</td>
</tr>
<tr>
<td>SBI Maintenance</td>
<td>SBIMT</td>
<td>33</td>
<td>R/W</td>
<td>CPU</td>
<td>yes</td>
</tr>
<tr>
<td>SBI Error Register</td>
<td>SBIER</td>
<td>34</td>
<td>R/W</td>
<td>CPU</td>
<td>yes</td>
</tr>
<tr>
<td>SBI Timeout Address</td>
<td>SBITA</td>
<td>35</td>
<td>R</td>
<td>CPU</td>
<td>—</td>
</tr>
<tr>
<td>SBI Quadword Clear</td>
<td>SIBQC</td>
<td>36</td>
<td>W</td>
<td>CPU</td>
<td>—</td>
</tr>
<tr>
<td>Micro Program Breakpoint</td>
<td>MBRK</td>
<td>3C</td>
<td>R/W</td>
<td>CPU</td>
<td>no</td>
</tr>
</tbody>
</table>

### VAX-11/750 Specific Registers

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Mnemonic</th>
<th>Number&lt;sub&gt;16&lt;/sub&gt;</th>
<th>Type</th>
<th>Scope</th>
<th>Init</th>
</tr>
</thead>
<tbody>
<tr>
<td>Console Storage Receive Status Register</td>
<td>CSRS</td>
<td>1C</td>
<td>R/W</td>
<td>CPU</td>
<td>Yes</td>
</tr>
<tr>
<td>Console Storage Receive Data Register</td>
<td>CSRD</td>
<td>1D</td>
<td>R</td>
<td>CPU</td>
<td>—</td>
</tr>
</tbody>
</table>

98
System Architectural Characteristics and Privileged Instructions

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Mnemonic</th>
<th>Number&lt;sub&gt;16&lt;/sub&gt;</th>
<th>Type</th>
<th>Scope</th>
<th>Init?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Console Storage Transmit Status Register</td>
<td>CSTS</td>
<td>1E</td>
<td>R/W</td>
<td>CPU</td>
<td>Yes</td>
</tr>
<tr>
<td>Console Storage Transmit Data Register</td>
<td>CSTD</td>
<td>1F</td>
<td>W</td>
<td>CPU</td>
<td>—</td>
</tr>
<tr>
<td>Translation Buffer Disable Register</td>
<td>TBDR</td>
<td>24</td>
<td>R/W</td>
<td>CPU</td>
<td>Yes</td>
</tr>
<tr>
<td>Cache Disable Register</td>
<td>CADR</td>
<td>25</td>
<td>R/W</td>
<td>CPU</td>
<td>Yes</td>
</tr>
<tr>
<td>Machine Check Error Summary Register</td>
<td>MCESR</td>
<td>26</td>
<td>R/W</td>
<td>CPU</td>
<td>Yes</td>
</tr>
<tr>
<td>Cache Error</td>
<td>CAER</td>
<td>27</td>
<td>R/W</td>
<td>CPU</td>
<td>Yes</td>
</tr>
<tr>
<td>I/O Reset Register</td>
<td>—</td>
<td>37</td>
<td>R/W</td>
<td>CPU</td>
<td>Yes</td>
</tr>
<tr>
<td>Translation Buffer</td>
<td>TB</td>
<td>3B</td>
<td>R/W</td>
<td>CPU</td>
<td>No</td>
</tr>
</tbody>
</table>

**EXTENDED FUNCTION CALL**

**Purpose:** provide for customer extensions to the instruction set

**Format:** opcode

**Operation:** |XFC fault|

**Condition**

- N ← 0;
- Z ← 0;
- V ← 0;
- C ← 0;

**Exceptions:** opcode reserved to customer

**Opcodes:** FC XFC Extended Function Call

**Description:** This instruction requests services of nonstandard microcode or software. If no special microcode is loaded, then an exception is generated to a kernel mode software simulator (see Chapter 3). Typically, the next byte would specify which of several extended functions are requested. Parameters would be passed either as normal operands, or, more likely, in fixed registers.
PART 2
VAX-11/750
## CHAPTER 6
### CONSOLE SUBSYSTEM

<table>
<thead>
<tr>
<th>FEATURES</th>
<th>BENEFITS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Console terminal</td>
<td>Dual function as console and user terminal results in hardware savings</td>
</tr>
<tr>
<td>Console command language</td>
<td>Gives the user a powerful, yet easy-to-use, debugging tool</td>
</tr>
<tr>
<td>Console terminal is a standard ASCII device</td>
<td>Provides a high degree of flexibility</td>
</tr>
<tr>
<td>EIA communications interfacing</td>
<td>Allows standard industry-compatible communications</td>
</tr>
<tr>
<td>Front panel switches and indicator lights</td>
<td>Offer the user easy system access and control</td>
</tr>
<tr>
<td>TU58 cartridge tape drive</td>
<td>Provides an inexpensive, reliable device and medium for booting, diagnostics, field updates to any software, and convenient personal data storage on cartridge</td>
</tr>
<tr>
<td>Unattended restart</td>
<td>The system restarts or reboots itself upon recovery of electricity after a power failure or other system crash</td>
</tr>
</tbody>
</table>

### INTRODUCTION
The VAX-11/750 Console subsystem has four major elements:

1. A front panel on the CPU cabinet with switches and indicator lights.
2. A separate ASCII terminal with keyboard, called the console terminal.
3. A TU58 tape cartridge drive in the CPU cabinet.
4. A special console command language with simple commands the user types from the console terminal.

This approachable Console subsystem is designed to allow the user to interactively communicate instructions to the CPU using the console terminal and the console command language. The integral TU58 tape
cartridge drive can be used optionally to boot the system, as well as to load diagnostics, updates to the operating system software, and compilers. DIGITAL also supports the TU58 under VAX/VMS for data storage and retrieval.

Figure 6-1 illustrates the hardware elements of the VAX-11/750 console subsystem.

![VAX-11/750 Console Subsystem Diagram]

**VAX-11/750 Front Panel**
The VAX-11/750's front panel contains switches which allow the processor to be halted, restarted, booted, and initialized. If a user wants the CPU to restart in case of a power failure, the POWER ON ACTION switch located on the front panel can be set to automatically restart the system. The OFF-LOCAL-REMOTE switch determines the operating mode of the console terminal: system terminal mode or console mode. Furthermore, by removing the key from the OFF-LOCAL-REMOTE switch, the user can fix the operating state, preventing anyone else from changing that state. The operator can use the console terminal as a user terminal in a protected environment once the system is running. Additional user terminals on the system cannot function as console terminals.

**Console Terminal/Console Command Language**
The console terminal and the console command language are two powerful tools within the VAX-11/750 Console subsystem. By typing in
simple console command language instructions through the console terminal, the user can instruct the CPU to perform a wide variety of functions. These instructions to the CPU can only be communicated when the console terminal is in console mode. In this mode, the console terminal is dedicated exclusively to controlling CPU functions such as examining and depositing data in memory. Console mode also allows the processor to be started, self-tested, initialized to a known state, single-stepped through instructions, or halted. In contrast, system terminal mode dedicates the console terminal to user application processes and the VAX/VMS operating system. In system terminal mode, the CPU ignores console commands.

Console Terminal Communications
The electrical interface for the console terminal is an industry-standard full-duplex EIA RS232-C line. The speed of the line is jumper-selectable to 300, 400, 600, 1200, 2400, 3600, 4000, 4800, 7200, 9600, 19,200, or 38,400 bits per second. This interface is a special port connected directly to the central processor.

CONSOLE MODES
The VAX-11/750 console runs in three modes: program I/O mode, console I/O mode, and RDM console mode. These modes are mutually exclusive. One of the three will always be enabled while there is power to the machine. In the program I/O mode (also known as the system terminal mode), the console functions like the other terminals on the VAX-11/750 system. In this mode the console passes characters between the terminal and the instruction set processor running in the CPU.

In the console I/O mode (also called console mode), the console program interprets and acts on commands typed on the console terminal. The VAX-11/750 will enter the console I/O mode whenever the instruction set processor halts.

Table 6-1 lists the actions that cause the CPU to halt and enter console I/O mode. Pressing the reset button simulates the power-down/power-up sequence. When the keylock switch is in either "secure" position, Control-P and the reset button are disabled.

When the processor enters the console I/O mode it types on the console terminal the address contained in the program counter (PC), a two-digit code which identifies the reason for the halt, and the console prompt symbol, >>>>. The prompt symbol shows that the console program is looping, waiting for a command.
Table 6-1  Console Halt Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Halt or Single Step from console</td>
</tr>
<tr>
<td>1</td>
<td>Successful T command</td>
</tr>
<tr>
<td>2</td>
<td>Control-P was typed on the console</td>
</tr>
<tr>
<td>4</td>
<td>Interrupt stack not valid or unable to read SCB</td>
</tr>
<tr>
<td>5</td>
<td>Double bus write error halt</td>
</tr>
<tr>
<td>6</td>
<td>Halt instruction with PSL&lt;CM&gt; = 0</td>
</tr>
<tr>
<td>7</td>
<td>SCB vector &lt;1:0&gt; = 3</td>
</tr>
<tr>
<td>8</td>
<td>SCB vector &lt;1:0&gt; = 2 and no UCS</td>
</tr>
<tr>
<td>A</td>
<td>CHMX while on the interrupt stack</td>
</tr>
<tr>
<td>B</td>
<td>CHMX SCB vector &lt;1:0&gt; .NE.0</td>
</tr>
<tr>
<td>11</td>
<td>Power up and can't find RPB, FPS1 at RE-START/HALT</td>
</tr>
<tr>
<td>12</td>
<td>Power up and warm start flag false FPS1 at RE-START/HALT</td>
</tr>
<tr>
<td>13</td>
<td>Power up and can't find good 64K of memory</td>
</tr>
<tr>
<td>14</td>
<td>Power up and booting, but bad Boot ROM or no ROM</td>
</tr>
<tr>
<td>15</td>
<td>Power up and cold start flag set during boot subroutine</td>
</tr>
<tr>
<td>16</td>
<td>Power up halt; Power On Action Switch at HALT position</td>
</tr>
<tr>
<td>FF</td>
<td>Micro Verify test failure</td>
</tr>
</tbody>
</table>

The third console mode is the RDM console mode. This mode is functional when the Remote Diagnosis Module (RDM) option is present in the system configuration. In the RDM console mode the microprocessor on the RDM board interprets and acts on the commands typed on the console terminal.
**Console Subsystem**

**FRONT PANEL CONTROLS AND INDICATORS**
The front panel of the VAX-11/750 has four switches and seven indicator lights. The switches and their functions are outlined as follows.

Figure 6-2 illustrates the VAX-11/750 front panel controls and indicators.

![Diagram of front panel controls and indicators]

**Figure 6-2**  VAX-11/750 Front Panel Controls and Indicators

**Off-Local-Remote**
This is a five-position keylock switch.

- **OFF**  No power to the CPU (except battery back up to the time-of-year clock) or to memory.
- **LOCAL**  The CPU responds to console commands and the remote diagnostic unit is completely disabled.
- **SECURE**  Console commands are ignored. The remote diagnostic unit is completely disabled, and the RESET switch is disabled.
- **REMOTE**  The console functions are enabled and may be activated from the remote line only. However, the remote line does have the capability to return control to the local terminal.
- **REMOTE/SECURE**  Console commands are ignored. The remote line replaces the console terminal, and the RESET switch is disabled. With the switch in this position, the remote diagnosis module may not be used to identify the source of system failures.
**Console Subsystem**

**Power On Action**
This four-position rotary switch controls the machine on a power-up sequence.

HALT  The machine comes up halted, in console mode.

BOOT  The machine bootstraps from the device selected by the boot device switch.

HALT/RESTART  A check is made to see if there is a valid Restart Parameter Block (RPB). If so, a re-start sequence is initiated. Otherwise, the machine halts.

BOOT/RESTART  Similar to Halt/Restart except that a bootstrap sequence occurs if RESTART is not successful.

**Boot Device**
This four-position rotary switch selects the device from which to boot. The VAX-11/750 CPU contains four sockets for built-in Read-Only Memory chips (ROMs) that contain the VAX-11/750 code needed to bootstrap a device. The switch selects which of the four ROMs is to provide the bootstrap code when a boot sequence is initiated. Typically, this switch is left in the position corresponding to the VAX/VMS system disk. Position A always causes a boot from the TU58 tape cartridge.

**Reset**
This pushbutton switch activates a system power-down sequence followed by a power-up sequence unless the keylock switch is on SECURE. The system will come up in the state selected by the POWER ON ACTION switch. It is usually used only if the machine appears to be hung and does not respond to console commands.

**CPU State Indicator Lights**

POWER  This green light indicates that DC power is present inside the CPU and that the keylock switch is not in the OFF position.

RUN  When lit, this green light indicates the machine is in the program I/O mode.

ERROR  When brightly lit, this red light indicates that the machine is stopped because of an unrecoverable control store parity error. To reset the machine, the RESET switch must be
Console Subsystem

used because console commands are not operational.
When glowing softly, the red light means that the CPU is functioning normally.

Remote Diagnostic Indicators
The following four lights are back-lit words, illuminated during various stages of remote diagnostic procedures. It should be noted that the REMOTE D, RD CARRIER, RD TEST, and RD FAULT lights only apply to systems having the optional remote diagnosis module installed by Field Service.

REMOTE D Remote diagnostics (RD) software illuminates this green light whenever the OFF-ON-LOCK switch is in one of the two remote positions.

RD CARRIER This green light is lit by the RD software whenever it detects that the remote port carrier is present. It indicates that the Remote Diagnostic Center (RDC) has established connection.

RD TEST The RDC software illuminates this amber light while tests are in progress.

RD FAULT This red light is lit by the Remote Diagnosis Module (RDM) if it detects a fault in its own logic. No tests should be attempted when the fault indicator is lit.

CONSOLE COMMAND LANGUAGE
The VAX-11/750 console command language gives the user an efficient way of communicating with the Console subsystem—using simple commands entered through the console terminal instead of the traditional toggle switches and lights. Thus, the VAX-11/750's console command language is a powerful tool, allowing the user to EXAMINE, DEPOSIT, INITIALIZE, CONTINUE, START, HALT, SINGLE STEP, SELF TEST, and BOOT as briefly described below.

<table>
<thead>
<tr>
<th>COMMAND</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Examine</td>
<td>Allows the user to look at physical and virtual memory, the processor registers, the general registers, and the Processor Status Longword</td>
</tr>
</tbody>
</table>
## Console Subsystem

<table>
<thead>
<tr>
<th>COMMAND</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deposit</td>
<td>Allows the user to make an entry into physical or virtual memory, the processor registers, the general registers, and the Processor Status Longword</td>
</tr>
<tr>
<td>Initialize</td>
<td>Allows the user to put the processor into a known state</td>
</tr>
<tr>
<td>Continue</td>
<td>Allows the user to restart a halted program without altering the state of the machine</td>
</tr>
<tr>
<td>Start</td>
<td>Initializes the processor and enables it to begin</td>
</tr>
<tr>
<td>Halt</td>
<td>No operation</td>
</tr>
<tr>
<td>Single Step</td>
<td>Allows the user to execute one instruction at a time</td>
</tr>
<tr>
<td>Boot</td>
<td>Allows the user to load the operating system and start the CPU</td>
</tr>
<tr>
<td>Self Test</td>
<td>Runs micro verify routines and initializes the processor</td>
</tr>
<tr>
<td>Binary Load</td>
<td>Permits the user to deposit a block of binary data in physical memory</td>
</tr>
<tr>
<td>Binary Unload</td>
<td>Allows the user to examine a block of binary data in physical memory</td>
</tr>
</tbody>
</table>

**NOTE**

The Binary Load/Unload commands are for special diagnostic use, and are not normally useful to the customer.

The console terminal utilizes the console command language to communicate instructions to the CPU. Commands are specified by a single letter with optional modifiers. When the CPU is not executing instructions, it is halted, and the console terminal is in console mode. In this mode, the CPU is receptive to console commands and can perform the functions previously listed. Direct Memory Access (DMA) activity to memory can also occur with the terminal in console mode, and all DMA transactions in progress will continue even when the CPU is halted. This allows the machine to be halted without destroying these transactions; however, interrupts will not be serviced while the
machine is halted. They will be serviced following a CONTINUE from the console.

Console Command Syntax and Semantics

Symbol | Function
---|---
< > | Angle brackets are used to denote category names. For example, the category name `<ADDRESS>` may be used to represent any valid address, instead of actually listing all the strings of characters that can represent an address.
[] | Parts of expressions in brackets do not need to be typed if defaults are used.
<SP> | Represents one typed space.
<COUNT> | Represents a numeric count in hexadecimal, 32 bits. Leading zeros may be omitted.
<ADDRESS> | Represents an address argument. Valid `<ADDRESS>` types are explained later in relation to specific commands. However, virtual addresses that reference nonexistent or nonresident pages will cause the console to abort execution of the console command which referenced that address. An appropriate error message will be displayed. Leading zeros may be omitted.
<DATA> | Represents a numeric argument. Leading zeros may be omitted.
<QUALIFIER> | Represents a command modifier (also called a switch). Valid `<QUALIFIER>` types are explained later in relation to specific commands.
<INPUT-PROMPT> | Represents the console’s input prompt string — `> >>`.
<CR> | Carriage return.
<LF> | Line feed.

Typing Errors and Illegal Characters

Typing errors may be corrected (before a `<CR>` sends the entire command to the CPU) using the DELETE key. When the key is typed, the console will echo the character being deleted (after printing a
backslash, \, upon receipt of the first deletion). The console will also add a backslash between the last deletion and the next input character.

Example:

operator types 1278<del><del>34
console prints 1278\87\34
console sees 1234

The console attempts to interpret each character as it is typed. If the console cannot interpret the next character in the context of the current command, it sounds a “beep” (bell) and ignores the character. This does not abort the entire command; instead, the command may be completed by typing the correct character(s).

Control Characters

Control-P This is typed by holding down the CONTROL key while typing a P. This command puts the machine in console I/O mode, halts the processor (see the HALT command), and causes the console terminal to type a halt message (PC<SP>02). Typing a Control-P while already in console I/O mode causes the console defaults to be reinitialized and the halt message to be typed on the console terminal.

Control-U Tells the system to ignore all characters typed since the last <CR> and types \<CR><LF><INPUT-PROMPT>.

Control-D Control-D causes the console to enter the RDM console mode, if the Remote Diagnosis Module is installed.

Console Command Language Instructions

BOOT Command

B[/X] [/n] [<SP><ddcu>] <CR>

This command boots the operating system or diagnostic software from the device specified. The hexadecimal number, <n>, specifies the boot control flags. The command deposits this number in register R5 before booting. For example, 200 sets bit 9 in R5, specifying boot control flag 9. <n> may be from one to four digits in length. If <n> is omitted, the default value for R5 is 0. The software boot control flags are actually implemented by VMS, not the VAX-11/750 CPU.
If /X is typed, the BOOT command will inhibit Micro Verify. This means that the Micro Verify tests are normally run at the beginning of the boot sequence.

<ddcu> represents the boot device, where dd is a two letter device mnemonic. Table 6-2 shows the boot device codes. The I/O channel adapter is specified by c. A, B, C, and D are possible values. U is a one-digit number that specifies the device drive number. If this is omitted, the device will be selected by the boot device switch, the channel adapter is A, and the drive number is 0. If /n or /X is used, then <ddcu> must be supplied.

**Table 6-2 Boot Device Codes**

<table>
<thead>
<tr>
<th>Device Code (dd)</th>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL</td>
<td>RL02</td>
</tr>
<tr>
<td>DM</td>
<td>RK06/7</td>
</tr>
<tr>
<td>DB</td>
<td>RP04/5/6/7, RM03</td>
</tr>
<tr>
<td>DD</td>
<td>TU58</td>
</tr>
</tbody>
</table>

Following a successful boot, the console enters program I/O mode (system terminal mode). If Micro Verify has been executed (/X was not typed), the console also prints a message telling whether the test revealed any errors (see the Test command).

**CONTINUE Command**

*C<CR>*

The CONTINUE command restarts execution of a halted program at the address currently contained in the Program Counter. The CPU is not initialized, and the console terminal enters system terminal mode once the CONTINUE command is issued.

**DEPOSIT and EXAMINE Commands**

*D[<QUALIFIER-LIST>]<SP><ADDRESS><SP><DATA><CR>*

*E[<QUALIFIER-LIST>][<SP><ADDRESS>]<CR>*

DEPOSIT and EXAMINE commands will be treated together because their formats are quite similar. Both commands require definition of the address space and the size of the operand in addition to the address. To make multiple EXAMINEs or DEPOSITs easier, there is a system of defaults for each of the items that must be specified for these commands. Some of the defaults are automatic (such as long-word size for general registers), and some are set up by the immediately preceding EXAMINE or DEPOSIT. All other console commands (except <CR>) cause the defaults to be set to their initial values. DEPOSIT and EXAMINE qualifiers are listed in Table 6-3.
### Console Subsystem

#### Table 6-3  DEPOSIT and EXAMINE QUALIFIERS

**Size Qualifiers**

<table>
<thead>
<tr>
<th>Qualifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/B</td>
<td>Byte</td>
</tr>
<tr>
<td>/W</td>
<td>Word</td>
</tr>
<tr>
<td>/L</td>
<td>Longword</td>
</tr>
</tbody>
</table>

**Space Qualifiers**

<table>
<thead>
<tr>
<th>Qualifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/V</td>
<td>Virtual Address</td>
</tr>
<tr>
<td>/P</td>
<td>Physical Address</td>
</tr>
<tr>
<td>/I</td>
<td>IPR</td>
</tr>
<tr>
<td>/G</td>
<td>GPR</td>
</tr>
</tbody>
</table>

**Address Values**

- nnnnnnnnn: Hex Number
- *: Last Location
- P: PSL
- +: Next Location
  - (DEPOSIT only)

**Data**

<table>
<thead>
<tr>
<th>Hex Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>nnn</td>
</tr>
</tbody>
</table>

DEPOSITs (writes) or EXAMINEs (reads) `<DATA>` at the `<ADDRESS>` specified. The address space and size used will depend upon the qualifier or qualifiers specified with the command. If no address space qualifier is used, the default is physical address space; following another DEPOSIT or EXAMINE, the same space as that of the previous command will be used as the default.

If no size qualifier is typed, the default for a physical or virtual DEPOSIT or EXAMINE is longword; however, following another EXAMINE or DEPOSIT, the size used in the previous command will be the default. The size for an IPR or GPR DEPOSIT/EXAMINE is always longword, and these commands do not change the current size default.

`<ADDRESS>` must be one to eight hexadecimal digits. The initial default is zero; however, the default is unpredictable when the address space is changed. Following another virtual or physical DEPOSIT or EXAMINE, the default is the sum of the address from the last EXAMINE/DEPOSIT plus size from the last EXAMINE/DEPOSIT. Typing a + for `<ADDRESS>` (for DEPOSIT only) will also get this default. Following another IPR or GPR EXAMINE/DEPOSIT, the default is the sum of the address from the last EXAMINE/DEPOSIT plus one. Using a P for `<ADDRESS>` does a longword reference of the Processor Status.
Console Subsystem

Longword, independent of address space and size. Using P has no effect on any of the defaults.

<Data> must be represented by one to eight hex digits. If more digits than specified by the size are supplied, the extra digits on the left are ignored; if fewer digits are supplied, zeros are appended to the left.

Sample DEPOSIT response: A successful DEPOSIT always receives <INPUT-PROMPT>.

Sample EXAMINE responses: console output underlined

>>> E/P 1234 ! Examine physical
! address 1234
P 00001234
ABCDEF89

>>> E/V 1234 ! Examine virtual
! address 1234
P 00005634 01234567 ! Note that
! virtual
! EXAMINENEs display
! the translated
! physical
! address

>>> E/G 0 ! Examine general
! register
! R0

G 00000000
98765432

Index of EXAMINENEs and DEPOSITNs

E<SP>*<CR> ! Examine the last location
! examined or deposited into
E<CR> ! Examine the next location
E<SP> ! Examine <ADDRESS>; all
<ADDRESS><CR> ! switches are defaulted to
! last EXAMINE
! or DEPOSIT
E/G<SP> ! Examine GPR; register
<ADDRESS><CR> ! number is <ADDRESS>;
! <ADDRESS> must be
! a value from 0 to
! F hexadecimal

115
Console Subsystem

E/I<SP><ADDRESS><CR>  ! Examine IPR; register number is <ADDRESS>.
E/P<SP><ADDRESS><CR>  ! Examine physical <ADDRESS>.
E/V<SP><ADDRESS><CR>  ! Examine virtual <ADDRESS>.
E<SP>P<CR>           ! Examine PSL
E/W/P<SP><ADDRESS><CR> ! Examine a word at physical <ADDRESS>.
E/P/W<SP><ADDRESS><CR> ! Examine a word at physical <ADDRESS>.
E/L/V<SP><ADDRESS><CR> ! Examine a longword at virtual <ADDRESS>.
D<SP>*<SP><DATA><CR>  ! Deposit <DATA> in the last location that was deposited into or examined.
D<SP>+<SP><DATA><CR>  ! Deposit <DATA> in the next sequential address.
D/G<SP><ADDRESS>-<SP><DATA><CR> ! Deposit <DATA> in GPR <ADDRESS>.
D/I<SP><ADDRESS>-<SP><DATA><CR> ! Deposit <DATA> in IPR <ADDRESS>.
D/P<SP><ADDRESS>-<SP><DATA><CR> ! Deposit <DATA> in physical <ADDRESS>.
D/V<SP><ADDRESS>-<SP><DATA><CR> ! Deposit <DATA> in virtual <ADDRESS>.
D<SP>P<SPACE><DATA><CR> ! Deposit <DATA> in PSL.
D<SP><ADDRESS>-<SP><DATA><CR> ! Deposit <DATA> in <ADDRESS>.
                          ! Switches are defaulted to previous switches.
D/V/W<SP><ADDRESS>-<SP><DATA><CR> ! Deposit a word of <DATA> in virtual <ADDRESS>.
D/L/P<SP><ADDRESS>-<SP><DATA><CR> ! Deposit a longword of <DATA> in physical <ADDRESS>.
Console Subsystem

INITIALIZE Command
I<CR>

This command performs the following functions:
- Initialize the processor
- Clear the translation buffer
- Clear the cache; turn on the cache
- Set the program counter to 0
- Set the processor status longword to 041F0000

HALT Command
H<CR>

The Halt command is implemented in the VAX-11/750 for the sake of consistency with the VAX-11/780. It does not actually halt the CPU since the CPU must already be halted to respond to the command. However, the Halt command does reset the console defaults.

Table 6-1 defines the various console halt codes.

NEXT Command
N<CR>

The NEXT command causes the CPU to execute one ISP level instruction. The CPU then halts and re-enters the console mode.

START Command
S[<SP><ADDRESS>]<CR>

The START command is normally used to start execution of programs that run without the operating system and without the diagnostic supervisor. START performs the equivalent of the following console commands:
1. Initialize the CPU.
2. Deposit <address> into the Program Counter (PC). If no address is specified, the current value of the PC is used.
3. Perform the Continue function to begin ISP level CPU execution.

Programs which may be started this way must be loaded into main memory before the START command is given. For example:

>>>S 1000  ! Start the program that begins at
          ! address 1000.
Consol e Subsystem

TEST Command
T<CR>
This command runs the Micro Verify program and initializes the processor. The Micro Verify program types a percent sign (%) on the terminal when it starts the test.

If all the tests run successfully, Micro Verify types a second %. If Micro Verify detects a failure, it types a single error character and then halts the processor in the console I/O mode. When the processor halts, the console types an error code in place of the PC and a halt code. The halt code shows that this halt is due to an error condition in Micro Verify.

Response if successful:
<CR><LF>%<CR><LF><INPUT-PROMPT>

Response if unsuccessful:
<CR><LF>%<ERROR INDICATOR><CONSOLE HALT MESSAGE>

BINARY LOAD/UNLOAD Command
X<SP><ADDRESS><SP><COUNT><CR><CHECKSUM>

<ADDRESS> starting address of load

<COUNT> number of bytes to be transferred

<CHECKSUM> 2's complement checksum of command string or binary data

The BINARY LOAD/UNLOAD command instructs the console to load binary data into or unload binary data from physical memory, starting from the location specified by <ADDRESS>. A <COUNT> with bit <31> set indicates BINARY UNLOAD (data sent to the register). The remaining bits in <COUNT> indicate the number of bytes to Load orUnload. After a correct 2's complement checksum calculation, the console issues an <INPUT-PROMPT>, but remains in binary mode and either sends data to the usepreparies to receive data. If the checksum shows an error, a message and an <INPUT-PROMPT> are issued.

In LOAD, a binary string of data, of length <COUNT> + 1, will be sent once the <INPUT-PROMPT> indicates that the console has accepted the command. When <COUNT> is exhausted, the final byte is a console-calculated block checksum of all the data. A successful transfer issues an <INPUT-PROMPT>. An error in the checksum will generate an error message (see Table 6-4 for error codes).

For UNLOAD, the console processes the command and <CHECK-
Console Subsystem

SUM>. Then, if the checksum is correct, the console responds with <INPUT-PROMPT>, followed by a string of bytes which is the binary data requested. A second checksum is calculated and processed as for the LOAD sequence.

Console Command Errors
When a command is given that the console cannot properly process, it responds by typing ?nn. nn is an error code that describes the nature of the problem. Error codes are listed in Table 6-4.

<table>
<thead>
<tr>
<th>Table 6-4  Console Command Error Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>10  Illegal GPR register number</td>
</tr>
<tr>
<td>11  Illegal access of an IPR</td>
</tr>
<tr>
<td>19  Reference to next location was preceded by an EX-AMINE or DEPOSIT to an IPR or PSL</td>
</tr>
<tr>
<td>20  ACV, TNV, or Machine Check during a read or write</td>
</tr>
<tr>
<td>30  Binary transfer checksum error</td>
</tr>
<tr>
<td>33  Unrecognizable boot device</td>
</tr>
</tbody>
</table>

NOTE
The console ignores illegal characters and echoes them as bell codes.

INTEGRAL TU58 CARTRIDGE TAPE DRIVE
The TU58 tape cartridge drive is an important part of the Console subsystem. Because the TU58 is connected directly to the CPU, it retains the ability to run diagnostics even with some system components inoperative. This feature significantly increases system maintainability. The TU58 may also be used to boot the system, to load files into physical memory, and to store files which describe and execute site-specific bootstrap procedures (see BOOTING THE VAX-11/750 SYSTEM later in this chapter).

The tape cartridge is preformatted to store 2048 records, each containing 128 bytes. The controller provides random access to any record. The TU58 searches at 60 inches per second (i/s) to find the file requested, then reads at 30 i/s. Data read from the tape are verified through checksums at the end of each record or header. All data transfers between the TU58 and the host are in 512-byte blocks, with the TU58 concatenating four 128-byte records. Data are transferred to the CPU at approximately 2 Kb per second.
BOOTING THE VAX-11/750 SYSTEM
Initializing or booting the VAX-11/750 system can be viewed from two different perspectives. First, there are the actual steps that the system operator must perform in order to boot the system. And second, there are the actual events that occur within the system during the boot process.

From the system operator’s standpoint, the booting process is quite simple. For a typical system boot:

1. Turn on power to the console terminal.
2. Turn the POWER ON ACTION switch on the VAX-11/750 front panel to the HALT position. The POWER ON light should now be lit. The console terminal will print:
   
   %
   00000000 16
   >>>

3. Check that the VMS operating system disk pack is in drive 0 and the READY light is on. (The disk READY light can take from 20 to 60 seconds to light from Power On until a Ready condition exists.)
4. Set the BOOT DEVICE switch to the position corresponding to the system device. This action matches the correct boot ROM with the VMS system device. The BOOT DEVICE switch settings are: A (TU58), B (RK07), C (RL02), and D (MASSBUS).
5. Set the POWER ON ACTION switch to the RESTART/BOOT position.
6. Press the RESTART button.
7. The console terminal should type:
   
   %
   00000000 16
   %

This is followed by the VAX/VMS boot message.

If this procedure does not work, it is possible that either drive 0 is not functioning properly or that the VAX system pack is no longer valid.

Console Subsystem Action on Boot
The overall flow of control for booting passes through a number of stages on the VAX-11/750. An overview of this flow is followed by a detailed specification for each part that is part of the VAX-11/750 hardware.
Console Subsystem

The Console subsystem initializes the CPU, finds 64 Kb of good memory, and passes the address of that memory to the Device ROM which brings in block 0 (the bootblock) of the boot device into the first page of good memory and then transfers control to it.

The bootblock code uses the Device ROM as a callable driver for the device and then uses it to bring in whatever file it needs to bring the system up.

There are five distinct ways that a boot sequence may be initiated on the VAX-11/750. They are:

1. Power-up sequence (RESET button or power restoration) with switch set to BOOT.
2. Typing the "B" command while in console mode.
3. Execution of a HALT instruction when the processor is in kernel mode and the POWER ON ACTION switch is in the BOOT position.
4. Executing the MTPR (Move to Processor Register) to the console register that invokes a boot.
5. Failure of a warm start (restart) while the POWER ON ACTION switch is in the RESTART/BOOT position.

All five of these mechanisms will initiate the following sequence of actions. The first two will not perform the first step listed. This implies that the first two will cause a boot independent of the state of the cold start bit, while the last three will not.

1. Clear the cold start flag.
3. Initialize PSL to 041F0000\textsubscript{16}.
4. Locate 64 Kb of page-aligned usable memory to be used during bootstrapping.
5. Initialize UBI0.
6. Load and validate the first 128 UBI0 map registers to address the first 64 Kb of usable memory.
7. Load contents of all four boot ROMS into memory (base + XFA00).
8. Check the cold start flag. If set, Halt. If clear, set it. This prevents cold start looping.
9. Load input arguments to be used by the Device ROM code and by VMS.
10. Select the boot ROM specified by the BOOT DEVICE switch.
11. Check for non-existent ROM; halt if no ROM.
The general registers receive the input arguments from the console subsystem.

R1  System bus address of a MASSBUS adapter (MBA0 unless otherwise specified in the BOOT command)

R2  Physical address of the UNIBUS I/O page associated with the UNIBUS adapter (UBIO unless otherwise specified in the BOOT command)

R3  Device unit number (0 unless otherwise specified in the BOOT command)

R5  Software boot control flags (0 unless otherwise specified in the BOOT command)

SP  <base address + ↑X200> of the 64 Kb of good memory

The Console subsystem then transfers control to the second word in the selected ROM code to begin macro instruction execution of the ROM code. The table below shows the four starting addresses.

<table>
<thead>
<tr>
<th>Device ROM</th>
<th>Starting Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>FA02</td>
</tr>
<tr>
<td>B</td>
<td>FB02</td>
</tr>
<tr>
<td>C</td>
<td>FC02</td>
</tr>
<tr>
<td>D</td>
<td>FD02</td>
</tr>
</tbody>
</table>

**Device ROM Code Function on Boot**

The Device ROM code consists of four components:

- An ASCII description of the device type
- A control routine: gains control from microcode
- A driver subroutine
- A checksum

ASCII device description:

- Located at bytes 0-1 of the ROM
- Consists of two ASCII characters describing device type; characters are written in reverse order

Examples:

```
.ASCII /MD/    ; RK06/7 disk drive
.ASCII /BD/    ; MASSBUS disk drive
```

Control Routine, Located at bytes 2-n of the ROM

Function: To prepare inputs for and then call the driver subroutine to bring in the bootblock from the device;
**Console Subsystem**

when the driver subroutine returns, check for errors; if no error, prepare inputs for and call bootblock code; if error, halt

**Inputs:**
R1: address of a MASSBUS adapter
R2: address of a UNIBUS I/O page
R3: unit number of a boot device
R5: software boot control flags
SP: base address of usable memory + \( \uparrow \times 200 \)

**Implicit inputs:**
UBIO map registers are marked as valid and mapped to the 64 Kb of usable memory

The bootblock is at LBN 0

The bootblock will be brought into the first page of good memory

The address stored in SP is mapped to the second UBI map register

Base of good memory + C: transfer address of bootblock code

**Outputs:**
R0: type of boot device
  0 MASSBUS Device
  1 RK06/7
  2 RL02
  3-63 Reserved for the future
  64 TU58

R1: (UNIBUS) address of the UNIBUS I/O page
(UNIBUS) address of the boot device's adapter

R2: (UNIBUS) address of the boot device's CSR
(MASSBUS) controller number of the boot device

R6: address of driver subroutine in ROM

**Preserves:** R3, R5, R10, R11, AP, SP

**Registers:**

**Driver Subroutine:**
Located after control subroutine in ROM.
Function is to read 1 block of data from device into memory. Called with JSB.

**Inputs:**
R1: (UNIBUS) address of a UNIBUS I/O page
(MASSBUS) address of the boot
Console Subsystem

device’s adapter
R2: (UNIBUS) address of the boot
device’s CSR
(MASSBUS) controller
number of the boot device
R3: Unit number of a boot device
R5: Offset from beginning of 64 Kb
block at which data are to
be written
R8: LBN of block to be read on the
boot device
4(SP) address in memory at which data
are to be written

Outputs: R0: <0>@1 indicates success
0 indicates failure

Preserves: R1-R6, R10, R11,AP,SP

Devices which use the UNIBUS map require the offset from the base of
good memory since the console will set up the UNIBUS map relative to
that base. This is provided in R5. Devices which need the actual physical
address in memory get it at 4(SP).

Checksum
Located at byte 255 (the last byte in the ROM)
Consists of the sum (discarding carries) of the first 255 bytes in the
ROM.

Checksum
Located at byte 255 (the last byte in the ROM)
Consists of the sum (discarding carries) of the first 255 bytes in the
ROM.

Console Subsystem Action on a Warm Start
When the Console subsystem gains control of the VAX-11/750 follow-
ing a power restoration, activation of the RESET switch, or CPU Halt, it
examines the POWER ON ACTION switch. The Console subsystem will
attempt to restart the CPU in either of two cases.

1. When the POWER ON ACTION switch is set to the RE-
START/HALT position.

2. When the POWER ON ACTION switch is set to the RE-
START/BOOT position.

If the POWER ON ACTION switch is set to either RESTART/HALT or
RESTART/BOOT, the Console subsystem searches through physical
memory for a valid restart parameter block (RPB), shown as follows.

124
Figure 6-3 represents a memory map of boot process.

![Memory Map of Boot Process Diagram](image)

**Figure 6-3  Memory Map of Boot Process**

<table>
<thead>
<tr>
<th>Physical Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Physical address of the RPB</td>
</tr>
<tr>
<td>4</td>
<td>Physical address of the restart routine</td>
</tr>
<tr>
<td>8</td>
<td>Checksum of $\uparrow$X1F longwords of restart routine</td>
</tr>
<tr>
<td>C</td>
<td>Bit 0; (\uparrow) warm start flag</td>
</tr>
</tbody>
</table>

Parameter Block, First Four Longwords

A valid RPB is defined as a block of four longwords, starting on a page boundary. The first longword points to itself. The second is a pointer to the address of the restart code and must not be zero. The third longword contains the checksum (sum, throwing away carries) of the 31 longwords pointed to by the second longword. The console subsystem starts at address zero and searches all available memory for a valid RPB. If it doesn't find one it performs the second action specified by the power on switch (HALT or BOOT).

If it does find an RPB, it examines bit 0 of the fourth longword of the RPB. If this bit is 1, it will halt or boot as specified by the power-on switch.

If this bit is 0, it sets it and then:
Console Subsystem

- Loads the stack pointer (SP) with the address of the RPB plus \( \uparrow X200 \).
- Loads the argument pointer (AP) with a value that indicates the cause of the restart. If the restart is occurring because of power restoration or reset switch activation, the AP is loaded with a 3. If it is because of a CPU Halt, it is loaded with one of the codes specified in Table 6-1.
- Starts execution of the restart routine, whose address is located at the second longword of the RPB.
<table>
<thead>
<tr>
<th>FEATURES</th>
<th>BENEFITS</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit microprogrammed</td>
<td>Provides the user with high-performance data throughput</td>
</tr>
<tr>
<td>processor</td>
<td></td>
</tr>
<tr>
<td>Memory management hardware</td>
<td>Allows the user to directly address up to four billion bytes of virtual</td>
</tr>
<tr>
<td></td>
<td>address space using a smaller physical memory</td>
</tr>
<tr>
<td>Gate array technology</td>
<td>Reduces the physical size and power consumption of the CPU and increases</td>
</tr>
<tr>
<td></td>
<td>system reliability</td>
</tr>
<tr>
<td>4 Kb direct mapped memory</td>
<td>Significantly improves memory access time, increasing overall performance</td>
</tr>
<tr>
<td>cache</td>
<td></td>
</tr>
<tr>
<td>PDP-11 compatibility mode</td>
<td>Gives the PDP-11 user an easy migration path to the VAX/VMS</td>
</tr>
<tr>
<td>User control store (optional)</td>
<td>Increases throughput by allowing the user to create routines in</td>
</tr>
<tr>
<td></td>
<td>microcode tailored to specific applications</td>
</tr>
</tbody>
</table>
INTRODUCTION
The VAX-11/750 central processing unit (CPU) performs the logic and arithmetic operations requested by the computer system user. The processor is a high-performance, microprogrammed computer that executes a large set of variable-length instructions in native mode, and nonprivileged PDP-11 instructions in compatibility mode.

The CPU uses 32-bit virtual addresses, allowing access to over four gigabytes (4 Gb, $2^{32}$) of virtual address space. These addresses are called virtual because each address is not necessarily the actual address in physical memory. The processor's memory management hardware translates virtual addresses to physical addresses.

The processor provides sixteen 32-bit registers that can be used for temporary storage, as accumulators, index registers, and base registers. Four of these registers have special significance: the Program Counter, and three registers that are used to provide an extensive CALL facility.

The native instruction set is highly versatile and bit-efficient. It includes integer, packed decimal, character string, bit field, and floating point instructions, as well as program control and special instructions. Instructions and data are variable in length and can start on any byte boundary or, in the case of bit fields, at any arbitrary bit in memory.

The VAX-11/750 processor includes these functional hardware components:

- 4 Kb direct mapped memory cache
- 8-byte prefetch instruction buffer
- 512-entry address translation buffer
- Time-of-year-clock
- Integral memory management
- Programmable real time clock
- Optional 10 Kb (80 bits wide) user control store

Figure 7-1 illustrates the central processing unit.
This chapter is divided into two sections. The first section discusses processor hardware and functionality, and the second section introduces the programming concepts and characteristics of the processing system.

GATE ARRAY TECHNOLOGY
The VAX-11/750 central processor uses state-of-the-art gate array technology. Three-quarters of the logic circuits in the CPU are custom LSI (Large Scale Integrated) circuits designed by DIGITAL. In the basic CPU and memory controller there are 55 LSI devices using 27 different circuit designs, with each LSI device having up to 488 logic gates. With gate array technology, the physical size of the processor and its power consumption can be reduced to approximately half of what they would be using conventional technology. In addition, this technology requires fewer components, for enhanced system reliability.

HARDWARE ELEMENTS
The VAX-11/750 CPU is a high-performance, 32-bit microprogrammed computer. Its speed and performance come from a CPU that can handle several independent functions simultaneously.

The CPU can process the following kinds of data:
- Bits (up to 32)
- Bytes (8 bits)
- Words (16 bits)
Central Processor

- Longwords (32 bits)
- Quadwords (64 bits)
- 32-bit floating point (single precision)
- 64-bit floating point (double precision)
- Packed decimal (up to 31 digits)
- Character strings (up to 64 Kb)

The following sections describe the VAX-11/750 processor hardware.

Control Store
The control store is a programmable read-only memory containing 6K 80-bit microwords. The control store contains the program that describes the operation and sequencing of the central processing unit. It also contains the native and PDP-11 compatibility mode instruction sets. In addition, the control store contains an 80-bit buffer, enabling it to execute one microword while simultaneously fetching the next.

Data Paths
The data path subsystem consists of three parallel sections used to process addresses and data specified by the instruction set. The arithmetic section is used to perform both arithmetic and logical operations on data and addresses, and is used for fast processing of floating point instructions. The data shift and rotate section packs and unpacks floating point and decimal string data. Finally, the address section calculates virtual addresses for the translation buffer.

4 Kb Direct Mapped Memory Cache
Cache memory is a small, high-speed memory that maintains a copy of frequently selected portions of main memory for faster access to instructions and data. The cache memory is loaded by memory management so that there is a high probability the desired data will be in the cache. Because the processor always looks for data in the cache first, the cache offers faster system speed for the cost of a small quantity of fast memory and associated logic.

The VAX-11/750 memory cache is a 4 Kb, direct mapped, write-through cache. It is used for all data coming from memory, including addresses and instructions. The write-through feature protects the integrity of memory because memory contents are updated immediately after the processor performs a write. Instead of tying up the processor while main memory is updated, VAX-11/750 processors buffer commands to avoid waiting while main memory is updated from the cache.
Central Processor

The memory cache (typically 90% hit rate) provides the central processor with high-speed access to main memory. The memory cache typically cuts memory access time to half what it would be with no cache feature. For increased integrity, the cache memory system carries byte parity for both data and addresses. Cache locations are allocated when data are read from memory or when a longword is written to memory. This cache also watches I/O transfers and updates itself appropriately. Thus, no operating system overhead is needed to synchronize the cache with I/O operations, and applications software does not need to be concerned with the cache.

Address Translation Buffer
The address translation buffer is a cache of frequently used physical address translations. It significantly reduces the amount of time the CPU spends on the repetitive task of dynamic address translation. The cache contains 512 virtual-to-physical page address translations which are divided into equal sections: 256 system space page translations and 256 process space page translations. Each of these sections is two-way associative and has parity on each entry for increased integrity.

8-Byte Prefetch Instruction Buffer
The 8-byte instruction buffer improves CPU performance by prefetching data in the instruction stream. The control logic fetches longword data from memory to keep the 8-byte buffer full.

Processor Clocks
The VAX-11/750 processor contains a programmable real-time clock, and a time-of-year and date clock. The interval or real-time clock permits the measurement of finely resolved variable intervals. The real-time clock is based on a crystal oscillator with accuracy of 0.01%, and resolution (update rate) of one microsecond. The time-of-year clock is used by software to perform various timekeeping functions. It provides the correct time to the system without operator intervention in the event of a power failure or other system crash. To insure continuous running, the time-of-year clock is equipped with its own battery backup.

Optional 10 Kb User Control Store (UCS)
The user control store consists of 1024 80-bit control words for a total of 10 Kb. These locations are available to the customer for augmenting the speed and power of the basic machine with customized microcode functions. For greater detail, see Chapter 12 of this book.
PROGRAMMING CONCEPTS
A program is a stream of instructions and data that a user can request the operating system to translate, link, and execute. An executable program is called an image. When the machine runs an image, the context or environment in which the image is executed is called a process. A process is the complete unit of execution in this computer system. Process context includes the definition of the virtual address space in which an image executes; it also includes the state of the image it is currently executing and the limitations on what an image is allowed to do. These limitations depend on the privileges of the user who runs the image. An image executing within a process context controls its execution by using one of the instruction sets, the general purpose registers, and the Processor Status Word. These hardware resources are discussed in greater detail later in this chapter.

VIRTUAL ADDRESS SPACE
Most data are located in memory using the address of an 8-bit byte. The programmer uses a 32-bit virtual address to identify a byte location. A virtual address is not necessarily a unique address of a location in memory, as is a physical memory address. For example, two processes might use the same virtual address to refer to two different physical memory locations. Similarly, a single physical memory location might be referenced using different virtual addresses.

The set of all possible 32-bit virtual addresses is called virtual address space. Virtual address space can be viewed as an array of $2^{32}$ byte locations, or over four billion bytes. The processor and operating system provide a multiprogramming environment by dividing virtual address space into two halves: one half for addressing process context-dependent code and data, and one half for addressing code and data which are process context-independent. The set of virtual addresses designated for use by a process, including an image it executes, is called process space. Addresses in the other half of virtual address space are used to refer to locations maintained, occupied, and protected by the operating system. This part of virtual address space is called system space.

Process Space Structure
Process space represents approximately two billion bytes. An image executing in the context of a process, and the operating system on behalf of the process, use addresses in process space to refer to code and data particular to that process context. A process cannot reference virtual addresses in any process space but its own. Thus, code and data belonging to a process are automatically protected from other processes in the system (although explicit sharing is per-
mitted—see Chapter 4, Memory Management, and Chapter 5, System Architectural Characteristics).

Process space is subdivided into two regions. Addresses in the first region, called the program region, identify the location of image code and data. Addresses in the second region, called the control region, usually refer to stacks and other temporary program image and permanent process control information maintained by the operating system on behalf of the process. Program region addresses are allocated from address 0 (zero) to higher virtual addresses, and control region addresses are allocated from address 7FFFFFFF₁₆ $(2^{31} - 1)$ to lower virtual addresses.

**System Space Structure**

As described above, the second half of virtual address space is called system space. The operating system assigns the meanings to addresses in system space. The significance of any address in system space is the same for every process, independent of process context.

System space is also subdivided into two regions. The operating system assigns the system region addresses for linkages to its service procedures, for memory management data, and for I/O processing routines. The second region is presently unavailable and is reserved for future use.

**INSTRUCTION SETS**

The VAX-11/750 processor is capable of executing instructions in either of two modes: native or compatibility. In native mode, the processor executes a large set of variable-length instructions, recognizes a variety of data types, and uses sixteen 32-bit general purpose registers. In compatibility mode, the processor executes the user set of PDP-11 instructions, recognizes integer data, and uses eight 16-bit general purpose registers. These instruction sets are closely related and their programming characteristics are similar. Therefore, a user process can execute both native mode and compatibility mode images. Each image, however, is constrained to a single instruction set—that is, native mode and compatibility mode instructions must not be mixed in the same image.

The native mode instruction set is considerably more extensive than that of compatibility mode execution. The native mode instruction set consists of 248 basic instructions, many of which correspond directly to high-level language statements. Additionally, the native mode instruction set includes floating point operations, character string manipulations, packed decimal arithmetic, and many instructions which improve both the performance and the memory utilization of systems and applications software.
Central Processor

A native instruction consists of an operation code (opcode) and zero or more operands, which are described by data type and addressing mode.

Data Types
The data type of an instruction operand identifies how many bits of storage are to be treated as a unit, and what the interpretation of that unit is. The processor's native instruction set recognizes four primary data types: integer, floating point, packed decimal, and character string. In addition, the processor can also manipulate a fifth data type, the bit field, in which the programmer defines the size of the field and its relative position. Table 7-1 illustrates the VAX-11/750 data types.

The address of any data item is the address of the first (lowest numbered) byte in which the item resides. All integer, floating point, packed decimal, and character string data can be stored starting on an arbitrary byte boundary. A bit field, however, does not necessarily start on a byte boundary. A field is simply a set of contiguous bits whose starting bit location is identified relative to a given byte address. The native instruction set can interpret a bit field as a signed or unsigned integer.

Addressing Modes
The processor's addressing modes allow an instruction operand to be located in main memory, in a register, in the instruction stream itself, or it can be an immediate constant. The way in which the programmer chooses to specify an operand location is called the operand addressing mode. The processor offers a variety of addressing modes and addressing mode optimizations. One addressing mode locates an operand in a register; and six addressing modes locate an operand in memory, using a register to point to the operand, point to a table of operands, or point to a table of operand addresses.

The seven basic addressing modes are:

- Register mode
- Register deferred mode
- Autodecrement mode
- Autoincrement mode
- Autoincrement deferred mode
- Displacement mode
- Displacement deferred mode

There also are six addressing modes that are indexed modifications of these addressing modes that locate an operand in memory. All the above modes, except register mode, can be modified by an index.
### Table 7.1 Data Types

<table>
<thead>
<tr>
<th>DATA TYPE</th>
<th>SIZE</th>
<th>RANGE (decimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Signed</td>
</tr>
<tr>
<td>Integer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Byte</td>
<td>8 bits</td>
<td>-128 to +127</td>
</tr>
<tr>
<td>Word</td>
<td>16 bits</td>
<td>-32768 to +32767</td>
</tr>
<tr>
<td>Longword</td>
<td>32 bits</td>
<td>-2³¹ to +2³¹</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-1</td>
</tr>
<tr>
<td>Quadword</td>
<td>64 bits</td>
<td>-2⁶³ to +2⁶³</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-1</td>
</tr>
<tr>
<td>Floating Point</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F_floating</td>
<td>32 bits</td>
<td>approximately seven decimal digits precision (.29 × 10⁻³⁸ to 1.7 × 10³⁸)</td>
</tr>
<tr>
<td>D_floating</td>
<td>64 bits</td>
<td>approximately sixteen decimal digits precision (same as F_floating)</td>
</tr>
<tr>
<td>G_floating</td>
<td>64 bits</td>
<td>approximately fifteen decimal digits precision (.56 × 10⁻³⁰⁸ to .9 × 10³⁰⁸)</td>
</tr>
<tr>
<td>H_floating</td>
<td>128 bits</td>
<td>approximately thirty-three decimal digits precision (.84 × 10⁻⁴⁹³² to .59 × 10⁴⁹³²)</td>
</tr>
<tr>
<td>Packed</td>
<td>0 to 16 bytes</td>
<td>numeric, two digits per byte</td>
</tr>
<tr>
<td>Decimal String</td>
<td>(31 digits)</td>
<td>sign in low half of last byte</td>
</tr>
<tr>
<td>Character String</td>
<td>0 to 65535 bytes</td>
<td>one character per byte</td>
</tr>
<tr>
<td>Variable-length Bit Field</td>
<td>0 to 32 bits</td>
<td>dependent on interpretation</td>
</tr>
</tbody>
</table>
Central Processor

register in this way. When an index register is used with a basic mode to identify an operand, the addressing mode is the name of the basic mode with the suffix *indexed*. For example, the indexed addressing mode for register deferred is called *register deferred indexed mode*.

Finally, there are two addressing modes that identify the location of the operand in the instruction stream: one for constant data (integer or floating point) and one for BRANCH instruction addresses. Table 7-2 summarizes the VAX-11/750 addressing modes.

**Table 7-2  Addressing Modes**

<table>
<thead>
<tr>
<th>Literal (Immediate)</th>
<th>Register</th>
<th>Register Deferred</th>
<th>Autodecrement</th>
<th>Autoincrement Deferred</th>
<th>(Absolute)</th>
<th>Displacement</th>
<th>Displacement Deferred</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Rn</td>
<td>(Rn)</td>
<td>- (Rn)</td>
<td>(Rn) +</td>
<td>@ # address</td>
<td>(Rn)</td>
<td>(Rn)</td>
</tr>
<tr>
<td>It</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>displacement</td>
<td>displacement</td>
</tr>
</tbody>
</table>

n = 0 through 15
x = 0 through 14
REGISTERS
A register is special hardware within the processor that can be used for temporary data storage and addressing. Instruction operands are often stored in the processor’s general registers or accessed through them; but instructions themselves are not stored in registers. Registers can be also be used as accumulators, as base registers, and as index registers. A base register contains the address of the base of a software data structure such as a table or queue, and an index register contains a logical offset into a data structure. Whenever a register is used to contain data, the data are stored in the register in the same format as would appear in memory. If a quadword or double floating datum is stored in a register, it is actually stored in two adjacent registers. For example, storing a double floating number in R7 loads both R7 and R8.

The programmer has sixteen 32-bit general registers available for use with the native instruction set. These registers, are labelled R0 through R15 (decimal). Registers 12 through 15 have special significance depending on the instruction being executed, and therefore have special names. These special registers are:

- The Program Counter (PC or R15), which contains the address of the next byte to be processed in the instruction stream
- The Stack Pointer (SP or R14), which contains the address of the base (also called the top) of a stack maintained for subroutine and procedure calls
- The Frame Pointer (FP or R13), which contains the address of the base of a software data structure stored on the stack called the stack frame, which is maintained for procedure calls
- The Argument Pointer (AP or R12), which contains the address of the base of a software data structure called the argument list, which is maintained for procedure calls

Except for the Program Counter, a register’s special significance does not preclude its use for other purposes. The Program Counter cannot be used as an accumulator, as a temporary register, or as an index register. In general, however, the Stack Pointer, Argument Pointer, and Frame Pointer are used expressly for the functions their names describe. These registers are discussed in greater detail later in this chapter.

STACKS, SUBROUTINES, AND PROCEDURES
A stack is an array of consecutively addressed data items that are referenced on a last-in/first-out basis using a general register. Data items are added to and removed from the low address end of the
Central Processor

stack. A stack grows toward lower addresses as items are added, and shrinks toward higher addresses as items are removed.

A stack can be created anywhere in user program address space and any register can point to the current item on the stack. The operating system, however, automatically reserves portions of each process address space for stack data structures. User software references this stack data structure—called the user stack—through a general register designated as the Stack Pointer.

A stack is a powerful data structure. It is reusable, recursive—and it enables the programmer to write re-entrant routines because the processor can handle routine linkages automatically using the Stack Pointer. The subroutine linkage of the VAX architecture also facilitates programming since the return address and other information about the processor status are pushed onto the stack. Routines can also be recursive because arguments can be saved on the stack for each successive call of the same routine.

The processor provides two kinds of routine CALL instructions: those for subroutines, and those for procedures.

The processor provides more elaborate routine linkages for procedures than for subroutines. The processor automatically saves and restores the contents of registers the programmer wants preserved across procedure calls. The processor provides two methods for passing argument lists to called procedures: by passing the arguments on the stack, or by passing the address of the arguments elsewhere in memory. The processor also constructs a list or record of procedure call nesting by using a general register as a pointer to the place on the stack where a procedure has its linkage data. This record of each procedure’s stack data, known as its stack frame or call frame, enables proper returns from procedures even when a procedure leaves data on the stack. In addition, user and operating system software can use the stack frame to trace back through nested calls to handle errors or debug programs.

PROGRAM COUNTER AND OTHER DEDICATED REGISTERS

Program Counter

The Program Counter (R15) contains the address of the next byte to be processed in the instruction stream. That is, just before the processor begins to execute an instruction, the Program Counter contains the address of the first byte of the next instruction. The Program Counter update is totally transparent to the programmer.

The Program Counter itself can be used to identify operands. The assembler translates many types of operand references into address-
ing modes using the Program Counter. Autoincrement mode using the Program Counter (also called immediate mode) is used to specify in-line constants other than those available with literal mode addressing. Autoincrement deferred mode using the Program Counter (also called absolute mode) is used to reference an absolute address. Displacement and displacement deferred modes using the Program Counter are used to specify an operand using an offset from the current location.

Appropriate addressing using the Program Counter enables the programmer to write position-independent code. Position-independent code can be executed anywhere in virtual address space after it has been linked, since program linkages can be identified as absolute locations in virtual address space and all other addresses can be identified relative to the current instruction.

The Stack Pointer, Argument Pointer and Frame Pointer
The Stack Pointer (R14) is a register designated for use with stack structures. To place items on a stack, the programmer can use autodecrement mode addressing through the Stack Pointer, and to remove them, can use autoincrement mode. The programmer can reference and modify the top element on a stack without removing it by using register deferred mode, and can reference other elements of the stack using displacement mode addressing.

The processor designates Register 14 as the Stack Pointer for use with the subroutine BRANCH or JUMP instructions, the procedure CALL instructions, and with some other instructions (see the VAX Architecture Handbook for greater detail). On subprogram entry, the processor automatically saves on the stack the address of the instruction following the routine call—using the Program Counter and the Stack Pointer to perform the operation. Before entering the subprogram, the Program Counter contains the address of the instruction following the BRANCH, JUMP, or CALL instruction. The Stack Pointer contains the address of the last item on the stack. The processor pushes the contents of the Program Counter onto the stack. On return from a subroutine, the processor automatically restores the Program Counter by popping the return address off the stack.

This process is somewhat more complex where procedures, rather than subroutines, are involved. For the procedure CALL instructions and some other instructions described in the VAX Architecture Handbook, the processor designates Register 12 as an Argument Pointer, and Register 13 as a Frame Pointer. The Argument Pointer is used to pass the address of the argument list to a called procedure,
and the Frame Pointer is used to keep track of nested CALL instructions.

An argument list is a formal data structure that contains the arguments required by the procedure being called. Arguments may be actual values, addresses of data structures, or addresses of other procedures. An argument list can be passed in either of two ways: by passing only its address, or by passing the entire list on the user stack. The first method is used to pass long argument lists, or lists to be preserved. The second method is generally used when building an argument list dynamically. Figure 7-2 illustrates the Argument Pointer and list.

![Figure 7-2 Argument Pointer and List](image)

The importance of the way these instructions work is that nested calls can be traced back to any previous level. The Frame Pointer contains the address of the items pushed on the stack during the procedure CALL. This set of items is known as a call frame (or stack frame). Figure 7-3 illustrates the call frame. Since the previous contents of the Frame Pointer are saved in each call frame, the nested frames form a linked data structure which can be unwound to any level when an error or exception condition occurs in any procedure (assuming that the Frame Pointer has not been used for other purposes).
Central Processor

CALL FRAME

STACK GROWTH

STACK BASE

CONDITION HANDLER

<table>
<thead>
<tr>
<th>CONTR.</th>
<th>REGISTER MASK</th>
<th>PSW</th>
</tr>
</thead>
<tbody>
<tr>
<td>OLD AP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OLD FP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RETURN PC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OLD R0...R11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

OLD SP

Figure 7-3 Call Frame

PROCESSOR STATUS WORD
The Processor Status Word (a portion of the Processor Status Long-word) is a special processor register that hardware uses to check the status of the CPU and to control synchronous error conditions. The Processor Status Word, illustrated in Figure 7-4, contains two sets of bit fields:
- Condition codes
- Exception enable flags

Figure 7-4 Processor Status Word
The condition codes are included to enable user programs to test the outcome of an arithmetic or logical operation. The condition codes indicate whether the previous arithmetic or logical operation produced a negative or zero result, or whether there was a carry, a borrow, or an overflow. For example, the SUBTRACT instruction sets the Negative bit if the result of the subtraction operation is a negative number; and it sets the Zero bit if the result produced zero. The programmer may then use a variety of branch on condition instructions, included in the VAX instruction set, to handle these occurrences. For further detail, please see the VAX Architecture Handbook.

**EXCEPTIONS**

There are some situations in which the programmer may not want to explicitly test the outcome of an operation. The processor recognizes many events for which testing is not necessary, and automatically changes normal program flow when they occur. These events, called *exceptions*, are the direct result of executing a specific instruction. Exceptions also include errors automatically detected by the processor, such as improperly formed instructions.

Exceptions are handled by the operating system software. In general, exceptions are described by vectors. Each vector is a counted list of longwords: the first longword contains the count of other longwords in the vector; the second identifies which type of exception occurred; the remaining longwords are the stack parameters, the Program Counter, and the Processor Status Longword.

There are three types of exceptions: aborts, faults, and traps. An abort is an exception condition that occurs during an instruction, potentially leaving memory and the registers indeterminate. Therefore, the instruction cannot necessarily be correctly restarted, completed, simulated, or undone.

A fault is an exception condition that occurs during an instruction and leaves the registers and memory in a consistent state. When the fault condition is eliminated, restarting the instruction will give the correct results. Note that faults do not always leave everything as it was prior to the instruction which caused the fault; they only restore enough to allow restarting. Thus, the state of a process that faults may not be the same as that of a process that was interrupted at the same point.

A trap is an exception condition that occurs at the end of the instruction that caused the exception. Therefore, the PC saved on the stack is the address of the next instruction that would normally have been executed.
**Central Processor**

**Arithmetic Exceptions and Trace Faults**
Two kinds of exceptions concern the user process: trace faults and arithmetic exceptions. The trace fault is used for debugging programs. Arithmetic exceptions include:

- Integer and decimal string overflow traps, in which the result was too large to be stored in the given format
- Floating point underflow faults, in which the result was too small to be expressed in the given format

**NOTE**
The preceding exception conditions occur only if the appropriate enable bits are set.

- Integer and decimal string divide by zero traps, in which the divisor supplied was zero
- Floating point overflow faults, in which the result was too large to be expressed in the given format
- Floating point divide by zero faults, in which the divisor supplied was zero

**Handling Exceptions**
When an exception occurs, the processor immediately saves the current state of execution and transfers control to the operating system. A vector tells the processor how to service the event by describing the starting address of an appropriate procedure to respond to the exception. These kinds of procedures are called *condition handlers*. The user can declare a condition handler for an entire image and for each individual procedure called; if there is no user-declared condition handler, control will not be returned to the program. In addition, because the processor keeps track of nested calls using the Frame Pointer register, it is possible to declare condition handlers for procedures that call other procedures in which exceptions might occur. The operating system automatically traces back through call frames to find a condition handler that will handle the exception. For additional detail on exceptions, see Chapter 3, Exceptions and Interrupts.
CHAPTER 8
MAIN MEMORY SUBSYSTEM

FEATURES

Expandable memory configuration
Error correcting memory controller

BENEFITS

Allows the addition of 256K byte array cards up to a maximum of 2M bytes.

Enhances data availability and reliability by correcting all single bit errors and detecting all double bit errors within the memory system.

Permits write operations to any combination of bytes within an aligned longword.

Boot ROMs

Allow bootstrapping from devices including the integral TU58 cartridge tape drive.

Optional battery backup

Maintains power to preserve data for a full 2M bytes memory for at least ten minutes.

INTRODUCTION

Main memory on the VAX-11/750 is implemented with array cards containing dynamic 16K bit array MOS memory devices interfaced to the system through an error correcting controller. The memory system is designed with a minimum capacity of 256K bytes and is expandable in increments of 256K bytes to a maximum capacity of 2M bytes. The controller corrects all single bit errors and detects all double-bit errors that may occur within the memory system. This feature enhances both data availability and reliability. All errors are reported back to the CPU where they may be recorded for use in preventive and corrective maintenance operations. Figure 8-1 illustrates the basic memory subsystem.
The memory controller module contains all of the logic needed to interface the array cards to the system. This includes the addressing logic, the refresh control, and the error correcting logic. The controller contains three longword registers that are accessible in I/O space. These registers allow a program to determine the size of the memory system, record the address of failing memory locations, isolate the error to an individual memory chip, and verify that the error correcting logic is working properly.

The memory system performance is optimized around longword read and write operations. The memory controller is designed to also allow write operations to any combination of bytes within an aligned longword; however, this operation reduces memory throughput.

**BOOT ROMS**

The memory controller module contains four read-only memories (ROMs) that are designed to allow bootstrapping from devices. One of the ROMs is always for the integral TU58 tape cartridge; the others are associated with various system devices used to boot the system. The ROMs are installed by Field Service to correspond to the positions of the BOOT DEVICE switch on the CPU cabinet front panel. Each ROM contains up to 256 bytes of code that are accessible in I/O space at the following physical addresses:

<table>
<thead>
<tr>
<th>ROM</th>
<th>Physical Address (hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>F20400 - F204FC</td>
</tr>
<tr>
<td>2</td>
<td>F20500 - F205FC</td>
</tr>
<tr>
<td>3</td>
<td>F20600 - F206FC</td>
</tr>
<tr>
<td>4</td>
<td>F20700 - F207FC</td>
</tr>
</tbody>
</table>
Main Memory Subsystem

The CPU is designed to execute out of these ROMs at bootstrap time as described in Chapter 6. In addition, each ROM contains a checksum of the ROM's contents to allow verification of proper operation and associated logic.

BATTERY BACKUP
The memory system has an optional battery backup unit, designed to maintain the contents of memory during power failures. This system will hold data for the full 2M bytes of memory for at least ten minutes. When power is restored, the memory controller checks to see if the contents of memory have been preserved by the battery unit. If they have not, the controller will write the contents of the entire memory system with zeros. On systems without the battery backup option, the controller always writes the memory system with zeros following a power-up sequence. The status of memory following a power-up sequence is noted in a control register so that the CPU can determine whether or not the contents of memory have been preserved throughout a power failure.

BASIC MEMORY OPERATIONS
The memory controller is an extended length, hex height, multilayer board with both LSI and gate array devices.

The memory system provides several key features for the VAX-11/750: timing and control for the three types of memory cycles, logic for refresh operations required by MOS memory, and memory system diagnostic features (under software control). In addition, the memory system performs ECC functions and provides data for error logging, and contains bootstrap functions and initialization logic.

The VAX-11/750 physical address space contained within the memory controller is divided into two areas: physical memory addresses and I/O addresses. Figure 8-2 illustrates the combined areas.

To access this address space, the system has three types of memory cycles.

Read
A READ cycle with no errors detected by ECC logic consists of four major steps. First, the device requesting the memory data presents ADDRESS and CONTROL information to the memory controller. Second, the memory controller initiates the memory cycle timing to the array cards. Third, the array cards present the memory data to the ECC logic. Finally, the memory controller puts out the data to the device requesting the READ and notifies the device that the data are available.
If, however, a correctable error is detected by the ECC logic, the corrected data are presented to the device doing the READ, and an interrupt is generated to notify the system that the error occurred. If an uncorrectable error is detected by the ECC logic, then the unmodified data are presented to the device along with an error message. If the READ reference came from an I/O adapter, then the adapter will record the error message. If the READ came from the CPU then the CPU will generate a machine check abort. When ECC detects errors during a READ cycle, the operation will be slightly slower than if no errors occurred.

**Longword Write**
For a LONGWORD WRITE, there are three major steps. First, the device requesting the LONGWORD WRITE presents ADDRESS and CONTROL information to the memory controller. Second, the memory controller initiates the memory cycle timing to the array cards. Third, while the array cards initiate their cycle, the memory controller generates seven check bits and presents them and the 32-bit longword to the array cards to complete the WRITE cycle.

**Byte and Word Write**
For these types of WRITE cycles, the sequence is the same as the first two steps of the READ cycle, with the added requirement of forming the new 32-bit data word and seven check bits into memory.
Main Memory Subsystem

If a correctable error was detected by the ECC, the data bit will be corrected. However, if the error is in a byte or word being written, the error will be ignored. If an uncorrectable error is detected by the ECC during the READ portion of the cycle, the original longword will be rewritten into memory without any changes. This will cause reporting of the error during a subsequent READ to this location.

CONTROL AND STATUS REGISTERS
There are three control and status registers with the following format in the VAX-11/750 main memory system. CSR 0 (address F20000₁₆) provides information to allow the recording of both correctable and uncorrectable errors. CSR 1 (F20004₁₆) aids in the verification and diagnosis of the error correcting hardware. CSR 2 (F20008₁₆) provides memory size and configuration information.

CSR 0 Bit Allocations
The following illustration shows the organization and status of CSR 0 following a power-up sequence:

![CSR 0 Bit Allocations Diagram](image)

**Figure 8-3   CSR 0**

**Bit: 31   Name:** Uncorrectable Error Flag  
**Function:** The occurrence of an uncorrectable error will set this bit to one. An uncorrectable error is a word which contains an even number of bits in error or an odd number of bits in error which map to an invalid syndrome. The address and syndrome of an uncorrectable error will overwrite the address and syndrome of a correctable error. This bit can only be cleared if the Uncorrectable Error Information Lost (CSR 0 <30>) is not set or if both bits are being cleared during the same write operation.

**Bit: 30   Name:** Uncorrectable Error Information Lost  
**Function:** When this bit equals one, an uncorrectable error has occurred when the Uncorrectable Error Flag (CSR 0 <31>) was already set. The second uncorrectable error page address will not overwrite the first uncorrectable error page address. This bit is cleared by writing a one to it.
Main Memory Subsystem

Bit: 29  Name: Correctable Error Flag
Function: This bit is set to one whenever a correctable (single-bit) error occurs during a READ operation. A single-bit error that occurs during the read portion of a BYTE WRITE cycle will have no effect on the flag. This bit is cleared by writing a one to it.

Bit: 28:24  Name: Not used

Bit: 23:9  Name: Page Address
Function: Identifies the page (512 bytes) on which an error occurred during a memory read cycle. The page address corresponds to the first error that occurs with the exception that the page address of an uncorrectable error will overwrite the page address of a correctable error. The page address of an uncorrectable error does not get overwritten by a more recent uncorrectable error. These bits are read-only.

Bit: 8:7  Name: Not used

Bit: 6:0  Name: Error Syndrome
Function: The error syndrome is stored in these bits. These bits are read-only.

CSR 1 Bit Allocations
The organization and status of CSR 1 following a power-up sequence is shown in the following illustration. All bits in CSR 1 are read/write.

![CSR 1 Bit Allocations](image)

Figure 8-4  CSR 1

Bit: 31:29  Name: Not used
Function:

Bit: 28  Name: Inhibit Reporting Correctable Errors
Function: When CSR 1 \(<28>\) equals zero, correctable (single-bit) errors will request the appropriate interrupt.

When CSR 1 \(<28>\) equals one, correctable (single bit) errors will still be corrected by the ECC logic, but the status lines will report No Error.

152
Main Memory Subsystem

In addition, the page address of the correctable error will not be logged in CSR 0 <23:9> and the Correctable Error Flag (CSR 0 <29>) will not get set.

CSR 1 <28> has no effect on logging and reporting uncorrectable errors. This bit is set to one on power up.

**Bit: 27**  **Name:** Page Mode  
**Function:** When CSR 1 <27> equals one, the ECC Disable Mode or the Diagnostic Check Mode operates on a 512-byte page whose address is contained in CSR 1 <23:9>.

The Diagnostic Check Mode must operate in the Page Mode.

The ECC Disable Mode can operate on a page or the entire memory (256K bytes to 2M bytes) depending on whether Page Mode is selected or not.

**Bit: 26**  **Name:** Diagnostic Check Mode  
**Function:** When CSR 1 <26> equals one, the contents of CSR 1 <6:0> are substituted for the check bits that come from memory during a read operation.

The Diagnostic Check Mode is constrained to operate on a single page whose address is stored in CSR 1 <23:9>. Therefore the Page Mode (CSR 1 <27>) must also be selected.

While operating in the Diagnostic Check Mode, read errors that occur in other pages of memory will not be logged into CSR 0.

Correct check bits are always put into memory during a write cycle.

The ECC Disable Mode and Diagnostic Check Mode cannot be selected at the same time.

**Bit: 25**  **Name:** ECC Disable Mode  
**Function:** When CSR 1 <25> equals one, no detection of uncorrectable errors will occur, no correction of single bit errors will take place, no error logging in CSR 0 will occur and No Error will be reported on the status lines.

In the ECC Disable Mode (CSR 1 <25> equals one), a read operation stores the seven check bits read from the array in CSR <6:0> and during a write operation, the generated check bits that are written into the array are also stored in CSR 1 <6:0>.

The ECC can be disabled for a 512-byte page or for the entire memory (256K bytes to 2M bytes) depending on whether the Page Mode (CSR 1 <27>) is selected or not.

Correct check bits are always put into memory during a write cycle. The ECC Disable Mode and Diagnostic Check Mode cannot be selected at the same time.
Main Memory Subsystem

Bit: 24  Name: Not used
Function:

Bit: 23:9  Name: Page Mode Address
Function: If the memory system is in the Page Mode (CSR 1 <27> equals one), CSR 1 <23:9> will contain the address of the 512-byte page to which the Diagnostic Check Mode or the ECC Disable Mode will be applicable.

Bit: 8:7  Name: Not used
Bit: 6:0  Name: Check Bits
Function: In Diagnostic Check Mode, the contents of CSR 1 <6:0> are substituted for the check bits that come from the memory during a read operation.

In ECC Disable Mode, a read operation takes the seven check bits read from the array and stores them in CSR 1 <6:0>.

CSR 2 Bit Allocations
The following illustration shows the power-up state of CSR 2 in a system with 768K bytes of memory (three array cards). This register is read-only.

![CSR 2 Bit Allocations](image)

Figure 8-5  CSR 2

Bit: 31:24  Name: Not used
Bit: 23:17  Name: Starting Address
Function: The starting address for the memory systems will be contained in these bits. The starting address is defined by jumper lines within the system and normally is set to zero.

Bit: 16  Name: Battery Backup Failure
Function: This bit is set on a power-up sequence if the battery back-up power has been exhausted.

The bit is cleared on a LONGWORD WRITE to any location in memory.

Bit: 15:0  Name: Memory Present Map
Function: These bits represent the amount of memory present in the system and the locations in the memory backplane where the array
boards are inserted. There are eight possible locations for the array boards. When a board is inserted, it causes two bits in this register to be set. Array cards are always inserted so that the bits are set beginning with bit \(<0\). Note that bits \(<5:0\)> in Figure 7-5 are set to indicate the presence of three array cards. No gaps in the use of array locations are allowed.

**DIFFERENCES BETWEEN THE VAX-11/750 AND VAX-11/780**

The information in this difference list will only be useful to system programmers who are not using VAX/VMS but are writing their own operating system software.

The register formats for error logging and memory configuration are different in the two machines. In addition, they differ in the following specifics:

**VAX-11/750**

2M bytes maximum physical memory, using up to eight 256K bytes array cards.

All physical memory uses a single memory controller and is contained in one cabinet.

Any LONGWORD WRITE will automatically clear a double-bit error occurring in memory.

**VAX-11/780**

4M bytes maximum physical memory per controller, using up to sixteen 256K bytes array cards.

For system maximum of 8M bytes physical memory, requires second memory controller and additional cabinet.

If a double-bit error (in a 64-bit word) occurs in memory, remedial operations have to clear it.
CHAPTER 9
UNIBUS SUBSYSTEM

FEATURES
Direct memory access (DMA) data transfers
Peer communication between UNIBUS devices
Total compatibility with the PDP-11 UNIBUS data path
Direct vectored hardware interrupts to servicing routines
UNIBUS device registers are addressed like memory locations
Supports a wide range of standard DIGITAL peripherals
Three buffered data paths

BENEFITS
Eliminates processor intervention for high data throughput
Allows direct data transfer between UNIBUS devices without CPU involvement
Gives the PDP-11 user an easy migration path to the VAX-11/750 processor
The CPU avoids time-consuming polling tasks when servicing interrupt requests
Simplifies I/O programming
Offers the user flexibility in peripheral selection to meet specific requirements
Increases data throughput by buffering data traffic to memory

INTRODUCTION
The UNIBUS subsystem connects most medium and low speed peripheral devices to the VAX-11/750 system. An asynchronous, bidirectional bus, the UNIBUS is used with all devices other than high-speed disk drives and magnetic tape transports. The UNIBUS lets the user select from a range of existing peripherals (those supported by VAX/VMS and diagnostics) and also provides easy connection for customer-designed special devices. Although the UNIBUS was originally designed for implementations of the PDP-11 architecture, its features and capabilities have been expanded by the VAX architecture. Thus, existing UNIBUS peripheral devices can be used with the new VAX family architecture without hardware modification. (UNIBUS addresses in this chapter are listed in both octal and hexadecimal. The PDP-11 uses octal while the VAX family uses hexadecimal.)
The integral UNIBUS Adapter (part of the CPU) connects the UNIBUS to the system and performs priority arbitration among UNIBUS devices. Furthermore, the UNIBUS Adapter lets the processor access UNIBUS peripheral device registers. Figure 9-1 illustrates the UNIBUS subsystem configuration.

![UNIBUS Subsystem Diagram]

**Figure 9-1  VAX-11/750 UNIBUS Configuration**

**VAX-11/750 UNIBUS SUMMARY**

The UNIBUS is a communication path that links I/O devices to the UNIBUS Adapter. Device-related addresses, data, and control information are passed along the 56 signal lines of the UNIBUS. The UNIBUS Adapter handles all communications between the UNIBUS and the system, and fields device-generated interrupts.

Conceptually, the UNIBUS is designed around memory elements with ascending addresses starting at UNIBUS address zero, while registers storing I/O data or individual peripheral device status information have addresses in the highest 8K bytes of addressing space (3E000₁₆ to 3FFE₁₆, or 76000₀₈ to 77777₆₈).
UNIBUS Subsystem

The UNIBUS consists of 56 signal lines, to which UNIBUS peripheral devices are connected. Figure 9-2 illustrates the signal line configuration. 51 of these lines are parallel and 5 are serial; 42 lines are bidirectional and 14 are unidirectional.

Communication between any two devices on the bus is in a master/slave relationship. During any bus operation, one device, the bus master, controls the bus when communicating with another device on the bus, called the slave. For example, the processor, as master, can fetch from a peripheral device, as slave; or the disk, as master, can transfer data to memory, as slave. Master/slave relationships are dynamic; the processor, for example, may pass bus control to a disk, then the disk may become master and communicate with slave memory. On the VAX-11/750, the main memory that the processor deals with for instructions and data is not actually on the UNIBUS, but is attached to the processor. The UNIBUS Adapter is the section of the processor that causes the VAX-11/750 memory to look like a slave to UNIBUS devices.

When two or more devices try to obtain control of the bus at once, priority circuits decide among them. Device priority levels are fixed at system installation. There are four priority levels among UNIBUS devices. A unit with a high priority level always takes precedence over
one with a lower priority level. In the case of units with equal priority levels, the one closest electrically to the processor on the bus takes precedence over those further away. The processor priority is raised and lowered according to the interrupt priority level (IPL).

Suppose the processor has control of the bus when three devices, all of higher priority than the processor, request bus control. If the requesting devices are of different priorities, the processor will grant use of the bus to the one with the highest priority. If they are all of the same priority, all three signals come to the processor along the same bus line, so that it sees only one request signal. Its reply, granting control of the bus, travels down the bus to the nearest requesting device, passing through any intervening non-requesting devices. The requesting device takes control of the bus, executes a single bus cycle and relinquishes the bus. Then the request grant sequence occurs again, this time going to the second device down the line, which has been waiting its turn. When all higher-priority requests have been granted, control of the bus returns to the processor.

The processor usually has the lowest priority because in general it can stop whatever it is doing without serious consequences. Peripheral devices may be involved with some kind of mechanical motion, or may be connected to a real-time process, either of which requires immediate attention to a request to avoid data loss.

Priority arbitration takes place asynchronously in parallel with the data transfer.

**Bus Communication**

Communication is interlocked, so that each control signal issued by the master must be acknowledged by a response from the slave to complete the transfer. This simplifies the device interface because timing is no longer critical.

**Using the Bus**

A device uses the bus if it needs to:

- Interrupt the processor. As a result, the processor stops what it is doing, enters a device service routine, and services the device.

- Transfer a word or byte of data to or from memory without involving the processor. Such functions—called NPRs or non-processor request transfers—are performed by direct memory access devices such as disks or tape units.

Whenever two devices communicate, it is called a bus cycle. Only one word or byte can be transferred per bus cycle.
UNIBUS Subsystem

Bus Control
There are two ways of requesting bus control: non-processor requests (NPRs) or bus requests (BRs).

An NPR is issued when a device wishes to perform a data transaction. An NPR does not use the CPU; therefore, DMA activity can occur while the CPU continues to execute instructions.

A BR is issued when a device needs to interrupt the CPU for service. A device can interrupt the CPU only if it has gained control via a BR.

Interrupts
Interrupt handling is automatic in the VAX-11/750. No device polling is required to determine which service routine to execute. When interrupting, the device supplies a vector which directs the CPU to a memory location which contains the starting address of an interrupt service routine. When servicing the interrupt, the processor automatically raises its priority to the level of the interrupting device. For a complete discussion of how the CPU responds to interrupts, see Chapter 3, Exceptions and Interrupts.

Priority Control
The UNIBUS priority system determines which device obtains the bus. Each UNIBUS device is assigned a specific location in the priority structure. Priority arbitration logic determines which device obtains the bus according to its position in the priority structure. The priority structure is two-dimensional, i.e., there are vertical priority levels and horizontal priorities at each level. Figure 9-3 illustrates UNIBUS priority control.

There are five UNIBUS vertical priority levels—NPR, and BR7, BR6, BR5, and BR4—which correspond to VAX interrupt priority levels (IPLs) 14 through 17. To accommodate several peripheral devices, it may be necessary to connect more than one device to a single level. When a number of devices are connected to the same level, the device electrically closest to the CPU has the highest priority.

Priority Assignments
When assigning priorities to a device, three factors must be considered: operating speed, ease of data recovery, and service requirements.

Data from some devices are available for only a short time period. Therefore, highest priorities are usually assigned to these devices for efficient data transfers. Lower priorities are assigned to devices whose data are available for longer periods of time, and to devices which have automatic data recovery features. For example, a disk or
magnetic tape device would be assigned a higher priority than a line printer or paper tape device. These priorities are assigned at installation time.

**CPU Priority Level**

In addition to device priority levels, the CPU has a programmable priority. The CPU can be set to any one of 32 priority levels. Levels 14 through 17 correspond to UNIBUS levels BR4 through BR7. For a complete discussion of CPU priority levels, see Chapter 3, Exceptions and Interrupts.

**Data Transactions**

There are four types of UNIBUS data transactions:

- **DATO**—a data word is transferred from the master to the slave.
- **DATOB**—a data byte is transferred from the master to the slave.
- **DATI**—a data word or byte is transferred from the slave to the master.
- **DATIP**—used to allow a read from slave/write from slave sequence to occur as a single bus cycle without any other UNIBUS activity intervening. This feature allows devices to synchronize their activities with the processor. DATIP must be followed by DATO or DATOB to the same location.
Users requiring more detailed information on the UNIBUS should refer to the PDP-11 UNIBUS Design Description.

**VAX-11/750 UNIBUS ADAPTER**

The VAX-11/750 UNIBUS Adapter serves three purposes. It allows the processor to access registers on the UNIBUS; it allows devices on the UNIBUS to perform DMA transfers to the VAX-11/750 memory; and it allows UNIBUS devices to interrupt the processor.

There are three characteristics of the VAX architecture and the VAX-11/750 memory system that require more than a straight-through connection from the UNIBUS to the system. First, addresses that are contiguous in the virtual address space may be noncontiguous in the physical address space on 512-byte boundaries. Since all UNIBUS non-processor request (NPR) devices broadcast sequential addresses, these device addresses must be broken up into noncontiguous 512-byte blocks by the UNIBUS Adapter.

Second, the VAX architecture imposes no restrictions on the alignment of data in memory. There are restrictions placed on UNIBUS word transfer NPR devices, however, in that data words are only transferred on even addresses. The VAX-11/750 UNIBUS Adapter provides an offset mechanism that allows the transfer to be effectively shifted by one byte to accommodate requests for I/O buffers on odd byte addresses. Third, because UNIBUS devices can only address a maximum of 256K bytes, the UNIBUS adapter allows these addresses to be expanded. This permits the devices to address the VAX-11/750's full 16M bytes of physical address space.

The adapter also provides three buffered data paths (similar to small caches) which allow UNIBUS devices to take advantage of the memory's four-byte accesses. This efficiency results in higher UNIBUS throughput and enhanced overall system operation.

**PROCESSOR ACCESS TO UNIBUS**

Any processor access with a physical address in the range of FC0000\(_{16}\) through FFFFFFF\(_{16}\) will map directly to the UNIBUS in the range 0 through 777777\(_{8}\). Any VAX-11/750 internal device (other than the processor) that references that address range will be ignored by the UNIBUS Adapter. The device will receive a non-existent memory confirmation.

**Processor Operations**

The VAX-11/750 UNIBUS Adapter responds only to aligned word or byte transactions. Any other type of access will yield UNPREDICTABLE results. The different types of operations mapped into UNIBUS
operations are: processor reads, processor reads with modify intent, processor read lock, write, and write unlock. Processor reads become UNIBUS Data In (UNIBUS read or DATI operations); processor reads with modify intent and processor read lock become Data In Pause (DATIP); and write and write unlock become UNIBUS Data Out operations (UNIBUS write, DATO or DATOB).

**UNIBUS Responses**
A processor read to the UNIBUS that causes a no-response timeout will result in a machine check abort with a non-existent memory (NXM) bus error indicated. A processor-write that causes such a timeout to occur will generate a write bus error interrupt request. If a UNIBUS device asserts PB in response to a processor access to it, it will cause a machine check abort, indicating an uncorrectable bus error.

**UNIBUS INITIATED DATA TRANSFERS**

**Overview**
As mentioned earlier in this chapter, the UNIBUS Adapter performs a number of functions to make the UNIBUS architecture compatible with the VAX architecture. A key element in this matching operation is the UNIBUS Map. This map translates UNIBUS device-generated addresses into physical memory addresses. It also identifies UNIBUS transfers so that other features, such as buffered data paths and odd byte addressing, can be used appropriately.

The UNIBUS has 18 address lines, creating an address space of 256K bytes. This is conceptually divided into two regions: the bottom 248 Kb used to address UNIBUS memory, and the top 8 Kb used to address UNIBUS peripheral device control and status registers (CSRs). A UNIBUS NPR device typically does a transfer to memory by doing a series of transactions placing addresses in the lower range on the bus. In the VAX-11/750, these are not the actual memory addresses, but serve as pointers to the UNIBUS Map, which in turn provides the actual physical memory address. If more than one device is doing transfers at the same time, each one should be set up to transfer into a different range within the 248 Kb address space.

The UNIBUS address space is divided into 512 pages of 512 bytes each. When an address arrives at the UNIBUS Adapter, the page number (the upper 9 bits of the 18-bit UNIBUS address) is taken as an index into the map, with the map then providing the actual Page Frame Number (PFN) of the physical memory address. This is concatenated with the address of the word or byte within the page from the UNIBUS address (the lower 9 bits of the 18-bit UNIBUS address) to create the final memory address. The map also provides a bit that specifies
UNIBUS Subsystem

whether this transaction should have its address incremented by one, allowing a device to address memory at an odd address. In addition, the map specifies which of the four data paths the transaction is to use—the direct data path or one of the three buffered data paths.

In using a direct data path, the UNIBUS Adapter insures that every UNIBUS transaction results directly in a memory transaction and that the UNIBUS is kept busy until the memory transaction is completed. However, the direct data path is not as efficient as the buffered data path; this is because memory can do four-byte transfers whereas the UNIBUS is limited to two-byte transfers.

Benefits of Buffered Data Paths
The buffered data paths allow UNIBUS devices to optimize use of memory. Because most NPR devices do sequential address transfers, every pair of two-byte transactions can be merged into a single long-word transfer. Therefore, the buffered data paths use only half the number of main memory transactions that the direct data path uses. The buffered data paths also increase performance for a large class of devices which access memory sequentially, but not exclusively so. Examples of such devices are graphics and intelligent communications devices. Furthermore, because requests are fulfilled from the buffer faster than a memory access can occur, the buffered data paths improve UNIBUS access time.

The buffered data path consists of an address register and a four-byte data buffer that together act as a cache. Only when a request cannot be met from this cache is a memory cycle required. If more than one NPR device is doing sequentially addressed transfers, the address stream that arrives at the UNIBUS Adapter will not be sequential. This is why three buffered data paths are provided. The UNIBUS Map directs different transfers to different data paths so that each data path sees only the activity of a single device and receives a sequential address stream.

Under VAX/VMS, these features are transparent to I/O drivers. VAX/VMS supplies standard system routines to allocate and release buffered data paths and blocks of map registers. Standard routines also set up the allocated map registers for a transfer and convert the buffer address to a UNIBUS space address for loading into the device buffer address register. For customer-developed operating systems and I/O drivers, the user would need to program these functions to manage the buffered data paths and map registers. A set of addresses is provided in I/O space for loading the map and controlling the buffered data paths.
UNIBUS Subsystem

UNIBUS Adapter Operating Detail
UNIBUS address bits <17:9> are used to enter a 512-byte by 19-bit wide memory location. The data coming out of that memory determine how the transaction will be handled. The map data field is divided into four sections: Page Frame Number (PFN), Data Path Number, Offset Bit, and Valid Bit. The format of the map data fields is shown in Figure 9-4.

![Figure 9-4 Map Data Fields](image)

On UNIBUS initiated transactions that cause a memory read or write cycle to occur, the map PFN is concatenated with UNIBUS address bits <8:0> to form a 24-bit physical address. Figure 9-5 shows this address translation process.

![Figure 9-5 UNIBUS to Physical Address Translation](image)
UNIBUS Subsystem

The Data Path Number is a two-bit field used to select among four data paths. Data paths one, two, and three are the buffered data paths, and data path zero is the direct data path.

If the Offset Bit is a one, it causes the transaction to behave as if the UNIBUS address supplied by the DMA device were incremented by one. This allows devices that only produce even byte addresses to access buffers on odd byte boundaries. A transaction that causes a word to cross page boundaries because the Offset Bit is set must have the Data Path Number, Offset Bit, and Valid Bit identical in both map entries. Any differences will yield UNPREDICTABLE results. If this is a DATI(P) or a DATO and the two bytes of UNIBUS data fall across a longword boundary, two memory cycles will occur.

If the Valid Bit is a one, the VAX-11/750 integral UNIBUS Adapter processes the transaction as described above. When it is zero, the integral UNIBUS Adapter ignores UNIBUS requests. The Valid Bit must be set to zero for map entries that correspond to sections of UNIBUS address space in which there are slaves expected to respond to transactions originating on the UNIBUS. Transactions from the CPU that cause a UNIBUS transaction to occur are always ignored by the integral UNIBUS Adapter and can never wrap back through the UNIBUS Adapter to memory.

Map Access — The map is accessible for both reading and writing. Each entry uses a longword address from $F30800_{16}$ to $F30FFC_{16}$. The logic that writes the map assumes that any write to the map addresses is a longword write. Any write other than a longword will make the contents of the map at the written location UNPREDICTABLE.

Direct Data Path — When the Data Path Number in the map is zero, the transaction uses the direct data path (DDP). This means that SSYN is not issued by the UNIBUS Adapter until the corresponding memory transaction is completed. Listed below are the types of memory functions that are initiated as a result of a UNIBUS cycle:

<table>
<thead>
<tr>
<th>UNIBUS Function</th>
<th>Memory Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATI</td>
<td>Read</td>
</tr>
<tr>
<td>DATIP</td>
<td>Read Lock</td>
</tr>
<tr>
<td>DATO(B)</td>
<td>Write or Write Unlock</td>
</tr>
</tbody>
</table>

If a DATO(B) follows a DATIP, then a write unlock will be issued; otherwise an ordinary write will occur.

Offset — If the offset in the map is set, then the transaction will be treated as if the UNIBUS address were incremented by one. If this is a DATI or a DATO and if the address crosses a longword boundary, two
cycles will occur. A device must not do a DATIP through the direct data path with the Offset Bit enabled.

**Buffered Data Paths** — When the Data Path Number in the map is one, two, or three, then a buffered data path has been selected. Each of the three BDPs consists of 4 bytes of data storage, 16 bits of address storage, 5 flag bits, and logic to make the BDP operate. These registers and flags are not accessible to software, but descriptions are included to permit precise definition of the BDP operation. The general intent of the BDP is that when UNIBUS transactions are occurring with sequential addresses (either ascending or descending), only one transfer is needed for every two UNIBUS transfers. When non-sequential transactions occur, then the correct data are provided, but no bandwidth saving occurs.

**Data Buffer** — Each buffered data path consists of a data storage buffer of four bytes. This storage buffer can be loaded from the UNIBUS or memory, and its contents can be output to either the UNIBUS or to memory. Data can be loaded into the buffer one or two bytes at a time from the UNIBUS, but is always loaded four bytes at a time from memory.

**Address Register** — Each buffered data path has a 16-bit address register that can be loaded from UNIBUS addresses <17:2> (or addresses <17:2>+1 if offset is on). Circuitry compares the stored address with the address on the UNIBUS to see if there is a match. The address held in the register is the UNIBUS longword corresponding to the data in the data buffer.

**Flags** — There are five flags that keep track of the data in the data buffer, named CD and BF3 through BF0. If CD = 1, then the buffer has four bytes of data from memory and BF3 through BF0 are always zero. If CD = 0, then BF3 through BF0 indicate which bytes in the data buffer have valid UNIBUS data. If they are all zero, the buffer is considered empty.

**Buffered Data Path Behavior**

- DATI(P). The buffered data paths treat DATI and DATIP identically. The behavior of the UNIBUS Adapter is primarily determined by the contents of the data and address buffers. If the buffer is empty or contains memory data but no address match, the UNIBUS Adapter performs five operations in the following sequence: 1) does a memory read; 2) puts the read data in the buffer and on the UNIBUS; 3) puts the UNIBUS address in the address register; 4) sets the flags to indicate memory data in the buffer; and 5) sends the data to the requesting UNIBUS device.
If there are UNIBUS data in the buffer, the UNIBUS Adapter first performs a memory write, using the stored data as data, the stored address as an address, and the byte flags as the byte mask. Operation then continues with the sequence described above, beginning with step 1.

If there are memory data in the buffer and an address match, the UNIBUS Adapter puts the data on the UNIBUS and issues SSYN.

- DATI(P) with byte offset. If the map specifies byte offset, the behavior depends on whether the transaction crosses a longword boundary. If it does not, then the response is identical to that described above for DATI(P). If the transaction does cross a longword boundary, then the UNIBUS Adapter acts as if two sequential transfers occurred—the first at the given UNIBUS address, the second at the address incremented by two. The two memory reads are pieced together to form the UNIBUS data word. The data buffer and address register hold the information from the second read at the end of the transaction.

- DATO(B). For a DATO(B) transaction, the contents of the buffer determine the behavior of the UNIBUS Adapter. If the buffer is empty or has memory data, the UNIBUS Adapter performs four functions: 1) puts the UNIBUS data in the data buffer; 2) puts the UNIBUS address in the address register; 3) sets the flags to indicate UNIBUS data in the buffer; and 4) indicates to the UNIBUS device that the transaction is completed.

If the buffer has UNIBUS data with an address match, behavior depends on whether the UNIBUS data, combined with the data in the buffer, form a full longword. If the data do not constitute a longword, the UNIBUS Adapter performs the same four operations as it does when the buffer is empty or has memory data (steps 1 through 4 above). If the data form a longword, the UNIBUS Adapter will do a memory write and clear the flags to show that the buffer is empty.

If the buffer has UNIBUS data with no address match, the UNIBUS Adapter will do a memory write and set the flags to show that the buffer is empty. Then the UNIBUS Adapter will perform the same operations described above for DATO(B) with the buffer empty.

- DATO with byte offset. If this transaction crosses a longword boundary, the UNIBUS Adapter treats it as two one-byte writes. If it does not cross a boundary, the UNIBUS Adapter handles it as a DATO(B) as described above.

- DATOB with byte offset. If this transaction does not cross a longword boundary, the UNIBUS Adapter treats it like a DATO(B) as described above, incrementing the address by one. If a longword
boundary is crossed, the UNIBUS Adapter treats it as if it were a DATOB in the next longword. In the latter case, address match is forced to no match.

Note: The behavior described for DATI(P) allows a device using a buffered data path to perform any sequence of transfers in either direction with any address sequence and still have the data end up in the desired location. A device that does repeated transfers within the same longword, however, will not cause any memory cycles. Thus, one may not use a buffered data path with a device that repeatedly reads one location in memory as a flag, waiting for the CPU to change it. In this case, the device would never see the change in memory since its reads will all be filled from the buffer.

**Control and Status Registers** — Proper transfers through BDPs require some intervention on the part of system software, aided by the implementation of control and status registers. The use of BDPs is not totally transparent. When a device has finished a series of DATO(B) transfers to memory, it is possible that some data will remain in the buffer if the transfer did not end on an even longword boundary. It is necessary for the software to initiate action to write this data to memory. When a device has finished a transaction that involves DATI(P)s, data are left in the buffer with a corresponding UNIBUS address in the address register. Should the contents of the map be changed at the location corresponding to the address in the address register, there will no longer be the correct association between address and data in the buffer. It is therefore necessary to clear the buffer following these transactions. A set of registers enable full control over these BDP transactions. The UNIBUS Adapter is assigned a block of 8K bytes of address space to map control and status registers. The bit assignments for these registers are described below and Figure 9-6 illustrates the format of a BDP control and status register.

![Buffered Data Path CSR](image)

**Figure 9-6  Buffered Data Path CSR**

**Bit: <31> Name:** Error  
**Function:** This bit on read is the OR of bits <30> and <29>. Writing to this bit has no effect.
UNIBUS Subsystem

Bit: <30> Name: NXM, Nonexistent Memory
Function: This bit is set when NXM status is received from memory. There is no response on the UNIBUS, and all future UNIBUS transactions through this BDP are ignored until this bit is cleared. This bit is cleared by writing a one to it.

Bit: <29> Name: UCE, Uncorrectable Error
Function: This bit is set when uncorrectable error status is received from memory. PB is asserted with the data that are passed back to the UNIBUS device on the first read from that location. It is not asserted on subsequent reads from this BDP. The bit is cleared by writing a one.

Bit: <28:1>
Name: Unused
Function: These bits yield UNPREDICATABLE values when read and are ignored when written.

Bit: <0> Name: Purge
Function: Writing a zero to this bit has no effect. Writing a one to it produces a result based on the contents of the buffer:

UNIBUS data The data are written to memory and the flags are set to mark the empty buffer empty
Memory data The flags are set to mark the buffer empty
Empty No action occurs

This bit will read as one following a write of one until the operation described above is completed. This allows software to determine if the write to memory is completed.

Interrupts — Interrupting devices on the UNIBUS are directly vectored through the System Control Block (SCB). The address of the vector is found at System Control Block Base (SCBB) + 200_{16} + device vector. The device vectors are the standard PDP-11 UNIBUS vectors. The VAX-11/750 integral UNIBUS Adapter interrupt mechanism will not operate properly with any UNIBUS device vector at or above 200_{16} (1000_{8}). The VAX-11/750 UNIBUS Adapter itself never generates an interrupt.

DIFFERENCES BETWEEN THE VAX-11/750 AND VAX-11/780
The information in this difference list will only be useful to system programmers who are not using VAX/VMS but are writing their own operating system software. Users of VAX/VMS can write I/O drivers that will run on both the VAX-11/780 and VAX-11/750. All of the UNIBUS control and status registers are different; in addition, the two machines differ in the following specifics:
### UNIBUS Subsystem

<table>
<thead>
<tr>
<th>VAX-11/750</th>
<th>VAX-11/780</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single UNIBUS Adapter is standard</td>
<td>One UNIBUS Adapter standard; up to three additional UNIBUS Adapters optional for systems with extensive UNIBUS I/O requirements</td>
</tr>
<tr>
<td>15-bit Page Frame Number and 24-bit physical byte address</td>
<td>21-bit Page Frame Number and 30-bit physical byte address</td>
</tr>
<tr>
<td>Direct vectored hardware interrupts to servicing routines; UNIBUS device interrupts vector to second page of System Control Block (SCB)</td>
<td>Not direct vectored; requires a read and test of the BR Receive Vector Register corresponding to the level of the interrupt</td>
</tr>
<tr>
<td>Three buffered data paths work on both sequential and non-sequential devices</td>
<td>15 buffered data paths work only on ascending sequential devices</td>
</tr>
<tr>
<td>No UNIBUS Adapter control and status bits (somewhat transparent to the operating system)</td>
<td>Implements UNIBUS Adapter control and status bits</td>
</tr>
</tbody>
</table>
CHAPTER 10
MASSBUS SUBSYSTEM

FEATURES
Direct memory access (DMA) data transfers
32-byte silo data buffer for each MASSBUS
Built-in diagnostic features
MASSBUS device registers are addressed like memory locations

BENEFITS
Eliminate processor intervention for high data throughput
Permits transfers at rates up to 2M bytes/second (5M bytes/second with three MASSBUSs).
Allow on-line diagnosis of the MASSBUS and MASSBUS drives
Simplifies I/O programming

INTRODUCTION
This chapter describes the hardware features of the MASSBUS subsystem. It outlines the kinds of CPU commands accepted by the MASSBUS Adapter and describes in detail the MASSBUS Adapter operation and the configuration of the various registers which control MASSBUS I/O operations. This INTRODUCTION and the section on MASSBUS ADAPTER OPERATION contain little technical detail and will be of interest to most readers. However, the section beginning with MBA REGISTERS will be of interest primarily to system designers not using VAX/VMS, who are writing their own operating systems or I/O device drivers. People writing their own device drivers under VAX/VMS should refer to the manual “How to Write a VMS Device Driver.”

The MASSBUS Adapter (MBA) is the hardware interface between the system and the high-speed MASSBUS storage devices on the VAX processors. The MASSBUS consists of two 16-bit wide communication paths linking the MASSBUS Adapter to the mass storage device drives. One path, the data bus, is used for data during data transfer operations. The other bus, the control bus, is used for accessing drive registers. The MBA will handle a MASSBUS drive with a maximum burst data transfer speed of 900 ns per 16 bits and a bandwidth of 2M bytes per second. Figure 10-1 illustrates the MASSBUS subsystem.
The MBA accepts and executes commands from the CPU, and reports the necessary status changes and fault conditions to the CPU. The MBA will accept the following commands from the CPU (see Appendix E of the VAX Architecture Handbook for command details).

- Read
- Read Lock (treated as Read)
- Read with Modify Intent (treated as Read)
- Write Unlock (treated as Write)

The MBA will send the following commands out to memory:

- Read
- Write
- Write Vector

The MBA will start a MASSBUS operation to transfer register data or a block of data to or from a MASSBUS device. 256 32-bit mapping registers store the page frame numbers of a block data transfer. A 32-byte deep data buffer is used to smooth the data transfer between memory and MASSBUS devices. The 32-bit memory data will be sent in two 16-bit words to the MASSBUS device.
Special diagnostic features are built into the hardware to allow online diagnosis of MBA and MASSBUS devices, and can be exercised through diagnostic features with no device on the MASSBUS.

The VAX-11/750 MBA handles these functions:
- Mapping addresses from virtual (program) to physical memory
- Data buffering for transfers between main memory and the MASSBUS
- Transferring interrupts from MASSBUS devices to the system

The VAX-11/750 can support up to three MASSBUS Adapters, with each adapter supporting up to eight device controllers. A MASSBUS Adapter supports any combination of mass storage devices, and is linked to devices by device controllers. There are controllers for different types of peripheral devices, with some controllers servicing several devices, while others support one device. For instance, each magnetic tape controller can support up to eight tape drives, but each disk controller supports a single disk drive. Regardless of the type of controller, only one controller can transfer data at any one given time. The data transfer rate depends on the particular mass storage device being accessed.

MASSBUS ADAPTER OPERATION
The MASSBUS Adapter includes an interface, internal registers, control paths and data paths. The MBA accepts and executes commands from the CPU and reports the necessary status changes and fault conditions to the CPU.

The MBA handles a MASSBUS drive with a maximum burst data transfer speed of 900 ns per 16 bits via the 16-bit wide MASSBUS data path. The MASSBUS Adapter controls data transfers between MASSBUS devices and physical memory. A MASSBUS Adapter transfers 16 bits at a time to a mass storage device or it receives 16 bits at a time from a MASSBUS drive. The MBA contains a 32-byte buffer used to store data enroute to either main memory or mass storage. Transfers (data only) to or from main memory, occur in 32-bit (4-byte) increments. Therefore, each memory transaction requires two MASSBUS transfers (16 bits each). The MASSBUS adapter will accept only aligned longword reads and writes to its external or internal registers. An attempt to address a nonexistent register in the MASSBUS adapter will prompt a no-response confirmation.

MBA REGISTERS
The MBA address space contains two sets of registers: internal and external. The MBA internal registers are the registers which are physi-
cally located in the MBA. The external registers are located in the
MASSBUS drives and are drive-dependent.

There are six internal registers and a 256 × 32-bit RAM. The primary
function of the internal registers is to monitor MBA and operating
status conditions. The internal registers also control phases of the
data transfers between the CMI and the MASSBUS device, such as:

- Maintaining a byte count to ensure that all of the data to be trans-
  ferred have been accounted for
- Converting virtual addresses to physical addresses for referencing
data in memory

The six internal registers are:
- MBA Control Register (CR)
- MBA Status Register (SR)
- MBA Virtual Address Register (VAR)
- MBA Byte Count Register (BCR)
- MBA Diagnostic Register (DR)
- MBA Command Address Register (CAR)

**NOTE**
The command address register is read-only and is valid only during data transfers.

The MBA contains 256 32-bit map registers which are used to map
program virtual addresses into physical addresses. The mapping reg-
isters allow transfers to or from contiguous or non-contiguous physi-
cal memory. Bits <30:15> of the map register are reserved and are not writable. Figure 10-2 illustrates mapping a virtual address to a
physical address.

**DATA PATH**
The data path controls the data transferred to and from the MASSBUS
device and memory. The 32-bit data word is divided into 16-bit (2-
byte) segments required for data on the MASSBUS. When performing
a read from a MASSBUS device, the data path assembles the two 16-
bit segments from the MASSBUS into the 32-bit format. A silo and
input/output data buffer provide the means for smoothing the data
transfer rate. The data path also contains a write check circuit which
can be used under program control to verify the accuracy of the data
transfer function.
MASSBUS Subsystem

![Diagram of MASSBUS Subsystem]

Figure 10-2 Virtual to Physical Address Translation

MBA ACCESS
Each device that interfaces to the VAX-11/750 has a block of addresses associated with it. Certain commonly addressed devices have preassigned address spaces, and other less frequently addressed devices use adapter codes. While the VAX-11/750 system handles up to three MASSBUS Adapters, it has address space reserved for four. Each of the four blocks of addresses contains 8K bytes, with the blocks beginning at addresses $F28000_{16}$, $F2A000_{16}$, $F2C000_{16}$, and $F2E000_{16}$. This space is accessible as part of the I/O address space. The command/address format used to access the MBA registers is illustrated in Figure 10-3.

179
**MASSBUS Subsystem**

Bit: <31:28>
Name: Bytes Mask

Bit: <27:25>
Name: Bus Function

Bit: <24>
Function: Reserved for future use, zero.

Bit: <23:15>
Name: MBA Address
Function: To select MBA Address Space, this field must be 1E5_{16}

Bit: <14:13>
Name: MBA Select
Function: Number of the MBA addressed.

Bit: <11:10>
Name: Register Select
Function:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>MBA internal register</td>
</tr>
<tr>
<td></td>
<td>Bit &lt;9:5&gt; must be zero</td>
</tr>
<tr>
<td></td>
<td>Bit &lt;4:2&gt; = register select offset</td>
</tr>
<tr>
<td>01</td>
<td>MBA external register</td>
</tr>
<tr>
<td></td>
<td>Bit &lt;9:7&gt; = device select</td>
</tr>
<tr>
<td></td>
<td>Bit &lt;6:2&gt; = register select</td>
</tr>
<tr>
<td>10</td>
<td>MBA MAP</td>
</tr>
<tr>
<td></td>
<td>Bit &lt;9:2&gt; = MAP address</td>
</tr>
<tr>
<td>11</td>
<td>Invalid (No response to an address with these bits on)</td>
</tr>
</tbody>
</table>

Bit: <1:0>
Function: Reserved for future use.

**MBA Control Register (Byte Offset = 4)**

180
Bit: <31:5>
Function: Reserved for future use, all zeros.

Bit: <4> Name: IBC Mode
Function: Ignore Byte Count Mode. When this bit is set, a data transfer will not be terminated by byte counter overflow. Instead, the data transfer terminates when a signal is received telling the MASSBUS Adapter that the last byte has been transferred. This feature allows the system to read from maagtapes with very long records (this requires a special device driver). In this mode, interrupts will be generated each time the byte counter overflows; the map registers are also writable. This bit is also cleared by writing a zero or by INIT.

Bit: <3> Name: MB Maintenance Mode (MMM)
Function: Setting this bit will put the MBA in the maintenance mode, which will allow the diagnostic programmer to exercise and examine the MASSBUS operations without using MASSBUS devices. When this bit is set, the MBA will send a signal to the MASSBUS so that all the devices on the MASSBUS will be detached from the MASSBUS. The MBA cannot be put in maintenance mode while a data transfer is in progress.

Bit: <2> Name: Interrupt Enable
Function: This bit is set by writing a one. This allows the MBA to interrupt the CPU when certain conditions occur. Cleared by writing a zero or by INIT.

Bit: <1> Name: Abort
Function: Abort data transfer. Write a one to set. Setting this bit will initiate the data transfer abort sequences which will stop sending commands, stop address and byte counter.

Setting this bit will also cause an interrupt to the CPU if the IE bit is one.

This bit will be cleared by writing a zero, or by INIT.

Bit: <0> Name: INIT
Function: Initialization. The bit is self-clearing. It will always read as zero. Setting this bit will:
MASSBUS Subsystem

1. Clear MBA Control register
2. Clear MBA Status register
3. Clear control and status bits of diagnostic registers
4. Cancel all pending commands
5. Abort data transfer
6. Assert MASSBUS INIT

MBA Status Register (Byte Offset = 8)

```
31 30 29 28  20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
   MBZ
```

Figure 10-5  MBA Status Register (Byte Offset = 8)

Bit: <31>  Name: DTBUSY  
Function: Data Transfer Busy. Bit is set when a data transfer command is received. It is cleared when data transfer is terminated normally or when a data transfer is aborted.

Bit: <30>  
Function:  Reserved for future use, zero.

Bit: <29>  Name: CRD  
Function: Corrected Read Data. This bit is set when the data received from memory were corrected. It is cleared by writing a one or by INIT. This bit is also cleared by subsequent receipt of a valid data transfer command.

Bit: <28:24>  
Function:  Reserved for future use, all zeros.

Bit: <23>  Name: CBHUNG  
Function: Control Bus Hung. This bit is set if the TRANSFER (TRA) signal has been stuck in the asserted state for 1.5 μs after the MBA receives a read or write command to an external register and any previous external register operation is complete. When this bit is set, no data transfer operations are initiated because the user cannot address the device drives or registers. This bit is cleared by writing a one or by INIT.

Bit: <22:20>  
Function:  Reserved for future use, all zeros.
MASSBUS Subsystem

Bit: <19> Name: PGE  
Function: Programming Error. The PGE bit is set when one or more of the following conditions exists:

1. Program tries to initiate a data transfer when MBA is currently performing one.
2. Program tries to load MAP, VAR, or Byte Counter when MBA is currently performing a data transfer operation.
3. Program tries to set MB Maintenance Mode during a data transfer operation.

The bit is cleared by writing a one to it, or by INIT. This bit is also cleared by subsequent receipt of a valid data transfer command. Setting this bit will cause an interrupt to the CPU if Interrupt Enable is set.

Bit: <18> Name: NED  
Function: Nonexistent Drive. This bit is set when a drive fails to assert the TRANSFER signal within 1.5 \( \mu s \) after asserting a DEMAND signal. The bit is cleared by writing a one or by INIT. Setting this bit will send zero read data back to memory and interrupt the CPU if Interrupt Enable is set.

Bit: <17> Name: MCPE  
Function: MASSBUS Control Parity Error. This bit is set when a MASSBUS control parity error occurs. It is cleared by writing a one or by INIT. Setting this bit will cause an interrupt to the CPU if Interrupt Enable is set.

Bit: <16> Name: ATTN  
Function: Attention from MASSBUS. Asserted when the attention line on the MASSBUS is asserted. Asserting this bit will cause an interrupt to the CPU if Interrupt Enable is set.

Bit: <15>  
Function: Reserved for future use, zero.

Bit: <14> Name: Silo Parity Error  
Function: This bit is set when a silo parity error occurs during a data transfer operation. Cleared by writing a one to this bit or by INIT. Setting this bit will abort the data transfer operation. Subsequent receipt of a valid data transfer command will also clear this bit.

Bit: <13> Name: DTCMP  
Function: Data Transfer Completed. This bit is set when the data transfer is terminated either due to an error or normal completion. It is cleared by writing a one or by INIT. This bit is also cleared by subsequent receipt of a valid data transfer command. Setting this bit will cause an interrupt to the CPU if Interrupt Enable is set.
Bit: <12> Name: DTABT
Function: Data Transfer Aborted. This bit is set with the trailing edge of the END OF BLOCK signal when the data transfer has been aborted. It is cleared by writing a one or by INIT. This bit is also cleared by subsequent receipt of a valid data transfer command. Setting this bit will cause an interrupt to the CPU if Interrupt Enable is set.

Bit: <11> Name: DLT
Function: Data Late. This bit is set when:

1. The data buffer is empty and WCLK (the WRITE CLOCK signal indicates when data written to a drive are to be strobed) is sent to the MASSBUS during a Write Data Transfer or Write Check Data Transfer.

2. The data buffer is full when SCLK (SYNCHRONIZE CLOCK is asserted during a read operation to indicate when data read from the drive are to be strobed) is received from the MASSBUS during a read data transfer.

This bit is cleared by writing a one or by INIT. It is also cleared by subsequent receipt of a valid data transfer command. Setting this bit will abort the data transfer operation.

Bit: <10> Name: WCK UP ERR
Function: Write Check Upper Error. This bit is set when a compare error is detected in the upper byte while the MBA is performing a write check operation. It is cleared by writing a one or by INIT. This bit is also cleared by subsequent receipt of a valid data transfer command. Setting this bit will abort the data transfer operation.

Bit: <9> Name: WCK LWR ERR
Function: Write Check Lower Error. This bit is set when a compare error is detected in the lower byte while the MBA is performing a write check operation. It is cleared by writing a one or by INIT. This bit is also cleared by subsequent receipt of a valid data transfer command. Setting this bit will abort the data transfer operation.

Bit: <8> Name: MXE
Function: Miss Transfer Error. This bit is set when an OCC (an OCCLUDED signal) is not received within 500 μs after data transfer busy is set. It is cleared by writing a one or by INIT. This bit is also cleared by subsequent receipt of a valid data transfer command. Setting this bit will cause an interrupt to the CPU if Interrupt Enable is set.

Bit: <7> Name: MBEXC
Function: MASSBUS Exception. This bit is set when EXC (the EXCEPTION signal indicates an error condition during a data transfer) is received from MASSBUS. It is cleared by writing a one or by INIT. This
MASSBUS Subsystem

bit is also cleared by subsequent receipt of a valid data transfer command. Setting this bit will abort the data transfer operation.

Bit: <6>  Name: MDPE  
Function: MASSBUS Data Parity Error. This bit is set when a MASSBUS data parity error is detected during a read data transfer operation. It is cleared by writing a one or by INIT. This bit is also cleared by subsequent receipt of a valid data transfer command. Setting this bit will abort the data transfer operation.

Bit: <5>  Name: MAPPE  
Function: Page Frame Map Parity Error. This bit is set when a parity error is detected on the page frame number read from the PF map. It is cleared by writing a one or by INIT. This bit is also cleared by subsequent receipt of a valid data transfer command. Setting this bit will abort the data transfer operation.

Bit: <4>  Name: INVMAP  
Function: Invalid Map. This bit is set when the valid bit of the next page frame number is zero when the byte count is not zero. It is cleared by writing a one or by INIT. This bit is also cleared by subsequent receipt of a valid data transfer command. Setting this bit will abort the data transfer operation.

Bit: <3>  Name: ERR STAT  
Function: Error Status. This bit is set when the MBA receives error status for the read command or write command. It is cleared by writing a one or by INIT. This bit is also cleared by subsequent receipt of a valid data transfer command. Setting this bit will cause the data transfer operation to be aborted.

Bit: <2>  
Function: Reserved for future use, zero.

Bit: <1>  Name: NRSTAT  
Function: No Response Status. This bit is set when the MBA receives no response status for a read or write command to memory. This bit is cleared by writing a one or by INIT. This bit is also cleared by subsequent receipt of a valid data transfer command. Setting this bit will abort the data transfer operation.

Bit: <0>  
Function: Reserved for future use, zero.

MBA Virtual Address Register (Byte Offset = 12)

185
MASSBUS Subsystem

Figure 10-6  MBA Virtual Address Register (Byte Offset = 12)

The program must load an initial virtual address (pointing to the first byte to be transferred) into this register before a data transfer is initiated. Bits <16:9> select one of the 256 map registers. The contents of the selected map and the values in bits <8:0> are used to assemble a physical address to be sent to memory. Bits <8:0> indicate the byte offset of the current data byte into the page. Note that the MBA virtual address register is incremented by four after every memory read or write and will not point to the next byte to be transferred if the transfer does not end on a longword boundary. (It will point four bytes ahead.) Also, upon a write check error, the virtual address register will not point to the failing data in memory due to the preloading of the silo data buffer. The virtual address of the bad data may be found by determining the number of bytes actually transferred compared to the MASSBUS (the difference between bits <31:16> of the MBA Byte Counter and their initial value) and adding that difference to the initial virtual address.

MBA Byte Counter (Byte Offset = 16)

Figure 10-7  MBA Byte Counter (Byte Offset = 16)

The program writes the 2's complement of the number of bytes for the data transfer to bits <15:0> of this register. MBA hardware will load these 16 bits into bits <31:16> and bits <15:0>. Bits <31:16> serve as the byte counter for the number of bytes transferred to or from the drive. Bits <15:0> serve as the byte counter for the number of bytes transferred to or from memory. The starting byte count with 16 bits of zero is the maximum number of bytes of a data transfer.

MBA Diagnostic Register (Byte Offset = 20)

Figure 10-8  Diagnostic Register (Byte Offset = 20)
MASSBUS Subsystem

The diagnostic register may be written only while in maintenance mode.

Bit: <31> Name: IMDPG
Function: Invert MASSBUS Data Parity Generator.

Bit: <30> Name: IMCPG
Function: Invert MASSBUS Control Parity Generator.

Bit: <29> Name: IMAPP
Function: Invert Map Parity Checking.

Bit: <28> Name: BLKSCOM
Function: Block Sending Command. During a data transfer, setting this bit will eventually cause the DLT (Data Late—bit <11> of the MBA Status Register) bit to be set and interrupt the CPU.

Bit: <27> Name: SIMSCLK
Function: Simulate SCLK. When the MB Maintenance Mode bit is set, writing a one to this bit will simulate the assertion of SCLK (SYNCHRONIZE CLOCK); and writing a zero to this bit will simulate the deassertion of SCLK.

Bit: <26> Name: SIMEBL
Function: Simulate EBL. When the MB Maintenance Mode bit is set, writing a one and writing a zero to this bit will simulate the assertion and deassertion of EBL (END OF BLOCK).

Bit: <25> Name: SIMOCC
Function: Simulate OCC. When the MB Maintenance Mode bit is set, writing a one and writing a zero to this bit will simulate the assertion and deassertion of OCC (OCCUPIED).

Bit: <24> Name: SIMATTN
Function: Simulate ATTN. When the MB Maintenance Mode bit is set, writing a one or a zero to this bit will simulate the assertion and deassertion of ATTN. (ATTENTION is asserted by drives to signal the MBA of changes in a drive’s status or abnormal conditions.)

Bit: <23> Name: MDIB SEL
Function: Maintenance MASSBUS Data Buffer Select. This bit selects what is to be sent out from bits <15:8> when the diagnostic register is read. When this bit is set to one, the upper byte of the MDIB (read-only) is selected. When this bit is set to zero, Maintenance Drive and Register Select (read-only) are selected.

Bit: <22> Name: ISPG
Function: Invert Silo Parity Generator.

Bit: <21> Name: SIMEXC
Function: Simulate EXC. When the MB Maintenance Mode bit is set,
writing a one or a zero to this bit will simulate the assertion and deassertion of the EXCEPTION signal.

**Bit: <20> Name: MFAIL**
**Function:** MASSBUS Fail (read-only). Fail is asserted when MMM is set.

**Bit: <19> Name: MRUN**
**Function:** Maintenance MASSBUS Run (read-only).

**Bit: <18> Name: MWCLK**
**Function:** Maintenance MASSBUS WCLK (read-only).

**Bit: <17> Name: MEXC**
**Function:** Maintenance MASSBUS EXC (read-only).

**Bit: <16> Name: MCTOD**
**Function:** Maintenance MASSBUS CTOD (read-only).

**Bit: <15:13>**
**Name:** MDS or MDIB <15:13>
**Function:** Maintenance MASSBUS Device Select (read-only) or MDIB <15:13> (read-only) as controlled by bit <23>.

**Bit: <12:8>**
**Name:** MRS or MDIB
**Function:** Maintenance MASSBUS Register Select (read-only) or MDIB <12:8> (read-only) as controlled by bit <23>.

**Bit: <7:0> Name: MDIB**
**Function:** Maintenance MDIB.

**MBA Command Address Register (Byte Offset = 28)**
This register is read-only and valid only when DT Busy (bit <31> of the MBA Status Register) is set. The bit assignments are as follows:

![Command Address Register Diagram](image)

**Figure 10-9  Command Address Register (Byte Offset = 28)**

**Bit: <31:28>**
**Name:** Byte Mask
**Function:** On a write or write unlock, these bits correspond to bytes zero through three and tell memory which bytes to write.
Bit: <27:25>
Name: Operation
Function: These bits define the operation to be performed. Possible operations are:

000  Read to Memory
100  Write
110  Interrupt

Bit: <24>
Function: Undefined

Bit: <23:0>
Name: Address
Function: These bits specify the physical address of the activity to memory.

MBA External Registers (Byte Offset = 400 to 7FC)
External registers are MASSBUS device-dependent. Each device has a maximum of 32 registers.

MBA Map Registers (Byte Offset = 800 to BFC)

```
31 30  16 15  0
V  RESERVED  PFN
```

Figure 10-10  MBA Map Registers (Byte Offset = 800 to BFC)

Bit: <31>  Name: Valid Bit

Bit: <30:15>
Function: Reserved for future use, all zeros.

Bit: <14:0>
Name: PFN
Function: Physical Page Frame Number. The MBA contains 256 map registers, each of which may be selected by address bits <9:2> when bits <11:10> are one and zero, respectively. Map registers can only be written when there is no data transfer operation in progress or the IBC bit (Ignore Byte Count—bit <4> of the MBA Control Register) is set. A write to a map register during a data transfer with IBC clear will be ignored and cause PGE to set.

DATA TRANSFER PROGRAM FLOW
1. Initialize MASSBUS Adapter
2. Mount pack into drive
MASSBUS Subsystem

3. Start drive spinning
4. Wait for ready light
5. Issue Pack ACK to drive
6. Load desired cylinder, sector, track, and registers in drive
7. Load starting virtual address into MBA's virtual address register
8. Load 2's complement of number of bytes to be transferred into byte count register in MBA
9. Load starting map (pointed to by bits $<16:9>$ of VAR) with physical page address
10. Load successive maps with physical addresses to rest of pages
11. Issue read/write command to drive

DIFFERENCES BETWEEN THE VAX-11/750 AND VAX-11/780

The information in this difference list will only be useful to system programmers who are not using VAX/VMS but are writing their own operating system software.

**VAX-11/750**

- Ignore Byte Count (IBC) Mode in the MBA Control Register enables the system to read from mag tapes with very long records; however, this capability requires a special I/O device driver.
- MBA generates and checks parity on the silo. When a silo parity error occurs, a bit in the MBA Status Register is set and the data transfer operation is aborted.
- Needs no Configuration Status Register because there are no SBI faults.
- Implements Control Bus Hung bit in the MBA Status Register. When set, this bit prevents the user from addressing the drive or registers, so that no control bus activity can be successfully initiated. However, a data transfer operation already in progress will be completed.

**VAX-11/780**

- No IBC mode implemented.
- No silo parity checking.
- Implements a Configuration Status Register and SBI faults.
- Control Bus Hung (CBHUNG) not implemented.
INTRODUCTION
The processor register space provides access to many types of CPU control and status registers such as the memory management base registers, the Processor Status Longword, and the multiple stack pointers. The benefit of privileged registers is that these key registers are explicitly accessible only by the Move to Processor Register (MTPR) and Move from Processor Register (MFPR) instructions which are controlled by the kernel executive. In a VAX/VMS environment, the operating system conveniently manages these registers for the user; therefore, the detailed privileged register information contained in this chapter will be useful only to system designers who will not be using VAX/VMS.

A complete list of VAX-11/750 internal processor registers may be found in Appendix F.

SYSTEM IDENTIFICATION REGISTER (SID)
The system identification register is a read-only constant register that specifies the processor type. The entire SID register is included in the error log and the type field may be used by software to distinguish processor types. Figure 11-1 illustrates the system identification register.

![System Identification Register IPR #3E16](image)

Figure 11-1  System Identification Register IPR #3E16

Type  A unique number assigned by engineering to identify a specific processor:

0  Reserved to DIGITAL (error)
1  VAX-11/780
2  VAX-11/750
3 through 127  Reserved to DIGITAL
128 through 255  Reserved to CSS and customers

For the VAX-11/750, the type-specific format is shown in Figure 11-2.
CONSOLE TERMINAL REGISTERS
The console terminal is accessed through four internal registers. Two are associated with receiving from the terminal and two with writing to the terminal. In each direction there is a control/status register and a data buffer register. Figure 11-3 illustrates the console receive control/status register, and the bit assignments are described.

Bit: 31:8  Name: MBZ
Function: Must be zero

Bit: 7  Name: Done
Function: This bit is read-only and is set by the console whenever a datum is received. Done is initialized to 0 at bootstrap time and is cleared whenever MFPR #RXDB.dst is executed.

Bit: 6  Name: IE
Function: Interrupt Enable. If this bit is set by software, an interrupt is generated at IPL 20 when Done becomes set. Similarly, if Done is already set and the software sets IE, an interrupt is generated. This bit is initialized to 0 at bootstrap time, and can be read or written by software.

Bit: 5:0  Name: MBZ
Function: Must be zero

Figure 11-4 illustrates the read-only console receive data buffer register. The bit assignments follow.
Privileged Registers

Figure 11-4  Console Receive Data Buffer Register (RXDB) IPR \#21\_{16}

Bit: 31:16  Name: MBZ  
Function: Must be zero

Bit: 15:  Name: ERR  
Function: Error bit. If the received data contained an error such as overrun or loss of connection, then ERR is set.

Bit: 14:12  Name: MBZ  
Function: Must be zero

Bit: 11:8  Name: ID  
Function: If ID are zero, then the data is from the console terminal. If ID is nonzero, then the entire register is implementation-dependent.

Bit: 7:0  Name: Data  
Function: This field contains the actual data received by the console.

Figure 11-5 illustrates the console transmit control/status register; the bit descriptions are also provided.

Figure 11-5  Console Transmit Control/Status Register (TXCS)  
IPR \#22\_{16}

Bit: 31:8  Name: MBZ  
Function: Must be zero

Bit: 7  Name: RDY  
Function: Ready. This bit is read-only and is set at bootstrap time. It is also set whenever the console transmitter is not busy. This bit is cleared when MTPR src,#TXDB is executed.
Privileged Registers

Bit: 6  Name: IE
Function: Interrupt Enable. If this bit is set by software, an interrupt is generated at IPL 20 when RDY becomes set. If RDY is already set and software sets IE, an interrupt is also generated. This bit is cleared when MTPR src,#TXDB is executed.

Bit: 5:0  Name: MBZ
Function: Must be zero

Figure 11-6 illustrates the read-only console transmit data buffer register. The bit assignments are described.

![Diagram of Console Transmit Data Buffer Register](image)

Figure 11-6  Console Transmit Data Buffer Register (TXDB) IPR #23_{16}

Bit: 31:12  Name: MBZ
Function: Must be zero

Bit: 11:8  Name: ID
Function: When an MTPR src,#TXDB is executed and these bits are written as 0, data will be sent to the console terminal. If ID is F_{16}, then the Data field can have five values. If ID is other than 0 or F_{16}, it indicates a reserved operand.

Bit: 7:0  Name: Data
Function: This field contains the actual data transmitted by the console. If ID is F, 0, 1 or 3 is a no-op; a 2 will cause a boot, and a 4 will clear a cold start flag. With ID = F, any other value will cause a reserved operand fault.

TU58 REGISTERS
The console TU58 tape cartridge subsystem is accessed through four internal registers. Two are associated with receiving from the TU58 and two with writing to it. In each direction there is a control/status register and a data buffer register. Figure 11-7 illustrates the console storage receive status register. Description of the bit assignments follow the figure.
Privileged Registers

Figure 11-7 Console Storage Receive Status (CSRS) IPR #1C₁₆

<table>
<thead>
<tr>
<th>Bit: 31:8</th>
<th>Name: MBZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function:</td>
<td>Must be zero</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit: 7</th>
<th>Name: Done</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function:</td>
<td>This bit is read-only and is set by the TU58 whenever a datum is received by the TU58 from the console storage receive data register. Done is initialized to 0 at bootstrap time and is cleared whenever MFPR #CSRD, dst is executed.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit: 6</th>
<th>Name: IE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function:</td>
<td>Interrupt Enable. If this bit is set by software, an interrupt is generated at IPL 23. Similarly, if Done is already set and the software sets IE, an interrupt is generated. This bit is initialized to 0 at bootstrap time, and can be read or written to by software.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit: 5:0</th>
<th>Name: MBZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function:</td>
<td>Must be zero</td>
</tr>
</tbody>
</table>

Figure 11-8 illustrates the console storage receive data register. Bit assignments follow the illustration.

Figure 11-8 Console Storage Receive Data (CSRSD) IPR #1D₁₆

<table>
<thead>
<tr>
<th>Bit: 31:8</th>
<th>Name: MBZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function:</td>
<td>Must be zero</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit: 7:0</th>
<th>Name: Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function:</td>
<td>This field contains the actual data received from the TU58 subsystem.</td>
</tr>
</tbody>
</table>

Figure 11-9 illustrates the console storage transmit status register. Bit descriptions follow the figure.
Privileged Registers

![Diagram of Privileged Registers]

Figure 11-9  Console Storage Transmit Status (CSTS) IPR #1E₁₆

Bit: 31:8  Name: MBZ  
Function: Must be zero

Bit: 7  Name: RDY  
Function: Ready. This bit is read-only and is set at bootstrap time. It is also set whenever the TU58 transmitter is not busy. This bit is cleared when MTPR src, #CSTD is executed.

Bit: 6  Name: IE  
Function: Interrupt Enable. If this bit is set by software, an interrupt is generated at IPL 23. If RDY is already set and software sets IE, an interrupt is also generated. This bit is cleared when MTPR src, #CSTD is executed.

Bit: 5:1  Name: MBZ  
Function: Must be zero

Bit: 0  Name: LB  
Function: Line Break. When this bit is written to a 1, a line break is issued to the TU58. This bit is write-only.

Figure 11-10 illustrates the console storage transmit data register. Bit descriptions follow the illustration.

![Diagram of Console Storage Transmit Data (CSTD) IPR #1F₁₆]

Figure 11-10  Console Storage Transmit Data (CSTD) IPR #1F₁₆

Bit: 31:8  Name: MBZ  
Function: Must be zero

Bit: 7:0  Name: Data  
Function: This field contains the actual data transmitted by the TU58 subsystem.
Privileged Registers

CLOCK REGISTERS
The clocks consist of a time-of-year clock and an interval clock. The time-of-year clock is used by the operating system to reinitialize the time and date after power has been off, without requiring an operator to type in the information. The time-of-year clock is also used for time stamping of user and operating system programs. The interval clock is used for accounting, for time-dependent events, and to maintain the software date and time.

Time-of-Year Clock
The time-of-year clock consists of one longword register. The register forms an unsigned 32-bit binary counter that is driven by a precision clock source. (Clock accuracy is typically .0025% but will vary slightly with ambient temperature and remaining life of the battery backup system.) The counter has a battery backup power supply sufficient for at least 100 hours of operation, and the clock does not gain or lose any ticks during transition to or from stand-by power. The battery is recharged automatically. The least significant bit of the counter represents a resolution of 10 ms. Thus, the counter cycles to zero after approximately 497 days.

If the battery has failed, so that time is not accurate, the register is cleared on power-up. It is held at zero until software writes a nonzero value to it. Thus, if software initializes this clock to a value corresponding to a large unit of time (e.g., a month), it can check for loss of time after a power restore by checking the clock value. The time-of-year clock register is illustrated in Figure 11-11.

![Figure 11-11 Time-of-Year Clock Register (TODR) IPR #1B₁₆](image)

Interval Clock
The interval clock provides an interrupt at IPL 24 at programmed intervals. The counter is incremented at 1 μs intervals, with a typical accuracy of .01% or 8.64 seconds per day. (Accuracy will vary slightly with ambient temperature.) The clock interface consists of three registers in the privileged register space: the read-only Interval Count Register, the write-only Next Interval Count Register, and the Interval Clock Control/Status Register.
Interval Count Register
The Interval Count Register is a read-only register incremented once every microsecond. It is automatically loaded from NICR (Next Interval Count Register) upon a carry out from bit 31 (overflow) which also causes an interrupt request at IPL 24 if the interrupt is enabled. Figure 11-12 illustrates the Interval Count Register.

Figure 11-12  Interval Count Register (ICR) IPR #1A_{16}

Next Interval Count Register
The reload register is a write-only register that holds the value to be loaded into ICR when it overflows. The value is retained when ICR is loaded. NICR is capable of being loaded regardless of the current values of ICR and ICCS (Interval Clock Control/Status Register). Figure 11-13 illustrates the Next Interval Count Register.

Figure 11-13  Next Interval Count Register (NICR) IPR #19_{16}

Interval Clock Control/Status Register (ICCS)
The ICCS register contains control and status information for the interval clock. Figure 11-14 illustrates the Interval Clock Control/Status Register, and the bit assignments are described in detail.

Figure 11-14  Interval Clock Control/Status Register (ICCS) IPR #18_{16}
Privileged Registers

Bit: 31  Name: ERR
Function: Whenever ICR overflows, if INT is already set, then ERR is set. Thus, ERR indicates one or more missed clock ticks. Attempts to set this bit via MTPR clears ERR.

Bit: 30:8  Name: MBZ
Function: Must be zero

Bit: 7  Name: INT
Function: This bit is set by hardware every time ICR overflows. If IE is set then an interrupt is also generated. An attempt to set this bit via MTPR clears INT, thereby re-enabling the clock tick interrupt (if IE is set).

Bit: 6  Name: IE
Function: When this bit is set, an interrupt request at IPL 24 is generated every time ICR overflows (INT is set). When clear, no interrupt is requested. Similarly, if INT is already set and the software sets IE, an interrupt is generated (i.e., an interrupt is generated whenever the function (IE .AND. INT) changes from 0 to 1).

Bit: 5  Name: SGL
Function: This bit is write-only. If Run is clear, each time this bit is set, ICR is incremented by one.

Bit: 4  Name: XFR
Function: This bit is write-only. Each time this bit is set, NICR is transferred to ICR.

Bit: 3:1  Name: MBZ
Function: Must be zero

Bit: 0  Name: Run
Function: When this bit is set, ICR increments each microsecond. When clear, ICR does not increment automatically. At bootstrap time, Run is cleared.

Thus, to set up the interval clock, load the negative of the desired interval into NICR. Then an MTPR #X51,#ICCS will enable interrupts, reload ICR with the NICR interval and set Run. Every "interval count" microseconds will cause INT to be set and an interrupt to be requested. The interrupt routing should execute an MTPR #XC1,#ICCS to clear the interrupt. If INT has not been cleared (i.e., if the interrupt has not been handled) by the time of the next ICR overflow, the ERR bit will be set.

At bootstrap time, bits <6> and <0> of ICCS are cleared. The rest of ICCS and the contents of NICR and ICR are UNPREDICTABLE.
MACHINE CHECK ERROR SUMMARY REGISTER (MCESR)
The Machine Check Error Summary Register (MCESR) logs information about causes of a machine check—for instance, a bus error. The entire register is read/write and all bits are set to 0 at bootstrap time. As shown in Figure 11-15, only bits <3:0> are implemented; descriptions are provided.

Bit: 31:4  Name: MBZ  
Function: Must be zero

Bit: 3  Name: Bus Error  
Function: This bit is set when a machine check results from either a read to nonexistent memory or a read of uncorrectable data. It is cleared by writing a 1.

![Machine Check Error Summary Register (MCESR)](image)

Figure 11-15  Machine Check Error Summary Register (MCESR)  
IPR #26_{16}

Bit: 2  Name: TB Error  
Function: This bit is set when a machine check results from a translation buffer error; it is cleared by writing a 1.

Bit: 1  Name: MBZ  
Function: Must be zero

Bit: 0  Name: XB Error  
Function: Execution Buffer Error. This bit is set when an error is detected trying to use data from the execution buffer. This bit is cleared by writing a 1.

MACHINE CHECK STATUS REGISTER (MCSR)
This register provides additional information about bus errors and translation buffer errors causing a machine check. The MCSR is illustrated in Figure 11-16 and the bit assignments are described.
Privileged Registers

<table>
<thead>
<tr>
<th>31</th>
<th>24</th>
<th>23</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- MEMORY DISABLE
- READ LOCK TIMEOUT
- TB GROUP 1 TAG ERROR
- TB GROUP 0 TAG ERROR
- TB GROUP 1 DATA ERROR
- TB GROUP 0 DATA ERROR
- TB HIT/MISS
- NON-EXISTENT MEMORY OR READ LOCK TIMEOUT
- UNCORRECTABLE DATA
- LOST ERROR
- CORRECTED DATA

Figure 11-16  Machine Check Error Type Register (MCSR) IPR #17

Bit: 31:17  Name: MBZ
Function: Must be zero

Bit: 16  Name: Memory Disable
Function: This bit is read-only by software and is initially 0. If set, only the cache is read and write (i.e., main memory is not referenced or modified). It is highly unlikely that this bit would ever be set; it would indicate a serious CPU failure.

Bit: 15:13  Name: MBZ
Function: Must be zero

Bit: 12  Name: Read Lock
Function: This bit is read-only by software and is initially zero. It is set by a Read Lock Timeout and defines how bits <3:0> are interpreted.

Bit: 11:8  Name: TBGPE
Function: These bits are read-only and are set to 0 at bootstrap time. They define the type of translation buffer group parity error and are cleared when bit <2> of the MCESR is reset.

Bit: 7:5  Name: MBZ
Function: Must be zero

Bit: 4  Name: Hit/Miss
Function: This bit is read-only and is set to 0 at bootstrap time. It indicates whether the last microcoded reference resulted in a hit or a miss.

Bit: 3:0  Name: Bus Error
Function: These bits are read-only and are set to 0 at bootstrap time. They define the type of bus error and are cleared when bit <3> of the MCESR is reset.

If bit <3> is set, a nonexistent memory or read lock timeout error is indicated. If bit <2> is set, an uncorrectable data error is indicated. If
Privileged Registers

bit<1> is set, a lost bus error is indicated. If bit<0> is set, the indication is corrected data.

TRANSLATION BUFFER GROUP DISABLE REGISTER (TBDR)
The Translation Buffer Group Disable Register (TBDR) controls the enabling and disabling of the translation buffer and controls which half of the buffer is replaced. This register is illustrated in Figure 11-17 and the bit assignments are described following.

![Diagram of TBDR Register]

Figure 11-17 Translation Buffer Group Disable Register (TBDR)
IPR #24_{16}

Bit: 31:4 Name: MBZ
Function: Must be zero

Bit: 3 Name: Replace
Function: This bit is read/write and is set to 0 at bootstrap time. When this bit is set to 1, bit <2> controls which half of the translation buffer is replaced.

Bit: 2 Name: Group Replace
Function: This bit is read/write and is set to 0 at bootstrap time. When bit <3> is 1 and this bit is 0, group zero will be replaced. When bit <3> is 1 and this bit is also 1, group one will be replaced.

Bit: 1 Name: DG1
Function: This bit is read/write and is set to 0 at bootstrap time. When this bit is a 1, group one of the translation buffer will be disabled.

Bit: 0 Name: DG0
Function: This bit is read/write and is set to 0 at bootstrap time. When this bit is a one, group zero of the translation buffer will be disabled.

NOTE
The machine cannot be run with mapping enabled and DG1 and DG0 set.
Privileged Registers

CACHE DISABLE REGISTER (CADR)
This register is used to turn off the cache. This bit is cleared on power-up. Figure 11-18 illustrates the cache disable register.

![Cache Disable Register](image)

Figure 11-18  Cache Disable Register (CADR) IPR #25_{16}

Bit: 31:1  Name: MBZ  
Function: Must be zero

Bit: 0  Name: Cache Disable Bit  
Function: If this read/write bit is 0, the cache is enabled to operate normally. If this bit is a one, the cache is turned off.

CACHE ERROR REGISTER (CAER)
The Cache Error Register (CAER) logs information about the nature of cache parity errors. This register is read/write, and only bits <3:0> are implemented. All bits are set to 0 at bootstrap time. This register is illustrated in Figure 11-19 and the bit assignments are described below.

![Cache Error Register](image)

Figure 11-19  Cache Error Register (CAER) IPR #27_{16}

Bit: 31:4  Name: MBZ  
Function: Must be zero

Bit: 3  Name: Tag Error  
Function: If this bit is set, it indicates that the parity error was in the tag portion of the cache.

Bit: 2  Name: Data Error  
Function: If this bit is set, it indicates that the parity error was in the data portion of the cache.

205
Privileged Registers

Bit: 1  Name: Lost Error
Function: If this bit is set, it indicates that there were multiple cache errors.

Bit: 0  Name: Hit/Miss
Function: If this bit is 0, it indicates that the last microcoded reference resulted in a miss; if the bit is 1, the last microcoded reference resulted in a hit.

TRANSLATION BUFFER REGISTER
The ability to read and write locations in the VAX-11/750 translation buffer is important for diagnostic purposes. The MTPR and MFPR instructions are two-operand instructions. One operand specifies a longword source or destination, and the other operand specifies the IPR number of TB. For accessing the translation buffer, a third operand containing the address of the translation buffer location to be referenced is needed. The virtual address field of the P0BR supplies this virtual address. For MTPR, the P0BR contains the virtual address whose page table entry (PTE taken from source operand) is to be written into the translation buffer. For MFPR, the P0BR contains the virtual address whose PTE is to be read from the translation buffer into the destination. Figure 11-20 illustrates how this register and the translation test works.

![Figure 11-20 Translation Buffer Register (TB) IPR #3B16](image)

NOTE
For this process to work correctly, memory management must be disabled by clearing the Map Enable Register. Furthermore, only stand-alone diagnostics may use this feature.
CHAPTER 12

MICROPROGRAMMING

The user microprogramming capability of the VAX-11/750 offers sophisticated users an opportunity to custom tailor the processor's performance to meet specific needs. This feature—called the user control store option—is best used by those who wish to increase the speed of a specific type of data handling (for example, certain scientific calculations). A scientist who is working with dynamic graphic display data may wish to increase the speed and specificity of the calculation by either permanently or temporarily modifying the way the processor implements the software.

The programmer who intends to use the microprogramming option on the VAX-11/750 should have extensive experience in assembly language programming and should be highly knowledgeable about the VAX family architecture and the VAX/VMS operating system.

The UCS option consists of 1024 80-bit words of random access memory for storing user programs and data.

MICROPROGRAMMING

Before explaining microprogramming as it relates to the VAX-11/750, it is helpful to consider some of the basic concepts of microprogramming and some of the variables which should influence the decision to use microprogramming.

Microprogramming is a method of controlling the functions of a computer. The essential ideas of microprogramming were first outlined by M.V. Wilkes in 1951 (Wilkes, M.V., "The Best Way to Design an Automatic Calculating Machine," Manchester University Inaugural Conference, 1951, pp. 16-21). Wilkes proposed a structured hardware design technique to replace prevailing methods of logic design. He observed that a machine-language instruction could be subdivided into a sequence of elementary operations which he called micro-operations, and he compared the execution of the individual steps to the execution of the individual instructions in a program. This concept is the basis of all microprogramming.

For many years, microprogramming was used almost exclusively by hardware designers. As new machines were designed that incorporated advances in theory and technology, the software for the older, slower machines became obsolete. Microprogramming could overcome this incompatibility. New machines could be provided with addi-
Microprogramming

tional read-only memory, or control store, which allowed them to emulate earlier computers. Emulation, or the interpretive execution of a foreign instruction set, was later extended to provide upward and downward compatibility among a number of computers in a family. It is this feature which allows the compatibility between the VAX-11/750 and the VAX-11/780.

Each step in the emulation of a foreign instruction is controlled by a microinstruction; a set of such microinstructions is known as a microprogram. Microprograms are held in a control store, a block of high-speed memory that can be accessed once per microcycle. A microcycle is the basic unit of time within a processor.

Microprogramming, as a user tool, has evolved slowly. Three things had to happen before its use became widespread. First, technological advances in the field of fast random-access memories were required. Using read-only memories in a user environment was troublesome and expensive, because correcting programming errors required new memories. Second, user microprogramming required specialized knowledge. Users and educators were at a severe disadvantage because only engineers had been actually involved in the design of microprogrammed computers. However, in recent years, microprogramming has found a place in computer science curricula, and has been widely used throughout the electronics and scientific industry. The third, and most important, factor in the expansion of user microprogramming is the inclusion of generality and extendability in computer design. A machine designed solely to implement a given instruction set, with no address space for user control programs, or no built-in features to support general microprogramming, makes writing user microcode extremely difficult. Equally important, software tools had to be developed so that the user would not have to work solely with binary patterns.

The UCS option and the software microprogramming tools developed for the VAX-11/750 represent DIGITAL's way of making VAX-11/750 user microprogramming a reality.

USES OF MICROPROGRAMMING
There are essentially three uses of microprogramming. They can be classified as follows:

Class 0

Instruction Set Extensions
Some functions were considered too specialized to be included in the original VAX design. However, by microprogramming these functions, they can become new
Microprogramming

VAX instructions. Their definition should conform to VAX instruction format and style.

Class 1

Application Kernels
Most applications and systems programs have sections which are executed much more frequently than others. A useful rule of thumb is that 10% of the code is executed 90% of the time. Kernels within these critical sections can be microprogrammed for better throughput. Examples include the Fast Fourier Transform, and an operating system's memory allocation routine.

Class 2

Emulation
The interpretive execution of an instruction set by software is generally called simulation. When this interpretation is done by hardware it is called emulation. Microprogramming provides a means for inexpensively emulating several different instruction sets on one piece of hardware. The tasks involved in emulation include instruction decode, address calculation, operand fetch, and I/O operation, as well as instruction execution.

Class 0 applications are relatively simple and straightforward uses of microprogramming. Class 1 applications require more intensive study and possibly statistical analysis to improve performance significantly.

The final class of applications, emulation, is best served by a machine specifically designed as a general purpose emulator. The VAX-11/750 was designed to implement the VAX architecture; hence, the organization is keyed to the 32-bit VAX longword and to the other characteristics of the VAX family architecture. These factors in large part determine which other computers the VAX-11/750 can emulate.

PROCESSOR STATE
The processor state of a computer is the set of registers and flags that store the information after one instruction completes. This information can then be used during the execution of the next instruction.

Programmers working at different levels of a machine see different machine states. An applications programmer may never be concerned with machine state at all. A machine-language or macro-level
Microprogramming

 programmer knows the VAX-11/750 processor state to be defined by
 the contents of R0 through R12, the FP, SP, PC and PSL and a few
 other per-process internal registers. However, nearly 100 registers are
 included in the machine state known to VAX-11/750 microprogram-
 mers. At the nano- or hardware level, even more machine state is
 seen.

 The concept of machine, or processor, state is fundamental to an
 understanding of microprogrammable processors like the VAX-
 11/750. State changes at the microprogramming level can affect the
 macro-level processor state.

 A computer is uniquely defined by the functions it performs and by the
 machine states it enters while performing those functions. Because of
 this, two machines can be built differently and yet perform identically.
 A microprogrammed machine changes state as it reads successive
 locations in the control store, emulating the state changes that would
 take place in a completely hard-wired machine. Additionally, the
 macro-level state, which is a subset of the micro-level machine state,
 changes as if there were no machine but the macro-level machine.

 ARCHITECTURE AND ORGANIZATION

 To distinguish the micro-level machine from the macro-level machine,
 it is useful to differentiate between the terms architecture and or-
 ganization.

 Architecture refers to that set of a computer's features that are visible
 to the programmer. To a machine-language programmer, this in-
 cludes the general registers, the instruction set, and the Processor
 Status Word.

 Organization describes a level below architecture, and is concerned
 with many items that are transparent to the programmer. The term
 architecture describes what facilities are provided, while organization
 is concerned with how those facilities are provided. Occasionally,
 another term is included in this hierarchy: realization. This term char-
 acterizes the components used in a particular machine implementa-
 tion, such as the type of logic and chips used.

 The macro-level organization, transparent to the macro-level pro-
 gramer, is the micro-level architecture of the machine. This concept
 is illustrated in Figure 12-1.

 The micro-level architecture of the VAX-11/750 is radically different
 from the standard VAX family structure visible to the machine lan-
 guage programmer.
To microprogram the VAX-11/750 successfully, the user must be familiar with the details of its micro-level architecture. An overview of that architecture is provided in this chapter. Detailed information may be obtained from Chapters 1 and 2 of the CPU Technical Description and from Introduction to the VAX-11/750 Microarchitecture.

**MICROARCHITECTURE OF THE VAX-11/750**

The VAX-11/750 is a microprogrammable computer which was specifically designed to emulate the VAX family architecture. It consists of up to 160 Kb of control store (10 Kb are accessible for user microprogramming), one 80-bit microinstruction per word, a microsequencer, a 32-bit data path, and a number of status and control registers which are needed to emulate VAX. Figure 12-2 is an overall block diagram of the VAX-11/750 microarchitecture.

There are three major internal buses in the VAX-11/750: the WBUS, the MBUS and the RBUS. The main bus is the WBUS. The output of the ALU, unless inhibited, goes on the WBUS. Data to be written into memory and data to be written into the scratch pad registers are taken from the WBUS. Status and control information is passed to and from the particular registers via the WBUS. The MBUS and RBUS provide sources for the super rotator and the ALU. Data on the MBUS are
Figure 12-2  VAX-11/750 Microarchitecture

Figure 12-3 shows the fields of a VAX-11/750 microinstruction. Primarily taken from the VAX-11/750 main memory interface registers and from the M scratch pad registers. Data on the RBUS are from the R scratch pad registers and the Long Literal Register.

The data path consists of two sets of scratch pad registers, the LONLIT (Long Literal) register, the super rotator, the ALU, and two special registers (D and Q). All are 32 bits wide. The ALU can perform 2's complement arithmetic, BCD arithmetic, and logical operations. There are two inputs to the ALU (A and B); both are multiplexed. The MUX field controls the sources of the ALU, the ALU field specifies the arithmetic or logical function to be performed, and the DQ field controls the destination of the output. The source of the carry input to the ALU is specified by the ALUC1 field. Input to the ALU can come from the RBUS, the MBUS, the super rotator, and the D and Q registers. The output of the ALU can be shifted or rotated (by itself, or combined with the Q register); the type of shift is determined by the DQ and
Microprogramming

ALUSHF fields. The output of the ALU can go on the WBUS and it can go to the D or Q register. In addition, the ALU can perform certain special functions (for example, a FAST MULTIPLY) where the input and output are specified as part of the function. This is done by coding the MUX, ALU and DQ fields as a single unit. We call this the ALPCTL field.

There are 56 scratch pad registers. Eight have two ports (to the MBUS and the RBUS), eight can be accessed by the MBUS only, and 40 can be accessed by the RBUS only. The particular registers accessed during any microcycle are specified by the MSRC and RSRC fields. Writing to the scratch pad registers is controlled by the SPW field.

The super rotator is a powerful combinational circuit. It can barrel shift a 64-bit data element, it can extract a desired field from a given piece of data, and it can construct a 32-bit data element according to a variety of specifications. This provides the VAX-11/750 with a very efficient bit manipulation capability. The super rotator is controlled by the ROT field.

Immediate data of 9 or 32 bits can be entered into the data path from the instruction stream under the control of the LIT field. The LIT/LONLIT micro-order causes the LONLIT register to be loaded with bits <62:31> of the microinstruction. (The notation <field>/<code> is used extensively throughout this discussion. LIT/LONLIT represents

Figure 12-3  VAX-11/750 Microinstruction Fields

215
the micro-order in which the LIT field contains the LONLIT code.) These data are then available in the next microinstruction; access to it is controlled by RSRC. The LIT/LITRL micro-order causes bits <39: 31> of the microinstruction be made available as input to the super rotator during the same microcycle.

The basic microcycle of the VAX-11/750 architecture is 320 ns. This is sufficient to read from the scratch pad registers, pass through the super rotator, perform an ALU operation, and write back to a scratch pad register. Certain activities can cause a microinstruction to need more than 320 ns. For example, suppose the address of the next microinstruction is to be computed partly from the results of the ALU operation. The micro-order BUT/WX.EQ.0 produces such a situation, and when this happens, the CLK field can extend the microcycle to 480 ns.

The VAX-11/750 has no microprogram counter. The address of the next microinstruction (CSA) is usually determined by the microsequencer in one of several ways. The microsequencer can generate a microbranch (up to 64-way conditional branch) based on the values of certain internal signals specified by the BUT field. The branch addresses are based on the contents of the NEXT field and the values of these specified signals. Alternatively, the CSA can be obtained by popping the microstack. The address can also be obtained from the IRD1 or IRDX ROMs. The above schemes are all under the control of the BUT field. In addition, the hardware can override these addressing schemes by forcing the CSA (by means of a microtrap) to a fixed address. This last technique is used to initiate some of the VAX exceptions and interrupts.

Control store addressing supports up to 16K 80-bit words of control store. Of this, the low order 6K words are required to emulate VAX and the next 2K words are available for the RDM (remote diagnostic module). This leaves a possible 8K 80-bit words of address space for UCS, of which 1K (or 10 Kb) are actually available as part of the user control store.

Finally, the VAX-11/750 contains several features specifically included to aid in the emulation of VAX. The PC, MDR, WDR, Translation Buffer, and Execution Buffer are a few of the registers which are used to control access to the memory cache and main memory. The BUS field controls this access. The PSL, Software IPR, status flags, console and TU58 registers, ASTLVL register, and the internal next interval register are a few of status and control registers which are used to control interrupt and exception processing. The WCTRL field controls these functions.
EMULATION OF VAX
The primary use of the VAX-11/750 microarchitecture is to emulate the VAX computer. This is done as follows.

The VAX registers are, for the most part, implemented in the M and R scratch pads. In particular, R0-R12, FP, and SP are implemented by R[10] through R[1E] in the R scratch pad. The stack pointers are R[20] through R[24]; the memory management registers are R[28] through R[2D]; the high order 16 bits of ICR and NICR are R[2E]; the PCBB is R[25]; and the SCBB and SISR are M[0E] and M[0F]. The rest of the VAX registers, including PC, PSL, and ASTLVL are implemented by VAX-11/750 registers designed specifically for those purposes.

Emulation starts with the BUT/IRD1 micro-order. This is the signal to begin the emulation of the next VAX machine instruction. In the micro-code which emulates each VAX instruction, this micro-order is present in the last microinstruction.

BUT/IRD1 first invokes a hardware routine DOSERVICE which checks for traps and interrupts. If any are pending, the processor will micro-trap to the appropriate control store address to initiate the trap or interrupt. Then, it causes one or two bytes (depending on the instruction) to be fetched from the instruction stream (i.e., from the execution buffer, XB) and loaded into the IR and OSR. (The opcode of the next VAX instruction is loaded into the IR and OSR.) The XB is an 8-byte register. The instruction stream is prefetched automatically, four bytes at a time, and stored in the XB. Each time the XB is accessed, the PC is automatically incremented.

The processor then begins the emulation of the VAX instruction. Usually it branches to common code to evaluate the first operand specifier. The branch address is obtained from a ROM (IRD1 ROM) which is indexed by the opcode, by whether the VAX instruction is being started from the beginning or whether it is being restarted after having been suspended in the middle (i.e., FPD set). The common code terminates with a BUT/IRDX micro-order which causes the next control store address to be obtained from the IRDX ROM. The rest of the operand addresses are computed and the VAX instruction is emulated, terminating in a BUT/IRD1 micro-order, which starts the cycle again.

If the VAX instruction requires a memory access, a BUS read or write initiates the access. The VAX-11/750 maintains a translation buffer (TB) of Page Table Entries (PTEs), which is used to map virtual addresses into physical addresses. If the PTE is not present in the TB, or if the memory access is unaligned (VAX allows the user to disregard natural word and longword boundaries), then the VAX-11/750 micro-
traps (forces the control store address) to a specific location to correct the problem. The executing microinstruction is not allowed to complete until the problem is corrected. If the problem is corrected, control returns to that microinstruction.

If the emulation of a VAX instruction requires enough time that pending interrupts cannot be ignored, the VAX emulation tests for interrupts under microprogram control. If an interrupt is to be initiated, the processor is put into a consistent state by either undoing whatever processing has occurred, or if that is not possible, by setting FPD and following a prescribed procedure for saving the relevant machine state so that the VAX instruction can be restarted at the point where it was suspended.

Exception and interrupt initiation is emulated in microcode. The branch to the starting address is caused by a microtrap if the condition was detected by the hardware (for example, by DOSERVICE), or by a microbranch in case the condition was detected under microprogram control. In either case, the microcode selects the appropriate stack to service the exception or interrupt, pushes the current PC and PSL as well as any necessary parameters on that stack, puts the processor into a consistent machine state, constructs a PSL for the service routine, performs any special tasks peculiar to that exception or interrupt, and loads PC with the starting virtual address of the service routine. The microcode terminates in BUT/IRD1, the signal to fetch the next VAX instruction. In this case, that instruction is usually the first instruction in the VAX service routine.

**USER MICROPROGRAMMING**

A number of VAX-11/750 features support writing application-specific code to augment the VAX instruction set or to refine software. These features are discussed below.

To support user microprogramming, the data path includes: 18 general purpose 32-bit scratch pad registers, 8 of which have ports to both the RBUS and the MBUS; the super rotator, which allows very efficient (in hardware) bit picking operations; and a flexible ALU. As is the case with many microprogrammable computers, inputs to the ALU are multiplexed. The output of the ALU can be shifted or rotated (alone or in combination with the Q register), and the output can be applied to several alternative destinations.

The microsequencer supports general microprogramming in three important ways: conditional branching, loop control, and subroutine control. The VAX-11/750 has six independent flag bits (FLAG0-FLAG5), four of which are always available for user microprogramming and two of which are conditionally available. Available bits can
Microprogramming

be set or cleared under microprogram control (e.g., MISC/SET.FLAG0), then later used for conditional branching (e.g., BUT/FLAG0). VAX-11/750 also has a 5-bit step counter which can be initialized to any arbitrary value \(0 \leq n \leq 31\). If this is followed by a loop which terminates with BUT/DBZ.SC, the loop will be performed \(n\) times. Each iteration will conclude with a "decrement the step counter and branch on zero." In the case of subroutine control, a 16-deep microstack is available for nested subroutine calls. The JSR/PUSH micro-order pushes the CSA (control store address) onto the microstack; the BUT/RETURN micro-order pops it.

The VAX-11/750 provides two independent paths to memory, one for data (read/write) and one for instructions (read-only). In the case of data, the virtual address is used as the storage address register, and the MDR (read) and WDR (write) are used as storage data registers. In the case of instructions, the PC points to memory and the XB can be used as a data storage register. Both are available to the user microprogrammer, and were used in the VAX-11/750 firmware where two distinct data paths to memory were needed.

Finally, the 80-bit VAX-11/750 microinstruction provides parallelism. A single microinstruction can introduce an immediate operand, perform an ALU function, push a control store address on the microstack, set or clear a flag bit, initiate a read or write to memory, and perform a multiway conditional branch, etc.

Access to the UCS is by the opcode FC in the VAX instruction stream. This causes an "opcode reserved to customers" fault. If UCS is present and if SCB[14] <1:0> = 10, this results in a branch to a location in user control store. The actual control store address is taken from SCB[14] <15:2>. From this point on, user microcode has control of the micromachine. The instruction stream, via the execution buffer, can be used if appropriate; the VAX firmware is also available.

Control can be returned to the VAX emulation by means of the BUT/IRD1 micro-order. When an exception or interrupt is to be handled in UCS, the PC and PSL of the suspended process are not pushed on the stack. Therefore, before BUT/IRD1 is invoked, the PC must be set to the memory location of the VAX machine language program where you want the firmware to take over. The micro-order WCTRL/PC-WB (load the PC with the contents of the WBUS) can accomplish this.

USER CONTROL STORE BENEFITS
To gain real benefit from use of the UCS option, time and resources should be invested in two areas of study before attempting any UCS
Microprogramming

microprogramming. These two areas are: 1) understanding the VAX-11/750 and 2) analyzing the proposed application.

To microprogram the VAX-11/750 effectively, the internal details of the microarchitecture must be studied—particularly the data path. Although this is not a difficult task, the largely unprotected nature of the microprogramming environment may seem complex and unpredictable.

Microprogramming will not always result in significant performance gains. Applications well suited to microprogramming may improve performance by a factor of 5 to 10; poorly suited ones may not improve performance at all. It is important to understand the application and to analyze the execution of its individual instructions.
CHAPTER 13
RELIABILITY, AVAILABILITY, AND MAINTAINABILITY FEATURES

INTRODUCTION
This chapter describes the extensive reliability, availability, and maintainability features that are an integral part of the VAX-11/750 system. The chapter provides a general overview (at a non-technical level) of the many hardware and software features, packaging improvements, and diagnostic aids, that prevent frequent system failures and allow the system to continue operation while ignoring certain types of failures. These features also make the VAX-11/750 easier and quicker to repair when it does fail. The reliability, availability, and maintainability features make the VAX-11/750 a true state-of-the-art system as well as a more reliable system than one without these important features. The net result is increased uptime and reduced overall cost of ownership. The chart below and on the following page provides a summary of reliability, availability and maintainability features and benefits.

Table 13-1  RAMP Features and Benefits

<table>
<thead>
<tr>
<th>Hardware Architecture</th>
<th>Reliability</th>
<th>Maintainability</th>
<th>Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access Modes</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Console Subsystem with RDM Option</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Consistency and Error Checking</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Special Instructions</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Fault Detection and Maintenance Features</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Fault Tolerance Features</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

223
### Reliability, Availability and Maintainability Features

<table>
<thead>
<tr>
<th></th>
<th>Reliability</th>
<th>Maintainability</th>
<th>Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Improved Packaging</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Air Flow</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Sensors and Indicators</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Modular Power Supply</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>and Option Replacement</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Cabling</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Power Supply</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><strong>Improved Diagnostic Aids</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remote Diagnosis</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>System Verification</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Diagnostics</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Functional and Fault-</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Isolation Diagnostics</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Software Architecture</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operating System</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Consistency Checking</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Redundant Recording of</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Critical Information</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exception Handling</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>On-Line Error Logging</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Automatic Restart</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Reconfiguration</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>On-Line Update and</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Maintenance</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Reliability
Product reliability implies a design that minimizes hardware and software failures and reduces the impact of failures that cannot be prevented. Therefore, the VAX-11/750 includes well-designed modules, thoughtful hardware packaging, efficient air flow, and implements error correcting code (ECC), parity checking, and other features which increase system reliability.

Availability
Features also have been implemented in the VAX-11/750 which contribute specifically to system availability. This makes the VAX-11/750 a general purpose system that is usable under conditions which would normally require a stand-alone environment. System management functions involving routine maintenance and update operations, which would usually occur as stand-alone activities, occur on-line under this system (both locally and remotely). Most peripheral diagnostics run on-line under VAX/VMS, and the unattended automatic restart capability avoids the time-consuming stand-alone environment normally required to bring the system on-line after a power failure or system crash. Because the VAX/VMS operating system implements power-fail recovery code, this usually allows program execution to resume automatically, with no operator intervention, if the contents of main memory were maintained by the optional battery back-up. If memory contents were not maintained, the VAX-11/750 attempts to reboot itself automatically. In addition, if parts of the system are down because components need servicing, the VAX-11/750 provides some configuration alternatives. These alternatives support a reliable subset of the system configuration—often sufficient to finish the job. For instance, the memory cache may be disabled, yet the system can continue to execute user processes.

Maintainability
Product maintainability implies a design that requires few maintenance steps which can be easily and quickly performed—one equipped with better diagnostic tools. In the VAX-11/750, maintainability is increased by thorough diagnostic programs and procedures—including a remote diagnosis option—and by a unique packaging scheme which facilitates repair.

The VAX-11/750 system is designed and built with integral hardware and software features which continually monitor and verify system integrity. System error logging and improved diagnostic aids help field engineers quickly locate and identify error conditions and verify a fault-free system. Furthermore, improved system packaging speeds replacement of faulty subassemblies. Overall, these reliability, avail-
ability, and maintainability features increase user productivity, while reducing down-time and maintenance costs.

DESCRIPTION OF RELIABILITY, AVAILABILITY, AND MAINTAINABILITY FEATURES
The VAX-11/750 reliability, availability, and maintainability features are described below under four categories: hardware features, improved packaging, diagnostic aids, and software features.

Hardware Features
Four hierarchical access modes — To protect system information and increase reliability, the memory management hardware defines four hierarchical modes of access privilege (from most to least privileged): kernel, executive, supervisor, and user. The hierarchy is implemented so that a read or write which is permissible in a less privileged mode will also be allowable in a more privileged mode. The VAX/VMS operating system is designed so that critical components run in highly privileged access modes (kernel and executive). This layered design increases protection and improves data reliability and integrity for both the system and the users. The operating system also provides a file protection scheme which prevents unauthorized users from accessing privileged data; this security feature will be appreciated by users handling sensitive material.

Gate array technology — 90% of the logic circuits in the VAX-11/750 central processor are custom LSI (large scale integration) circuits designed by DIGITAL using the latest gate array technology. Because this technology requires fewer components than off-the-shelf circuitry for equivalent performance, system reliability is increased. In addition, physical size and power consumption are reduced to about one third of what they would be using conventional technology. The size reduction achieved using gate arrays rather than off-the-shelf components also means that fewer printed circuit boards are needed for field spares. Therefore, because there are fewer boards to diagnose and repair, these maintenance activities require less time than they would with conventional technology.

Console subsystem and optional remote diagnosis module — The console subsystem and optional remote diagnosis module (RDM) provide both local and remote diagnosis of system errors. This results in faster and easier maintenance procedures for increased system availability. The console subsystem also simplifies system bootstrap procedures; and the TU58 tape cartridge provides a convenient medium for software update distribution. Furthermore, simple console commands (entered through the console terminal) replace traditional front panel switches and lights. This console command language is a more
Reliability, Availability and Maintainability Features

powerful tool than the switches and lights; in addition, it makes it easier for the user to communicate with the console subsystem.

**Consistency and error checking** — Automatic consistency and error checking detects abnormal instruction uses and illegal arithmetic conditions. Continual checking by hardware and uniform exception handling by software increase data reliability. If the checks fail, an exception is signalled and the current instruction sequence is suspended. A condition handler is entered and the system software or the user program provides an appropriate response to the condition. Such checks increase data reliability by preventing error conditions from propagating through a data base or a system. Some of the checks performed include:

- Arithmetic traps. Traps occur when overflow, underflow and divide by zero arithmetic conditions are detected. Hardware detection of these error conditions allows checking to be used in high performance software, where software checking would normally be prohibitively slow. Overflow and underflow traps may be enabled or disabled by setting bits in the Processor Status Longword (PSL), allowing these arithmetic exception conditions to be ignored, if appropriate.

- Limit checking traps. Decimal string instructions all have length limit checks (0-31 decimal digits) performed on output strings to ensure that instructions do not overwrite adjacent data.

- Reserved operand traps. *Reserved to customer* and *reserved to DIGITAL* fields and opcodes ensure that user extensions to the VAX architecture (such as user-defined instructions or data structures) do not conflict with future DIGITAL expansions.

**Special instructions** — Special instructions, such as CALLx and RETURN, provide a standard interface which is identical on user routine calls and system calls. Furthermore, these instructions have hardware-implemented register save/restore and consistency checking. The CRC instruction also provides powerful block checking error code calculations, important in communications applications. These features increase system reliability.

**Integral fault detection and maintenance features** — These features increase system availability and make system maintenance more efficient by assisting in hardware error diagnosis. Specific features include the following:

- **Memory error correcting code (ECC)** will correct all single-bit memory errors, will detect all double-bit memory errors, and will detect all greater than double-bit errors if the number of errors is an even number. ECC provides protection from non-repeatable errors by
automatically correcting data. Detections and corrections are noted in the error log as a preventive maintenance aid.

- **Disk error correcting code** detects and corrects most errors. Detections and corrections are noted in the error log as a preventive maintenance aid.

- **Microverify code** detects faults in internal CPU data paths, registers, and in the cache and translation buffer parity logic. All of the cache memory is also tested. Microverify routines are not intended to be of general diagnostic use; rather, they test the boot path so that the user can be confident of a successful boot. If errors are reported to the console, extensive diagnostics should be run to further isolate any hardware problems that may exist.

- A **system identification (SID) hardware register** maintains information pertinent to the system processor: type and revision number. This information may be examined (during the software logging process, for example) to determine the engineering status of the processor.

- Several **maintenance registers** contain bus-specific maintenance information and can be examined at the time of an error to help determine the cause. These registers are:
  - Machine Check Error Summary Register, which logs information about the cause of a machine check.
  - Cache Error Register, which indicates the type of cache error and whether the cache reference resulted in a hit or a miss.
  - Machine Check Status Register, which indicates certain types of hardware errors.

- A **high resolution interval clock** (1 μs) is used by diagnostics to test time-dependent functions without requiring machine-specific timing loops in programs. It is also used by VAX/VMS to time stamp operations.

- **Parity checks** are performed on MASSBUS data, control, silo, memory cache data and addresses, address translation buffer transactions, microcode, and user control store.

- **Selective hardware disabling**. Several hardware elements—the memory management, cache, and translation buffer—can be disabled by diagnostics to aid in isolating hardware problems. These elements may also be disabled by VAX/VMS to allow continued operation at a reduced level of performance.

**Fault tolerance features** — The VAX-11/750 has fault tolerance features which allow processing to continue without loss of information,
Reliability, Availability and Maintainability Features

even though hardware errors may be occurring. Specific features include:

• **VAX/VMS performs dynamic bad block handling** to increase the reliability of the medium. Bad blocks can occur when a disk surface becomes worn. When hardware detects a bad block during a read, the VAX operating system marks the header of the file in which the error occurred. When the file is eventually de-allocated, the system checks the file header to see if any bad blocks exist in the file. If so, they are designated *permanently in use* and are not allocated for use by other files.

• **Write-verify checking hardware** in mass storage peripherals ensures data reliability by verifying all input and output disk and tape operations. This hardware, supported by the device drivers, compares each block for errors immediately after the block is read or written. Checking may be performed on all reads or writes to a file or volume, or checking may be specified for a single read or write operation, or not at all.

• **Track offset retry hardware** enables programmed software recovery from disk transfer errors, thereby increasing data reliability and availability. If an error occurs during a read operation, the error is signalled by the disk hardware and the operation is retried. If the retry fails, the disk head may be repositioned slightly (offset) on either side of the normal track location in an attempt to read the data correctly.

**Improved Packaging**

**Cabinet** — The VAX-11/750 is self-contained in a cabinet which is 41.8" (106.2 cm) high, 29" (73.7 cm) wide, and 30" (76.2 cm) deep. The system is designed to meet the applicable Underwriters Laboratories (USA), Canadian Standards Association (CSA), and international standards. This guarantees VAX-11/750 customers product safety and integrity. Furthermore, the VAX-11/750 printed circuit card cage and backplane are fixed mounted into the system cabinet. Because these elements are not on slides, no service loops or internal cables are required.

**Easy access** — The physical packaging of the VAX-11/750 has been designed so that all components are easily accessible for servicing. All VAX-11/750 CPU modules, CPU options (user control store, MASSBUS adapters) and memory arrays have no cables connected to them. This allows the modules to be easily installed and removed without cable handling and its associated wear. For example, the MASSBUS adapter cables come off the backplane, not from the module. Therefore, the cables remain in place and are not bent or damaged when the modules are installed or removed.
Improved air flow — System reliability is increased by a powerful blower designed to pull cool air in at the top of the cabinet, drive it down through the modules, and force the air out at the bottom rear of the cabinet. By pulling the air in from the top in this fashion, the system receives clean air rather than air contaminated with dirt particles from the floor. The blower operates well below maximum ratings, which extends its operating life significantly. Furthermore, the cooling air flow is adequate to operate even when all cabinet doors are open, allowing on-line servicing of modules.

Power loss, temperature, and air flow alarms — The VAX-11/750 has sensors to detect emergency conditions and to protect the system from damage. Indicators which are visible on the back of the cabinet signal abnormal operating conditions. (See the photograph at the beginning of this chapter.) Both external conditions (short circuit or over voltage, loss of power) and internal conditions (regulator or main power board failure) are signalled.

Option replacement — The MASSBUS adapter, memory arrays (256 Kb), user control store, and remote diagnosis module are all single printed circuit board options and can be removed or replaced in a few minutes.

Modular power supply — The power system for the VAX-11/750 consists of three modules: a power controller and two power regulators. Any of the three modules can be replaced in less than ten minutes, meaning increased maintainability and availability. In addition, the power supply indicators are visible at the rear of the cabinet. This allows the user to determine when a power supply has failed so that Field Service can be notified.

Improved Diagnostic Aids
Remote diagnosis of system hardware errors — Remote diagnosis is a customer option which can substantially lower maintenance contract prices, as well as increase system availability and maintainability. (The remote diagnosis option is only available to those customers with maintenance contracts, and applies solely for the duration of the contract.) The experienced staff at DIGITAL's Diagnostic Center (DDC) are available seven days a week, 24 hours a day to quickly and efficiently apply their expertise to problems. The DDC will answer non-technical, operational questions, but more importantly the Center has a host diagnostic computer to aid in pinpointing system faults. With these capabilities, VAX-11/750 customers who have had the optional remote diagnosis module (RDM) installed can have their systems connected to the DDC host computer within 15 minutes after a problem call is placed. Then, special diagnostic routines exercise hardware to isolate
and reveal faults. Because some diagnostics can even be run on-line, faults can often be diagnosed without bringing the entire system down. Furthermore, with the customer's permission, the error log can also be run from the remote port. This kind of diagnosis is fast, accurate, and automatic, because the host diagnostic computer generates results which can be analyzed immediately—giving Field Service a head start in determining what part or parts need to be replaced.

These remote diagnosis capabilities contribute significantly to both system maintainability and availability. Before remote diagnosis, when a failure occurred, a service representative had to diagnose the problem on site. The replacement of a faulty part sometimes meant a return to the local Field Service office to obtain the necessary spare part. Now, with the assistance of the remote diagnosis option, problems are known in advance and the engineer can carry the right parts. On-site repair time is also reduced because extensive diagnosis is performed in advance. With overall repair time reduced, the user benefits from increased system availability. Availability is also increased by the DDC's ability to answer operational questions immediately. This minimizes down-time caused by power failures, intermittent errors, and operator-induced errors. These kinds of failures can be quickly identified by remote diagnosis and rectified without a visit from the Field Service account representative.

Maintainability is further increased by the preventive maintenance diagnostic sessions offered through remote diagnosis. Through the DDC, these sessions can be pre-scheduled for off-peak periods, also contributing to system availability during high usage times.

After receiving a problem call from a user, the DIGITAL Diagnostic Center dials up the customer site and logs onto the system. Unauthorized remote diagnosis access by the DDC is impossible because a customer's system will not answer the remote diagnosis line unless the local system operator has performed specific procedures. Then connection can be achieved only through a complex communication protocol in the DDC host computer.

When the connection is established, the response technician directs the host computer to merge the customer's description of the problem with a file containing installation-specific configuration and historical data (i.e., memory size, peripherals, options, previous system failures, etc.). This combined information is used to generate a script, or control file, of relevant tests. For example, if a customer described a problem with an RP06 disk drive, the script generated would include tests appropriate to the RP06 and I/O activities.
Reliability, Availability and Maintainability Features

Next, this script is broken down into a more specific list of tests, which is further subdivided into a sequence of actual diagnostic procedures. These diagnostics can then be executed either manually or automatically from the host computer. As mentioned previously, in some cases complete diagnosis can be accomplished without bringing the system down. When diagnosis is complete, the DDC notifies the customer and the appropriate branch office, which then dispatches a Field Service engineer to the site to make repairs.

NOTE
Remote diagnosis requires the installation of a remote diagnosis module and phone line; therefore remote diagnosis cannot be performed without the cooperation of the customer.

System verification — For system verification, VAX/VMS implements an automated, reusable collection of test software called the User Environment Test Package (UETP). These tests are designed, not as a replacement for any other kind of test, but as a means of validating the installation and proper functioning of any VAX/VMS system. The UETP is not a hardware or software diagnostic; it ensures that hardware and software are working properly together. It assumes that hardware diagnostics have already been run to verify the stand-alone operation of the various hardware components, and it assumes that the functional integrity of the software has been validated. The UETP provides a comprehensive and systematic test of the major system components (both hardware and software) by running customer-type routines (compilations and assemblies) and comparing the results to known answers. The program is non-interactive once called and reports errors to the error log and to the console terminal as execution proceeds. These tests are used to demonstrate correct operation after installation and can be run by the user anytime to check system functioning, perform preliminary diagnosis, or to serve as a demonstration of system capabilities. This User Environment Test Package contributes significantly to overall maintainability.

VAX/VMS on-line diagnostics — One of the most impressive features of the VAX-11/750 system is the ability to run functional diagnostics on-line under VAX/VMS. This capability means that many problems can be isolated without the time-consuming task of taking the system down. Furthermore, a special system diagnostic even lets users run specific I/O device, CPU, and bus functional diagnostics concurrently with user processes, or to run several VAX/VMS diagnostics concurrently. In addition to the system diagnostics, the following specific functional diagnostics are implemented:
Reliability, Availability and Maintainability Features

- Line printer (LP11)
- Card reader (CR11)
- Synchronous link (DMC, DUP)
- Terminal (one at a time)
- Terminal exerciser (multiple terminals concurrently)
- Bus interaction
- Tape reliability (TM03/TE16/TU45)
- Disk reliability (RP0x/RK0x/RM03)
- Disk formatter
- User mode cluster exerciser (a CPU test which runs in user mode and executes all instructions except privileged instructions)

NOTE
The last four diagnostics listed may also be run in a stand-alone environment.

Fault-isolation diagnostics — Once macro level diagnostics have isolated faults to a particular subsystem or device, fault-isolation diagnostics may be run to pinpoint the problem to the smallest possible element. These diagnostics may be run at the site or via a remote diagnostic port if the system has the optional remote diagnostic module (RDM). Fault-isolation diagnostics, which run in a stand-alone environment, exercise the hardware and are capable of isolating problems to a field-replaceable unit—in some cases to the board level, in a small percent of the time to the chip level. This means that Field Service spends less time diagnosing the fault. It also means that the cost of replacement parts is kept to a minimum by isolating the fault to the smallest possible unit. In addition, with the optional RDM, the user can run microdiagnostics even when the VAX-11/750 CPU is completely down. The RDM microprocessor has its own visibility bus which allows it to exercise the hardware to pinpoint faults. This also greatly reduces repair time, contributing to both maintainability and availability.

Microverify routines — For greater system reliability, additional tests, known as microverify routines, check specific hardware elements on a power-up sequence. These routines are not designed as a major stand-alone diagnostic tool; rather, they are intended to test the internal CPU data paths, all 16 general purpose registers, most internal CPU registers, and the instruction prefetch buffer. In addition, micro-verify routines test the parity logic of the cache and the translation buffer and test all of the cache memory. Single character codes sent to
the console indicate either error conditions or successful completion of the microverifies. The routines are executed automatically on a power-up sequence, when the system is booted, or when the console front panel RESET switch is pushed. When the microverify routines execute successfully, the user can be confident that the system will also boot successfully.

Software Features

Operating system consistency checks — Internal consistency checks increase system reliability by helping the VAX/VMS operating system determine when data are not valid. In particular, consistency checks are performed to determine the validity of system control information. System malfunctions will cause a trap to an exception handling routine, with appropriate information recorded in the error log file.

Critical data structure information — For greater reliability, critical information, such as the home block and index file header disk information, is redundantly recorded to permit recovery in case of accidental destruction.

Uniform exception handling — A uniform condition handling facility is used in VAX/VMS to signal both hardware and software exceptions. All of the exception conditions cause vectored traps. Because the exception handling facility is uniform, the VAX system design is more reliable.

Automatic on-line error logging — Error logging is a software tool used to monitor error occurrences, and is an integral part of every VAX/VMS operating system. The VAX error logging process is continual. The operating system accepts signals from the hardware and records CPU, memory, I/O and software errors in an on-line log file. In addition, the error logger notes as much information as possible about the state of the system at the time of an error. If no errors occur over a period of time, the error logger notes the time of year in the log file. In the special case of ECC memory errors, there is a threshold value and if the error rate exceeds this threshold, no more ECC log entries are made for a period of time. A special utility program is available to convert the log file into a meaningful format which can be printed for later study.

Error logging is useful for efficient maintenance of the hardware. It provides field service engineers with a report which can help them diagnose impending or persistent hardware problems, and it assists them in detecting trends in fault occurrences. Another benefit of this error logging feature is that certain types of intermittent failures can be detected that would not be apparent without it. This is a significant
Reliability, Availability and Maintainability Features

help to Field Service and increases both the availability and maintainability of the system. Error logging also helps Field Service pinpoint specific subsystems which need to be examined using diagnostics.

**Unattended automatic system restart** — Automatic system restart facilities can bring the system on-line, without operator intervention, after a system failure. The system operator can select automatic restart by setting the front panel POWER ON ACTION switch to the BOOT/RESTART position. The system can then restart itself after a failure caused by power interruption. If battery back-up was installed, a bit in a special memory configuration register indicates to the recovery microcode whether data in memory were lost. Thus, the VAX-11/750 can be used in unattended environments, and the automatic restart capability can substantially reduce the cost of operation.

After a power failure, all possible input and output activity in progress before the failure is restarted—a feature which helps guarantee system data integrity. However, it should be noted that the power-fail asynchronous trap facility may be used to initiate user-specific power failure recovery processing. If battery back-up was not installed, or if the power outage exceeded the battery back-up time, the contents of memory are not valid and the system attempts to reboot itself. If the system crashes because of a machine check hardware malfunction (time-out on bus, uncorrected bus or memory error) or a fatal software error, the system attempts to reboot itself, aborting all previously running processes. Following a system crash or a power failure (with successful battery back-up), the latest error log entries are written to the error log file.

**Reconfiguration** — The VAX/VMS operating system allows users to operate the system even though some of its hardware components have failed. Modification of the system configuration, both manually and automatically at system start-up time, provides a reliable subset of the system which can be used until maintenance is performed on the failed components. This means that under certain failure conditions, the VAX-11/750 can continue to operate (with some performance degradation) once reconfigured through the operator console. For example:

- If the usual system disk is unavailable, the system can be bootstrapped from any other disk, if provisions have been made ahead of time.
- If memory units are defective, memory is configured so that defective modules are not referenced (i.e., bad page mapping). Faulty memory units can be disabled or removed, but the user must
physically reconfigure the remaining units (see Chapter 8, Main Memory Subsystem). Physical address 000 is not required for bootstrapping the system; however, 64 Kb of contiguous good memory are required. When running VAX/VMS on-line diagnostics, physical address 000 is required.

- If the memory cache is down for some reason, it may be disabled since the cache is transparent to the software. The system will continue to operate properly with a certain degradation in performance.
- The VAX/VMS operating system automatically determines the presence of peripherals on the processor at bootstrap time and flags nonresponding devices as unavailable.
- Software spooling allows output to be generated even if the usual output device is not available. The system operator can reroute data to an alternate device using system operating commands.

**On-line software update and maintenance facilities** — The system operator can perform software update and maintenance activities without bringing the system down for stand-alone use. Software updates are distributed in machine readable form on a TU58 tape cartridge. Software modules on disk can be updated with patches and replacement modules, concurrent with normal system activities; however, a reboot may be necessary to activate newly installed modules.

Software maintenance procedures may also be performed on-line. The operator could perform a disk back-up procedure concurrent with normal activities. Because these activities are performed on-line, both system maintainability and availability are significantly increased.
PART 3

VAX-11/780
INTRODUCTION

The Console Subsystem serves as the interface between the operator and the VAX-11/780 system. The console subsystem provides the user with improved system maintenance features and greater operating system flexibility. The user interface to the subsystem is via the console command language, which is quite similar to the system command language. The traditional lights and toggle switch functions have been replaced by simple English language commands entered into the system terminal. The system terminal (OPA0) is the logical first terminal of the system. The floppy disk, an integral part of the subsystem, stores microdiagnostics and system software. This facilitates fast diagnosis (initiated both locally and remotely), simplified system bootstrapping and initialization, and improved software update distribution. Figure 14-1 functionally illustrates the console subsystem.

Figure 14-1  Console Subsystem
Console Subsystem

The console subsystem is comprised of six major components:

- An LSI-11 microprocessor (KD11-F) including a 4K by 16-bit semiconductor random access memory (RAM).
- A floppy disk drive (RX01) and controller (RXV11).
- A system terminal and two serial line interface units, one serial line unit provided for optional remote diagnosis port.
- Console interface board (CIB) including 4K by 16-bit read only memory (ROM) for the LSI-11 microprocessor.
- The control panel on the VAX-11/780 cabinet.
- Bus structure. The internal data (ID) bus is a high speed data path connecting major functional areas of the VAX-11/780 CPU.

CONSOLE INTERFACE BOARD

The Console Interface Board links the console subsystem to the VAX-11/780 central processor. The CIB contains interfaces to the console subsystem bus structures; registers accessible to each bus; and all the hardware necessary to implement the console functions. In addition, the CIB contains a 4K by 16-bit ROM which provides the core of the console LSI-11 software.

All data transfer operations between the VAX-11/780 processor and the console LSI-11 are routed via the TO Internal Data and FM Internal Data privileged registers on the CIB. The interaction of the console subsystem and the VAX-11/780 processor, however, is directly related to the states of the two processors. The VAX-11/780 processor may be either running or halted. When running, the VAX processor is executing normal VAX code. The processor can then be halted in one of two ways:

- Internal system error
- Halt command via console (console must be in console command mode to activate halt command)

If the processor is halted via an error detection, the console subsystem automatically enters the console command mode (e.g., CPU double-error halt).

The LSI-11 may perform in either the program I/O mode or the console command mode. When the LSI-11 is in the program I/O mode, it passes console terminal input character by character to the VAX-11/780 software. Data sent from the VAX-11/780 software to the console terminal is passed by the LSI-11 software directly to the terminal. When the LSI-11 is in the console command mode, it interprets all console terminal output in order to perform diagnostic and mainte-
nance functions and to implement the console command language (CCL). Therefore, four possible system states could exist. They are:

- VAX-11/780 running—LSI-11 program I/O mode
- VAX-11/780 running—LSI-11 console command mode
- VAX-11/780 halted—LSI-11 program I/O mode
- VAX-11/780 halted—LSI-11 console command mode

Figure 14-2 illustrates the VAX-11/780 and LSI-11 interaction and operating mode combinations.

![Diagram](image)

Figure 14-2 VAX-11/780 and LSI-11 Interaction and System Operating States

**VAX-11/780 Running — LSI-11 in Program I/O**
In this mode of operation, the console terminal acts like any other user terminal and may be used in conjunction with normal user application programming. The Console Interface Board (CIB) passes character data between both processors. In this mode, the LSI-11 console software does not interpret commands typed at the console terminal.

**VAX-11/780 Running — LSI-11 in Console Command Mode**
In this mode, the operator is able to halt the VAX-11/780 processor via the console terminal by typing the HALT command, and resume execution of the processor by entering the CONTINUE command. However, by entering the CONTINUE command, the console is automatically updated to program I/O mode. When the VAX-11/780 is
executing instructions and the LSI-11 is in the program I/O mode, to halt the VAX processor, the operator must change console modes

- SHOW
- SET
- WAIT DONE
- HELP
- EXAMINE /VBUS
- CLEAR
- HALT

Note that the functions which may be performed by the console are limited to those requiring no direct response by the VAX-11/780 processor (except HALT). The console software does not pass commands to the executing VAX processor software. Conversely, the console will not accept output from the executing software of the VAX-11/780 processor. Therefore, the VAX-11/780 software cannot communicate with the console floppy disk or console terminal.

VAX-11/780 Halted — LSI-11 in Program I/O Mode
This mode of operation contains no system functionality and should not be utilized.

VAX-11/780 Halted — LSI-11 in Console Command Mode
In this mode, the full functionality of the console command set is available to the system operator. Through the use of the console command language, the system operator has the capability to:

- Initiate and terminate software being executed by the VAX-11/780 processor.
- Display and modify memory elements including main memory, I/O, general register and process register address space.
- Control the processor clock to provide single step clock modes for use in basic hardware or program development.
- Initiate macro and micro diagnostics.

For further information regarding the console language, a complete listing of the console commands is included in this chapter.

CONSOLE BUS STRUCTURE
Communication between the elements of the console is achieved by three separate bus configurations. The ID (Internal Data) Bus links together the major functional areas of the central processor. The Q
bus interfaces the LSI-11 microprocessor and its peripheral hardware to the VAX CPU via the CIB (Console Interface Board). The V bus is utilized by the console, while the LSI-11 is in the console I/O mode, to access the Central Processor’s major buses and key control points.

**Internal Data Bus**
The Internal Data Bus is a high speed data path between the major functional areas of the CPU. The ID bus may be controlled from the console interface logic in a maintenance mode operation. This allows access to writable control store and internal registers from the console.

When the Console Interface Board generates the ID MAINT signal, it initiates a maintenance operation, allowing the console to assert ID bus address and control signals (and data, if appropriate). The ID Bus Registers are described in Appendix E.

**Q Bus**
The Q bus (LSI-11 bus) connects the LSI-11 processor (and its ROM and RAM memories), the console terminal interfaces, and the floppy disk interface to the Console Interface Board, and thus to the VAX CPU. The 16 address signals and 16 data signals share the same bus lines. Fourteen other LSI-11 signal lines are used in the VAX-11/780 configuration for control signals (note that the DMA control lines are not used).

Note that the serial line interface and the floppy disk interface cannot communicate directly with the Console Interface Board, nor can the CIB communicate directly with them. All transfers initiated from the interfaces begin with interrupts to the LSI-11 processor.

**V Bus**
The V bus consists of eight serial data lines, a load signal line, a clock signal line, and a self test line. Each of the participating VAX CPU modules contains a V bus shift register. The data input lines to the shift register monitor specific test points on the CPU module, as shown in Figure 14-3. The LOAD signal causes the shift register to parallel load from the test points when the VAX CPU is in a stable condition. The clock signal can then be used to read the latched data serially from each of the shift registers into a register on the CIB. The LSI-11 must read the register before clocking in the next serial bit from each of the shift registers.

**CONSOLE/VAX-11 INTERACTION**
All data transfer operations between the VAX CPU and the console LSI-11 are routed via the TO and FM ID Registers on the CIB. The LSI-
Console Subsystem

11 Console may look at various points in the VAX CPU via the V Bus or it may look at data on the ID Bus. The TO ID Register is a data buffer, serving two functions. First, it may be loaded by the LSI-11 with data from the console terminal, one ASCII character to be read by the VAX-11 microcode. The low order eight bits of the TO ID register contain the ASCII character (RXDB <7:0>). Bits <11:8> specify the console unit at which the data originated. Logical unit 00 is reserved for the operator terminal. Second, the LSI-11 may write to any ID bus address through the TO ID register by executing an ID maintenance cycle.

The terms TO and FR (FROM) are used with respect to the VAX-11 CPU.

The FM ID Register is also a data buffer, serving a dual function. First, it may be loaded by the VAX-11 microcode with data to be passed to the console subsystem. The low order eight bits of the FM ID register contain the ASCII character to be passed to the LSI-11. Bits <11:8> specify one of the logic units in the console subsystem. Second, the LSI-11 may read any ID bus register through the FM ID register by executing an ID maintenance cycle when the VAX CPU is halted.

The TO and FM internal data registers are illustrated in Figure 14-4.

Figure 14-3  V Bus Block Diagram

Figure 14-4  TO and FM ID Registers
Console Subsystem

READ ONLY MEMORY (ROM)
The Console Interface Board contains 4K words of ROM. This ROM contains the core of the LSI-11 console operating system, including the power up routines, the terminal and the floppy drivers. The LSI-11 begins executing instructions in the ROM when power is applied to the system.

THE CONSOLE COMMAND LANGUAGE
The console command language commands are listed and described below in alphabetical order.

SYNTAX: ALL COMMANDS ARE TERMINATED BY CARRIAGE RETURN.

'EXAMINE' AND 'DEPOSIT' <QUALIFIERS> SWITCHES FOR ADDRESS SPACE:

'P' = PHYSICAL MEMORY (THE DEFAULT)
'V' = VIRTUAL MEMORY
'I' = INTERNAL (PROCESSOR) REGISTERS
'G' = GENERAL REGISTERS 0 THRU F (R0 THRU PC)
'VR' = VBUS REGISTERS
'ID' = IDBUS REGISTERS

'EXAMINE' AND 'DEPOSIT' <QUALIFIERS> SWITCHES FOR DATA-LENGTH:

'B' = BYTE (8 BITS)
'W' = WORD (2 BYTES)
'L' = LONGWORD (2 WORDS)
'Q' = QUADWORD (4 WORDS)

<ADDR> IS A <NUMBER>, OR ONE OF THE FOLLOWING SYMBOLIC ADDRESSES

'R0,R1,R2,......,R11,AP,FP,SP,PC' (GENERAL REGISTERS)
'PSL' = PROCESSOR STATUS WORD
'*' = LAST ADDRESS
'+' = ADDRESS FOLLOWING 'LAST' ADDRESS
'-' = ADDRESS PRECEDING 'LAST' ADDRESS
'@' = USES LAST EXAMINE/DEPOSIT DATA FOR ADDRESS

<number> = STRING OF DIGITS IN CURRENT DEFAULT RADIX,
Console Subsystem

STRING OF DIGITS PREFIXED WITH A DEFAULT RADIX OVERRIDE (%O FOR OCTAL, %X FOR HEX).

'BOOT' - BOOTS THE CPU FROM DEFAULT DEVICE

'BOOT <DEVNAM>' - TAKES THE FIRST THREE ALPHANUMERIC CHARACTERS OF <DEVNAM>, AND EXECUTES THE INDIRECT FILE '<DEVNAM>BOO.CMD'

'CLEAR SOMM' - CLEAR 'STOP ON MICRO-MATCH' ENABLE. NOTE: ID REGISTER 21 IS THE MICRO-MATCH REGISTER

'CLEAR STEP' - ENABLE NORMAL (NO STEP) MODE

'CONTINUE' - ISSUES A CONTINUE TO THE ISP

'DEPosit [/<SWITCH(ES)>] <ADDR> <DATA>' - DEPOSIT <DATA> TO <ADDR>

'DIAGNOSE' - BOOTS THE DIAGNOSTIC SUPERVISOR FROM DEFAULT DEVICE

'DIAGNOSE <DEVNAM>' - TAKES THE FIRST THREE ALPHANUMERIC CHARACTERS OF <DEVNAM>, AND EXECUTES THE INDIRECT FILE '<DEVNAM>SUP.CMD'

'ENABLE DX1.' - ENABLES CONSOLE SOFTWARE TO ACCESS FLOPPY DRIVE 1 ON THOSE SYSTEMS WITH DUAL FLOPPIES

'EXAMINE [/<SWITCH(ES)>] <ADDR>' - DISPLAY CONTENTS OF <ADDR>

'EXAMINE IR' - EXAMINES INSTRUCTION REG (IR), DISPLAYS OP-CODE, SPECIFIER, & EXECUTION POINT COUNTER

'HALT' - HALTS THE CPU

'HELP' - PRINTS THIS FILE

'INITIALIZE' - INITIALIZES THE CPU

'LINK' - CAUSES CONSOLE TO BEGIN COMMAND LINKING. CONSOLE PRINTS REVERSED PROMPT TO INDICATE LINK-
Console Subsystem

ING. ALL COMMANDS TYPED BY USER WHILE LINKING ARE STORED IN AN INDIRECT COMMAND FILE FOR LATER EXECUTION. TYPING CONTROL C TERMINATES LINKING. (ALSO REFERENCE THE PERFORM COMMAND)

'LOAD[/START: <ADDR>]
  <FILENAME>']
  -LOAD FILE TO MAIN MEMORY, STARTING AT ADDRESS 0, OR DDR> IF SPECIFIED

'LOAD/WCS
  <FILENAME>']
  -LOAD FILE SPECIFIED TO WCS

'NEXT <NUMBER>']
  -<NUMBER> STEP CYCLES ARE DONE, TYPE OF STEP DEPENDS ON LAST 'SET STEP' COMMAND

'PERFORM'
  -EXECUTE A FILE OF LINKED COMMANDS PREVIOUSLY GENERATED VIA A 'LINK' COMMAND.

'QCLEAR
  <ADDRESS>']
  -DOES A QUAD CLEAR TO <ADDRESS>, WHICH IS FORCED TO A QUAD WORD BOUNDARY (CLEARING ECC ERRORS)

'REBOOT'
  -CAUSES A CONSOLE SOFTWARE RELOAD

'REPEAT<ANY-
  CONSOLE-
  COMMAND>']
  -CAUSES THE CONSOLE TO REPEATEDLY EXECUTE THE <CONSOLE-COMMAND>, UNTIL STOPPED BY A (CONTROL C) ↑C

'SET CLOCK SLOW'
  -SET CPU CLOCK FREQ TO SLOW

'SET CLOCK FAST'
  -SET CPU CLOCK FREQ TO FAST

'SET CLOCK NORMAL'
  -SET CPU CLOCK FREQ TO NORMAL

'SET DEFAULT
  <OPTION>
  [,...,<OPTION>]' 
  -SET CONSOLE DEFAULTS. NOTE: <OPTIONS> ARE: OCTAL, HEX, PHYSICAL, VIRTUAL, INTERNAL GENERAL, VBUS, IDBUS, BYTE, WORD, LONG, QUAD

'SET RELOCATION:
  <NUMBER>']
  -PUT <NUMBER> INTO CONSOLE RELOCATION REGISTER. RELOCATION REGISTER IS ADDED TO EFFECTIVE ADDRESS OF PHYSICAL AND VIRTUAL EXAMINES AND DEPOSITS
Console Subsystem

'SET SOMM' -SET 'STOP ON MICRO-MATCH' ENABLE

'SET STEP BUS' -ENABLE SINGLE BUS CYCLE CLOCK MODE

'SET STEP INSTRUCTION' -ENABLES SINGLE INSTRUCTION MODE

'SET STEP STATE' -ENABLE SINGLE TIME STATE CLOCK MODE

'SET TERMINAL FILL: <NUMBER>' -SET FILL COUNT FOR # OF BLANKS WRITTEN TO THE TERMINAL AFTER <CRLF>

'SET TERMINAL PROGRAM' -PUT CONSOLE TERMINAL INTO 'PROGRAM I/O MODE'

'SHOW' -SHOWS CONSOLE AND CPU STATE

'SHOW VERSION' -SHOWS VERSIONS OF MICROCODE AND CONSOLE

'START <ADDRESS>' -INITIALIZES THE CPU, DEPOSITS <ADDRESS> TO THE PC, ISSUES A CONTINUE TO THE ISP

'TEST' -RUNS MICRO-DIAGNOSTICS

'TEST/COM' -LOADS MICRO-DIAGNOSTICS, AWAITS COMMANDS

'UNJAM' -UNJAMS THE SBI

'WCS' -CALLS MICRODEBUGGER. (FOR DEBUGGER HELP, TYPE '@WCSMON.HLP')

'WAIT DONE' -WHEN EXECUTED FROM AN INDIRECT COMMAND FILE, THIS COMMAND WILL CAUSE COMMAND FILE EXECUTION TO STOP UNTIL: A) A 'DONE' SIGNAL IS RECEIVED FROM THE PROGRAM RUNNING IN THE VAX-11/780 (COMMAND FILE EXECUTION WILL CONTINUE), OR B) THE VAX-11/780 HALTS, OR OPERATOR TYPES A 'C (COMMAND FILE EXECUTION WILL TERMINATE)

'P'(CONTROL-P) -PUT CONSOLE TERMINAL INTO 'CONSOLE I/O' MODE

(UNLESS MODE SWITCH IN 'DISABLE')

248
Console Subsystem

'@@<FILENAME>'' -PROCESS AN INDIRECT COMMAND FILE

CONSOLE ERROR MESSAGES
This section lists all console error messages and defines their format and meaning. All console error messages are prefixed by a question mark, to distinguish them from informational messages. Where user interaction is required, the necessary steps appear in parentheses following the respective error description.

Syntactic Errors
?‘<TEXT-STRING>’ IS INCOMPLETE
The <TEXT-STRING> is not a complete console command.

?‘<TEXT-STRING>’ IS INCORRECT
The <TEXT-STRING> is not recognized as a valid command.

? FILE NAME ERR
A <FILENAME> given with a command cannot be translated to RAD50. (<FILENAME> is invalid)

?IND-COM ERR
The console detected an error in the format of an indirect command file. Possible errors are:
1) More than 80 characters in an indirect command line or

2) An indirect command line did not end with a CARRIAGE-RETURN and LINE FEED.

Command Generated Errors
?FILE NOT FOUND
A <FILENAME> given with a ‘LOAD’ or ‘@’ command does not match any file on the currently loaded floppy disk. This error can also be generated by a ‘HELP’, ‘BOOT’ or an attempted WCS load if HELP FILE, BOOT FILE or WCS FILE is missing from Floppy.

?NO CPU RESPONSE
The console timed out waiting for a response from the CPU. (Retry, indicates possible CPU-related hardware fault)

?CPU NOT IN CONSOLE WAIT LOOP,COMMAND ABORTED
A console command requiring assistance from the CPU was issued while the CPU was not in the console service loop. (HALT CPU, re-issue command)

?CPU CLOCK
A console command that requires the CPU
STOPPED, COMMAND ABORTED clock to be running was issued with the clock stopped. (Clear step mode; re-issue command)

CANT DISABLE BOTH FLOPPY's, FUNCTION ABORTED An attempt was made to disable both the remote and local floppy.

Micro-Routine Errors
The console uses various micro-code routines in the CPU’s control store to perform console functions. The following errors are generated by micro-routine failures:

?MIC-ERR ON FUNCTION A micro-error occurred in the CPU while servicing a console request. SBI error registers are dumped after this message is printed. (Action dependent upon error)

?INT-REG ERR A micro-error occurred while attempting to reference a CPU internal (processor) register. An illegal address will cause this error.

?MICRO-ERROR, CODE=X An unrecognized micro-error occurred. The code returned by the CPU is not in the range of recognized error codes. ‘X’ is the code returned by the CPU.

?MEM-MAN FAULT, CODE=XX A virtual examine or deposit caused an error in the memory management micro-routine. ‘XX’ is a one byte error code returned by the routine, with the following bit assignments:

Bit 0 = Length violation (bits numbered from right)

Bit 1 = Fault was on a PTE reference

Bit 2 = Write or modify intent

Bit 3 = Access violation

Bits 4 through 7 should be ignored

CPU Fault Generated Error Messages

?INT-STACK INVALID The CPU halted because the interrupt stack was marked invalid.

?CPU DOUBLE-ERR HALT A machine check occurred before a previous machine check had been handled,
causing the CPU to execute a ‘Double Error’ Halt. (Examine ID Registers 30-3F (hex); contents will aid in locating cause of machine check).

?ILL I/E VECTOR The CPU detected an illegal Interrupt/Exception vector.

?NO USR WCS CPU detected an Interrupt/Exception vector to user WCS and no user WCS exists.

?CHM ERR A change mode instruction was attempted from the interrupt stack.

INT PENDING This is not actually an error, but indicates that an error was pending at the time that a console-requested halt was performed. (Continue CPU to clear interrupt).

?MICRO-MACHINE TIME OUT Indicates that the VAX-11/780 micro-machine has failed to strobe interrupts within the max time period allowed.

Messages Generated by Floppy Errors

?FLOPPY ERROR, CODE = X The console Floppy driver detected an error. Codes are as follows: (Codes always printed in HEX Radix).

CODE 1-Floppy hardware error. (CRC, Parity, etc.)

CODE 2-File not found.

CODE 3-Floppy driver queue overfull.

CODE 4-Console software requested an illegal sector number.

?FLOPPY NOT READY The console floppy drive failed to become ready when booting. (Retry)

?NO BOOT ON FLOPPY Console attempted to boot from a floppy that does not contain a valid boot block. (Change floppy disk)

?FLOPPY ERROR ON BOOT A floppy error was detected while attempting a console boot. (Retry)

Messages Related to Version Compatibility

?WARNING-WCS & FPLA VER MISMATCH The microcode in WCS is not compatible with FPLA. This message is printed on each
Console Subsystem

ISP START or CONTINUE, but no other action taken by console.

?FATAL-WCS & PCS VER MISMATCH
The microcode in PCS is not compatible with that in WCS. ISP START and CONTINUE are disabled by console.

?REMOTE ACCESS NOT SUPPORTED
Printed when console mode switch enters a 'REMOTE' position, and the remote support software routines are not included in the console.

Console Generated Errors

?TRAP-4, RESTARTING CONSOLE
The console took a time-out trap. Console will restart.

?UNEXPECTED TRAP MOUNT CONSOLE FLOPPY, THEN TYPE ↑C
Console trapped to an unused vector. Console reboots when ↑C typed.

?Q-BLKD
Console's terminal output queue is blocked. Console will reboot.
CHAPTER 15
CENTRAL PROCESSOR

INTRODUCTION
The VAX-11/780 Central Processing Unit (CPU) is the hardware responsible for performing the logic and arithmetic operations requested of the computer system. The processor is a high-performance, microprogrammed computer that executes a large set of variable-length instructions in native mode, and non-privileged PDP-11 instructions in compatibility mode.

The CPU maintains 32-bit addressing and data capability, thereby allowing it direct access to four billion bytes of virtual address space ($2^{32}$). That is, the CPU references a location in terms of a 32-bit virtual address. This address is termed virtual because it is not the actual address in physical memory. The processor's memory management hardware translates a virtual address to a physical address under operating system control.

The processor provides 16 32-bit registers that can be used for temporary storage, as accumulators, index registers, and base registers. Four of these registers have special significance: the Program Counter, and three registers that are used to provide an extensive CALL facility. The processor offers a variety of addressing modes that use the general registers to identify instruction operand locations, including an indexed addressing mode that provides true post-indexing capability.

The native instruction set is highly bit efficient. It includes integral decimal, character string, and floating point instructions, as well as integer, logical, and bit field instructions. Instructions and data are variable length and can start at any arbitrary byte boundary or, in the case of bit fields, at any arbitrary bit in memory. Floating point instruction execution can be enhanced by an optional floating point accelerator.

The processor's instruction set is defined by the microcode loaded into the programmable read-only memory (control store).
Central Processor

The VAX-11/780 processor includes the following functional hardware components:

- 8K byte two-way set associative memory cache
- 8 byte prefetch instruction buffer
- 128 entry address translation buffer
- 12K byte writable diagnostic control store (WDCS)
- time-of-year clock
- programmable real time clock
- integral memory management
- optional floating point accelerator (FPA)
- optional 12K byte customer writable control store (WCS)

This chapter is divided into three sections. The first section discusses processor hardware, functionality and example processor operation. The second section discusses the programming characteristics of the processing system from the user's point of view. And the last section looks at the processing system, but from an operating system viewpoint.

HARDWARE ELEMENTS

The VAX-11/780 CPU is a fast, high-performance, 32-bit microprogrammed computer. The CPU derives its speed and performance from the fact that it can handle several independent functions simultaneously.

The CPU can process both 32-bit data and addresses while maintaining the ability to manipulate:

- bits (up to 32)
- bytes
- words
- longwords
- quadwords
- 32-bit floating point (single precision)
- 64-bit floating point (double precision)
- packed decimal (up to 31 digits)
- character strings (up to 64K bytes)

The following sections describe the VAX-11/780 processor hardware:

Control Store

The control store is a read-only memory containing 4K 96 bit microwords plus 3 parity bits per microword. The control store contains the program that describes the operation and sequencing of the central
processing unit. It also contains the native, compatibility, and floating point instruction sets. The control store contains a 96 bit buffer, enabling it to execute one microword while simultaneously fetching the next.

Data Paths
The data path subsystem consists of four independent and parallel sections used to process addresses and data specified by the instruction set. The arithmetic section is used to perform both arithmetic and logical operations on data and addresses. The exponent and sign section is used for fast exponent processing of floating point instructions. The data shift and rotate section packs and unpacks floating point and decimal string data. And finally, the address section calculates virtual addresses for the translation buffer.

8K Byte Two-Way Set Associative Memory Cache
The memory cache is the primary cache system for all data coming from memory, including addresses, address translations, and instructions. The memory cache is an 8K byte, two-way set associative, write-through cache.

Write-through provides reliability because the contents of main memory are updated immediately after the processor performs a write. Most write-through cache systems tie up the processor while main memory is updated. However, this processor buffers its commands to avoid waiting while main memory is updated from the cache. Therefore, while providing the reliability of a write-through cache, this system also provides much the same performance as a write-back cache.

The memory cache also reduces the average time the processor waits to receive main memory data by reading eight bytes at a time from main memory, and transferring four bytes to the CPU data paths, or instruction buffer. Since the remaining four bytes are already available, the memory cache also provides pre-fetching. The cache memory system carries byte parity for both data and addresses for increased integrity. Cache locations are allocated when data is read from memory. When both of the possible locations for a particular datum are already filled, one of the previously cached data is randomly replaced.

Address Translation Buffer
The address translation buffer is a cache of likely-to-be-used physical address translations. It significantly reduces the amount of time spent by the CPU on the repetitive task of dynamic address translation. The cache contains 128 virtual-to-physical page address translations
which are divided into equal sections: 64 system space page translations and 64 process space page translations. Each of these sections is two-way associative. There is byte parity on each entry for increased integrity.

8 Byte Prefetch Instruction Buffer
The 8 byte instruction buffer improves CPU performance by prefetching data in the instruction stream. The control logic continuously fetches data from memory to keep the 8 byte buffer full. It effectively eliminates the time spent by the CPU waiting for two memory cycles where bytes of the instruction stream cross 32-bit longword boundaries. In addition, the instruction buffer processes operand specifiers in advance of execution and subsequently routes them to the CPU.

12K Byte Writable Diagnostic Control Store (WDCS)
The writable diagnostic control store consists of 1024 96-bit (12K byte) control words plus three parity bits per control word. These locations are used to contain basic instruction microcode, diagnostic microcode, and reserved space to accommodate future additions or improvements made by DIGITAL to the instruction set.

Processor Clocks
The VAX-11/780 processor contains a programmable real-time clock and a time-of-year clock. The interval or real-time clock was designed to permit the measurement of finely resolved variable intervals which are identified by interrupts (i.e., scheduling, diagnostics, etc.). The real-time clock is based upon a crystal oscillator with an accuracy of 0.01%, and a resolution of one μsec. The time-of-year clock is used by software to perform various timekeeping functions. Its major function is to provide the correct time to the system after power failure or other system interruptions.

Optional Floating Point Accelerator
The floating point accelerator is an optional high-speed processor extension. When included in the processor, the floating point accelerator executes the addition, subtraction, multiplication, and division instructions that operate on single- and double-precision floating point operands, including the special EMOD and POLY instructions in both single- and double-precision formats. Additionally, the floating point accelerator enhances the performance of 32-bit integer multiply instructions.

The processor does not have to include the floating point accelerator to execute floating point operand instructions. The floating point accelerator can be added or removed without changing any existing software.
Central Processor

When the floating point accelerator is included in the processor, a floating point operand register-to-register add instruction takes as little as 800 nanoseconds to execute. A register-to-register multiply instruction takes as little as one μsec. The inner loop of the POLY instruction takes approximately one μsec per degree of polynomial.

Optional 12K Byte Customer Writable Control Store (WCS)
The user writable control store consists of 1024 96-bit (12K byte) control words plus three parity bits per control word. These locations are optionally available to the customer for augmenting the speed and power of the basic machine with customized functions.

Figure 15-1 illustrates the central processing unit.

PROCESSOR OPERATION
For those interested in the hardware operations and interfaces of the VAX-11/780 CPU elements, the execution of a sample piece of code is described below. A FORTRAN IV DO LOOP is first expanded into its VAX-11 MACRO equivalent, and then into VAX machine specific implementation. For the purposes of this description, virtual to physical translation values, although valid, have been assumed.

Example: FORTRAN IV DO LOOP

\[ J = 0 \]
\[ \text{Do } 100 \text{ I = 1, 10} \]
\[ 100 \quad J = J + N(I) \]

VAX MACRO EXPANSION

| 1000 | CLRL | R0 |
| 1002 | MOVL | #1, R1 |
| 1005 |      |     |
| 1$:  | ADDL2| N<R1>, R0 |
| 100B | AOBLEQ| #10, R1, 1$ |
| 1FFC |      |     |
| N:   | BLKL | 11 |

CENTRAL PROCESSOR IMPLEMENTATION

<table>
<thead>
<tr>
<th>CPU Component</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU, R</td>
<td>1000 → PC</td>
</tr>
<tr>
<td>TB</td>
<td>Translate virtual 1000 to physical 1F600</td>
</tr>
<tr>
<td>Cache</td>
<td>Does Cache presently contain address 1F600? (NO) therefore,</td>
</tr>
</tbody>
</table>
Figure 15-1  The Central Processing Unit
<table>
<thead>
<tr>
<th>CPU Component</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBI</td>
<td>Fetch 1F600-1F607 from memory</td>
</tr>
<tr>
<td>Cache</td>
<td>Store address range 1F600-1F607 and corresponding contents in Data Cache</td>
</tr>
<tr>
<td>(IB)</td>
<td>Obtain instructions from physical addresses 1F600-1F603</td>
</tr>
<tr>
<td>ALU, R</td>
<td>Ask IB for instruction -- (CLRL)</td>
</tr>
<tr>
<td>ALU, R</td>
<td>Clear R0</td>
</tr>
<tr>
<td>(IB asks Cache for more)</td>
<td>IB retrieves physical addresses 1F604-1F607 from Cache</td>
</tr>
<tr>
<td>ALU, R</td>
<td>Ask IB for next instruction -- (MOVIL)</td>
</tr>
<tr>
<td>ALU, R</td>
<td>Ask IB for destination specifier -- (R1)</td>
</tr>
<tr>
<td>(IB asks Cache for more)</td>
<td>Cache asks SBI for 1F608</td>
</tr>
<tr>
<td>SBI</td>
<td>Asks memory for physical addresses 1F608-1F60F</td>
</tr>
<tr>
<td>ALU, R</td>
<td>Store 1 in R1</td>
</tr>
<tr>
<td>ALU, R</td>
<td>Ask IB for next instruction -- (ADDL2)</td>
</tr>
<tr>
<td>ALU, R</td>
<td>Calculate base address of N (virtual 1FFC)</td>
</tr>
<tr>
<td>ALU, R</td>
<td>Adds 4*R1 to address of N to yield virtual address 2000</td>
</tr>
<tr>
<td>TB Cache</td>
<td>Look up address of N[1]: physical address A00</td>
</tr>
<tr>
<td>SBI</td>
<td>The SBI enters a wait mode because it is currently completing the fetch operation of physical addresses 1F608-1F60F</td>
</tr>
<tr>
<td>SBI</td>
<td>Finishes the prefetch operation of physical addresses 1F608-1F60F</td>
</tr>
<tr>
<td>IB</td>
<td>Grabs 1F608-1F60B</td>
</tr>
</tbody>
</table>
**Central Processor**

<table>
<thead>
<tr>
<th>CPU Component</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache</td>
<td>Gets 1F608-1F60F</td>
</tr>
<tr>
<td>SBI</td>
<td>Starts fetch of physical addresses A00-A07</td>
</tr>
<tr>
<td>Cache</td>
<td>Gets A00-A07</td>
</tr>
<tr>
<td>ALU, R</td>
<td>A00-A03</td>
</tr>
<tr>
<td>ALU, R</td>
<td>Asks IB for destination specifier (R6)</td>
</tr>
<tr>
<td>(IB asks for 1F60C)</td>
<td>Cache sends 1F60C-1F60F to IB</td>
</tr>
<tr>
<td>ALU, R</td>
<td>Add (A00-A03) (i.e., N[1]) to R0</td>
</tr>
<tr>
<td>ALU, R</td>
<td>Ask IB for instruction (AOBLEQ)</td>
</tr>
<tr>
<td>(IB asks for more data)</td>
<td>Cache asks SBI to get 1F610 from memory</td>
</tr>
<tr>
<td>ALU, R</td>
<td>Asks IB for next specifier (R1)</td>
</tr>
<tr>
<td>ALU, R</td>
<td>Add 1 to R1, compare to 10, if less than or equal to 10 then branch</td>
</tr>
<tr>
<td>ALU, R</td>
<td>Flush (clear) IB, load virtual 1005 into PC</td>
</tr>
<tr>
<td>IB</td>
<td>Fetch 1005 from cache (resumption of loop)</td>
</tr>
<tr>
<td>ALU, R</td>
<td>Ask IB for next instruction (ADDL2)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>SBI</td>
<td>Memory data (1F610) arrives</td>
</tr>
<tr>
<td>Cache</td>
<td>Takes data, but IB doesn't grab it</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

on the 11th increment,

| ALU, R        | Add 1 to R1, compare to 10, now however R1 = 11 |

262
Central Processor

<table>
<thead>
<tr>
<th>CPU Component</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>and do not branch, but fall through to the next instruction</td>
</tr>
<tr>
<td>ALU, R</td>
<td>Ask IB for next instruction</td>
</tr>
</tbody>
</table>

USER PROGRAMMING CONCEPTS
A program is a stream of instructions and data that a user can request the operating system to translate, link, and execute. An executable program is called an image. When a user runs an image, the context in which the image is executed is called a process. A process is the complete unit of execution in this computer system. Process context includes the state of the image it is currently executing and it includes the limitations on what an image is allowed to do, which primarily depend on the privileges of the user who runs the image.

Process Virtual Address Space
Most data are located in memory using the address of an 8-bit byte. The programmer uses a 32-bit virtual address to identify a byte location. A virtual address is not a unique address of a location in memory, as are physical memory addresses. Two programs using the same virtual address might refer to two different physical memory locations, or refer to the same physical memory location using different virtual addresses.

The set of all possible 32-bit virtual addresses is called virtual address space. Virtual address space (mass storage) can be viewed as an array of byte "locations" ($2^{32}$ or over four billion bytes in length). Virtual address space is divided into two halves. The set of virtual addresses designated for use by a process, including an image it executes, is one half of the total virtual address space. Addresses in the remaining half of virtual address space are used to refer to locations maintained and protected by the operating system.

Instruction Sets
The VAX-11/780 processor is capable of executing instructions in either of two modes: native or compatibility. In native mode the processor executes a large set of variable-length instructions, recognizes a variety of data types, and uses 16 32-bit general purpose registers. In compatibility mode the processor executes a set of PDP-11 instructions, recognizes integer data, and uses 8 16-bit general purpose registers. Both instruction sets are closely related and their programming characteristics are similar. Thus, a user process can execute both native mode and compatibility mode images. However, the native mode instruction set is considerably more extensive than that of com-
patibility mode execution. The native mode instruction set consists of 248 basic instructions, many of which correspond directly to high-level language statements. Additionally, the native mode instruction set includes floating point operations, character string manipulations, packed decimal arithmetic, and many instructions which improve the performance and memory utilization of systems and applications software.

A native instruction consists of an operation code (opcode) and zero or more operands, which are described by data type and addressing mode.

Data Types
The data type of an instruction operand identifies how many bits of storage are to be treated as a unit, and what the interpretation of that unit is. The processor's native instruction set recognizes four primary data types: integer, floating point, packed decimal, and character string. In addition, the processor can also manipulate a fifth data type, the variable bit field, in which the programmer defines the size of the field and its relative position. Table 15-1 illustrates the VAX-11/780 data types.

The address of any data item is the address of the first byte in which the item resides. All integer, floating point, packed decimal, and character data can be stored starting on an arbitrary byte boundary. A bit field, however, does not necessarily start on a byte boundary. A field is simply a set of contiguous bits (0-32) whose starting bit location is identified relative to a given byte address. The native instruction set can interpret a bit field as a signed or unsigned integer.

Registers and Addressing Modes
A register is a location within the processor that can be used for temporary data storage and addressing. The assembly language programmer has 16 32-bit general registers available for use with the native instruction set. Some of these registers have special significance. One register is designated as the Program Counter, and it contains the address of the next instruction to be executed. Three general registers are designated for use with routine linkages: the Stack Pointer, the Argument Pointer, and the Frame Pointer.

An instruction operand can be located in main memory, in a general register, or in the instruction stream itself. The way in which the programmer chooses to specify an operand location is called the operand addressing mode. The processor offers a variety of addressing modes and addressing mode optimizations. There is one addressing mode that locates an operand in a register. There are six addressing modes

264
### Table 15-1  Data Types

<table>
<thead>
<tr>
<th>DATA TYPE</th>
<th>SIZE</th>
<th>RANGE (decimal)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Signed</td>
<td>Unsigned</td>
</tr>
<tr>
<td>Integer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Byte</td>
<td>8 bits</td>
<td>-128 to +127</td>
<td>0 to 255</td>
</tr>
<tr>
<td>Word</td>
<td>16 bits</td>
<td>-32768 to +32767</td>
<td>0 to 65535</td>
</tr>
<tr>
<td>Longword</td>
<td>32 bits</td>
<td>-2^{31} to +2^{31}</td>
<td>0 to 2^{32} - 1</td>
</tr>
<tr>
<td>Quadword</td>
<td>64 bits</td>
<td>-2^{63} to +2^{63}</td>
<td>0 to 2^{64} - 1</td>
</tr>
</tbody>
</table>

Floating Point

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>F_floating</td>
<td>32 bits</td>
<td>approximately seven decimal digits precision (0.29 \times 10^{-38} to 1.7 \times 10^{38})</td>
<td></td>
</tr>
<tr>
<td>D_floating</td>
<td>64 bits</td>
<td>approximately sixteen decimal digits precision (same as F_floating)</td>
<td></td>
</tr>
<tr>
<td>Packed Decimal String</td>
<td>0 to 16 bytes (31 digits)</td>
<td>numeric, two digits per byte sign in low half of last byte</td>
<td></td>
</tr>
<tr>
<td>Character String</td>
<td>0 to 65535 bytes</td>
<td>one character per byte</td>
<td></td>
</tr>
<tr>
<td>Variable-length Bit Field</td>
<td>0 to 32 bits</td>
<td>dependent on interpretation</td>
<td></td>
</tr>
</tbody>
</table>
that locate an operand in memory using a register to:
• point to the operand
• point to a table of operands
• point to a table of operand addresses

There also are six addressing modes that are indexed modifications of the addressing modes that locate an operand in memory. Finally, there are two addressing modes that identify the location of the operand in the instruction stream: one for constant data, and one for branch instruction addresses.

Stacks, Subroutines, and Procedures
A stack is an array of consecutively addressed data items that are referenced on a last-in, first-out basis using a general register. Data items are added to and removed from the low address end of the stack. A stack grows toward lower addresses as items are added, and shrinks toward higher addresses as items are removed.

A stack can be created anywhere in user program address space and can utilize any register to point to the current item on the stack. The operating system, however, automatically reserves portions of each process address space for stack data structures. User software references its stack data structure, called the user stack, through a general register designated as the Stack Pointer.

A stack is a powerful data structure because it is able to pass arguments to a routine in a highly efficient manner. In particular, the stack structure enables the programmer to write reentrant routines because the processor can handle routine linkages automatically, using the Stack Pointer. Routines can also be recursive because arguments can be saved on the stack for each successive call of the same routine.

The processor provides two kinds of routine call instructions: those for subroutines, and those for procedures. In general, a subroutine is a routine entered using a Jump to Subroutine or Branch to Subroutine instruction, while a procedure is a routine entered using a Call instruction.

The processor provides more elaborate routine linkages for procedures than for subroutines. The processor automatically saves and restores the contents of registers the programmer wants preserved across procedure calls. The processor provides two methods for passing argument lists to called procedures: by passing the arguments on the stack, or by passing the address of the arguments elsewhere in memory. The processor also constructs a list or record of procedure call nesting by using a general register as a pointer to the place on the stack where a procedure has its linkage data. This record
of each procedure's stack data, known as its stack frame, enables proper returns from procedures even when a procedure leaves data on the stack. In addition, user and operating system software can use the stack frame to trace back through nested calls to handle errors or debug programs.

**Condition Codes**
A user program can test the outcome of an arithmetic or logical operation. The processor provides a set of condition codes and branch instructions for this purpose. The condition codes indicate whether the previous arithmetic or logical operation produced a negative or zero result, or whether there was a carry or borrow, or an overflow. There are a variety of branch on condition instructions: those for overflow and carry or borrow, and those for signed and unsigned relational tests.

**Exceptions**
There are some situations in which the programmer may not want to test the outcome of an operation. The processor recognizes many events for which testing is not necessary, and automatically changes normal program flow when they occur. These events, called exceptions, are the direct result of executing a specific instruction. Exceptions also include errors automatically detected by the processor, such as improperly formed instructions.

All exceptions transfer program control to the operating system software. There are essentially no fatal exceptions. All exceptions either wait for the instruction that caused them to complete before trapping or they restore the processor to the state it was in just prior to executing the instruction that caused the exception. In either case, the instruction can be retried after the cause of the exception is cleared. Depending on the exception, it may be desirable to correct the situation and continue. If not, the operating system issues an appropriate message and aborts the instruction stream in progress. To continue, the programmer can request the operating system software to start execution of a condition handler automatically when an exception occurs.

**USER PROGRAMMING ENVIRONMENT**
A process context includes the definition of the virtual address space in which it executes an image. An image executing within a process context controls its execution through the use of one of the instruction sets, the general purpose registers, and the Processor Status Word. These hardware resources are discussed in detail in the following sections.
Process Virtual Address Space Structure
The processor and operating system provide a multiprogramming environment by dividing virtual address space into two halves: one half for addressing context-dependent code and data, and one half for addressing context-independent code and data.

The first half, termed per-process space, is capable of addressing approximately two billion bytes. An image executing in the context of a process and the operating system on behalf of the process use addresses in process space to refer to code and data particular to that process context. A process cannot represent virtual addresses in any process space but its own. Thus, code and data belonging to a process are automatically protected from other processes in the system.

The second half of virtual address space is called system space. The operating system assigns the meanings to addresses in system space. The significance of any address in system space is the same for every process, independent of process context.

Per-process space is further subdivided into two regions. Addresses in the first region, called the program region, are used to identify the location of image code and data. Addresses in the second region, called the control region, are used to refer to stacks and other temporary program image and permanent process control information maintained by the operating system on behalf of the process. Program region addresses are allocated from address 0 and up, and control region addresses are allocated from address $2^{31} - 1$ and down.

System space is also subdivided into two regions. The operating system assigns the system region addresses for linkages to its service procedures, for memory management data, and for I/O processing routines. The second region is reserved for future use.

General Registers
Instruction operands are often either stored in the processor's general registers or accessed through them. The first 12 of these 32-bit programmable general registers, labelled R0 through R11 (decimal), can be used for temporary storage, accumulators, base registers, and index registers. A base register contains the address of the base of a software data structure such as a table or queue, and an index register contains a logical offset into a data structure.

Whenever a register is used to contain data, the data are stored in the register in the same format that would appear in memory. If a quad-word or double floating datum is stored in a register, it is actually stored in two adjacent registers. For example, storing a double floating number in register R7 loads both R7 and R8.
Central Processor

Some registers have special significance depending on the instruction being executed. Registers R12 through R15 have special significance for many instructions, and therefore have special labels. These special registers are:

- The Program Counter (PC or R15), which contains the address of the next byte to be processed in the instruction stream.
- The Stack Pointer (SP or R14), which contains the address of the base (or top) of a stack maintained for subroutine and procedure calls.
- The Frame Pointer (FP or R13), which contains the address of the base of a software data structure stored on the stack called the stack frame, which is maintained for procedure calls.
- The Argument Pointer (AP or R12), which contains the address of the base of a software data structure called the argument list, which is maintained for procedure calls.

A register’s special significance does not preclude its use for other purposes, except for the Program Counter. The Program Counter cannot be used as an accumulator, as a temporary register, or as an index register. In general, however, most users do not use the Stack Pointer, Argument Pointer, or Frame Pointer for purposes other than those designated.

Addressing Modes
The processor’s addressing modes allow almost any operand to be stored in a register or in memory, or as an immediate constant. There are seven basic addressing modes that use the general registers to identify the operand location. They include:

- Register mode
- Register Deferred mode
- Autodecrement mode
- Autoincrement mode
- Autoincrement Deferred mode
- Displacement mode
- Displacement Deferred mode

Of these seven basic modes, all except register mode can be modified by an index register. When an index register is used with a basic mode to identify an operand, the addressing mode is the name of the basic mode with the suffix “indexed”. For example, the indexed addressing mode for register deferred is called “register deferred indexed” mode.

Therefore, in addition to the seven basic addressing modes, the processor recognizes six indexed addressing modes.

269
The processor also provides literal mode addressing, in which an unsigned 6-bit field in the instruction is interpreted as an integer or floating point constant. Table 15-2 summarizes the VAX-11/780 addressing modes.

**Program Counter**
The program counter contains the address of the next byte to be processed in the instruction stream. That is, just before the processor begins to execute an instruction, the program counter contains the address of the first byte of the next instruction. General register 15 contains this address. The program counter update is totally transparent to the programmer.

The Program Counter itself can be used to identify operands. The assembler translates many types of operand references into addressing modes using the Program Counter. Autoincrement mode using the Program Counter, or immediate mode, is used to specify in-line constants other than those available with literal mode addressing. Autoincrement Deferred mode using the Program Counter, or absolute mode, is used to reference an absolute address. Displacement and Displacement Deferred modes using the Program Counter are used to specify an operand using an offset from the current location.

Addressing using the Program Counter enables the programmer to write position independent code. Position independent code can be executed anywhere in virtual address space after it has been linked, since program linkages can be identified as absolute locations in virtual address space and all other addresses can be identified relative to the current instruction.

**The Stack Pointer, Argument Pointer, and Frame Pointer**
The Stack Pointer is a register specifically designated for use with stack structures. To place items on a stack, the programmer can use autodecrement mode addressing through the Stack Pointer, and to remove them, use Autoincrement mode. The programmer can reference and modify the top element on a stack without removing it by using Register Deferred mode, and can reference other elements of the stack using Displacement mode addressing.

The processor designates Register 14 as the Stack Pointer for use with both the subroutine Branch or Jump instructions, and the procedure Call instructions. On routine entry, the processor automatically saves the address of the instruction that follows the routine call on the stack. It uses the Program Counter and the Stack Pointer to perform the operation. Before entering the subroutine, the Program Counter contains the address of the instruction following the Branch, Jump, or Call
Table 15-2  Addressing Modes

<table>
<thead>
<tr>
<th>Literal (Immediate)</th>
<th>Register Deferred</th>
<th>Autodecrement</th>
<th>Autoincrement Deferred (Absolute)</th>
<th>Displacement</th>
<th>Displacement Deferred</th>
</tr>
</thead>
<tbody>
<tr>
<td>S[^^]</td>
<td>R[^^]</td>
<td>(R[^^])</td>
<td>(R[^^]) +</td>
<td>(R[^^]) #address</td>
<td>displacement (R[^^])</td>
</tr>
<tr>
<td>R[^^]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[^^]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Indexed [Rx]

n = 0 through 15
x = 0 through 14
instruction. The Stack Pointer contains the address of the last item on
the stack. The processor pushes the contents of the Program Counter
on the stack. On return from a subroutine, the processor automatically
restores the Program Counter by popping the return address off the
stack.

Also for the procedure Call instructions, the processor designates
Register 12 as an Argument Pointer, and Register 13 as a Frame
Pointer. The Argument Pointer is used to pass the address of the
argument list to a called procedure, and the Frame Pointer is used to
keep track of nested Call instructions.

An argument list is a formal data structure that contains the arguments
required by the procedure being called. Arguments may be actual
values, addresses of data structures, or addresses of other pro-
cedures. Figure 15-2 illustrates the argument pointer and list. An argu-
ment list can be passed in either of two ways: by passing only its
address, or by passing the entire list on the user stack. The first
method is used to pass long argument lists, or lists that you want to
preserve. The second method is generally used when calling pro-
cedures that do not require arguments, or when building an argument
list dynamically.

![Argument Pointer and List](image)

The importance of the way the Call instructions work is that nested
calls can be traced back to any previous level. The Call instructions
always keep track of nested calls by using the Frame Pointer register.
The Frame Pointer contains the address on the stack of the items
pushed on the stack during the procedure call. The set of items
pushed on the stack during a procedure call is known as a call frame
or stack frame. Figure 15-3 illustrates the Call Frame. Since the previ-
ous contents of the Current Frame register are saved in each call
frame, the nested frames form a linked data structure which can be
unwound to any level when an error or exception condition occurs in
any procedure.
Processor Status Word
The Processor Status Word (a portion of the Processor Status Long-word) is a special processor register that a program uses to check its status and to control synchronous error conditions. The Processor Status Word, illustrated in Figure 15-4, contains two sets of bit fields:
- The condition codes
- The exception enable flags

![Figure 15-4 Processor Status Word](image)

The condition codes indicate the outcome of a particular logical or arithmetic operation. For example, the Subtract instruction sets the Negative bit if the result of the subtraction operation produced a negative number, and it sets the Zero bit if the result produced zero. The
programmer can then use the Branch on Condition instructions to transfer control to a code sequence that handles the condition.

There are two kinds of exceptions that concern the user process: trace exceptions and arithmetic exceptions. The trace fault is used for debugging programs. Arithmetic exceptions include:

- integer, floating point, or decimal string overflow, in which the result was too large to be stored in the given format
- integer, floating point, or decimal string divide by zero, in which the divisor supplied was zero
- floating point underflow, in which the result was too small to be expressed in the given format

Handling Exceptions
When an exception occurs, the processor immediately saves the current state of execution and traps to the operating system. The operating system automatically searches for a procedure that wants to handle the exception. Procedures that respond to exceptions are called condition handlers. The user can declare a condition handler for an entire image and for each individual procedure called. In addition, because the processor keeps track of nested calls using the Frame Pointer register, it is possible to declare condition handlers for procedures that call other procedures in which exceptions might occur. The operating system automatically traces back through call frames to find a condition handler that wants to handle an exception that occurs.

SYSTEM PROGRAMMING CONCEPTS
The processor is specifically designed to support a high-performance multiprogramming environment. The characteristics of the hardware system that support multiprogramming are:

- rapid context switching
- priority dispatching
- virtual addressing and memory management

As a multiprogramming system, VAX-11/780 not only provides the ability to share the processor among processes, but also protects processes from one another while enabling them to communicate with each other and to share code and data.

Context Switching
In a multiprogramming environment, several individual streams of code can be ready to execute at any one time.

To support multiprogramming for a high-performance system, the
Central Processor

processor enables the operating system to switch rapidly between individual streams of code. A process is a stream of instructions and data defined by a hardware context. Each process has a unique identification in the system. At any one time, the stream of code being executed is determined by its hardware context. Hardware context includes the information loaded in the processor's registers that identifies:

- where the stream's instructions and data are located
- which instruction to execute next
- what the processor status is during execution

The operating system switches between processes by requesting the processor to save one process hardware context and load another. Context switching occurs rapidly because the processor instruction set includes save hardware context and load hardware context instructions. The operating system's context switching software does not have to individually save or load the processor registers which define the hardware context.

Priority Dispatching
To share processor, memory, and peripheral resources among many processes, the processor provides two arbitration mechanisms that support high-performance multiprogramming: exceptions and interrupts. Exceptions are events that occur synchronously with respect to instruction execution, while interrupts are external events that occur asynchronously.

The flow of execution can change at any time, and the processor distinguishes between changes in flow that are local to a process and those that are system-wide. Process-local changes occur as the result of a user software error or when user software calls operating system services. Process-local changes in program flow are handled through the processor's exception detection mechanism and the operating system's exception dispatcher.

System-wide changes in flow generally occur as the result of interrupts from devices or interrupts generated by the operating system software. Interrupts are handled by the processor's interrupt detection mechanism and the operating system's interrupt service routines.

System-wide changes in flow take priority over process-local changes in flow. Furthermore, the processor uses a priority system for servicing interrupts. To arbitrate between all possible interrupts, each kind of interrupt is assigned a priority, and the processor responds to the highest priority interrupt that is pending.
Central Processor

The processor services interrupts between instructions, or at well-defined points during the execution of long, iterative instructions. When the processor acknowledges an interrupt, it switches rapidly to a special system-wide context to enable the operating system to service the interrupt. System-wide changes in the flow of execution are totally transparent to individual processes.

Virtual Addressing and Virtual Memory
The processor’s memory management hardware enables the operating system to provide an environment that allows users to write programs without having to know where the programs are loaded in physical memory, and to write programs that are too large to fit in the physical memory allocated.

The processor provides the operating system with the ability to provide virtual addressing. A virtual address is a 32-bit integer that a program uses to identify storage locations in virtual memory. Virtual memory is the set of all physical memory locations in the system plus the set of disk blocks that the operating system designates as extensions to physical memory.

A physical address is an address that the processor uses to identify physical memory storage locations and peripheral controller registers. It is the address that the processor sends out on the SBI bus to which the memory and peripheral adapters respond.

The processor must be able to translate the virtual addresses provided by the programs it executes into the physical addresses recognized by the memory and peripherals. To provide virtual to physical address mapping, the processor has address mapping registers controlled by the operating system and an integrated address translation buffer.

The mapping registers enable the operating system to relocate programs in physical memory, to protect programs from each other, and to share instructions and data between programs transparently or at their request. The address translation buffer ensures that the virtual address to physical address translation takes place rapidly.

SYSTEM PROGRAMMING ENVIRONMENT
Within the context of one process, user-level software controls its execution using the instruction sets, the general registers, and the Processor Status Word. Within the multiprogramming environment, the operating system controls the system’s execution using a set of special instructions, the Processor Status Longword, and the internal processor registers.
Processor Status Longword
A processor register called the Processor Status Longword (PSL) determines the execution state of the processor at any time. The low-order 16 bits of the Processor Status Longword is the Processor Status Word available to the user process. The high-order 16 bits provide privileged control of the system. Figure 15-5 illustrates the Processor Status Longword.

The fields can be grouped together by functions that control:
- The instruction set the processor is executing
- The access mode of the current instruction
- Interrupt processing

![Diagram of Processor Status Word](image)

**Figure 15-5** Processor Status Longword

The instruction set the processor executes is controlled by the compatibility mode bit in the Processor Status Longword. This bit is normally set or cleared by the operating system. The initial environment is established by the operating system but any process, including user, can change it. In particular, compatibility mode processes switch to native mode with EMTs and native processes can perform an REI instruction to get into compatibility mode.

Processor Access Modes
In a high-performance multiprogramming system, the processor must provide the basis for protection and sharing among the processes competing for the system's resources. The basis for protection in this system is the processor's access mode. The access mode in which the processor executes determines:
- instruction execution privileges: what instructions the processor will execute
- memory access privileges: which locations in memory the current instruction can access
At any one time, the processor is executing code in the context of a particular process, or it is executing in the system-wide interrupt service context. In the context of a process, the processor recognizes four access modes: kernel, executive, supervisor, and user. Kernel is the most privileged mode and user the least privileged.

The processor spends most of its time executing in user mode in the context of one process or another. When user software needs the services of the operating system, whether for acquisition of a resource, for I/O processing, or for information, it calls those services.

The processor executes those services in the same or one of the more privileged access modes within the context of that process. That is, all four access modes exist within the same virtual address space. Each access mode has its own stack in the control region of per-process space, therefore each process has four stacks: one for each access mode. Note that this makes it easy for the operating system to context switch a process even when it is executing an operating system service procedure.

In any mode except kernel, the processor will not execute instructions that:

- halt the processor
- load and save process context
- access the internal processor registers that control memory management, interrupt processing, the processor console, or the processor clock

These instructions are privileged instructions that are generally reserved to the operating system.

In any mode, the processor will not allow the current instruction to access memory unless the mode is privileged to do so. The ability to execute code in one of the more privileged modes is granted by the system manager and controlled by the operating system. The memory protection the privilege affords is enforced by the processor. In general, code executing in one mode can protect itself and any portion of its data structures from read and/or write access by code executing in any less privileged mode. For example, code executing in executive mode can protect its data structures from code executing in supervisor or user mode. Code executing in supervisor mode can protect its data structures from access by code executing in user mode. This memory protection mechanism provides the basis for system data structure integrity.

**Protected and Privileged Instructions**
The processor provides three types of instructions that enable user
mode software to obtain operating system services without jeopardiz-
ing the integrity of the system. They include:
- the Change Mode instructions
- the PROBE instructions
- the Return from Exception or Interrupt instruction

User mode software can obtain privileged services by calling operat-
ing system service procedures with a standard CALL instruction. The
operating system’s service dispatcher issues an appropriate Change
Mode instruction before actually entering the procedure. Change
Mode allows access mode transitions to take place from one mode to
the same or more privileged mode only. When the mode transition
takes place, the previous mode is saved in the Previous Mode field of
the Processor Status Longword, allowing the more privileged code to
determine the privilege of its caller.

A Change Mode instruction is simply a special trap instruction that can
be thought of as an operating system service call instruction. User
mode software can explicitly issue Change Mode instructions, but
since the operating system receives the trap, non-privileged users
cannot write any code to execute in any of the privileged access
modes. User mode software can include a condition handler for
Change Mode to User traps, however, and this instruction is useful for
providing general purpose services for user mode software. The sys-
tem manager ultimately grants the privilege to write any code that
handles Change Mode traps to more privileged access modes.

For service procedures written to execute in privileged access modes
(kernel, executive, and supervisor), the processor provides address
access privilege validation instructions. The PROBE instructions en-
able a procedure to check the read (PROBER) and write (PROBEW)
access protection of pages in memory against the privileges of the
caller who requested access to a particular location. This enables the
operating system to provide services that execute in privileged modes
to less privileged callers and still prevent the caller from accessing
protected areas of memory.

The operating system’s privileged service procedures and interrupt
and exception service routines exit using the Return from Exception or
Interrupt (REI) instruction. REI is the only way in which the privilege of
the processor’s access mode can be decreased. Like the procedure
and subroutine return instructions, REI restores the Program Counter
and the processor state to resume the process at the point where it
was interrupted.

REI performs special services, however, that normal return instruc-
Central Processor

tions do not. For example, REI checks to see if any asynchronous system traps have been queued for the currently executing process while the interrupt or exception service routine was executing, and ensures that the process will receive them. Furthermore, REI checks to ensure that the mode to which it is returning control is the same as, or less privileged than, the mode in which the processor was executing when the exception or interrupt occurred. Thus REI is available to all software, including user-written trap handling routines, but a program cannot increase its privilege by altering the processor state to be restored.

When the operating system schedules a context switching operation, the context switching procedure uses the Save Process Context (SVPCTX) and Load Process Context (LDPCTX) instructions to save the current process context and load another. The operating system’s context switching procedure identifies the location of the hardware context to be loaded by updating an internal processor register.

Internal processor registers include not only those that identify the process currently executing, but also the memory management and other registers, such as the console and clock control registers. The Move to Processor Register (MTPR) and Move from Processor Register (MFPR) instructions are the only instructions that can explicitly access the internal processor registers. MTPR and MFPR are privileged instructions that can be issued only in kernel mode.

Memory Management

The processor is responsible for enforcing memory protection between access modes. Memory protection, however, is only a part of the processor’s memory management function. In particular, the memory management hardware enables the operating system to provide an extremely flexible and efficient virtual memory programming environment. Virtual and physical address space definitions provide the basis for the virtual memory available on a system.

Virtual address space consists of all possible 32-bit addresses that can be exchanged between a program and the processor to identify a byte location in physical memory. The memory management hardware of the VAX-11/780 translates a virtual address into a 30-bit physical address. A physical address is the address exchanged between the processor and the memory and peripheral adapters over the SBI bus. Physical address space is the set of all possible physical addresses the processor can use to express unique memory locations and peripheral control registers.

Physical address space is an array of addresses which can be used to
Central Processor

represent $2^{30}$ byte locations, or approximately one billion bytes. Half of
the addresses in physical address space can be used to refer to real
memory locations and the other half can be used to refer to peripheral
device control and data registers. The lowest addressed half of physi-
cal address space is called memory space, and the highest-addressed
half I/O space.

Chapter 4, Memory Management, describes the way in which the
memory management hardware enables the operating system to map
virtual addresses into physical addresses to provide the virtual memo-
ry available to a user process.

Virtual to Physical Page Mapping
Virtual address space is divided into pages, where a page represents
512 bytes of contiguously addressed memory. The first page begins at
byte zero and continues to byte 511. The next page begins at byte 512
and continues to byte 1023, and so forth. The first eight pages of
virtual address space, and their addresses in both decimal and hexa-
decimal are:

<table>
<thead>
<tr>
<th>PAGE</th>
<th>ADDRESS (10)</th>
<th>ADDRESS (16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000-0511</td>
<td>0000-01FF</td>
</tr>
<tr>
<td>1</td>
<td>0512-1023</td>
<td>0200-03FF</td>
</tr>
<tr>
<td>2</td>
<td>1024-1535</td>
<td>0400-05FF</td>
</tr>
<tr>
<td>3</td>
<td>1536-2047</td>
<td>0600-07FF</td>
</tr>
<tr>
<td>4</td>
<td>2048-2559</td>
<td>0800-09FF</td>
</tr>
<tr>
<td>5</td>
<td>2560-3071</td>
<td>0AO0-0BFF</td>
</tr>
<tr>
<td>6</td>
<td>3072-3583</td>
<td>0CO0-DDFF</td>
</tr>
<tr>
<td>7</td>
<td>3584-4095</td>
<td>0EO0-OFFF</td>
</tr>
</tbody>
</table>

The size of a virtual page exactly corresponds to the size of a physical
page of memory, and the size of a block on disk.

To make memory mapping efficient, the processor must be able to
translate virtual addresses to physical addresses rapidly. Two features
providing rapid address translation are the processor's internal ad-
dress translation buffer, which is described later, and the translation
algorithm itself.

Figure 15-6 compares the virtual and physical address format. The
high-order two bits of a virtual address immediately identify the region
to which the virtual address refers. Whether the address is physical or
virtual, the byte within the page is the same. Thus, the processor has to
know only which virtual pages correspond to which physical pages.
Central Processor

Virtual Address

Virtual Page Number

Byte Within Page

Memory Address

I/O Space Address

Physical Address

Page Frame Number

Byte Within Page

Memory Address

I/O Space Address

Figure 15-6 Virtual and Physical Address Format

The processor has three pairs of page mapping registers, one pair for each of the three regions actively used. The operating system's memory management software loads each pair of registers with the base address and length of data structures it sets up called page tables. The page tables provide the mapping information for each virtual page in the system. There is one page table for each of the three regions.

A page table is a virtually contiguous array of page table entries. Each page table entry is a longword representing the physical mapping for one virtual page. To translate a virtual address to a physical address, therefore, the processor simply uses the virtual page number as an index into the page table from the given page table base address. Each translation is good for 512 virtual addresses since the byte within the virtual page corresponds to the byte within the physical page.

Exception and Interrupt Vectors

The processor can automatically initiate changes in the normal flow of program execution. The processor recognizes two kinds of events that cause it to invoke conditional software: exceptions and interrupts. Some exceptions affect an individual process only, such as arithmetic traps, while others affect the system as a whole, for example, machine check. Interrupts include both device interrupts, such as those signaling I/O completion, and software-requested interrupts, such as those signaling the need for a context switch operation.

The processor knows which software to invoke when an exception or interrupt occurs because it references specific locations, called vectors, to obtain the starting address of the exception or interrupt dispatcher. The processor has one internal register, the System Control Block Base Register, which the operating system loads with the physical address of the base of the System Control Block, which contains
the exception and interrupt vectors. The processor locates each vector by using a specific offset into the System Control Block. Figure 15-7 illustrates the vectors in the System Control Block. Each vector tells the processor how to service the event, and contains the system region virtual address of the routine to execute. Note that vector 14 (hex) can be used as a trap to writable control store to execute user-defined instructions, and the vector contains information passed to microcode.

**Interrupt Priority Levels**

Exceptions do not require arbitration since they occur synchronously with respect to instruction execution. Interrupts, on the other hand, can occur at any time. To arbitrate between interrupt requests that may occur simultaneously, the processor recognizes 31 interrupt priority levels.

The highest 16 interrupt priority levels are reserved for interrupts generated by hardware, and the lowest 16 interrupt priority levels are reserved for interrupts requested by software. Table 15-3 lists the assignment of each level, from highest to lowest priority. Normal user software runs at process level, which is interrupt priority level zero.

To handle interrupt requests, the processor enters a special system-wide context. In the system-wide context, the processor executes in kernel mode using a special stack called the interrupt stack. The interrupt stack cannot be referenced by any user mode software because the processor only selects the interrupt stack after an interrupt, and all interrupts are trapped through system vectors.

The interrupt service routine executes at the interrupt priority level of the interrupt request. When the processor receives an interrupt request at a level higher than that of the currently executing software, the processor honors the request and services the new interrupt at its priority level. When the interrupt service routine issues the REI (Return from Exception or Interrupt) instruction, the processor returns control to the previous level.

**I/O Space and I/O Processing**

An I/O device controller has a set of control/status and data registers. The registers are assigned addresses in physical address space, and their physical addresses are mapped, and thus protected, by the operating system's memory management software. That portion of physical address space in which device controller registers are located is called I/O space.

I/O space occupies the highest-addressed half of physical address
### Central Processor

#### Exception Vectors

<table>
<thead>
<tr>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Machine Check</td>
</tr>
<tr>
<td>10</td>
<td>Kernel Stack Not Valid</td>
</tr>
<tr>
<td>14</td>
<td>Power Fail</td>
</tr>
<tr>
<td>24</td>
<td>Reserved or Privileged Instruction</td>
</tr>
<tr>
<td>30</td>
<td>Customer Reserved Instruction</td>
</tr>
<tr>
<td>34</td>
<td>Reserved or Illegal Operand</td>
</tr>
<tr>
<td>40</td>
<td>Reserved or Illegal Addressing Mode</td>
</tr>
<tr>
<td>44</td>
<td>Access Violation</td>
</tr>
<tr>
<td>48</td>
<td>Translation Not Valid (page fault)</td>
</tr>
<tr>
<td>4C</td>
<td>Trace Trap</td>
</tr>
<tr>
<td>52</td>
<td>Breakpoint Trap</td>
</tr>
<tr>
<td>68</td>
<td>Compatibility Mode Trap</td>
</tr>
<tr>
<td>74</td>
<td>Arithmetic Trap</td>
</tr>
</tbody>
</table>

#### Interrupt Vectors

<table>
<thead>
<tr>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>Change Mode to Kernel</td>
</tr>
<tr>
<td>84</td>
<td>Change Mode to Executive</td>
</tr>
<tr>
<td>88</td>
<td>Change Mode to Supervisor</td>
</tr>
<tr>
<td>92</td>
<td>Change Mode to User</td>
</tr>
<tr>
<td>8C</td>
<td>Software Level 1</td>
</tr>
<tr>
<td>90</td>
<td>Software Level 2</td>
</tr>
<tr>
<td>98</td>
<td>Software Level F</td>
</tr>
<tr>
<td>CO</td>
<td>Interval Timer</td>
</tr>
<tr>
<td>100</td>
<td>Device Level 14, device 0</td>
</tr>
<tr>
<td>101</td>
<td>Device Level 14, device 1</td>
</tr>
<tr>
<td>105</td>
<td>Device Level 14, device 2</td>
</tr>
<tr>
<td>13F</td>
<td>Device Level 14, device 15</td>
</tr>
<tr>
<td>140</td>
<td>Device Level 15, device 0</td>
</tr>
<tr>
<td>144</td>
<td>Device Level 15, device 1</td>
</tr>
<tr>
<td>148</td>
<td>Device Level 15, device 2</td>
</tr>
<tr>
<td>17F</td>
<td>Device Level 15, device 15</td>
</tr>
<tr>
<td>180</td>
<td>Device Level 16, device 0</td>
</tr>
<tr>
<td>184</td>
<td>Device Level 16, device 1</td>
</tr>
<tr>
<td>1BF</td>
<td>Device Level 16, device 15</td>
</tr>
<tr>
<td>1C0</td>
<td>Device Level 17, device 0</td>
</tr>
<tr>
<td>1C4</td>
<td>Device Level 17, device 1</td>
</tr>
<tr>
<td>1D0</td>
<td>Device Level 17, device 2</td>
</tr>
<tr>
<td>1DF</td>
<td>Device Level 17, device 15</td>
</tr>
</tbody>
</table>

Offset from System Control Block Base Register (HEX)

**Figure 15-7** System Control Block

284
### Table 15-3  Interrupt Priority Levels

<table>
<thead>
<tr>
<th>PRIORITY</th>
<th>HARDWARE EVENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hex</td>
<td>Decimal</td>
</tr>
<tr>
<td>1F</td>
<td>31</td>
</tr>
<tr>
<td>1E</td>
<td>30</td>
</tr>
<tr>
<td>1D</td>
<td>29</td>
</tr>
<tr>
<td>1C</td>
<td>28</td>
</tr>
<tr>
<td>1B</td>
<td>27</td>
</tr>
<tr>
<td>1A</td>
<td>26</td>
</tr>
<tr>
<td>19</td>
<td>25</td>
</tr>
<tr>
<td>18</td>
<td>24</td>
</tr>
<tr>
<td>17</td>
<td>23</td>
</tr>
<tr>
<td>16</td>
<td>22</td>
</tr>
<tr>
<td>15</td>
<td>21</td>
</tr>
<tr>
<td>14</td>
<td>20</td>
</tr>
<tr>
<td>13</td>
<td>19</td>
</tr>
<tr>
<td>12</td>
<td>18</td>
</tr>
<tr>
<td>11</td>
<td>17</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PRIORITY</th>
<th>SOFTWARE EVENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F</td>
<td>15</td>
</tr>
<tr>
<td>0E</td>
<td>14</td>
</tr>
<tr>
<td>0D</td>
<td>13</td>
</tr>
<tr>
<td>0C</td>
<td>12</td>
</tr>
<tr>
<td>0B</td>
<td>11</td>
</tr>
<tr>
<td>0A</td>
<td>10</td>
</tr>
<tr>
<td>09</td>
<td>09</td>
</tr>
<tr>
<td>08</td>
<td>08</td>
</tr>
<tr>
<td>07</td>
<td>07</td>
</tr>
<tr>
<td>06</td>
<td>06</td>
</tr>
<tr>
<td>05</td>
<td>05</td>
</tr>
<tr>
<td>04</td>
<td>04</td>
</tr>
<tr>
<td>03</td>
<td>03</td>
</tr>
<tr>
<td>02</td>
<td>02</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

285
space, and is $2^{28}$ bytes in length. A portion of I/O space is specifically mapped into UNIBUS addresses, and is called UNIBUS space.

No special processor instructions are needed to reference I/O space. The registers are simply treated as locations containing integer data. An I/O device driver issues commands to the peripheral controller by writing to the controller’s registers as if they were physical memory locations. The software reads the registers to obtain the controller status. Note that accesses to UNIBUS registers may be made with byte or word instructions only. The driver controls interrupt enabling and disabling on the set of controllers for which it is responsible. When interrupts are enabled, an interrupt occurs when the controller requests it. The processor accepts the interrupt request and executes the driver’s interrupt service routine if it is not currently executing on a higher priority interrupt level.

**Process Context**

For each process eligible to execute, the operating system creates a data structure called the software process control block. Within the software process control block is a pointer to a data structure called the hardware process control block. It contains the hardware process context, i.e., all the data needed to load the processor’s programmable registers when a context switch occurs. To give control of the processor to a process, the operating system loads the processor’s Process Control Block Base register with the physical address of a hardware process control block and issues the Load Process Context instruction. The processor loads the process context in one operation and is ready to execute code within that context.

A process control block not only contains the state of the programmable registers, it also contains the definition of the process virtual address space. Thus, the mapping of the process is automatically context-switched.

Furthermore, the process control block provides the mechanism for triggering asynchronous system traps to user processes. The Asynchronous System Trap field enables the processor to schedule a software interrupt to initiate an AST routine and ensure that it is delivered to the proper access mode for the process.
CHAPTER 16
SYNCHRONOUS BACKPLANE INTERCONNECT

INTRODUCTION
The Synchronous Backplane Interconnect (SBI) is the data path that links the central processor, the memory subsystem, and the hardware adapters provided for the UNIBUS and MASSBUS. The VAX-11/780 bus structure is illustrated in Figure 16-1.

![Figure 16-1 Basic Bus Configuration](image)

When interfaced to the SBI, the central processor, memory subsystem, and I/O controllers are known as NEXUSs.

A NEXUS is a physical connection to the SBI and is capable of acting as any of the following:

- Commander — A NEXUS which transmits command and address information.
- Responder — A NEXUS which recognizes command and address information as directed to it and requiring a response.
- Transmitter — A NEXUS which drives the signal lines.
- Receiver — A NEXUS which samples and examines the signal lines.
A NEXUS also performs priority arbitration for its access to the SBI. A NEXUS may perform more than one function, as illustrated in the two following examples.

When the CPU issues a read command it is a commander since it issues command/address information. At the same time it is a transmitter since it is driving the signal lines. When the device (responder) returns the requested data, the CPU is considered a receiver since it examines the signal lines.

In the case of a memory read exchange, memory is the responder since it recognizes and responds to command/address information. Also, since it examines the signal lines, it is a receiver. When memory returns the requested data by driving the signal lines, it is a transmitter.

All NEXUSs receive every SBI transfer. Logic in each NEXUS determines whether the NEXUS is the designated receiver for this transfer.

Data may be exchanged between the following system elements:

- The central processor and memory subsystem.
- I/O controllers and memory subsystem.
- Central processor and I/O controllers.

The communication protocol allows the information path to be time-multiplexed in such a way that up to 32 data exchanges may be in progress simultaneously.

The SBI provides checked, parallel information transfer synchronous with a common system clock. In each clock period or cycle (duration of 200 nsec) interconnect arbitration, information transfer and transfer confirmation may occur. Utilizing the 200 nsec clock period, the SBI achieves a maximum information transfer rate of 13.3 million bytes per second.

**SBI STRUCTURE**

The SBI is comprised of 84 signal lines as illustrated in Figure 16-2. Its maximum physical length may not exceed 3 meters (9.8 ft). The lines of the SBI are divided into the following functional groups:

- Arbitration
- Information
- Response
- Interrupt
- Control
Arbitration Lines
There are 16 bus arbitration lines. Each arbitration line, TR (transfer request) <15:01>, is assigned to one NEXUS, thereby establishing a fixed priority access to the information path (refer to Figure 16-3). Access priority increases from TR15 to TR00, where TR00 is reserved for use as a hold signal for the following reasons:

1. NEXUS requires two or three adjacent cycles for a write type exchange.
2. NEXUS requires two adjacent cycles for an Extended Read exchange.
3. Central processors for Interrupt Summary Read exchanges.
4. TR00 is reserved for an SBI UNJAM operation.
To acquire control of the information path, a NEXUS asserts its assigned (transfer request) line at the beginning of a cycle.

At the end of the cycle, the NEXUS examines the state of all transfer request lines of higher priority. If no higher priority NEXUS is arbitrating for control of the SBI, the NEXUS will remove its transfer request and assert information path signals. The lowest priority NEXUS arbitrating for control of the SBI is the central processing unit. The CPU does not require a transfer request signal, since by default it will gain control of the SBI when no higher priority NEXUS is arbitrating.

**Information Lines**

The information transfer group exchanges command/addresses, data, and interrupt summary information. Each exchange consists of one to three information transfers.

For write commands, the commander uses two or three successive SBI cycles. The number of successive cycles required depends on whether one or two data words are to be written in the exchange. In the first case, the commander transmits the command/address in the first cycle, and a data word in the second cycle. In the second case, the commander transmits the command/address in the first cycle, data word 1 in the second cycle, and data word 2 in the third cycle.

Read commands are also initiated with a command/address transmitted from the commander. However, since data emanates from the responder, the requested data may be delayed by the characteristic access time of the responder. As in a write exchange, the read exchange will transmit data using one or two successive cycles depending on whether one or two data words were requested.
An interrupt summary exchange is in response to a device-generated interrupt to the CPU. The exchange is initiated with an interrupt summary read transfer from the CPU. The exchange is completed two cycles later with an interrupt summary response transfer containing the interrupt information.

The Information Transfer Group is subdivided into the following five fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Length in Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parity check</td>
<td>2 (P &lt;1:0&gt;)</td>
</tr>
<tr>
<td>Information Tag</td>
<td>3 (TAG &lt;2:0&gt;)</td>
</tr>
<tr>
<td>Source/Destination Identity</td>
<td>5 (ID &lt;4:0&gt;)</td>
</tr>
<tr>
<td>Mask</td>
<td>4 (M &lt;3:0&gt;)</td>
</tr>
<tr>
<td>Information</td>
<td>32 (B &lt;31:00&gt;)</td>
</tr>
</tbody>
</table>

**PARITY FIELD**

The parity field (P<1:0>) provides even parity for detecting single bit errors in the information transfer group. A transmitting NEXUS generates P0 as parity for the Tag, Identifier, and Mask fields and P1 as parity for the Information field. The parity field is illustrated in Figure 16-4.

Figure 16-4  Parity Field Configuration

P0 and P1 are generated in such a way that the sum of all logical one bits in the checked field, including the parity bit, is even.

When the SBI is idle, the information transfer path assumes an all-zero state, therefore, the parity field should always carry an even parity.

**TAG FIELD**

The tag field (TAG<2:0>) is asserted by a transmitting NEXUS to indicate the information type being transmitted on the information
lines. The tag field determines the interpretation of the ID and B fields. In addition, the tag field, in conjunction with the mask field, further defines special read and write data conditions.

Four tag fields and four reserved fields are defined as:

<table>
<thead>
<tr>
<th>TAG</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>READ DATA</td>
</tr>
<tr>
<td>011</td>
<td>COMMAND ADDRESS</td>
</tr>
<tr>
<td>101</td>
<td>WRITE DATA</td>
</tr>
<tr>
<td>110</td>
<td>INTERRUPT SUMMARY READ</td>
</tr>
</tbody>
</table>

The remaining tag fields, 001, 010, 100, and 111, are reserved.

- Read Data Tag

A tag field content of 000 specifies that the information field B<31:00> contains data requested by a previous read type command. The retrieved data may be one of three types: read data, corrected read data, and read data substitute. The retrieved data type is identified by the mask field M<3:0>. Read data is the normal expected error-free data type, where M<3:0> = 0000. Corrected read data (CRD) is represented by M<3:0> = 0001, and read data substitute is represented by M<3:0> = 0010. The recipient of the read data is designated by ID<4:0>. The read data tag formats are illustrated in figure 16-5.
Synchronous Backplane Interconnect

Figure 16-5  Read Data Tag Formats

- Command/Address Tag

A tag field content of 011 specifies that the data lines contain a command/address word, and that ID<4:0> is a unique code identifying the logical source (commander) of that command. As illustrated in Figure 16-6, B<31:00> is divided into a function field and an address field to specify the command and its associated address.

Figure 16-6  Command/Address Format

The ID field code represents the logical source of the data in a write command, and the address field specifies the address of where the data is to be written. For a read command, the ID field represents the logical destination of the data at the location specified in the address field.

The 28 bits of the address field define a 268, 435, 456 longword address space (1, 073, 741, 824 bytes) which is divided into two sections. Addresses 0-7FFFFFF (hex) (A27=0) are reserved for primary memory. Addresses 8000000 (hex) - FFFFFFF (hex) (A27=1) are reserved for device control registers. Primary memory begins at address 0, the address space is dense and consists only of storage elements. Figure 16-7 illustrates the VAX-11/780 physical address space. Note that both physical and SBI addresses are provided.
Synchronous Backplane Interconnect

Figure 16-7  SBI Physical Address Space

The user has access to the physical address space via the 30-bit physical byte address. However, since NEXUS registers are accessible only as longword addresses, system hardware converts the physical byte address (30 bits) to the SBI longword address (28 bits). This translation is described in Figure 16-8.
Synchronous Backplane Interconnect

Figure 16-8  Physical To SBI Address Translation

The low order two bits of the physical to SBI translation are not lost, but are represented by the mask field adjoining the SBI command address format.

The control address space is sparse with address assignments based on device type. Each NEXUS is assigned a 2048, 32-bit longword address space for control and status. The addresses assigned are determined by the TR number as shown in Figure 16-9.

Figure 16-9  Control Address Space Assignment
Synchronous Backplane Interconnect

The command/address tag formats are illustrated in Figure 16-10.

![Diagram of command/address tag formats]

- **READ MASKED FORMAT**
  - **READ DATA AT THIS ADDRESS**
  - **Parity (P1)**
  - **Tag (P0)**
  - **ID (0)**
  - **Mask (11)**
  - **Function (0)**
  - **Physical Address (1000)**

- **READ INTERLOCKED MASKED FORMAT**
  - **READ DATA AT THIS ADDRESS**
  - **Parity (P1)**
  - **Tag (P0)**
  - **ID (0)**
  - **Mask (11)**
  - **Function (0)**
  - **Physical Address (1000)**

- **READ EXTENDED FORMAT**
  - **READ DATA AT THIS ADDRESS**
  - **Parity (P1)**
  - **Tag (P0)**
  - **ID (0)**
  - **Mask (0)**
  - **Function (0)**
  - **Physical Address (1000)**

- **WRITE MASKED FORMAT**
  - **ADDRESS WHERE DATA IS TO BE WRITTEN**
  - **Parity (P1)**
  - **Tag (P0)**
  - **ID (0)**
  - **Mask (11)**
  - **Function (0)**
  - **Physical Address (1000)**

- **WRITE MASKED INTERLOCKED FORMAT**
  - **ADDRESS WHERE DATA IS TO BE WRITTEN**
  - **Parity (P1)**
  - **Tag (P0)**
  - **ID (0)**
  - **Mask (11)**
  - **Function (0)**
  - **Physical Address (1011)**
Synchronous Backplane Interconnect

Figure 16-10  Command/Address Tag Formats

- Write Data Tag
A tag field content of 101 specifies that B<31:00> contains the write data for the location specified in the address field of the previous write command. The write data will be asserted on B<31:00> in the SBI cycle immediately following the command/address cycle. The ID field transmitted is that of the commander. Figure 16-11 illustrates the write data tag format.

Figure 16-11  Write Data Tag Format

- Interrupt Summary Tag
A tag field content of 110 defines B<31:00> as the interrupt level mask for an interrupt summary read command. The level mask (B<07:04>) is used to indicate the interrupt level being serviced as the result of an interrupt request. In this case, the ID field identifies the commander, which is the CPU. Although unused, M<3:0> is to be transmitted as zero.
Synchronous Backplane Interconnect

The interrupt sequence consists of two exchanges:

The first exchange indicates the interrupt level being serviced. The interrupt level is determined by:

- I/O controller asserts interrupt

CPU strobes the interrupt, and if level 7 is the current level, interrupt code is called which performs the Interrupt Service Request.

The second exchange is the response, where the device requesting the interrupt identifies itself. From the identity of the device and the interrupt level the starting address of the service routine can be determined.

Figure 16-12 illustrates the interrupt summary tag format.

![Figure 16-12 Interrupt Summary Tag Format](Image)

- Reserved Tags

Tag \((<2:0> = 111)\) is reserved for diagnostic purposes. Tag codes 001, 010, and 100 are unused and reserved for future definition.

300
SOURCE/DESTINATION IDENTITY FIELD
The ID field (ID<4:0>) contains a code which identifies the logical source or logical destination of the information contained in B<31:00>. ID codes are assigned only to commander and responder NEXUSs (which issue/recognize command/address information). Each NEXUS is assigned an ID code which corresponds to the TR line which it operates. For example, a NEXUS assigned TR05 would also be assigned ID code=5.

MASK FIELD
The mask field (M<3:0>) has two interpretations. In the primary interpretation, M<3:0> is encoded to specify operations on any or all bytes appearing on B<31:00>. The mask is used with the read masked, write masked, interlock read masked, interlock write masked, and extended write masked commands. As shown in Figure 16-13 each bit in the mask field corresponds to a particular byte of B<31:00>.

![Diagram of Mask Field Format]

**Figure 16-13  Mask Field Format**

As previously mentioned, the secondary interpretation is used when Tag<2:0> = 000 (read data). Figure 16-14 illustrates the read data mask field.

![Diagram of Read Data Mask Format]

**Figure 16-14  Read Data Mask Format**
Response Lines
There are three response lines, broken down into two fields, confirmation CNF<1:0> and Fault (FAULT). CNF<1:0> informs the transmitter whether the information was correctly received, or if the receiver can process the command. FAULT is a cumulative error indication of protocol or information path malfunction, and is asserted with the same timing as the confirmation field. The CPU latches the fault signal, which in turn latches all the fault status registers and the SBI silo. The silo is a hardware mechanism used to record the last 16 SBI transactions. The silo aids in rapid error detection. The fault is then cleared by the software.

Either field is transmitted to the receiver two cycles after the associated information transfer. Confirmation is delayed to allow the information path signals to propagate, be checked, decoded by all receivers, and to be generated by the responder. During each cycle every NEXUS in the system receives, latches, and makes decisions on the information transfer signals. Except for multiple bit transmission errors or NEXUS malfunction, one of the NEXUSs receiving the information path signals will recognize an address or ID code. This NEXUS then asserts the appropriate response in CNF. The confirmation codes and their definitions are listed in Table 16-1.

<table>
<thead>
<tr>
<th>CNF Code</th>
<th>Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>00, No Response (N/R)</td>
<td>The unasserted state and indicates no response to a commander’s selection.</td>
</tr>
<tr>
<td>01, Acknowledge (ACK)</td>
<td>The positive acknowledgement to any transfer.</td>
</tr>
<tr>
<td>10, Busy (BSY)</td>
<td>The response to a command/address transfer, and indicates successful selection of a NEXUS which is presently unable to execute the command.</td>
</tr>
<tr>
<td>11, Error (ERR)</td>
<td>The response to a command/address transfer, and indicates selection of a NEXUS which cannot execute the command.</td>
</tr>
</tbody>
</table>

A BSY (10) or ERR (11) response to transfers other than command/address transfers will be considered as no response from the transmitter.
Synchronous Backplane Interconnect

Interrupt Request Lines
The interrupt request group consists of four request lines (REQ <7:4>) and an alert (ALERT) line. A request line is assigned to each NEXUS that interrupts and represents its assigned CPU interrupt level. The lines are used by NEXUS to invoke a CPU to service a condition requiring processor intervention. The request lines are priority encoded in an ascending order of REQ4-REQ7. A requesting NEXUS asserts its request lines synchronously with the SBI clock to request an interrupt. Any of the REQ lines may be asserted simultaneously by more than one NEXUS, and any combination of REQ lines may be asserted by the collection of requesting NEXUSs.

The alert signal is asserted by NEXUSs which do not implement interrupt request lines. Its purpose is to indicate to the CPU a change in NEXUS status of power condition or operating environment. NEXUSs which implement the REQ lines report these changes by requesting an interrupt.

Control Lines
The control group functions synchronize system activities and provide specialized system communications. The group includes the system clock which provides the universal timebase for any NEXUS connected to the SBI. The group also provides initialization, power fail, and restart functions for the system. In addition, a path is provided for coordinating memories to assure access to shared structures.

The control lines are comprised of the following subgroups: clock functions, interlock line, dead signal function, fail function, and the UNJAM function.

CLOCK FUNCTIONS
Six control group lines are clock signals and are used as a universal time base for all NEXUSs connected to the SBI. All SBI clock signals are generated on the CPU clock module and provide a 200 nsec clock period.

INTERLOCK LINE
The interlock line will be discussed in the command code description section.

DEAD SIGNAL FUNCTION
The Dead signal indicates an impending power failure to the clock circuits on bus terminating networks. NEXUSs will not assert any SBI signal while Dead is asserted. Thus, NEXUSs prevent invalid data from being received while the bus is in an unstable state.

The assertion of the power supply DC LO to the clock circuits or terminating networks causes the assertion of Dead. Dead is asserted
asynchronously to the SBI clock and occurs at least two \( \mu \)sec before the clock becomes inoperative. With power restart, the clock will be operational for at least two \( \mu \)sec before DC LO is negated. The negation of DC LO negates Dead.

**FAIL FUNCTION**
A NEXUS asserts the Fail (FAIL) function asynchronously to the SBI clock when the power supply AC LO signal is asserted to that NEXUS. The assertion of Fail inhibits the CPU from initiating a power up service routine. Fail is negated asynchronous to the SBI clock when all NEXUSs that are required for the power up operation have detected the negation of AC LO. The CPU samples the Fail line following the power down routine (assertion of FAIL) to determine if the power down routine should be initiated.

**UNJAM FUNCTION**
The UNJAM function restores (initializes) the system to a known, well defined state. The UNJAM signal is asserted only by the console of the CPU, and is detected by all NEXUSs. The CPU asserts UNJAM only when a console key is selected. The duration of the UNJAM pulse is 16 SBI cycles and is negated at T0.

When the CPU intends to assert UNJAM it will assert TR00 for 16 SBI cycles. The CPU will continue to assert TR00 for the duration of UNJAM and for 16 SBI cycles after the negation of UNJAM. This use of TR00 insures that the SBI is inactive preceding, during, and after the UNJAM operation.

**COMMAND CODE (Function \(<3:0>\) DESCRIPTION**
Information bits B<31:00> carry most of the information on the SBI. Information appears on these lines in command/address format, data format, interrupt summary read format, or interrupt summary response format. In command/address format, information is grouped in three fields: M<3:0>, the byte mask; F<3:0>, the function code; and A<27:00>, a 28-bit physical address. Function codes are shown in Figure 16-15. Bit 27 of the SBI address field determines whether the longword address A<27:00> is located in memory or I/O space (refer to Chapter 17, Figure 2).

**Read Mask Function (F=0001)**
Once the commander has arbitrated for and gained control of the SBI, it asserts the information transfer lines at T0. The receiver latches open at T2 and closes at T3. Information in these latches is stable from T3 to the next T2.
**Synchronous Backplane Interconnect**

<table>
<thead>
<tr>
<th>MASK USE</th>
<th>FUNCTION CODE</th>
<th>FUNCTION DEFINITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>IGNORED</td>
<td>0000</td>
<td>RESERVED</td>
</tr>
<tr>
<td>USED</td>
<td>0001</td>
<td>READ MASKED</td>
</tr>
<tr>
<td>USED</td>
<td>0010</td>
<td>WRITE MASKED</td>
</tr>
<tr>
<td>IGNORED</td>
<td>0011</td>
<td>RESERVED</td>
</tr>
<tr>
<td>USED</td>
<td>0100</td>
<td>INTERLOCK READ MASKED</td>
</tr>
<tr>
<td>IGNORED</td>
<td>0101</td>
<td>RESERVED</td>
</tr>
<tr>
<td>IGNORED</td>
<td>0110</td>
<td>RESERVED</td>
</tr>
<tr>
<td>USED</td>
<td>0111</td>
<td>INTERLOCK WRITE MASKED</td>
</tr>
<tr>
<td>IGNORED</td>
<td>1000</td>
<td>EXTENDED READ</td>
</tr>
<tr>
<td>IGNORED</td>
<td>1001</td>
<td>RESERVED</td>
</tr>
<tr>
<td>IGNORED</td>
<td>1010</td>
<td>RESERVED</td>
</tr>
<tr>
<td>USED</td>
<td>1011</td>
<td>EXTENDED WRITE MASKED</td>
</tr>
<tr>
<td>IGNORED</td>
<td>1100</td>
<td>RESERVED</td>
</tr>
<tr>
<td>IGNORED</td>
<td>1101</td>
<td>RESERVED</td>
</tr>
<tr>
<td>IGNORED</td>
<td>1110</td>
<td>RESERVED</td>
</tr>
<tr>
<td>IGNORED</td>
<td>1111</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>

**Figure 16-15  SBI Command Codes**

The command/address format instructs the NEXUS selected by A<27:00> to retrieve the addressed data word, and transfer it to the logical destination specified in the ID field. The addressed NEXUS will respond to the command/address transfer with ACK (assuming the NEXUS can perform the command at this time) two SBI cycles after the assertion of command/address. Figure 16-16 illustrates the SBI read function.

**Write Masked Function (F=0010)**

The write masked function instructs the NEXUS selected by A<27:00> to modify the bytes specified by M<3:0> in the storage element addressed by A<27:00>, using data transmitted in the next succeeding cycle. Figures 16-13 and 16-14 illustrate SBI write functions. Figure 16-17 illustrates a single SBI write transaction while Figure 16-18 illustrates two SBI write transactions from devices A and B.
Figure 16-16  SBI Read Transaction

Figure 16-17  Single SBI Write Transaction
Interlock Read Masked (F=0100)
This command, used to insure exclusive access to a particular memory location, causes the NEXUS selected by A<27:00> to retrieve and transmit the addressed data as for Read Masked. In addition, this command causes the selected memory controller NEXUS to set an Interlock flip-flop. Only memory NEXUSs have the ability to assert Interlock. While this flip-flop is set the NEXUS will assert the INTLK signal synchronously at time T0. Interlock is asserted during the same cycle as the confirmation signal. In the preceding cycle, the commander must assert Interlock. While the INTLK signal is asserted, the NEXUS will respond with BSY confirmation to Interlock read masked commands. The Interlock flip-flop is cleared on receipt of an Interlock write masked function. Interlock read masked and Interlock write masked are always paired by commanders utilizing them.

Interlock Write Masked (F=0111)
The Interlock write masked function instructs the NEXUS selected by A<27:00> to modify the bytes specified by M<3:00> in the storage element addressed, using data transmitted in the succeeding cycle with TAG=101. Additionally, the Interlock flip-flop is cleared.

Extended Read (F=1000)
The Extended Read function instructs the NEXUS selected by A<27:00> to retrieve the addressed 64-bit data and transmit it to the
Synchronous Backplane Interconnect

ID accompanying the command as in the read masked function. The responder transmits the data in two successive cycles with the low 32 bits (A00=0) preceding the high 32 bits (A00=1). Two data words are always transmitted. M<3:0> and A00 of the received command/address word are ignored. M<3:0> must be transmitted as 0000 by the commander. Figure 16-19 illustrates extended read transactions by two separate devices, A and B, reading memory via a single memory controller.

<table>
<thead>
<tr>
<th>SB1 ACTIVITY</th>
<th>EVENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARBITRATION</td>
<td>A</td>
</tr>
<tr>
<td>MEMORY HOLD</td>
<td>MEMORY HOLD</td>
</tr>
<tr>
<td>INFORMATION TRANSFER</td>
<td>EXT READ C/A A</td>
</tr>
<tr>
<td>DATA 1 (TO A)</td>
<td>DATA 2 (TO A)</td>
</tr>
<tr>
<td>DATA 1 (TO B)</td>
<td>DATA 2 (TO B)</td>
</tr>
<tr>
<td>CONFIRM (TO A)</td>
<td>CONFIRM (TO B)</td>
</tr>
<tr>
<td>CONFIRM (BY A)</td>
<td>CONFIRM (BY B)</td>
</tr>
</tbody>
</table>

Figure 16-19  Extended Read Transactions Via Single Memory Controller

Figure 16-20 illustrates extended read transactions by two separate devices, A and B, reading memory via separate memory controllers, M1 and M2.

<table>
<thead>
<tr>
<th>ACTIVITY</th>
<th>EVENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARBITRATION</td>
<td>A</td>
</tr>
<tr>
<td>M1 HOLD</td>
<td>M2 HOLD</td>
</tr>
<tr>
<td>INFORMATION TRANSFER</td>
<td>EXT READ C/A A</td>
</tr>
<tr>
<td>DATA 1 (TO A)</td>
<td>DATA 1 (TO A)</td>
</tr>
<tr>
<td>CONFIRM (TO A)</td>
<td>CONFIRM (TO B)</td>
</tr>
</tbody>
</table>

Figure 16-20  Extended Read Transactions Via Separate Memory Controllers

308
Extended Write Masked (F=1011)
The Extended Write Masked function instructs the NEXUS selected by A<27:00> that 64 bits of data are to be written. The receiver ignores A00 of the command/address transfer. A<27:00> indicates the low 32 bit word address. The write data is transmitted in two 32 bit words. The first word corresponds to A00=0 and the second word corresponds to A00=1. M<3:0> that accompanies the command address transmission indicates bytes to be written in the first write data word. M<3:0> that accompanies the first write data word transmission indicates bytes to be written in the second write data word. The M<3:0> field of the second data word cycle is ignored by receivers but must be transmitted as zeros. The assertion of a particular mask bit signifies that the byte corresponding to that mask bit is to be modified. The NEXUS implementing the Extended Write Masked function must implement all combinations of M<3:0>.

SYNCHRONOUS BACKPLANE INTERCONNECT THROUGHPUT
The following is a derivation of the aggregate throughput rate of the SBI:

200 nanoseconds/cycle = 5 million cycles/second.

Each cycle can carry an address (memory request) or four bytes of data.

Thus, one cycle is used to request eight bytes of data (to be read or written), and two cycles are used to carry data (at four bytes/cycle).

5 million cycles/second x 4 bytes/cycle = 20 million bytes/second.

20 x 2/3 (1 of every 3 cycles is an address) = 13.3 million bytes/second.
CHAPTER 17
MAIN MEMORY SUBSYSTEM

INTRODUCTION
Main Memory is a dynamic MOS (metal oxide semiconductor), random access memory designed to interface with the VAX-11/780 Synchronous Backplane Interconnect.

The memory subsystem consists of a controller and one to sixteen array boards utilizing 16K N-channel MOS IC storage elements. Each array board contains 256K bytes of memory, giving the system a capacity of four megabytes. Addition of a second controller increases system capacity to eight megabytes.

Memory is capable of random access read and write operations to a single 32-bit longword or extended 64-bit quadword. Memory is also capable of random access write to an arbitrary byte, series of contiguous bytes, or a series of noncontiguous bytes. The memory array board has been organized to optimize eight byte read/write access.

Memory features an error checking and correcting scheme (ECC) which can detect all double bit errors and detect and correct all single bit errors. The error detection and correction algorithm requires an entire quadword of data and thus during any type of read or write operation an entire quadword of data is fetched from the array.

Eight ECC check bits are stored with each quadword and accessed with the data to determine its integrity. Therefore, a total of 72 bits are accessed at once.

MEMORY CONTROLLER
The memory controller is the NEXUS interfacing main memory to the SBI. The controller examines the command and address lines of the SBI for each SBI cycle. To initiate and complete a memory write masked, interlock write masked, or extended write masked transaction, the controller must receive a recognizable command or address and data in two or three SBI cycles. However, to perform a read masked, extended read, or interlock read masked operation, the controller need only recognize an appropriate command/address. The controller provides the necessary timing and control to complete all memory transactions. The controller informs a commander, via a confirmation, of a successful write operation and contends and arbitrates for SBI bus control to transmit information during a read masked, extended read, or interlock read operation. However, before the controller will initiate a memory cycle operation, it checks for bus
Main Memory Subsystem

The basic memory subsystem is illustrated in Figure 17-1.

transmission parity errors and other fault conditions and reports these conditions to the commander, conforming to the SBI protocol. Data transfers to and from main memory are protected by ECC logic, i.e., main memory contains single bit error detection and correction and double bit error detection logic to improve system reliability.

Error reporting provides an early warning to protect the system from performance degradation. The system error logging feature requires tagging single bit errors and uncorrectable errors during memory read transmission from the memory subsystem. Also saved in the memory controller are the syndrome bits for the first memory read error and the error address for error logging and servicing. The memory controller retains this information until the first error is serviced. There are ten bits in register B that are used for ECC diagnostics only. In addition to its error detection capabilities, the controller provides the logic to buffer commands, addresses, and data, thus improving memory throughput.

A system may require more than one memory controller. If the system requires a two-controller interleaved memory configuration, the memory controllers must have consecutive TR selects. The interleave bit will be cleared on power up and must be set by writing to configuration register A in each controller. Each controller must have the same array size and be issued the same starting address. In the case of multiple
memory controllers, (up to four) each controller will assume a different starting address on power up. The proper starting address will be written into the configuration B register from the SBI bus.

A read-only memory that can be addressed on the SBI bus resides in the memory subsystem. The address, timing, and control logic to read the information from the ROM for booting the system is also contained in the subsystem.

The memory controller provides power up initialization logic and refresh control logic for the dynamic MOS memory devices. The dynamic MOS memory cell is a capacitor in which the stored charge represents a data bit. As the stored charge tends to diminish over a period of time, each cell requires a refresh cycle every 2 msec to retain the charge reliability.

**BASIC MEMORY OPERATIONS**

The memory subsystem operates synchronously with the SBI clock cycles, satisfying the system communication protocol. As discussed in Chapter 16, *Synchronous Backplane Interconnect*, the physical address space is divided into two equal areas, memory address space, and I/O address space. Figure 17-2 illustrates the physical address space.

The 28-bit (A<27:00>) SBI longword address field (refer to figure 8-2) is capable of accessing up to 512 M bytes of main memory. The hardware, however, will currently support a maximum of 8 Mbytes utilizing the 16K chip design. Physical memory operations are performed when bit <27> of Figure 8-2 is zero. I/O operations occur when bit <27> is one. The operation field identifies one of the following six transactions performed by the memory subsystem:

- Read Masked
- Extended Read
- Interlock Read Masked
- Write Masked
- Extended Write Masked
- Interlock Write Masked

A write mask operation is executed to transfer one to four bytes of data to memory. A read mask operation, however, is only capable of transferring four bytes of data from memory.

An Extended Read is executed to transfer eight bytes of data (two longwords) from memory to a requesting NEXUS. An Extended Write Masked, on the other hand, provides a byte-selectable transfer of up
Main Memory Subsystem

Figure 17-2  Physical Address Space

to eight bytes to memory. Interlock Read Masked and Interlock Write Masked perform the same function as Read Masked and Write Masked but also provide process synchronization.

Read Cycle
The read cycle will fetch 32 bits of data from the addressed location in the memory subsystem, will check for a single or double bit error, will correct a single bit error if it exists, and will transmit the data word along with the proper tag and ID code of the commander that requested the data. In the event a single bit error occurs during the read operation, corrected data would be rewritten into the memory in a subsequent memory cycle. In the case of a double bit error, the exact data and check code read is rewritten to ensure that the double bit error recurs on subsequent reads. In either case, an indication of the error condition would be tagged and transmitted with the corrected data or uncorrectable bad data during the next available bus cycle to the requester. The sequence of events that initiates a read cycle in a memory subsystem is as follows:

Any commander on the SBI (central processor or I/O controller) that wants to initiate a read cycle in any one of the memory subsystems on
the bus will arbitrate and gain control of the bus. Having gained control, the commander then transmits a command or address tag and identification code information on the bus. All subsystems on the bus monitor and decode the tag and command or address lines prior to initiating any action. If the decoded address corresponds to the memory subsystem and if no faults are detected, it would immediately (unless already busy) initiate a memory cycle. If the memory is presently executing a cycle, the command will be stored in the queue, if there is room in the buffer, until the present cycle is complete. Either way, the memory will notify the commander that the message has been received. The address under interrogation would be fetched from the memory, while the memory controller in the meantime would request, arbitrate, and gain control of the bus to transmit the data along with the commander’s identification code. Read cycles with single bit errors require an extra bus cycle to correct the error, and therefore the controller would re-request the bus and transmit data after gaining control of the bus.

**Extended Read**
The extended read cycle is the same as the read cycle, except that it fetches 64 bits of data from the addressed location. Also, the data would be transmitted on the SBI in two successive bus cycles; the lower 32 bits are transmitted first and then the upper 32 bits. In the event a single bit error occurs, the start of data transmission would be delayed until the memory controller re-requests the bus and gains control of the bus.

**Write Masked**
The write masked function instructs the memory controller selected by the address (A<27:00>) to modify the bytes specified by (M<3:0>) in the storage element addressed, using data transmitted in the next succeeding cycle.

This is accomplished in the memory subsystem in two successive memory cycles, a read followed by a write cycle as the memory is organized as a 32K x 72, with an ECC over 64-bit width. During the read portion of the cycle, the 64-bit word is retrieved, the error code checked, and the appropriate bytes are modified in the upper or lower half of the word. New check bits are then encoded and the modified word is written into the memory. If a single bit error occurs during the read portion of the cycle it would be corrected. In the case of an uncorrectable error the bad data would be rewritten into the memory with the bad check code and the new data would not be used.

Up to four bytes in the upper or lower word can be modified in a write masked cycle.
Extended Write Masked

The extended write masked function instructs the memory controller selected by the address (A <27:00>) to first modify the bytes specified by (M<3:0>) in the low 32 bits of storage element addressed, using data transmitted in the next succeeding cycle. Then the controller is to modify the bytes of the high 32-bit storage element specified by the masks (M<3:0>) field found in the first data word cycle, using data transmitted in the next succeeding cycle. The mask field in the second data word transmission is ignored.

The implementation of this cycle is similar to write masked except in the following areas:

- One to eight bytes of an address can be modified during this operation in the upper and lower word.
- An extended write masked that specified modification to all eight bytes does not execute a read cycle first but unconditionally writes the new 64 bits and eight check bits to the designated address. This is described as a full write cycle.

INTERLOCK CYCLES

The interlock cycles are special memory cycles used for process synchronization. They consist of the interlock read cycle and the interlock write cycle. The memory controller treats the interlock cycles as a pair of cycles, with an interlock read masked always followed (an arbitrary number of cycles after) by an interlock write masked. Interlock read and write cycles are 32-bit operations. The interlock line on the SBI is used to coordinate activity between memory controllers. An interlock timer of 512 bus cycles is started with the acceptance of an interlock write. If the interlock write is not found, after 512 bus cycles, the interlock line is cleared.

Interlock Read

The interlock read masked cycle is implemented in the same manner as the read masked cycle, with the following exception. The interlock read has a special function code which the memory controller decodes and also monitors the interlock line on the bus to verify any interlock activity elsewhere in the system. If the interlock line is not asserted, the memory controller addressed would acknowledge the cycle and set its interlock line on the SBI until a valid interlock write has been received.

In the case of a single-bit error, the controller corrects the data and transmits it with the proper tag. If an uncorrectable error occurs, the read data substitute tag with the bad data would be transmitted and the memory would rewrite the bad data and bad ECC.
Every commander on the SBI capable of issuing an interlock command should also assert the interlock line on the bus for one cycle immediately following the interlock read mask command. This is to insure cooperation among memory controllers responding to interlock reads without ambiguity.

Interlock Write
The interlock write masked cycle is similar to the write masked cycle with the following exceptions:
The set state of the interlock line on the SBI would verify the integrity of the command prior to acknowledging the cycle and implementing it. The interlock flip-flop would be cleared and consequently the interlock line on the bus would be deasserted.

If the interlock line was not asserted, the write interlock command would not be executed and the interlock sequence fault would be set.

ERROR CHECKING AND CORRECTION (ECC)
The ECC scheme used in the memory subsystem is capable of detecting a single or double bit error. It is also capable of correcting all single bit errors. This is accomplished by storing eight parity bits, called check bits, along with the 64 data bits in each memory location. Each check bit is generated by parity-checking selected groups of data bits in the given data quadword. When parity is again checked during a read, an incorrect bit will be detected by the parity-checking logic and will develop a unique 8-bit syndrome which will identify the bit in error. Error correction logic may thus correct the bit in error. There are 72 unique syndromes pointing to individual bits in the coded quadword.

MEMORY CONFIGURATION REGISTERS
There are three configuration registers in the memory controller to provide configuration-dependent information to the operating system and diagnostic software. These are addressable registers with read and write access.

Memory Configuration Register A
Register A contains the following information:
Bit: <31:26>
Name: SBI Fault Status
Function:

Bit: <23:22>
Name: Power Up/Power Down Status
Function: These bits work in conjunction with the Alert line. If the memory is strapped to inhibit ROM decode, the assertion of AC LO will set the power down status, clear the power up status and activate the
Main Memory Subsystem

Figure 17-3 illustrates memory configuration register A.

16 15 | 9 8 7 | 5 4 3 | 2 1 0
---|---|---|---
| MEMORY SIZE | 000 | TYPE | ILV

- PWR UP
- PWR DWN
- XMT FLT
- MLT XMT
- INTLK SEQ ERR
- WR SEQ ERR
- PAR ERR

Figure 17-3  Memory Configuration Register A

Alert line. The deassertion of the AC LO signal will set power up status, clear power down status and assert the Alert line. Writing a one to the active status bit will clear it and deassert Alert.

**Bit: <15:9>**

**Name:** Memory Size

**Function:** These bits contain the binary representation of the memory size in 64K byte increments, zero inclusive. For older 4K chip implementations, bits <12:9> are used. For the 16K chip, bits <14:09> are used. These bits are read-only.

**Bit: <4:3> Name:** Memory Type

**Function:** These bits specify the memory type. This refers to the 4K MOS chip implementation or the 16K MOS chip implementation. These bits are read-only.

<table>
<thead>
<tr>
<th>Bit &lt;4&gt;</th>
<th>Bit &lt;3&gt;</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Error condition, no array cards plugged in</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4K chip</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>16K chip</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Error condition, both 4K and 16K chip array boards are being used.</td>
</tr>
</tbody>
</table>

**Bit: <2:0> Name:** Interleave

**Function:** These bits contain interleave information. If bit <0> is 0, the memory is not interleaved. If bit <0> is a 1, the system is interleaved. Bits <1> and <2> are not used at this time and should be 0. Bit <8> is the interleave write enable bit. When bit <8> is written to with a one, bit <0> will take on whatever state bit <0> in the written data is. Bit <8> will always read as a 0. If bit <8> is written to with a 0,
interleave bit \(<0>\) will be unchanged. The interleave flip-flop receives its power from the +5V BAT supply so it retains its state during battery backup. On a cold start, this bit will come up 0.

**Memory Configuration Register B**

Figure 17-4 illustrates memory configuration register B.

![Figure 17-4 Memory Configuration Register B](image)

Register B contains the following information:

**Bit:** \(<31:30>\)

**Name:** File Output Pointer (File Read Address Counter)

**Function:** These two bits point to the address that would be read from the command and data file and operated on by the timing and control logic for starting a new memory cycle at the appropriate time, depending on the state of the memory busy line. This information is also useful for diagnosing the file control logic problems in the file read path. Bit \(<31>\) is the most significant bit, bit \(<30>\) is the least significant bit.

**Bit:** \(<29:28>\)

**Name:** File Input Pointer (File Write Address Counter)

**Function:** The memory controller command buffer (File) is four addresses deep and the Write Address Counter state can be read via these two bits. These bits point to the next available file address into which the command address or data information will be written after accepting the command address and data from the SBI. These bits assist in diagnosing the file control logic problems in the file write path. Bit \(<29>\) is the most significant bit, bit \(<28>\) is the least significant bit.

**Bit:** \(<27:15>\)

**Name:** Memory Starting Address

**Function:** These bits indicate the starting address of the memory controller in 64K byte granularity or increments. These bits are writable and can be altered by the system after power up. During a cold
Main Memory Subsystem

start the memory controller would come up with a default starting address depending on the starting address jumpers in the memory backplane. A cold start is defined as a CPU power up from inactive battery backup and memory power supply. There are two starting address jumpers in the memory controller backplane, and in a four controller system the default starting address assignments are as follows for cold starts.

<table>
<thead>
<tr>
<th>Controller No.</th>
<th>Starting Address Jumpers</th>
<th>Starting Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SA 01 SA 00</td>
<td>zero</td>
</tr>
<tr>
<td>2</td>
<td>OPEN OPEN</td>
<td>4 Megabyte</td>
</tr>
<tr>
<td>3</td>
<td>OPEN GND</td>
<td>8 Megabyte</td>
</tr>
<tr>
<td>4</td>
<td>GND OPEN</td>
<td>12 Megabyte</td>
</tr>
</tbody>
</table>

Also during battery backup the contents of the starting address bits are saved.

**Bit: <14> Name:** Write Enable to Memory Starting Address  
**Function:** This bit must be at a one state during a write to register B in order to alter the state of the memory starting address. If bit <14> is a zero, writes to register B will leave the starting address unchanged.

**Bit: <13:12>**  
**Name:** Memory Initialization Status  
**Function:** These are read-only bits and contain the recovery mode information necessary to determine whether or not the memory has recovered from battery backup and therefore contains valid data.

<table>
<thead>
<tr>
<th>Bit &lt;12&gt;</th>
<th>Bit &lt;13&gt;</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Initialization cycle in process. This means the memory is presently writing a known data pattern and check code throughout the storage area. A command issued to the array at this time will receive a busy response.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Invalid state</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>This state means the memory contains valid data. This state after a CPU Power Fail implies that all memory data was saved.</td>
</tr>
</tbody>
</table>
Main Memory Subsystem

Bit <12> Bit <13> Description
1 1 This state signifies that initialization is complete and that the power restoration was from a cold state. No data was preserved.

Bit: <10> Name: Refresh Indication
Function: This bit is used for diagnostic purposes only, and will verify the access time delay due to refresh collision.

Bit: <9> Name: Force ERR
Function: This bit is used in conjunction with bits <7:0>. When it is set, it will enable the ECC substitute bits to replace the actual check bits for the ECC computation when operating on an address with SBI bits 3 and 12 active. Writing a one sets this bit and writing a zero clears it.

Bit: <8> Name: ECC BYPS
Function: This bit is set to totally bypass (BYP S) the ECC check function. If this bit is set, the data that is read from the memory will be placed on the SBI exactly as it is found. Also, no CRD or RDS flag will accompany the data if it is in error but the error log will continue to operate normally (register C). Writing a one sets this bit, writing a zero clears it. This bit is used for diagnostics only.

Bit: <7:0> Name: Substitute ECC Bits
Function: These bits can be substituted for the eight check bits read from the memory, providing that bit <9> in Register B is set to a one and the address read contains SBI bits 3 and 12 active. These bits are for diagnostics only and can be used to simulate any single bit or multiple bit error, thereby checking the entire ECC path. Writing a one sets the bits, writing a zero will clear them.

Memory Configuration Register C
Figure 17-5 illustrates memory configuration register C.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ERROR ADDRESS</td>
<td>ERROR SYNDROME</td>
<td></td>
</tr>
</tbody>
</table>

Figure Figure 17-5 Memory Configuration Register C

321
This register gathers all the ECC error information:

**Bit: <30> Name: Inhibit CRD**
**Function:** This bit is used to prevent constant CRD flags from being sent to the commander when working in sequential memory locations with single cell failures, thus preventing repeated error service invocation by the operating system. Writing a one to this bit prevents subsequent CRD flags from being transmitted to the commander until such a time as the commander writes a zero to bit <30>. However, in the event an uncorrectable error occurs in the memory it would be reported right away regardless of the state of this bit.

**Bit: <29> Name: High Error Rate**
**Function:** This bit flags the high error rate in the memory by setting this bit if an error occurs between the time the first error message was sent and the time the error service subroutine was invoked by the operating system. This bit can be cleared by writing a one.

**Bit: <28> Name: Error Log Request Flag**
**Function:** This bit is set when the first error occurs during the memory controller's response to an SBI read cycle. This would indicate to the error service subroutine whether the controller has logged an error during its operation or not. When this bit is set, any subsequent CRD reports to the bus commander will be inhibited. In a multiple memory controller system, this is needed in determining which controller sent the error message. This can be cleared by writing a one.

**Bit: <27:8>**
**Name: Error Address**
**Function:** The SBI longword address at which the first read error occurred during memory controller response to an SBI read command is saved in these bits. Subsequent error addresses, if they occur, are not saved until the first one is serviced.

The address field is described as follows: (bit order is least to most)

- Bit <8> indicates the word in error.
  - 0 = lower word
  - 1 = upper word

- Bits <20:9> indicate the 4K chip address in error.
- Bit <21> indicates the 4K chip array bank in error.
  - 0 = lower 4K chip
  - 1 = upper 4K chip

- Bits <23:22> are unused for 4K chip.

Used in 16K chip for two necessary extra chip address bits. All chip address and bank address bits shift left two.
- Bits <27:24> indicate the array card in error.
Main Memory Subsystem

Bit: <7:0> Name: Error Syndrome
Function: These eight bits store the error syndrome of the first error word that was read from memory in response to an SBI read command. The syndrome will be saved until the error service routine has serviced the error. Subsequent error syndromes will not be saved but will be indicated by bit <29>.

MEMORY INTERLEAVING
The memory subsystem is capable of operating in the non-interleaving or two-way interleaved mode. Interleaving improves memory subsystem throughput on the bus.

In a single memory controller system the starting address is assigned by the ROM bootstrap. The size of the memory subsystem is encoded from the number of array cards plugged into the backplane. Array boards must be mounted contiguously in the backplane. If boards are misplugged an indicator light would indicate configuration error.

Interleaving can be used to increase the overall speed of the memory subsystem when there are two memory controllers with equal amounts of MOS memory on each. The effectiveness of interleaving is based on the principle that most memory operations are performed on consecutive memory locations. While one controller is fetching data, the other controller is available to decode an address for the next operation. On VAX-11/780, the two memory controllers access alternate quadwords.

With an interleaved memory system, both controllers must have contiguous bus TR select levels (odd and even pairs), the same array size, the same starting address, and both controllers must have their interleaved bits set.

It is also possible to have two two-way interleaved memory systems, four controllers, by following the rules just listed and assigning the second interleaved memory system a starting address that is one location above the final address of the first interleaved set.

Four memory controllers on one bus may require reassigning of bus TR select levels of the other SBI NEXUS.

ROM BOOTSTRAP
A four kilobyte programmable read-only memory to boot the system resides in the memory controller and it uses a 1K x 4, bipolar, high speed device. The memory is organized as a 1K x 32 and is assigned 4K byte I/O address space. The ROM can be addressed via the SBI interface in the memory controller during system initialization. All the address, data and control logic for addressing the ROM bootstrap is in
the memory controller. The ROM is packaged in such a way that ECOs can be easily handled by providing sockets in the PROM locations. ROM access time = 5 bus cycles (with respect to the commander).

BATTERY BACKUP
A battery backup option is available that will maintain the contents of 4 Mb of MOS memory for a minimum of 10 minutes, or less memory for longer than 10 minutes.
INTRODUCTION

The UNIBUS subsystem is the hardware developer’s primary interface to VAX-11/780. All devices other than high-speed disk drives, magnetic tape transports and devices which use the high performance DR780 interconnect are connected to the UNIBUS, an asynchronous bidirectional bus. The UNIBUS is connected to the SBI through the UNIBUS adapter. The UNIBUS adapter does priority arbitration among devices on the UNIBUS.

The UNIBUS adapter provides access from the processor to the UNIBUS peripheral device registers and to UNIBUS memory by translating UNIBUS addresses, data, and interrupt requests to their SBI equivalents, and vice versa. The UNIBUS adapter address translation map translates an 18-bit UNIBUS address to a 30-bit SBI address. The map provides direct access to system memory for nonprocessor request UNIBUS peripheral devices and permits scatter/gather disk transfers.

The UNIBUS adapter enables the processor to read and write the peripheral controller status registers. In the case of processor interrupt request devices, this constitutes the transfer.

This chapter is organized to provide the reader with an understanding of the UNIBUS and the VAX-11/780 UNIBUS Adapter. The UNIBUS subsystem is comprised of the UNIBUS adapter logic, the UNIBUS, and associated peripheral devices. Figure 18-1 illustrates the UNIBUS subsystem configuration.

UNIBUS SUMMARY

The UNIBUS, a high-speed communication path, links together I/O devices to the UNIBUS adapter. Device-related address, data, and control information are passed along the 56 lines of the UNIBUS. The UNIBUS adapter handles all communications between the UNIBUS and the SBI, and fields device-generated interrupts.

The following UNIBUS summary description takes into account the presence of the UBA, which performs the following UNIBUS functions:

- arbitration
- interrupt fielding
- power fail/restart
- initialization
UNIBUS Subsystem

![UNIBUS Configuration Diagram](image)

Figure 18-1 UNIBUS Configuration

For example, the UBA enables the system to accept device interrupts and transfer the requests from the UNIBUS to the SBI. However, UBA and SBI operations between the VAX-11/780 CPU and UNIBUS are transparent to the UNIBUS devices.

Communications and Control
A master/slave relationship defines all communications between devices on the UNIBUS. The device in control of the bus is considered the master; the device being addressed is the slave. Communication on the UNIBUS is interlocked, that is, each control signal issued by the master device must be acknowledged by a corresponding response from the slave to complete the transfer.

Bus Request Levels
Each device uses one of five priority levels for requesting bus control: Non-Processor Requests (NPR) and four Bus Requests (BR). The NPR is used when a device requests a direct access data transfer to memory or another device (i.e., a transfer not requiring processor intervention). Normally, NPR transfers are made between a mass storage device (e.g., disk drive) and memory. Two bus lines are associated with the NPR priority level. The device issues its request on the NPR line; the UBA responds by issuing a grant on the Non-Processor Grant (NPG) line.
A BR level is used when a device interrupts the VAX-11/780 CPU in order to request service. The device may require the CPU to initiate a transfer. Or it may need to inform the CPU that an error condition exists. Two lines are associated with each of four BR levels. The bus request is issued on a BR line (BR7-BR4); the bus grant is issued on the corresponding Bus Grant line (BG7-BG4).

Priority Structure and Chaining
When a device requests use of the bus, the handling of that request depends on the location of that device in a two-dimension device-priority structure. Priority is controlled by the priority arbitration logic of the CPU and the UBA.

The device-priority structure consists of five priority levels: NPR and BR7-4. Bus requests from devices can be made on any one of the request lines. The NPR has highest priority; BR7 is the next highest priority, and BR4 is the lowest. The priority arbitration logic is structured so that if two devices on different BR levels issue simultaneous requests, the priority arbitration logic grants the bus to the device with the highest priority. However, the lower priority device keeps its request up and will gain bus control when the higher-priority device finishes with the bus (providing that no other higher-priority device issues a BR).

Since there are only five priority levels, more than one device may be connected to a specific request level. If more than one device makes a request at the same level, the device closest (electrically) to the UBA has highest priority. The grant for each BR level is connected to all devices on that level in a daisy-chain arrangement (chaining). When a corresponding BG is issued it goes to the device closest to the UNIBUS adapter. If that device did not make the request, it permits the BG to pass to the next closest device. When the BG reaches the device making the request, that device captures the grant and prevents it from passing on to any subsequent device in the chain. Functionally, NPG chaining is similar to BG chaining.

Device Register Organization
The actual transfer of data and status information over the UNIBUS is accomplished between status, control, and data buffer registers located within the peripheral devices and their control units. All device registers are assigned addresses similar to memory addresses. These registers can therefore be accessed by word type memory reference instructions (i.e., MOVW, BITW, etc.).

Control and status functions are assigned to the individual bits within the corresponding addressable registers. Since the register content
UNIBUS Subsystem

can be controlled, setting and clearing register bits can control service operations. Internal device status may be loaded into the appropriate register and retrieved when a program instruction addresses that register. Depending on the function, register bits may be read/write, read only, or write only. The number of addressable registers in a device (and control unit) varies depending on the device’s function.

UNIBUS Line Definitions
The UNIBUS consists of 56 signal lines which may be divided into three function groups: bus control, data transfer, and miscellaneous signals. The 13 lines of the bus control group comprise those signals required to gain bus control through an NPR/BR or for a priority arbitration to select the next bus master while the current bus master is still in control of the bus. The 40 bidirectional lines of the data transfer group are those signals required during data transfers to or from a slave device. The miscellaneous group are the initialization and power fail signals required on the UNIBUS. Table 18-1 describes the bus signals within each group.

Table 18-1  UNIBUS Signal Descriptions

<table>
<thead>
<tr>
<th>SIGNAL LINE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Transfer Group</strong></td>
<td></td>
</tr>
<tr>
<td>Address Lines (A&lt;17:0&gt;)</td>
<td>These lines are used by the master device to select the slave (actually a unique memory or device register address). A &lt;17:1&gt; specifies a unique 16-bit word; SA00 specifies a byte within the word.</td>
</tr>
<tr>
<td>Data Lines (D&lt;15:0&gt;)</td>
<td>These lines transfer information between master and slave.</td>
</tr>
<tr>
<td>Control (C1,C0)</td>
<td>These signals are coded by the master device to control the slave in one of the four possible data transfer operations specified below. Note that the transfer direction is always designated with respect to the master device.</td>
</tr>
</tbody>
</table>

**Data Transfer Designation Description**

<table>
<thead>
<tr>
<th>C1</th>
<th>C0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
**UNIBUS Subsystem**

<table>
<thead>
<tr>
<th>C1</th>
<th>C0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Parity A-B (PA,PB)**

These signals transfer UNIBUS device parity information. PA is currently unused and not asserted. PB, when true, indicates a device parity error.

**Master Synchronization (MSYN)**

MSYN is asserted by the master to indicate to the slave that valid address and control information (and data on a DATO or DATOB) are present on the UNIBUS.

**Slave Synchronization (SSYN)**

SSYN is asserted by the slave. On a DATO it indicates that the slave has latched the write data. On a DATI or DATIP it indicates that the slave has asserted read data on the UNIBUS.

**Interrupt (INTR)**

This signal is asserted by an interrupting device, after it becomes bus master, to inform the UBA that an interrupt is to be performed, and that the interrupt vector is present on the data (D) lines. INTR is negated upon receipt of the assertion of SSYN by the UBA at the end of the transaction. INTR may be asserted only by a device which obtained bus mastership under the authority of a BG signal.

**Priority Arbitration Group**

**Bus Request (BR7-BR4)**

These signals are used by peripheral devices to request control of the bus for an interrupt operation.

**Bus Grant (BG7-BG4)**

These signals form the UBA's response to a bus request.

Only one of the four will be asserted at any time.
UNIBUS Subsystem

Nonprocessor Request (NPR)  This is a bus request from a device for a transfer not requiring CPU intervention (i.e., direct memory access).

Nonprocessor Grant (NPG)  This is the grant in response to an NPR.

Select Acknowledge (SACK)  SACK is asserted by a bus-requesting device after having received a grant. Bus control passes to this device when the current bus master completes its operation.

Bus Busy (BBSY)  BBSY indicates that the data lines of the UNIBUS are in use and is asserted by the UNIBUS master.

Initialization Group

Initialize (INIT)  This signal is asserted by the UBA when DC LO is asserted on the UNIBUS, and it stays asserted for ten msec following the negation of DC LO. It is used to initialize UNIBUS peripherals.

AC Line Low (AC LO)  This is a signal which indicates that a power failure is about to occur on the UNIBUS. The assertion of this signal initiates the UNIBUS power fail sequence of the UBA and can cause an interrupt to the VAX-11/780 CPU. It may also be used by peripheral devices to terminate operations in preparation for power loss.

DC Line Low (DC LO)  This signal is available from each system power supply and remains clear as long as all DC voltages are within the specified limits. If an out-of-voltage condition occurs, DC LO is asserted.

THE UNIBUS ADAPTER

The UNIBUS Adapter provides the interface between the asynchronous UNIBUS and the Synchronous Backplane Interconnect in the VAX-11/780. The UNIBUS Adapter provides the following functions:

- Access to UNIBUS address space (i.e., UNIBUS device registers) from the SBI
- Mapping of UNIBUS addresses to SBI addresses for UNIBUS DMA transfers to SBI memory
- Data transfer paths for UNIBUS device access to random SBI memory addresses and high-speed transfers for UNIBUS devices that transfer to consecutive increasing memory addresses
- UNIBUS interrupt fielding
- UNIBUS priority arbitration

332
UNIBUS Subsystem

- UNIBUS power fail sequencing

The UNIBUS Subsystem is illustrated in Figure 18-2.

Figure 18-2 UNIBUS Subsystem

VAX-11/780 hardware will support a UNIBUS Adapter in one of four physical address spaces. The UNIBUS adapter maintains two independent address spaces within the Synchronous Backplane Interconnect I/O address space. The first area of addressable space is within the area reserved for all NEXUSs (i.e., UBA, MBA, memory controller) internal registers. Each NEXUS (UBA) register address space occupies 8K bytes (16 pages of 512 bytes/page). This address space contains all control and status registers of the UBA, registers required for UNIBUS interrupt fielding, and registers required for mapping UNIBUS device transfers to the SBI address space. The second address space is the UNIBUS address space associated with the UBA. The UNIBUS address space occupies a total of 256K bytes (512 pages of 512 bytes/page). Figure 18-3 illustrates the SBI I/O address space.
SBI ACCESS TO UNIBUS ADDRESS SPACE

The UNIBUS Address Space (248K bytes of memory space and 8K bytes of device register space) is accessible as part of the SBI I/O Address Space. The UBA translates SBI command(addresses to UNIBUS command(addresses, thereby giving the software the ability to read and write UNIBUS device registers using word type memory reference instructions (MOVW, BITW, etc.).

Device Registers are assigned I/O addresses within the UNIBUS Address Space spanning 760000h—777777h. In VAX-11/780 physical byte address terms, the device registers occupy address space 201XE000₁₆—201XFFFF₁₆. The hexadecimal digit 3,7,B or F₁₆ is substituted in place of the X value within the physical address, depending upon which one of four UNIBUS address spaces the UBA is configured for (refer to Figure 16-7, SBI Physical Address Space, Chapter 16). Table 18-2 illustrates the UNIBUS device register address structure.
Table 18-2  UNIBUS Device Address Space

<table>
<thead>
<tr>
<th>UNIBUS I/O ADDRESS SPACE</th>
<th>UNIBUS ADDRESS (OCTAL)</th>
<th>PHYSICAL BYTE LOCATIONS (HEX)</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIBUS 0 Address Space</td>
<td>760000-777777</td>
<td>2013E000-2013FFFF</td>
</tr>
<tr>
<td>UNIBUS 1 Address Space</td>
<td>760000-777777</td>
<td>2017E000-2017FFFF</td>
</tr>
<tr>
<td>UNIBUS 2 Address Space</td>
<td>760000-777777</td>
<td>201BE000-201BFFFF</td>
</tr>
<tr>
<td>UNIBUS 3 Address Space</td>
<td>760000-777777</td>
<td>201FE000-201FFFF</td>
</tr>
</tbody>
</table>

Table 18-3 illustrates the translation of SBI to UNIBUS transfer operations involved in accessing the UNIBUS address space.

Table 18-3  CPU-Initiated Transfer

<table>
<thead>
<tr>
<th>SBI FUNCTION</th>
<th>TRANSFER DIRECTION</th>
<th>UNIBUS FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read-masked (word or byte)</td>
<td>device to UBA</td>
<td>DATI</td>
</tr>
<tr>
<td>Write-masked (word or byte)</td>
<td>UBA to device</td>
<td>DATO or DATOB</td>
</tr>
<tr>
<td>Interlock Read-masked then Inter-</td>
<td>device to UBA then UBA</td>
<td>DATIP then DATO or</td>
</tr>
<tr>
<td>locked Write-masked</td>
<td>to device</td>
<td>DATOB</td>
</tr>
</tbody>
</table>

During such transfers, the UNIBUS Adapter becomes the highest priority UNIBUS Non-Processor request (NPR) device.

Address and Function Translation

Figure 18-4 shows the SBI command/address format for accessing the UNIBUS address space for UBAs 0 through 3. Each SBI address (longword address) covers two 16-bit UNIBUS addresses (word addresses). In addition to the SBI address being decoded, the SBI function and byte mask is decoded to determine the word or byte to be accessed. The SBI to UNIBUS address and command translation is shown on the following page.
Table 18-4 illustrates the translation from SBI Mask and Function fields to UNIBUS Control and Address fields.

Only the function byte mask combinations shown will be valid. All other function byte mask combinations addressed to the UNIBUS address space will be given an ERR confirmation. The UNIBUS address space will respond only to word or byte SBI references. Note that extended transfers cannot be made to either the UNIBUS address space or the UNIBUS Adapter Registers.

The translation from SBI Mask and Function to UNIBUS control and byte address is handled by the UNIBUS control and byte address encoder illustrated in Figure 18-4.

When the VAX-11 software initiates a data transfer, reading from or writing to a UNIBUS device register, the UBA will recognize the address as being an address within the UNIBUS address space and will pass the lower 16 SBI address bits through to the UNIBUS as UNIBUS address bits UA <17:2>. The UNIBUS Adapter generates UNIBUS address bits UA <1:0> and control bits C <1:0> by decoding the SBI mask and function bits. Table 18-5 shows the relationship of the UNIBUS space controlled by UBA #0 to the SBI address space.
Table 18-4  SBI Function-Mask Translation To UNIBUS Control-Address

<table>
<thead>
<tr>
<th>Function &lt;3:0&gt;</th>
<th>Mask 3 2 1 0</th>
<th>Control C&lt;1:0&gt;</th>
<th>Address UA&lt;1:0&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Mask</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>DATI</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>DATI</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>DATI</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>DATI</td>
<td>1 0</td>
<td></td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>DATI</td>
<td>1 0</td>
<td></td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>DATI</td>
<td>1 0</td>
<td></td>
</tr>
<tr>
<td>Write Mask</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>DATOB</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>DATOB</td>
<td>0 1</td>
<td></td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>DATOB</td>
<td>1 0</td>
<td></td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>DATOB</td>
<td>1 1</td>
<td></td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>DATO</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>DATO</td>
<td>1 0</td>
<td></td>
</tr>
<tr>
<td>Interlock Read Mask (Sets Interlock)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>DATIP</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>DATIP</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>DATIP</td>
<td>1 0</td>
<td></td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>DATIP</td>
<td>1 0</td>
<td></td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>DATIP</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>DATIP</td>
<td>1 0</td>
<td></td>
</tr>
<tr>
<td>Interlock Write Mask</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>DATOB</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>DATOB</td>
<td>0 1</td>
<td></td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>DATOB</td>
<td>1 0</td>
<td></td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>DATOB</td>
<td>1 1</td>
<td></td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>DATO</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>DATO</td>
<td>1 0</td>
<td></td>
</tr>
</tbody>
</table>

SBI to UNIBUS Transfer Failures
If, during a read sequence to the UNIBUS address space, data is received from the UNIBUS device with UNIBUS PB asserted (UNIBUS Device Parity Error) then the data will be sent to the SBI as a Read Data Substitute.

If, for some reason, an access is made to the UNIBUS address space and the transfer is not completed on the UNIBUS (i.e., nonexistent device), the following will occur:

1. An all-zeroes word will be sent as a read data for a read transfer.
2. The UNIBUS address bits <17:2> will be stored in the Failed UNIBUS Address Register (FUBAR).
3. The bit indicating the cause of failure (UBA Select Time Out or SSYN Time Out) will be set in the UBA Status Register. Note that in the case of a Write Transfer to the UNIBUS, the error bit is set at least 13 μs after the command was issued and acknowledged by
<table>
<thead>
<tr>
<th>System Address Space (not to scale)</th>
<th>30 bit Physical Byte Address (Hex)</th>
<th>10 bit Unibus Address Space (Hex)</th>
<th>10 bit Unibus Address Space (Octal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Address Space</td>
<td>20100002 20100000</td>
<td>00002 00000</td>
<td>00002 00000 Unibus Memory Address</td>
</tr>
<tr>
<td></td>
<td>20100006 20100004</td>
<td>00006 00004</td>
<td>00006 00004 Space Reserved For Expansion (496 pages) 1 page = 512 bytes</td>
</tr>
<tr>
<td></td>
<td>2010000A 20100008</td>
<td>0000A 00008</td>
<td>000012 000010 Memory Address Space</td>
</tr>
<tr>
<td></td>
<td>2010000E 2010000C</td>
<td>0000E 0000C</td>
<td>000016 000014 Address Space</td>
</tr>
<tr>
<td></td>
<td>20100012 20100010</td>
<td>00012 00010</td>
<td>000022 000020 Reserved For Expansion (496 pages) 1 page = 512 bytes</td>
</tr>
<tr>
<td>Other Adaptor Registers</td>
<td>2013DFF6 2013DFF4</td>
<td>3DFF6 3DFF4</td>
<td>757766 757764 1 page = 512 bytes</td>
</tr>
<tr>
<td>Unibus Adaptor Registers</td>
<td>2013DFFA 2013DFF8</td>
<td>3DFFA 3DFF8</td>
<td>757772 757770 1 page = 512 bytes</td>
</tr>
<tr>
<td></td>
<td>2013DFFE 2013DFFC</td>
<td>3DFFE 3DFFC</td>
<td>757776 757774 1 page = 512 bytes</td>
</tr>
<tr>
<td>Other Adaptor Registers</td>
<td>2013E002 2013E000</td>
<td>3E002 3E000</td>
<td>760002 760000 Unibus I/O Address Space</td>
</tr>
<tr>
<td></td>
<td>2013E006 2013E004</td>
<td>3E006 3E004</td>
<td>760006 760004 Address Space</td>
</tr>
<tr>
<td></td>
<td>2013E00A 2013E008</td>
<td>3E00A 3E008</td>
<td>760012 760010 Address Space</td>
</tr>
<tr>
<td></td>
<td>2013E012 2013E011 2013E010</td>
<td>3E012 3E011 3E010</td>
<td>760023 760022 760021 760020 Unibus I/O Address Space</td>
</tr>
<tr>
<td>Unibus I/O Address Space</td>
<td>3E013</td>
<td>757766 757764</td>
<td>757772 757770 Address Space</td>
</tr>
<tr>
<td></td>
<td>3E012 3E011 3E010</td>
<td>757772 757770</td>
<td>757774 757774 Address Space</td>
</tr>
<tr>
<td>Other I/O Address Space</td>
<td>2013FFF6 2013FFF4</td>
<td>3FFF6 3FFF4</td>
<td>777766 777764 Address Space</td>
</tr>
<tr>
<td></td>
<td>2013FFFFA 2013FFFF8</td>
<td>3FFFFA 3FFFF8</td>
<td>777772 777770 Address Space</td>
</tr>
<tr>
<td></td>
<td>2013FFFFE 2013FFFFC</td>
<td>3FFFFE 3FFFFC</td>
<td>777774 777774 Address Space</td>
</tr>
</tbody>
</table>

Note: These addresses refer to UBAO.
the UBA. It will therefore not be immediately known to the software. If the software has set the SUFFIE (SBI to UNIBUS Error Interrupt Enable), the setting of UB select Time Out or SSYN Time Out will initiate an adapter interrupt request (13 \( \mu s \) for SSYN timeout, 50 \( \mu s \) for select Time Out).

This method gives the VAX-11 software an opportunity to exit gracefully from a transfer failure rather than being trapped out of a program due to a Read Data timeout. The method is also consistent for read and write failures.

**UNIBUS ACCESS TO THE SBI ADDRESS SPACE**

UNIBUS initiated transfers to UNIBUS memory addresses are mapped by the UBA to SBI addresses on a page-by-page basis, allowing UNIBUS data transfers to discontiguous pages of SBI memory. The SBI uses a 30-bit addressing scheme and a 32-bit wide data path, while the UNIBUS uses an 18-bit addressing scheme and a 16-bit data path. The SBI is synchronous, supporting a maximum of 16 NEXUSs while UNIBUS functions are asynchronous, supporting a large number of devices.

The UNIBUS Adapter accepts one of two forms of input from the UNIBUS:

- Hardware-generated interrupts
- Direct memory access transfers

Terminal input, for example, is an interrupt-driven process in which the DZ-11 (terminal interface) initiates an interrupt sequence. The interrupt service routine for the terminal driver will accept and process the data resulting from the terminal input. This process is therefore classified as a non-direct memory transfer.

In contrast, once initiated by the software, an RK06 disk will transfer its data directly to or from SBI memory via the UBA without processor intervention. The RK06, therefore, is a direct memory access (DMA) device. The direct memory access transfer may be further divided into two groups:

- Random access—access of noncontiguous addresses
- Sequential access—access of sequentially increasing addresses

The UNIBUS adapter can channel data through any one of 16 data paths for UNIBUS devices performing DMA transfers. The UBA provides a direct data path to allow UNIBUS transfers to random SBI addresses. Each UNIBUS transfer through the direct data path is mapped directly to an SBI transfer, thereby allowing only one word of information to be transferred during an SBI cycle. The UBA provides
15 buffered data paths (BDP), each of which allows a sequential access device on the UNIBUS (a device that transfers to consecutive increasing addresses) access to the SBI in a more efficient manner than that offered by the direct data path. Each of the BDPs stores data for the UNIBUS, so that four UNIBUS transfers are performed for each SBI transfer, making more efficient use of the SBI and memory. Using the BDPs, the UBA can support high-speed DMA block transfer devices such as the RK06 disk subsystem and the DMC-11. The Buffered Data Paths also allow a UNIBUS device to operate on random longword aligned 32-bit data.

**UNIBUS to SBI Address Translation**

The UNIBUS Adapter provides for direct memory access transfers to main memory via the memory controllers connected to the Synchronous Backplane Interconnect. The UNIBUS Adapter translates UNIBUS memory addresses to SBI addresses through a UNIBUS to SBI address translation map. The UNIBUS Adapter physically contains 496 (decimal) hardware map registers utilized in mapping UNIBUS memory page addresses to SBI page addresses (longwords). Each map register is assigned an SBI longword address. The map register contains the SBI page address and the data path required to transfer data between the UNIBUS and the SBI.

Each UNIBUS address is mapped to an SBI address in three sections:

1. SBI page address (one page equals 512 bytes).
2. Longword within an SBI page (one longword equals four bytes).
3. Word or byte within a longword.

**NOTE**

To avoid confusion between UNIBUS and SBI address bits, UNIBUS address bits will be shown as UA <bit num> and SBI address bits will be shown as SA <bit num>.

As illustrated in Figure 9-5, the UNIBUS to SBI page map translates UNIBUS memory page addresses to any SBI page address. The map allows the transfer of data to discontiguous pages of SBI memory. The map translates the nine UNIBUS page address bits (UA<17:9>) to the 21 SBI page address bits (SA<27:7>).

340
There are 496 map registers provided to map the entire UNIBUS memory address space at once, thereby reducing the problem of register allocation. Each map register corresponds to the UNIBUS page which is to be mapped. The map registers are available to the VAX-11 software as part of the SBI I/O address space. These registers are discussed in detail in the section titled SBI ADDRESSABLE UNIBUS ADAPTER REGISTERS.

UNIBUS address bits UA <08:02> determine the longword within a page and are seen by the SBI as address bits SA <06:00>. These seven bits are concatenated with the mapped page address to form the 28-bit SBI address.

The two low order UNIBUS address bits (UA<01:00>) and the two control bits (C<1:0>) determine the SBI function and byte mask (F<3:0>, M<3:0>).

The mask field points to either one or two bytes within the longword address. The function field selects either read or write and the associated qualifier. The mask and the function fields are illustrated in the following table. Table 18-6 illustrates the translation from the UNIBUS control and byte address fields to the SBI function and mask fields.
### UNIBUS Subsystem

#### Table 18-6 UNIBUS Field To SBI Field Translation

<table>
<thead>
<tr>
<th>UNIBUS</th>
<th>SBI</th>
<th>MASK</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTROL C&lt;1:0&gt; ADDRESS A&lt;1:0&gt;</td>
<td>FUNCTION FUNC&lt;3:0&gt;</td>
<td>M&lt;3:0&gt;</td>
</tr>
<tr>
<td>DATI</td>
<td>READ MASK</td>
<td>0 0 1 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 1 0 0</td>
</tr>
<tr>
<td>DATO</td>
<td>WRITE MASK</td>
<td>0 0 1 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 1 0 0</td>
</tr>
<tr>
<td>DATOB</td>
<td>WRITE MASK</td>
<td>0 0 0 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 1 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 1 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 0 0 0</td>
</tr>
<tr>
<td>DATIP</td>
<td>INTERLOCK READ MASK</td>
<td>0 0 1 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 1 0 0</td>
</tr>
<tr>
<td>followed by</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DATO</td>
<td>INTERLOCK WRITE MASK</td>
<td>0 0 1 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 1 0 0</td>
</tr>
<tr>
<td>OR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DATOB</td>
<td>INTERLOCK WRITE MASK</td>
<td>0 0 0 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 1 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 1 0 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 0 0 0</td>
</tr>
</tbody>
</table>

### UNIBUS ADAPTER DATA TRANSFER PATHS

Data is transferred between the UNIBUS and the SBI through one of the 16 data paths of the UNIBUS Adapter:

1. The direct data path (DDP) translates each UNIBUS data transaction (DATI, DATIP, DATO, DATOB) directly to an SBI function for each UNIBUS word (or byte) transfer, thereby transferring data between SBI memory and a UNIBUS device in 16-bit quantities.

2. The Buffered Data Paths allow fast, sequential access UNIBUS devices to access the SBI in a more efficient manner than is offered by the Direct Data Path. Each buffered data path (BDP1-15) accumulates data and transfers the data as words or bytes to or from the UNIBUS device. The BDPs perform quadword transfers (64 bits) to SBI memory addresses. The BDPs will respond to UNIBUS DATI, DATO, and DATOB functions but will not respond to the DATIP function.
UNIBUS Subsystem

3. The Buffered Data Paths also allow a UNIBUS device to operate on random 32-bit longword-aligned data.

The data path to be used by a particular device is assigned by the software when setting up the map registers. The data paths are numbered from DP0 to DP15. DP0 is the direct data path (DDP) and DP1 through DP15 are the buffered data paths, BDP1 through BDP15 respectively. One or more transferring UNIBUS devices can be assigned to DP0. No more than one transferring UNIBUS device, however, can be assigned to any one of the BDPs at any time. If, during a DMA transfer, the UNIBUS address points to an invalid map register or a map register that has a parity error within the high order 16 bits, the UNIBUS transfer will be aborted (SSYN Timeout in the UNIBUS device), and the bit indicating the problem will be set in the UBA status register (IVMR or MRPF). Note that for this implementation, the low order 16 bits of the map register are accessed only when an SBI transfer is required, and only at that time is parity checked on the low 16 bits of the map register.

Direct Data Path (DDP)

The Direct Data Path (DP0) translates each UNIBUS data transfer function (DATI, DATO, DATOB) to a unique SBI function (Read Mask, Write Mask). The DDP can transfer words or bytes directly between the UNIBUS and SBI memory. In addition, the DDP allows a UNIBUS device to interlock its operation with the system by translating a DATIP-DATO/DATOB UNIBUS sequence to an Interlock Read Mask Interlock Write Mask SBI sequence, thereby setting and clearing the memory interlock.

Each UNIBUS word (or byte) transfer is translated by the UNIBUS adapter to an SBI transfer. The UNIBUS transfer does not complete until the SBI transfer has been completed. The SBI address, function and byte mask are mapped directly from the UNIBUS address and control lines, and the state of an internal interlock flip flop in the case of a DATIP-DATO sequence.

Use of the Direct Data Path

- The Direct Data Path can be assigned to more than one transferring UNIBUS device.
- The DDP must be used by any device wanting to execute an interlock sequence (DATIP-DATO/DATOB) to the SBI.
- The Direct Data Path must be used by devices not transferring to consecutive increasing addresses or devices that mix read and write functions.
UNIBUS Subsystem

- The maximum throughput via the DDP is approximately 425K words per second for write operations and 316K words per second for read operations. These rates will decrease as other SBI activity increases.
- The DDP is the simplest data path, as far as programming goes, since the map registers are the only UNIBUS adapter registers required to be accessed when initiating a UNIBUS device transfer.

Table 18-7 illustrates the translation of UNIBUS to SBI data transfer operations.

<table>
<thead>
<tr>
<th>UNIBUS FUNCTION</th>
<th>TRANSFER DIRECTION</th>
<th>SBI FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATI</td>
<td>UBA to device</td>
<td>Read-masked (16 bits)</td>
</tr>
<tr>
<td>DATO or DATOB (byte)</td>
<td>device to UBA</td>
<td>Write-masked (8 or 16 bits)</td>
</tr>
<tr>
<td>DATIP then DATO or DATOB</td>
<td>UBA to device then device to UBA</td>
<td>Interlock Read-masked then Interlock Write-masked</td>
</tr>
</tbody>
</table>

Buffered Data Path (BDP)
There are 15 Buffered Data Paths, DP1-DP15. The Buffered Data Paths are provided for the following reasons:

1. To be used by fast DMA block transfer devices such as the RK07. The BDPs allow UNIBUS devices to make more efficient use of the SBI and memory and therefore improve system performance. The use of BDPs improves the effective UNIBUS bandwidth. The maximum throughput via the BDP is 695K words per second for both read and write operations. This rate will decrease as other SBI activity increases.
2. To enable word-aligned block transfer devices to begin and end on an odd byte of SBI memory. (Byte offset operation will be discussed under Byte Offset Data Transfers).
3. To allow a UNIBUS device to operate on random longword-aligned 32-bit data from SBI memory so that all 32 bits of the longword are read or written at the same time.

The software assigns a UNIBUS Transfer to a Buffered Data Path when it sets up the map registers corresponding to the transfer.
The software must assure that no more than one active transfer is assigned to a particular BDP at any time.

A UNIBUS device transfer using the Buffered Data Path must have the following properties:

1. It must be a block transfer. (A block is greater than or equal to one byte). BDP maintenance (purge) will be initiated by the software following each block transfer. The purge operation is a software-initiated function of the UBA that clears the BDPs of any remaining bytes of data. These bytes will be transferred to SBI Memory for UNIBUS to Memory Write operations or cleared for UNIBUS to Memory Read Operations.

2. All transfers within a block must be to consecutive increasing addresses.

3. All transfers within a block must be of the same function type, Memory Read (DATI) or Memory Write (DATO or DATOB). The DATIP UNIBUS function will not be recognized by the BDP. A SSYN Timeout will result in a device attempting a DATIP to a BDP.

Each BDP contains eight bytes of DATA buffering, forming a quad-word-aligned memory image. DATA is transferred between the UNIBUS and a BDP as words or bytes. Data is transferred between the BDP and SBI memory as quadwords or between the BDP and an SBI I/O register as longwords.

The Buffered Data Paths are transparent to the UNIBUS device. The device will perform its transfer as if transferring directly to memory.

The operation of the BDPs is described in the following section.

**UNIBUS Data Transfers to Memory**

As a UNIBUS device transfers data to memory (DATO,DATOB) via a BDP, the BDP will store the data and complete the UNIBUS cycle. The Buffered Data Paths are implemented so that a quadword image is formed in the BDP before an SBI cycle is initiated. When the UNIBUS device addresses the last byte or word of a physical quadword, the UBA will complete the data cycle and the BDP will perform an extended write operation, thereby transferring the stored bytes of data. The SBI transfer will be completed before recognizing additional UNIBUS transfers. The BDP will set its Buffer Not Empty (BNE) bit whenever a UNIBUS Write to the BDP is performed, and clear the BNE bit each time the SBI transfer is executed. The BNE bit indicates whether or not valid data is contained in the BDP. Figure 18-6 illustrates a Buffered Data Path transfer. In this illustration, a Buffered Data Path transfer of four 16-bit data words to the Buffered Data Path takes place. The fourth data transfer initiates the extended write transfer of all 64 bits to memory.
UNIBUS Subsystem

UNIBUS TRANSFERS

DATA TO ADDRESS (HEX)

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

BNE STATE

EMPTY

DATA XXXX0
16 BITS

WORD 0

DATA XXXX2
16 BITS

WORD 1

DATA XXXX4
16 BITS

WORD 2

DATA XXXX6
16 BITS

WORD 3

DATA XXXX8
16 BITS

WORD 4

DATA XXXXA
16 BITS

WORD 5

DATA XXXX2
16 BITS

WORD 6

DATA XXXXXE
16 BITS

WORD 7

64 BITS

SBI EXTENDED WRITE TO MAPPED UNIBUS ADDRESS (MEMORY)

Figure 18-6  UNIBUS Transfer to Memory
The BDP stores the UNIBUS address of data contained in the BDP. The BDP stores the UNIBUS address of the current transfer in order to transfer the remaining bytes to memory at the end of a block transfer. This is the purge function that will be discussed in a later section.

The BDP also stores the type of function and the state of each byte of the data buffer (buffer state). The buffer state is transmitted as the SBI mask bits during the BDP to SBI write cycle so that only the correct bytes will be written into memory.

**UNIBUS Data Transfers From Memory**
As a UNIBUS device performs Memory Read operations (DATI) via a BDP, the BDP tests the state of its data buffers. If the buffers do not contain data for the UNIBUS transfer, the BDP will initiate an Extend Read operation to memory. The BDP will then transfer data for the current cycle to the UNIBUS, thereby completing the UNIBUS cycle, store the remaining bytes in its buffers, and set the BNE bit. If the data for the current UNIBUS cycle is available in the data buffers, then the BDP will pass the data to the UNIBUS and complete the cycle. The BDP will prefetch the next quadword of data (Extended Read Transfer) after each UNIBUS access to the last word of a quadword-aligned group. The Buffer Not Empty (BNE) bit is cleared by the BDP before the prefetch and set when the Read Data returns, thereby indicating the state of the BDP. Figure 18-7 illustrates the Buffered Data Path transfer from memory to the UNIBUS.

**Programming Note:**
Since the prefetch allows the possibility of the UBA crossing a page boundary into nonexistent memory, resulting in a 100 μs timeout, it is recommended that the software allocate an additional map register following a block. This map register must be invalidated. When the prefetch crosses this page boundary to the invalid map register, the prefetch will be aborted immediately, thereby eliminating the 100 μs timeout. The UBA does not record any UBA or SBI errors that may occur during the prefetch operations since this is an anticipatory function based on the next probable address. If an error does occur then the prefetch will be aborted and the BDP will not be filled with data. If the UNIBUS device accesses the same BDP again, then the BDP to SBI read will be initiated and any errors that occur will be logged at this time.

**Byte Offset Data Transfers**
The BDPs enable word-aligned UNIBUS devices (devices beginning transfers on word boundaries and transferring an integral number of words) to begin and end a block transfer at an ODD byte of SBI memory. To use this feature, the software will set the Byte Offset bit of
UNIBUS Subsystem

Figure 18-7  UNIBUS Transfers From Memory
the map registers involved during the devices transfer.

When the Byte Offset bit is set for a transfer using the BDPs, the BDP will, in effect, increase the SBI memory address by one byte. The data will appear on the UNIBUS in the byte or word indicated by the UNIBUS Address. The data will appear on the SBI shifted to the left (increased) by one byte. The UNIBUS adapter will distribute the data, and adjust the SBI address and byte mask so that the data will get to or come from the correct memory location. This operation is transparent to the UNIBUS device.

Figure 18-8 shows the relative position of data being transferred between a UNIBUS device and SBI memory. Figure 18-8 top shows the relative positions without Byte Offset and Figure 18-8 bottom shows the position with Byte Offset.

---

**Figure 18-8  Relative Position of Data Between UNIBUS and SBI**

"EACH LETTERED BOX REPRESENTS 1 BYTE (8 BITS)"
UNIBUS Subsystem

Purge Operation
The purge operation is a software-initiated function of the UBA in which the Buffered Data Paths are purged of data and initialized. The Buffered Data Path used by a UNIBUS device must be purged at the completion of the device's transfer. The software initiates the purge by writing a one to the BNE bit of the data path register (DPR) corresponding to the Buffered Data Path to be purged. The UBA will perform the following, depending on the transfer function that was being performed by the BDP:

1. Writes to memory. If there are any remaining bytes of data in the BDP, this data will be transferred to memory. The UBA will then clear the BNE bit, function bit and buffer state bits and leave the BDP in its initialized state. If an error occurs during this transfer, the Buffer Transfer Error bit of the data path register will be set, indicating that the data was not successfully transferred to memory. Software must clear this bit before the BDP can be used again. If there were no data remaining in the Buffered Data Path, then the buffer is left in its initialized state.

2. Reads from memory. The UBA will initialize the BDP by clearing the BNE bit of the DPR.

Longword-Aligned 32-Bit Random Access Mode
The UNIBUS adapter can be used in a mode so that a UNIBUS device can operate on random longword-aligned 32-bit quantities without requiring purge operations. This mode is selected by setting the longword access enable (LWAE) bit 26 of the map register corresponding to the UNIBUS transfer. A Buffered Data Path must be selected for this operation.

In this mode, a UNIBUS device must first operate on the low order word of the longword and then the high order word. An operation is considered to be a read from memory (DATI) or a write to memory (DATO) or a read/write (DATI/DATO). The UNIBUS DATIP function code is not valid for transfers using Buffered Data Paths, and any device performing the DATIP through a Buffered Data Path will receive an SSYN timeout (NXM).

The Buffered Data Path will not perform the prefetch operation when this mode is enabled, thereby allowing for random access of longword-aligned 32-bit quantities. This mode eliminates the need for
UNIBUS Subsystem

the purge operation at the completion of the transfer, providing the
UNIBUS device operates on both words of the longword and operates
on them in order (i.e., low word, then high word).

Maximum throughput in this mode is approximately 1.7 Mbyte/sec as
illustrated in Figure 18-9.

<table>
<thead>
<tr>
<th>FIRST WORD TO BDP</th>
<th>SECOND WORD TO BDP TO SBI</th>
</tr>
</thead>
<tbody>
<tr>
<td>800 ns</td>
<td>2.6 US MIN.</td>
</tr>
</tbody>
</table>

3.4 US MIN. PER WORD (4 BYTES) = 1.17 MBYTE/SEC. MAX.

Figure 18-9  Random Access Mode Throughput

The operation of the UBA for the longword-aligned 32-bit access
mode is determined by the function (DATI, DATO/DATOB) and ad-
dress (A1, A0) received from the UNIBUS and the state of the buffer
not empty (BNE) bit of the data path register, corresponding to the
Buffered Data Path being used for this operation, within the UBA.
(BNE SET = buffer not empty, BNE CLEAR = buffer empty).

The following statements summarize the operation of the UBA for the
longword-aligned 32-bit random access mode of operation.

DATI Functions
1. SBI reads will occur when a DATI operation is received and the
   Buffered Data Path is empty (BNE = 0).
2. The BNE bit will be set in response to a successful SBI read
   generated by a DATI operation to the low order word (A1 = 0).
   Longword data from memory is stored in the Buffered Data Path.
3. The BNE bit will be cleared by a DATI operation to the high order
   word (A1 = 1).
4. If the BNE bit is set, data from the Buffered Data Path will be
   returned to the UNIBUS device.

DATO/DATOB Functions
1. The BNE bit will be set by a DATO or DATOB operation. The data
   from the UNIBUS device will be stored in the Buffered Data Path
   and the byte mask bit is set within the data path register to indi-
   cate the bytes or words that have been written by the UNIBUS
   device.
2. SBI writes will occur when a DATO operation occurs to the high
order word or when a DATOB operation occurs to the high order byte. The bytes or words that were written (i.e., those for which the byte mask bits are set) are written into main memory.

3. The BNE bit will be cleared after a SBI write operation.

The UBA operations per UNIBUS access, as a function of BNE and received UNIBUS function and address for this mode of operation are:

<table>
<thead>
<tr>
<th>Present BNE State</th>
<th>Function</th>
<th>A1,A0</th>
<th>UBA Operations</th>
<th>Next BNE State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DATI</td>
<td>0 X</td>
<td>SBI READ,RETURN LOW WORD, STORE DATA</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>DATI</td>
<td>1 X</td>
<td>SBI READ, RETURN HIGH WORD</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>DATI</td>
<td>0 X</td>
<td>RETURN LOW WORD</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>DATI</td>
<td>1 X</td>
<td>RETURN HIGH WORD</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>DATO</td>
<td>0 X</td>
<td>STORE LOW WORD</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>DATO</td>
<td>1 X</td>
<td>STORE HIGH WORD,SBI WRITE</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>DATO</td>
<td>0 X</td>
<td>STORE LOW WORD</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>DATO</td>
<td>1 X</td>
<td>STORE HIGH WORD,SBI WRITE</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>DATOB</td>
<td>0 0</td>
<td>STORE BYTE 0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>DATOB</td>
<td>0 1</td>
<td>STORE BYTE 1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>DATOB</td>
<td>1 0</td>
<td>STORE BYTE 2</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>DATOB</td>
<td>1 1</td>
<td>STORE BYTE 3,SBI WRITE</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>DATOB</td>
<td>0 0</td>
<td>STORE BYTE 0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>DATOB</td>
<td>0 1</td>
<td>STORE BYTE 1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>DATOB</td>
<td>1 0</td>
<td>STORE BYTE 2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>DATOB</td>
<td>1 1</td>
<td>STORE BYTE 3,SBI WRITE</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>DATIP</td>
<td>X X</td>
<td>UBA DOES NOT RESPOND (NXM TO UNIBUS DEVICE) NO CHANGE</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>DATIP</td>
<td>X X</td>
<td>UBA DOES NOT RESPOND (NXM TO UNIBUS DEVICE) NO CHANGE</td>
<td>1</td>
</tr>
</tbody>
</table>

To enable this mode of operation, Bit 26 of the map register has been changed to the Longword Access Enable (LWAE) bit. This bit when set and when a buffered data path is selected, will enable the longword-
aligned 32-bit random access mode. It is a read/write bit and is cleared on initialization.

Programming the UBA for longword-aligned random access mode requires loading the map registers with the following data:

BIT<31> MRV Map register valid, must be set.
BIT<30:27> LWAE Longword access enable, must be set. Ignored during Direct Data Path transfers.
BIT<25> BO Byte offset, must be zero.
BIT<24:21> DPDB Data path designator bits, must use a buffered data path, BDP1-BDP15. LWAE bit is ignored when DPDB = 0 (Direct Data Path).
BIT<20:0> PFN Page frame number, SBI page address.

The allowed UNIBUS sequences for this mode of operation are:

A1,A0

1. DATI 0 0 SBI READ—Low word is returned to UNIBUS device. Both words are stored in BDP. BNE bit is set.
   DATI 1 0 High word from BDP is returned to UNIBUS device. BNE is cleared.

2. DATO 0 0 Low word is written to BDP. BNE bit is set.
   DATO 1 0 SBI WRITE—High word is written to BDP—then low word and high word are transferred to memory. BNE bit is cleared.

3. DATOB 0 0 Byte 0 is written to BDP, BNE is set.
   DATOB 0 1 Byte 1 is written to BDP, BNE is set.
   DATOB 1 0 Byte 2 is written to BDP, BNE is set.
   DATOB 1 1 Byte 3 is written to BDP, SBI WRITE.

4. DATI 0 0 SBI READ—Low word is returned to UNIBUS device. Both words are stored in BDP.
   DATO 0 0 Low word of BDP is written by UNIBUS device.
   DATI 1 0 High word from BDP is returned to UNIBUS device.
UNIBUS Subsystem

DATO 1 0 SBI WRITE-High word of BDP is written by UNIBUS device and modified longword is returned to the memory.

DATO 1 0 SBI WRITE-High word of BDP is written by UNIBUS device and modified longword is returned to the memory.

Additional BDP Software Information

1. For purge operations in which data is transferred to memory, the SBI transfer takes about 2 μs. The UBA will not respond to Data Path Register Read during this period (Busy Confirmation), thus preventing a race condition when testing for the BNE bit to be cleared.

2. The Buffer Transfer Error bit (BTE) of the data path registers indicates that an error occurred during an operation involving a buffered data path. Once this bit is set, UNIBUS transfers using the BDP will be aborted until the bit is cleared by the software. The purge operation does not clear the BTE bit.

3. Any purge operations initiated by the software to BDPs for which the purge or initialization is not required are treated by the UBA as a NO-OP.

4. A purge operation to Data Path Register 0 (Direct Data Path) is treated by the UBA as a NO-OP.

INTERRUPTS

SBI interrupts can be generated from two sources within the UNIBUS subsystem: either from a UNIBUS device or from the UNIBUS adapter.

Interrupts from the UNIBUS can occur at any one of the four request levels, as determined by the UNIBUS BR lines. Interrupts from the UNIBUS adapter will occur at one assigned request level. This level is assigned by backplane jumper.

The UNIBUS adapter contains one request sublevel. The UBA will therefore require four of the 64 possible SBI interrupt vectors (1 for each of the 4 required levels). The four vectors will each “point” to a UBA Service Routine corresponding to an interrupt request level. Each UBA service routine must read and test the BR Receive Vector Register corresponding to the level of interrupt:

- BRRVR 7 for Req Level 7
- BRRVR 6 for Req Level 6
- BRRVR 5 for Req Level 5
- BRRVR 4 for Req Level 4
From the contents of the BRRVRs, the UBA service routine will determine whether the interrupt was generated from within the UBA Status Register, from the UNIBUS device, or from both. The UBA service routine can then service the interrupt as determined by testing the contents of the BRRVR.

<table>
<thead>
<tr>
<th>Bit &lt;31&gt;</th>
<th>Bits &lt;15:0&gt;</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>No service required.</td>
</tr>
<tr>
<td>0</td>
<td>V</td>
<td>UNIBUS service as indicated by vector V received from the UNIBUS device (UNIBUS device Interrupt Service Routine).</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>UNIBUS Adapter service required. Read configuration register and status register to determine the service required.</td>
</tr>
</tbody>
</table>
| 1        | V           | UNIBUS and UNIBUS Adapter service required.  
|          |             | 1. Save the vector V (received from the UNIBUS device).  
|          |             | 2. Read UBA configuration register and status register.  
|          |             | 3. Perform UBA service as indicated by configuration and status register.  
|          |             | 4. Index into UNIBUS device service routine with vector V. |

V is the vector field of the BRRVR received from the UNIBUS device. Zero is the null vector indicating that a vector was not received from the UNIBUS device.

Software Note: The zero vector resulting from an SBI Interrupt Summary Read must be reserved and interpreted as a Passive Release Condition.

**Interrupts from the UNIBUS**

The UBA will translate the UNIBUS BR interrupts to SBI request interrupts, providing the Interrupt Fielder Switch (IFS) bit and the BR Interrupt Enable (BRIE) bit of the UBA control register are set. The assertion of the SBI request lines will initiate an SBI interrupt transaction vectoring to the UBA interrupt service routine. This routine will then read the BR Receive Vector Register (BRRVR) corresponding to the level of the interrupt. On receiving the read BRRVR command, the UBA will test that the following conditions are true:

1. The UNIBUS BR line corresponding to the BRRVR number is asserted.
2. The BRRVR does not contain an already valid vector.
3. UNIBUS AC LO is not asserted.

If all of the three conditions are met, then the UBA will issue the UNIBUS Bus Grant and complete the UNIBUS interrupt transaction. The BRRVR is loaded with the interrupt vector by the successful completion of the interrupt transaction. The device vector received during the transaction will be sent as the read data to the BRRVR Read Command. If a UBA interrupt is active then the vector will be sent as a negative quantity (bit 31 sent as a one).

The BRRVR is cleared by the successful completion of the SBI Read Data Cycle, otherwise the vector is saved and the BRRVR remains full. If conditions 1, 2, 3 are not met then the contents of the BRRVR (either the stored vector, from a previously failing SBI read data cycle, or zero) will be sent as read data. If a UBA interrupt is active then bit 31 will be sent as a one.

The following sequence is performed for UNIBUS device interrupts:

1. A bus request line is asserted by the UNIBUS device.
2. The UNIBUS adapter asserts the SBI request line, corresponding to the UNIBUS BR line, to initiate the interrupt transaction in the CPU.
3. When the interrupt summary read corresponding to the above request level is seen by the UNIBUS adapter, the UNIBUS adapter asserts the request sublevel assigned to the UBA.
4. The CPU will then transfer control to the UNIBUS adapter interrupt service routine.
5. The UNIBUS interrupt service routine will execute a read to the BR receive vector register corresponding to the level of interrupt.
6. The UNIBUS adapter will issue the UNIBUS Bus Grant corresponding to the level of the interrupt being serviced providing the following conditions are met: Adapter interrupt is not pending; BR line corresponding to the BRRVR is asserted; the BRRVR does not contain a previous vector.
7. The UNIBUS interrupt transaction is completed, the vector is loaded into the corresponding BRRVR, and the vector is given to the UNIBUS Interrupt Service Routine as a Read DATA.
8. The BRRVR will be cleared when the ACK Confirmation is received for the Read DATA.
9. The UNIBUS interrupt service routine will then dispatch to the UNIBUS device service routine (or service the UBA) as indicated by the received interrupt vector.
UNIBUS Subsystem

NOTE
The UNIBUS adapter interrupt service routine (UBASR) is the routine that will interface the CPU interrupt process to the individual UNIBUS device service routines. This routine will provide the additional level of dispatch required for UNIBUS-initiated interrupts.

Failure to Complete the UNIBUS Interrupt Transaction
If for some reason, the UNIBUS initiated an interrupt transaction and then fails to complete (i.e., passive release), the interrupt vector will not be loaded into the interrupt vector register. The following mechanism will allow the UNIBUS interrupt service routine to gracefully exit. The idle state of the BRRVR is zero. If, when reading the interrupt vector register, the UNIBUS interrupt service routine receives the zero vector, it will log an error (if desired) and return from the service routine.

Once successfully loaded, the BRRVR will maintain the interrupt vector until an ACK confirmation to the BRRVR Read Data has been received, or an Adapter Init sequence is initiated. If the ACK confirmation is not received for the Read Data then the BRRVR full bit will not be cleared, and subsequent reads to that BRRVR will result in the stored vector being returned for the Read Data until ACK is received for the Read Data.

Interrupts from the UNIBUS Adapter to the SBI
When the UNIBUS adapter interrupt enable bit is set, and a condition warranting an interrupt occurs in the UNIBUS Adapter, the following sequence occurs:
1. The UNIBUS adapter asserts its assigned request line.
2. When the Interrupt Summary Read, corresponding to the above request level, is seen by the UNIBUS adapter, the request sublevel assigned to the UNIBUS adapter is sent to the CPU as an Interrupt Summary Response.
3. With this information, request level and request sublevel, the CPU can dispatch to the UNIBUS adapter service routine, which will then read the BR Receiver Vector Register corresponding to the level of interrupt. The BRRVR will contain a negative value (bit 31 set).
4. The UBA service routine will detect the negative value and branch to a routine that will read the Configuration Register and Status Register to determine the service required.

The request line will remain asserted until all conditions (bits of the UNIBUS adapter status register) have been cleared by the software.
UNIBUS Subsystem

UNIBUS ADAPTER (NEXUS) REGISTER SPACE
Each NEXUS register address space occupies 16 pages (512 bytes/page) of Synchronous Backplane Interconnect I/O address space. The address location of the UNIBUS adapter is determined by the transfer request priority number assigned to the adapter. The transfer request number is determined by electrical jumpers on the NEXUS backplane and may vary from one system configuration to the next.
Table 18-8 illustrates the physical base address and SBI base address for a NEXUS assigned to any one of the SBI transfer request numbers.

Table 18-8 Transfer Number Address Assignments

<table>
<thead>
<tr>
<th>SBI Transfer Request Number</th>
<th>Address Base Physical (Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20002000</td>
</tr>
<tr>
<td>2</td>
<td>20004000</td>
</tr>
<tr>
<td>3</td>
<td>20006000</td>
</tr>
<tr>
<td>4</td>
<td>20008000</td>
</tr>
<tr>
<td>5</td>
<td>2000A000</td>
</tr>
<tr>
<td>6</td>
<td>2000C000</td>
</tr>
<tr>
<td>7</td>
<td>2000E000</td>
</tr>
<tr>
<td>8</td>
<td>20010000</td>
</tr>
<tr>
<td>9</td>
<td>20012000</td>
</tr>
<tr>
<td>10</td>
<td>20014000</td>
</tr>
<tr>
<td>11</td>
<td>20016000</td>
</tr>
<tr>
<td>12</td>
<td>20018000</td>
</tr>
<tr>
<td>13</td>
<td>2001A000</td>
</tr>
<tr>
<td>14</td>
<td>2001C000</td>
</tr>
<tr>
<td>15</td>
<td>2001E000</td>
</tr>
</tbody>
</table>

Table 18-9 lists each of the UNIBUS Adapter registers and its associated physical address offset.

Table 18-9 UNIBUS Adapter Register Address Offset

<table>
<thead>
<tr>
<th>UNIBUS Adapter Register</th>
<th>Byte Offset (Physical Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration Register</td>
<td>000</td>
</tr>
<tr>
<td>UNIBUS Adapter Control Register</td>
<td>004</td>
</tr>
<tr>
<td>UNIBUS Adapter Status Register</td>
<td>008</td>
</tr>
<tr>
<td>Diagnostic Control Register</td>
<td>00C</td>
</tr>
</tbody>
</table>

358
### UNIBUS Subsystem

#### UNIBUS Adapter Register

<table>
<thead>
<tr>
<th>Description</th>
<th>Byte Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failed Map Entry Register</td>
<td>010</td>
</tr>
<tr>
<td>Failed UNIBUS Address Register</td>
<td>014</td>
</tr>
<tr>
<td>Failed Map Entry Register</td>
<td>018</td>
</tr>
<tr>
<td>Failed UNIBUS Address Register</td>
<td>01C</td>
</tr>
<tr>
<td>Buffer Selection Verification Register 0</td>
<td>020</td>
</tr>
<tr>
<td>Buffer Selection Verification Register 1</td>
<td>024</td>
</tr>
<tr>
<td>Buffer Selection Verification Register 2</td>
<td>028</td>
</tr>
<tr>
<td>Buffer Selection Verification Register 3</td>
<td>02C</td>
</tr>
<tr>
<td>Buffer Receive Vector Register 4</td>
<td>030</td>
</tr>
<tr>
<td>Buffer Receive Vector Register 5</td>
<td>034</td>
</tr>
<tr>
<td>Buffer Receive Vector Register 6</td>
<td>038</td>
</tr>
<tr>
<td>Buffer Receive Vector Register 7</td>
<td>03C</td>
</tr>
<tr>
<td>Data Path Register 0</td>
<td>040</td>
</tr>
<tr>
<td>Data Path Register 1</td>
<td>044</td>
</tr>
<tr>
<td>Data Path Register 14</td>
<td>078</td>
</tr>
<tr>
<td>Data Path Register 15</td>
<td>07C</td>
</tr>
<tr>
<td>Reserved</td>
<td>080</td>
</tr>
<tr>
<td>Reserved</td>
<td>7EC</td>
</tr>
<tr>
<td>Map Register 0</td>
<td>800</td>
</tr>
<tr>
<td>Map Register 1</td>
<td>804</td>
</tr>
<tr>
<td>Map Register 494</td>
<td>EB8</td>
</tr>
<tr>
<td>Map Register 495</td>
<td>EBC</td>
</tr>
<tr>
<td>Reserved</td>
<td>EC0</td>
</tr>
<tr>
<td>Reserved</td>
<td>EFC</td>
</tr>
</tbody>
</table>

The offset within the UNIBUS Adapter Address Space is shown for each of the UNIBUS adapter registers with respect to the physical address. As described in Table 18-9, the addresses of all other UNIBUS Adapter Registers are relative to the configuration register address by an offset. The base address of the configuration register is
the physical base address described in Table 18-8. Therefore, the byte offset for the configuration register in Table 18-9 is 000.

**SBI ADDRESSABLE UNIBUS ADAPTER REGISTERS**
The UNIBUS adapter registers occupy eight pages of the SBI I/O address space. These registers fall into four categories: map registers, data path registers, interrupt vector registers and control and status registers. The UNIBUS adapter registers are all 32-bit registers and can only be written as longwords. These registers will, however, respond to byte or word read commands. These registers will also respond to the Interlock Read/Interlock Write sequence but will not affect the interlock of the SBI. The following sections discuss the function and content of each of the UNIBUS adapter registers.

**Configuration Register (CNFGR)**
The configuration register contains the SBI fault bits, the UNIBUS adapter and UNIBUS environment status bits, and the UNIBUS adapter code. This register is required to interface with the SBI. Figure 18-10 illustrates the configuration register.

![Configuration Register Bit Configuration](image)

Figure 18-10  Configuration Register Bit Configuration

The contents of the Configuration Register are as follows:

**Bit: 31:27  Name: SBI faults**

**Function:** These bits are set when the UNIBUS adapter detects specific fault conditions on the SBI. These bits cannot be set once FAULT has been asserted. The negation of FAULT and the disappearance of the fault conditions clear the bits. The setting of any of the bits <31:
26> will cause the UNIBUS adapter to assert the FAULT signal on the SBI.

**Bit: 31**  **Name:** Parity Fault (PAR FLT)
**Function:** PAR FLT is set when the UNIBUS adapter detects an SBI parity error.

**Bit: 30**  **Name:** Write Sequence Fault (WSQ FLT)
**Function:** WSQ FLT is set when the UNIBUS adapter receives a Write Masked, Extended Write Masked, or Interlock Write Masked command which is not immediately followed by the expected write data.

**Bit: 29**  **Name:** Unexpected Read Data Fault (URD FLT)
**Function:** URD FLT is set when the UNIBUS adapter receives data for which a Read Masked, Extended Read, or Interlock Read Masked command has not been issued.

**Bit: 28**  **Name:** Interlock Sequence Fault (ISQ FLT)
**Function:** ISQ FLT is set when an Interlock Write Masked command or a UNIBUS address space is received by the UNIBUS adapter without a previous Interlock Read Masked command.

**Bit: 27**  **Name:** Multiple Transmitter Fault (MXT FLT)
**Function:** MXT FLT is set when the UNIBUS adapter is transmitting on the SBI and the IB bits transmitted by the UNIBUS adapter do not match those latched from the SBI. The lack of correspondence indicates a multiple transmitter condition.

**Bit: 26**  **Name:** Transmit Fault (XMT FLT)
**Function:** XMT FLT is set if the UNIBUS adapter was the transmitter during a detected fault condition. When the software subsequently reads the configuration and status registers of each of the NEXUSs on the SBI in order to identify the source of the fault, the UNIBUS adapter will be identified as that source if bit <26> is set.

**Bit: 25:24**  **Name:** Reserved and zero
**Function:** Bits <23,22,18,17,16> are UNIBUS Subsystem Environmental Status Bits. If any of these bits are set and the Configuration Interrupt Enable bit (CNFIE) of the control register is also set, then the UNIBUS adapter will initiate an SBI interrupt request at the level assigned to the UNIBUS adapter.

**Bit: 23**  **Name:** Adapter Power Down (AD PDN)
**Function:** This bit is set when the UNIBUS Adapter power supply asserts AC LO. It is cleared by writing a one to the bit location or when the Adapter Power Up bit is set.

**Bit: 22**  **Name:** Adapter Power Up (AD PUP)
**Function:** This bit is set by the negation of power supply AC LO. It is cleared by writing a one to the bit location or by setting the Adapter
Power Down bit.

Bit: 21:19  Name:  Reserved and zero

Function:

Bit: 18  Name:  UNIBUS INIT Asserted (UB INIT)
Function:  The assertion of UNIBUS Init will set this bit. It is cleared by the setting of the UNIBUS Initialization Complete bit (UBIC) or by the writing of a one to this bit location.

Bit: 17  Name:  UNIBUS Power Down (UB PDN)
Function:  This bit is set when UNIBUS AC LO is asserted. It indicates that the UNIBUS has initiated a power down sequence. The setting of the UNIBUS initialization complete bit or writing a one to this location will clear UB PDN.

Bit: 16  Name:  UNIBUS Initialization Complete (UBIC)
Function:  This bit is set by a successful completion of a power up sequence on the UNIBUS. It is the last of the status bits to be set during a UNIBUS adapter initialization sequence, and it can be interpreted to mean that the UNIBUS adapter and the UNIBUS are ready. The assertion of UNIBUS AC LO or UNIBUS INIT, or the writing of a one to this bit location will clear UBIC.

Bit: 15:8  Name:  Reserved

Function:

Bit: 7:0  Name:  Adapter Code
Function:  These bits define the code assigned to the UNIBUS adapter. Table 18-10 shows the bit assignment.

### Table 18-10  Adapter Code Bit Assignment

<table>
<thead>
<tr>
<th>BIT NUMBER</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIBUS ADDRESS SPACE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Adapter code bits 1 and 0 are determined by backplane jumpers and indicate the starting address of the UNIBUS Address space associated with the UNIBUS Adapter, as shown in Table 18-11.
Table 18-11  Selectable UNIBUS Starting Addresses

<table>
<thead>
<tr>
<th>UNIBUS ADDRESS SPACE</th>
<th>STARTING ADDRESS OF THE UNIBUS ADDRESS SPACE, BASE 16 (PHYSICAL BYTE ADDRESS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20100000(16)</td>
</tr>
<tr>
<td>1</td>
<td>20140000(16)</td>
</tr>
<tr>
<td>2</td>
<td>20180000(16)</td>
</tr>
<tr>
<td>3</td>
<td>201C0000(16)</td>
</tr>
</tbody>
</table>

Note that the lowest two bits of the Configuration Register (Vb and Va) correspond to SBI address bits 16 and 17.

Control Register (UACR)
The UNIBUS Adapter Control Register enables the software to control operations both on the UNIBUS Adapter and on the UNIBUS. All bits except for the Adapter INIT bit are set by writing a 1 and cleared by writing a 0 to the bit location. The Adapter INIT bit is set by writing a one to the bit location and is self clearing. Figure 18-11 shows the Control Register bit configuration.

![Control Register Bit Configuration](image)

Figure 18-11  Control Register Bit Configuration

The contents of the control register are as follows:

**Bit: 31**  **Name:** Reserved and zero

**Function:**

**Bit: 30:26**  **Name:** Map Register Disable <4:0> (MRD)

**Function:** This field of five read/write bits disables map registers in groups of 16, according to the binary value contained in the field. The MRD bits prevent double addressing if UNIBUS memory is used. This field is loaded with a binary value equal to the number of 8 Kbyte units of memory attached to the UNIBUS, as shown in Table 18-12.
DMA transfers to addresses controlled by disabled map registers are not recognized by the UNIBUS adapter. No error bits are set and no transfers are initiated. However, SBI access to disabled map registers is permitted. The MRD field is initialized as zero, with all map registers enabled.

**Bit:** 25:7  **Name:** Reserved and zero  
**Function:**

**Bit:** 6  **Name:** Interrupt Field Switch (IFS)  
**Function:** This bit determines whether interrupts from a UNIBUS device on the UNIBUS outside of the UNIBUS adapter will be fielded by the VAX-11 CPU or passed to the UNIBUS inside of the UNIBUS adapter. If the bit is set (1), then the interrupt will be passed to the SBI, if the BR Interrupt Enable bit of the control register is set. If the bit is cleared (0), then the interrupt will be passed to the UNIBUS inside of the UNIBUS adapter, where it is in effect ignored.

The power up state of the IFS bit is 0. The bit is also cleared by the adapter unit and SBI dead signals. This bit and BRIE must be set by the software to receive UNIBUS device interrupts.

**Bit:** 5  **Name:** Bus Request Interrupt Enable (BRIE)  
**Function:** When this bit is set it allows the UNIBUS adapter to pass interrupts from the UNIBUS to the VAX-11 CPU. The power up state of the BRIE bit is 0. The bit is also cleared by the Adapter INIT, SBI UNJAM, and SBI Dead signals. This bit and IFS must be set by the software to receive UNIBUS device interrupts.

**Bit:** 4  **Name:** UNIBUS to SBI Error Field Interrupt Enable (USEFIE)  
**Function:** The USEFIE bit enables an interrupt request to the VAX-11

---

**Table 18-12 Map Register Disable Bit Functions**

<table>
<thead>
<tr>
<th>MRD &lt;4:0&gt;</th>
<th>AMOUNT OF UNIBUS</th>
<th>MAP REGISTERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>0 K</td>
<td>NONE</td>
</tr>
<tr>
<td>00001</td>
<td>4 K</td>
<td>0 TO 15 (10)</td>
</tr>
<tr>
<td>00010</td>
<td>8 K</td>
<td>0 TO 31 (10)</td>
</tr>
<tr>
<td>00011</td>
<td>12 K</td>
<td>0 TO 47 (10)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00100</td>
<td>120 K</td>
<td>0 TO 480 (10)</td>
</tr>
<tr>
<td>00101</td>
<td>320 K</td>
<td>0 TO 495 (10)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01000</td>
<td>512 K</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01001</td>
<td>1024 K</td>
<td></td>
</tr>
<tr>
<td>01010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01111</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

364
UNIBUS Subsystem

CPU whenever any of the following Status Register bits is set on a DMA transfer.

RDTO (Read Data Time Out)
RDS (Read Data Substitute)
CXTER (Command Transmit Error)
CTTO (Command Transmit Time Out)
DPPE (Data Path Parity Error)
IVMR (Invalid Map Register)
MRPF (Map Register Parity Failure)

The power up state of the USEFIE (UNIBUS Error Field Interrupt Enable) bit is zero. SBI UNJAM and Adapter Init will clear the bit.

Bit: 3 Name: SBI To UNIBUS Error Field Interrupt Enable (SUEFIE)
Function: If this bit is set, the UNIBUS adapter will generate interrupt requests to the VAX-11 CPU when one of the two bits in the SBI to UNIBUS data transfer error field is set.

UBSTO (UNIBUS Select Time Out)
UBSSYNTO (UNIBUS Slave Sync Time Out)

The power up state of this bit is zero. SBI UNJAM, SBI Dead, and Adapter INIT will clear the bit.

Bit: 2 Name: Configuration Interrupt Enable (CNFIE)
Function: If this bit is set, the UNIBUS adapter will initiate an interrupt request to the VAX-11 CPU whenever any one of the environmental status bits of the configuration register is set.

AD PDN (Adapter Power Down)
AD PUP (Adapter Power Up)
UB INIT (UNIBUS INIT Asserted)
UB PDN (UNIBUS Power Down)
UBIC (UNIBUS Initialization Complete)

The power up state of this bit is set (1). The bit is cleared by Adapter INIT, SBI UNJAM, and SBI Dead.

Bit: 1 Name: UNIBUS Power Fail (UPF)
Function: When this bit is set, it initiates a power fail sequence on the UNIBUS, asserting AC LO, DC LO, and INIT. The software uses this bit to initialize the UNIBUS. The UNIBUS will remain powered down as long as UPF is set. Thus the software can initialize the UNIBUS by setting and then clearing the UPF bit.

Bit: 0 Name: Adapter INIT (ADINIT)
Function: When this bit is set it will completely initialize the UNIBUS adapter and the UNIBUS. The map registers, the data path registers,
the status register, and the control register will be cleared. The UNIBUS adapter will initialize all of its control logic, and will generate a power fail sequence on the UNIBUS. The adapter initialization sequence takes only 500 µs to complete, while the UNIBUS power fail sequence requires 25 ms.

Only the configuration register and the diagnostic control register can be read during the adapter initialization sequence. And only the configuration register, the diagnostic control register, and the control register can be written during the adapter initialization sequence.

Once the sequence has been completed, all UNIBUS adapter registers can be accessed. However, the UNIBUS cannot be accessed until the UNIBUS initialization sequence has been completed as well. The software can test for this condition by reading the UBIC bit of the configuration register, or by setting the configuration interrupt enable bit of the control register and looking for the interrupt generated by the setting of the UBIC bit. Note that the assertion of UNIBUS INIT (UBINIT) can also initiate an interrupt. The Adapter INIT bit can be set by writing a one to the bit location, and it is self-clearing.

Status Register (USAR)
The UNIBUS Adapter Status Register contains program status and error information. Bits <27:24> are read only bits which are set and cleared by operations within the UNIBUS adapter. Bits <10:0> can be read and cleared by writing a one to the appropriate bit location. Specific conditions which occur on the UNIBUS adapter will set these bits. Writing a zero has no effect on any of the bits. Figure 18-12 shows the Status Register bit configuration.

The contents of the status register are:

**Bit:** 31:28 **Name:** Reserved and zero

**Function:**

**Bit:** 27:24 **Name:** BR Receive Vector Register Full
**Function:** These bits indicate the state of the SBI addressable BR receive vector registers. Each bit is set when the interrupt vector is loaded into the corresponding BRRVR during a UNIBUS interrupt transaction, providing that the SBI processor is fielding UNIBUS device interrupts.
Figure 18-12 Status Register Bit Configuration

Each bit is cleared by the successful completion of a read data transmission following a read BRRVR command. The software will see these bits set only after a read data failure has occurred during the execution of the read BRRVR command, and the UNIBUS interrupt vector has been saved by the UNIBUS adapter.

- Bit 27 = BRRVR 7 Full
- Bit 26 = BRRVR 6 Full
- Bit 25 = BRRVR 5 Full
- Bit 24 = BRRVR 4 Full

**Bit: 23:11 Name:** Reserved and zero  
**Function:** The remaining bits identify specific data transfer errors. They are read and write-one-to-clear bits.

- **Bit: 10 Name:** Read Data Time Out (RDTO)  
  **Function:** The UNIBUS Adapter sets the Read Data Time Out bit when the following conditions are met: When a UNIBUS device has initiated a DMA read transfer, when the UNIBUS adapter has successfully transmitted a read command on the SBI, and the SBI memory has not returned the requested data within 100 μs, and when the UNIBUS device has not timed out. Note that the normal UNIBUS timeout is 10-20 μs, and that after 10-20 μs, the UNIBUS device will set its non-existent memory bit. Thus, the Read Data Time Out bit will be set on the
UNIBUS Subsystem

UNIBUS adapter status register only if the UNIBUS device timeout function is inoperative, or takes more than 100 $\mu$s.

**Bit: 9  Name: Read Data Substitute (RDS)**
**Function:** This bit is set if a read data substitute is received in response to a UNIBUS to SBI read command (DMA read transfer). No data will be sent to the UNIBUS device, and when the device timeout occurs, the nonexistent memory bit will be set within the UNIBUS device.

**Bit: 8  Name: Corrected Read Data (CRD)**
**Function:** The UNIBUS adapter sets this bit when it receives corrected read data in response to an SBI read command during a DMA read transfer. The setting of this bit has no effect on the completion of the UNIBUS transfer.

**Bit: 7  Name: Command Transmit Error (CXTER)**
**Function:** The UNIBUS adapter sets this bit when it receives an error confirmation in response to an SBI command transmission during a DMA transfer.

**Bit: 6  Name: Command Transmit Timeout (CXT0)**
**Function:** This bit is set when a command transmission times out during a UNIBUS to SBI data transfer or during a BDP to SBI write or purge.

Note that the normal UNIBUS timeout is 10 $\mu$s, which will result in the UNIBUS device setting its nonexistent memory bit and will also abort the UBA to SBI transfer. The CXT0 bit will therefore only be set for a UNIBUS to SBI transfer if the device's timeout mechanism is inoperative. The UBA will, however, attempt to perform a BDP to SBI write or purge operation for the full 100 $\mu$s timeout period if busy or no response confirmation is received, thereby setting the CXT0 bit. The bit is not set for a prefetch operation since the prefetch works by anticipated addresses (i.e., the next address) and any errors resulting from the prefetch are considered to be invalid.

**Bit: 5  Name: Data Path Parity Error (DPPE)**
**Function:** This bit is set when a parity error occurs in the Buffered Data Path during either a UNIBUS to BDP DATI, a BDP to SBI write, or a purge.

Note that during a purge operation the address to be mapped is also obtained from the BDP and it is possible for a parity error to occur when fetching the address from the BDP. This parity error will also set the DPPE bit and abort the SBI transfer that would have taken place. Also note that any condition that sets the DPPE bit will also set the buffer transfer error bit in the DPR of the Buffer Data Path in which the
error occurred, thereby aborting any SBI transfers in progress and any future UNIBUS transfers through that BDP until the buffer transfer error is cleared.

**Bit: 4**  **Name:** Invalid Map Register (IVMR)  
**Function:** The UNIBUS adapter sets this bit during a DMA transfer or purge operation when the UNIBUS address points to a map register which has not been validated by the software, or when the DMA transfer crosses an SBI page boundary for which the map register has not been validated.

**Bit: 3**  **Name:** Map Register Parity Failure (MRPF)  
**Function:** This bit is set with the occurrence of a map register parity failure when a UNIBUS address is being mapped to an SBI address on a DMA transfer operation or a purge operation.

Seven of the bits just listed (RDTO, RDS, CXTER, CXTO, DPPE, IVMR, and MRPF) form an error-locking field. If any one of these bits is set, the field is locked until the bit indicating the error is cleared. The Failed Map Entry Register (FMER) is also locked and unlocked with this field. And the setting of any of these bits will cause the UNIBUS adapter to initiate an interrupt request if the interrupt enable bit for the UNIBUS to SBI data transfer error field (USEFIE) in the control register is set.

**Bit: 2**  **Name:** Lost Error Bit (LEB)  
**Function:** The UNIBUS adapter sets this bit if the locking error field is locked and another error within this field occurs. The lost error bit does not initiate an interrupt request.

**Bit: 1**  **Name:** UNIBUS Select Time Out (UBSTO)  
**Function:** The UNIBUS adapter sets this bit if it cannot gain access to the UNIBUS within 50 μs in the execution of a software initiated transfer (SBI to UNIBUS transfer). When UBSTO is set it indicates that the UNIBUS Adapter has issued NPR on the UNIBUS but has not become bus master. This condition indicates the presence of a hardware problem on the UNIBUS. The UNIBUS may be inoperative, or one device may be holding it for extended periods. Note that if the UNIBUS does become inoperative, it may be possible to clear the problem with the assertion of UNJAM on the SBI, by setting and clearing of the UNIBUS power fail bit (control register bit 1) or by setting Adapter INIT (control register bit 0).

**Bit: 0**  **Name:** UNIBUS Slave Sync Time Out (UBSSYNT0)  
**Function:** This bit is set when an SBI to UNIBUS transfer (software-initiated transfer) times out during the data transfer cycle on the UNIBUS. The timeout occurs after 12.8 μs. UBSSYNT0 indicates a transfer failure resulting when a nonexistent memory or device on the UNIBUS is addressed.
The two bits just discussed, UBSTO and UBSSYNT0, form an SBI to UNIBUS transfer error-locking field. They are set by the occurrence of the conditions mentioned and cleared by writing a (1) to the bit location. The setting of either bit will cause the UNIBUS adapter to make an interrupt request on the SBI if the SBI to UNIBUS error interrupt enable bit (SUEFIE) in the control register is set. The setting of either UBSTO or UBSSYNT0 will lock the failed UNIBUS address register (FUBAR), thus storing the high 16 bits of the UNIBUS address identified with the failure. The FUBAR will remain locked until the UBSTO and UBSSYNT0 bits are cleared.

**Diagnostic Control Register (DCR)**

The Diagnostic Control Register provides control and status bits which aid in the testing and diagnosis of the UNIBUS adapter. The bits of this register, when set, will defeat certain vital functions of the UNIBUS adapter. The DCR is therefore not intended for use during normal system operation. Figure 18-13 shows the bit configuration of the DCR.

![Diagnostic Control Register Bit Configuration](image)

**Figure 18-13  Diagnostic Control Register Bit Configuration**

**Bit: 31  Name:** Spare  
**Function:** This read/write bit has no effect on any UNIBUS adapter operation. It can be set by writing a one and cleared by writing a zero to the bit location. SBI Dead, Adapter INIT, and a power up sequence on the UNIBUS adapter will clear this bit.

**Bit: 30  Name:** Disable Interrupt (DINTR)  
**Function:** When it is set, this bit will prevent the UNIBUS adapter from recognizing interrupts on the UNIBUS. It is useful in testing the response of the UNIBUS adapter to the passive release condition during a UNIBUS interrupt transaction. This bit is set by writing a one and
UNIBUS Subsystem

cleared by writing a zero to the bit location. SBI Dead, Adapter INIT, and the power up sequence on the UNIBUS adapter will also clear DINTR.

Bit: 29    Name: Defeat Map Parity (DMP)
Function: When it is set, this read/write bit will inhibit the parity bits of the map registers from entering the map register parity checkers. The map register parity generator checkers generate and check parity on eight bit quantities. Each parity field (eight data bits and one parity bit) is implemented so that the total number of ones in the field is odd.

For example, if bits <7:0> of a map register equal zero, then the parity bit equals one. However, if the DMP bit is set, then the parity bit is disabled and the parity checkers will see all zeros. This results in a map register parity failure. Then, if the DMP bit is set, the parity checkers will see correct parity. Note, however, that if bits <7:0> of the map register contain an odd number of ones, the generated parity bit will be zero. The state of the DMP bit will therefore have no effect on the parity result in this case.

When the integrity of the parity generator checkers is to be tested, the map register must contain data so that at least one of the bytes contains an even number of ones. The DMP bit, when set, will disable the parity bit, and the map register parity failure can be detected during a DMA transfer. SBI Dead, Adapter INIT, and the power up sequence on the UNIBUS adapter will clear this bit.

Bit: 28    Name: Defeat Data Path Parity (DDPP)
Function: The DDPP bit functions in the same way as the DMP bit. When it is set, the DDPP bit will inhibit the parity bits of the data path RAM from entering the parity checkers. The data path parity generator checkers generate and check parity on eight bit data units. Each parity field (eight data bits and one parity bit) is implemented so that the total number of ones in the field is odd. When the integrity of the parity generator checkers is to be tested through use of the DDPP bit, the total number of ones in at least one of the bytes of data must be even. With the parity bit disabled by the DDPP bit, a data path parity failure will result during a DMA transfer via that buffered data path. SBI Dead, Adapter INIT, and the power up sequence on the UNIBUS adapter will clear the DDPP bit.

Bit: 27    Name: Microsequencer OK (MIC OK)
Function: The MIC OK bit is a read-only bit which indicates that the UNIBUS adapter microsequencer is in the idle state. The microsequencer will enter the idle state after it has completed the initialization sequence or once it has completed a UNIBUS adapter function.
The MIC OK bit can be used by diagnostics to determine whether or not the microsequencer has completed a successful power up sequence and whether or not it is caught up in any loops. Note that SBI dead, UNIBUS adapter power supply DC LO, and Adapter INIT force the microsequencer into the initialization routine. Once the routine has been completed and the microsequencer has entered the idle state, MIC OK will be true (1).

Bit: 26:24  Name:  Reserved and zero
Function:

Bit: 23:0  Name:
Function:  Same as bits <23:0> of the Configuration Register

Failed Map Entry Register (FMER)
The Failed Map Entry Register contains the map register number used for either a DMA transfer or a purge operation which has resulted in the setting of one of the following error bits of the status register: IVMR, MRPF, DPPE, CXTO, CXTER, RDS, RDTO. This register is locked and unlocked with the UNIBUS to SBI data transfer error field of the status register. The contents of the FMER are valid only when the register is loaded. The FMER is a read-only register. Attempts to write to the FMER will result in an SBI error confirmation. No signals or events will clear the register.

The software can read the FMER to obtain the map register number associated with the failure. It can then read the contents of the failing map register to determine the number of the data path which failed.

Figure 18-14 shows the bit configuration for the Failed Map Entry Register.

![Figure 18-14  Failed Map Entry Register Bit Configuration](image_url)

Bit: 31:9  Name:  Reserved and zero
Function:

Bit: 8:0  Name:  Map Register Number (MRN)
Function:  These bits contain the number of the map register which was in use at the time of a failure. Bits <8:0> correspond to bits <17:9> of the UNIBUS address.
Failed UNIBUS Address Registers (FUBAR)
The FUBAR contains the upper 16 bits of the UNIBUS address translated from an SBI address during a previous software-initiated data transfer. The occurrence of either of two errors indicated in the status register will lock the FUBAR: UNIBUS Select Time Out (UBSTO) and UNIBUS Slave Sync Time Out (UBSSYNT0). When the error bit is cleared, the register will be unlocked.

The FUBAR is a read-only register. Attempting to write to the register will result in an error confirmation. No signals or conditions will clear the register. Figure 18-15 shows the bit configuration of the FUBAR. The contents of the FUBAR are listed below.

![Figure 18-15 Failed UNIBUS Address Register Bit Configuration](image)

**Bit: 31:16 Name:** Reserved and zero  
**Function:**

**Bit: 15:0 Name:** Failed UNIBUS to SBI Address  
**Function:** These bits correspond to UNIBUS Address bits <17:2>.

Buffer Selection Verification Registers 0-3 (BRSVR)
These four read/write do-nothing registers are provided in order to give the diagnostic software a means of accessing and testing the integrity of the data path RAM. Four spare locations in the data path RAM have been assigned to these registers. Writing and reading the BRSVRs has no effect on the behavior of the UNIBUS adapter. The BRSVR bit configuration is shown in Figure 18-16.

![Figure 18-16 Buffer Selection Verification Register Bit Configuration](image)
UNIBUS Subsystem

The contents of the BRSVRs are listed below.

Bit: 31:16  Name:  
Function:  Always zero  

Bit: 15:0  Name:  
Function:  Read/write bits  

BR Receive Vector Registers 4-7 (BRRVR)
The UNIBUS adapter contains four BR receive vector registers: BRRVR 7, BRRVR 6, BRRVR 5, and BRRVR 4. Each BRRVR corresponds to a UNIBUS interrupt bus request level: 7, 6, 5, 4. Each BRRVR is a read-only register and will contain the interrupt vector of a UNIBUS device interrupting at the corresponding BR level. Each BRRVR is read by the software as part of the UNIBUS adapter interrupt service routine. Note that the UNIBUS adapter interrupt service routine is the routine to which the VAX-11 CPU will transfer control once it has determined that the UNIBUS adapter has transmitted an interrupt request on the SBI.

If the IFS and BRIE bits on the control register are set, so that UNIBUS interrupt requests are passed to the SBI, then the VAX-11 CPU responds with an Interrupt Summary Read command. The UBA sends its request sublevel as an Interrupt Summary Response. The software then invokes the UBA interrupt service routine, initiating a read transfer to the appropriate BRRVR. The UNIBUS adapter will assert the contents of the BRRVR on the SBI as read data if the corresponding BRRVR Full bit in the status register is set. If the BRRVR Full bit is not set, the Read BRRVR command causes the UNIBUS adapter to fetch the interrupt vector from the interrupting UNIBUS device. The interrupt vector is loaded into the BRRVR only at the successful completion of a UNIBUS interrupt transaction. The UNIBUS adapter will then send the contents of the BRRVR to the SBI as read data. The BRRVR used is cleared only when the UNIBUS adapter receives an ACK confirmation for the read data. Following this exchange, the UNIBUS adapter interrupt service routine will use the contents of the BRRVR to branch to the appropriate UNIBUS device service routine.

Four types of failure conditions can occur when the software is reading a BRRVR and the VAX-11 CPU is servicing a UNIBUS device interrupt:

1. If the software attempts to read a BRRVR for which a BR interrupt line is not asserted, and BRRVR is not full, the zero vector (all zeroes data) will be sent as read data.

2. If the BR line asserted by the interrupting UNIBUS device is released during the interrupt summary read transfer, and the vector is not received from the device (passive release), then the zero vector will be sent as read data.
3. If the vector has been received from the interrupting device, but an ACK confirmation is not received following the read data transmission, then the BRRVR will not be cleared, and the BRRVR Full bit will remain set. Subsequent read commands to the full BRRVR will cause the UNIBUS adapter to send the stored vector, but the BRRVR will remain full until the UNIBUS adapter receives an ACK confirmation for the read data. Note that the BRRVR Full bits always reflect the state of the BRRVRs.

4. If the IFS bit in the control register is cleared and the software reads a BRRVR, then the zero vector will be sent as read data.

The contents of the BRRVR are also used by the software to determine whether or not the UNIBUS adapter itself has an interrupt pending. Bit 31 of the BRRVR is the Adapter Interrupt Request Indicator. Although the bit is present in all four BRRVRs, it will be active only in the BRRVR corresponding to the interrupt request level that has been assigned to the UNIBUS adapter. If bit 31 is set when the software reads the BRRVR, then an adapter interrupt request is pending.

Figure 18-17 shows the BR Receive Vector Register bit configuration.

![UNIBUS Device Interrupt Vector]

**Figure 18-17. BR Receive Vector Register Bit Configuration**

The contents of the four BRRVRs are as follows:

**Bit: 31**  **Name:** Adapter Interrupt Request Indicator
**Function:** 0=No UBA interrupt pending.
1=UBA interrupt pending.

**Bit: 30:16**  **Name:** Reserved and zero
**Function:**

**Bit: 15:0**  **Name:** Device Interrupt Vector Field
**Function:** These bits contain the device interrupt vector loaded by the UNIBUS adapter during a UNIBUS interrupt transaction.

**Data Path Registers 0-15 (DPR)**
The UNIBUS adapter contains 16 data path registers (DPR 0 to DPR 15), each of which corresponds to one of the 16 data paths. DPR 0, corresponding to the direct data path, is always 0.
UNIBUS Subsystem

Figure 18-18 shows the Data Path Register bit configuration.

![Data Path Register Bit Configuration]

Figure 18-18  Data Path Register Bit Configuration

The DPR bit functions are as follows:

- Buffer Not Empty and the Purge Operation

**Bit:** 31    **Name:** Buffer Not Empty (BNE)
**Function:** Each DPR contains a data path status bit called Buffer Not Empty. This bit is read/write one to clear bit.

0 = Buffer empty
1 = Buffer not empty

If this bit is set (1), the BDP contains valid data. If clear, then the BDP does not contain valid data. The UNIBUS adapter uses the bit to determine the proper action for DMA transfers via the BDP. If bit <31> is set as a DATI transfer begins, the data in the BDP will be asserted on the UNIBUS. If bit <31> is clear on a DATI, the UNIBUS adapter will initiate a read transfer to SBI memory, load the read data into the BDP, whereby setting bit <31>, and gate the addressed data to the UNIBUS.

For a DMA write transfer via the associated BDP, the BNE bit is set each time UNIBUS data is loaded into the BDP. The bit is then cleared when the contents of the BDP are transferred to SBI memory.

The software will write a one to this bit to initiate the purge operation. The purge operation is required at the completion of a UNIBUS device block transfer and is performed in the following way:

1. Write transfers to memory. If any bytes of data remain in the corresponding BDP (BNE is set), the UNIBUS adapter will transfer this data to memory. The UNIBUS adapter will then initialize the BDP and clear the BNE bit. If no data remains to be transferred (BNE is cleared), the purge operation will be treated as a no-op (it is a legal do-nothing function).
2. Read transfers to memory. If any bytes of data remain in the BDP, the UNIBUS adapter will initialize the BDP by clearing the BNE bit. If no data remains, the purge will be treated as a no-op.

In addition, the following considerations apply to the purge operation:

- For purge operations in which data are transferred to memory, the SBI transfer takes about 2 $\mu$s. The UBA will not respond to Data Path Register read transfers during this period (busy confirmation), thereby preventing a race condition when testing for BNE bit.
- A purge operation to Data Path Register Zero (Direct Data Path) is treated by the UBA as a NO-OP.

**Bit: 30  Name:** Buffer Transfer Error (BTE)

**Function:** This is a read-write one to clear bit. The UNIBUS adapter sets the BTE bit if a failure occurs during a DMA write transfer, or a purge, or for a data path parity failure during a DMA read transfer via the associated BDP. If bit $<30>$ is set, any additional DMA transfers via the BDP will be aborted until the bit is cleared by the software. Note that if a parity error on the UNIBUS occurs during a DMA read, the UNIBUS signal PB will be asserted, giving the UNIBUS device the opportunity to abort its own DMA transfer. The purge operation does not clear the BTE bit.

**Bit: 29:0  Name:**

**Function:** Read-only bits.

**Bit: 29  Name:** Data Path Function (DPF)

**Function:** This bit indicates the function of the DMA transfer using this data path.

0 = DMA Read
1 = DMA Write

**Bit: 28:24  Name:**

**Function:** Unused

**Bit: 23:16  Name:** Buffer State (BS)

**Function:** These eight bits indicate the state of each of the eight byte buffers of the associated BDP during a DMA write transfer. They are included in the Data Path Register for diagnostic purposes only. The UNIBUS adapter generates the SBI mask bits from the BS bits during a DMA write transfer or purge operation. The bits are set as each byte is written from the UNIBUS. The bits are cleared during the SBI write operation.

0 = Empty
1 = Full

**Bit: 15:0  Name:** Buffered UNIBUS Address (BUBA)
**UNIBUS Subsystem**

**Function:** This portion of each DPR contains the upper 16 bits of the UNIBUS address, UA \(<17:2>\), asserted during the DMA transfer using the associated BDP. If the transfer through the associated BDP is in the byte offset mode, and the last UNIBUS transfer has spilled over into the next quadword, then these bits contain UA \(<17:2>\). This is the UNIBUS address from which the SBI address will be mapped should a purge operation occur before the next UNIBUS transfer.

**Map Registers 0-495(10)**
The UNIBUS adapter contains 496 map registers, one for each UNIBUS memory page address (a page = 512 bytes).

<table>
<thead>
<tr>
<th>REG</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR0</td>
<td>800</td>
</tr>
<tr>
<td>MR1</td>
<td>804</td>
</tr>
<tr>
<td>MR2</td>
<td>808</td>
</tr>
<tr>
<td>MR3</td>
<td>80C</td>
</tr>
<tr>
<td>REG</td>
<td>Offset</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>MR494</td>
<td>FB8</td>
</tr>
<tr>
<td>MR495</td>
<td>FBC</td>
</tr>
</tbody>
</table>

When a DMA transfer begins, the upper nine address bits asserted by the UNIBUS device select one of the map registers which the software has set up. The map register in turn tests for the validity of the current UNIBUS transfer, steers the transfer through one of the 16 data paths, determines whether or not the transfer will take place in the byte offset mode if a BDP has been selected, and maps the UNIBUS page address to an SBI page address.

The map registers are numbered sequentially from 0 through 495\(_{10}\). There is a one to one correspondence between each map register and UNIBUS memory page address (i.e., MR0 corresponds to UNIBUS memory page 0, MR1 to UNIBUS Memory Page 1.....MR495 to UNIBUS memory page 495). Each map register contains the information required to effect the data transfer of the UNIBUS device addressing that page:

1. The fact that the software has loaded the map register (map register valid).
UNIBUS Subsystem

2. The number of the data path to be used by the transfer and, if a BDP is used, whether it is in byte offset mode.
3. The SBI page to which the transfer will be mapped.

Since the map register is implemented with a bipolar RAM, the contents of the map registers will be checked by parity. If, during a UNIBUS transfer, the parity test fails, the map register parity fail bit of the UNIBUS adapter status register will be set and the UNIBUS transfer will be aborted.

Figure 18-19 illustrates the map register bit configuration.

![Map Register Bit Configuration](image)

NOTE

For brevity, for the map register description, “this UNIBUS page” refers to “the UNIBUS memory page corresponding to this map register.”

The contents of a map register are as follows:

**Bit: 31  Name:** Map Register Valid (MRV)
**Function:** 0 = not valid—initialized state
              1 = valid

The MRV is set by the software to indicate that the contents of the map register are valid. The MRV is tested each time that “this UNIBUS page” is accessed. If the bit is set (1), the transfer continues. If the bit is not set, the UNIBUS transfer is aborted (nonexistent memory error in the UNIBUS device) and the invalid map register bit is set in the UNIBUS adapter status register.

The MRV can be set and cleared by the software.

379
**Bit: 30:27  Name:** Reserved read/write bits

**Function:** This is a read/write bit. If set, and the map register selects a BDP, then the longword-aligned 32-bit random access mode is enabled for the BDP. The longword-aligned 32-bit random access mode has been discussed above. This bit has no effect if the Direct Data Path is selected by the map register. This bit is cleared on initialization.

**Bit: 26  Name:** Longword Access Enable (LWAE)

**Function:** This is a read/write bit. If set, and the map register selects a BDP, then the longword-aligned 32-bit random access mode is enabled for the BDP. The longword-aligned 32-bit random access mode has been discussed above. This bit has no effect if the Direct Data Path is selected by the map register. This bit is cleared on initialization.

**Bit: 25  Name:** Byte Offset Bit (BO)

**Function:** This is a read/write bit. If set, and “this UNIBUS page” is using one of the BDPs, and the transfer is to an SBI memory address, then the UNIBUS adapter will perform a byte offset operation on the current UNIBUS data transfer. The software can interpret this operation as increasing the physical SBI memory address, mapped from the UNIBUS address, by one byte. This allows word-aligned UNIBUS devices to transfer to odd byte memory addresses.

UNIBUS transfers via the DDP or to SBI I/O addresses will ignore the Byte Offset bit.

This bit is cleared on initialization.

**Bit: 24:21  Name:** Data Path Designator Bits (DPDB)

**Function:**

<table>
<thead>
<tr>
<th>0000</th>
<th>Direct Data Path (DDP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>Buffered Data Path 1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>Buffered Data Path 15</td>
</tr>
</tbody>
</table>

The DPDBs are read/write bits that are set and cleared by the software to designate the data path that “this UNIBUS page” will be using.

The software can assign more than one UNIBUS transfer to the DDP.

The software must assure that no more than one active UNIBUS transfer is assigned to any BDP.

The DPDBs are cleared on initialization.

**Bit: 20:0  Name:** SBI Page Address (SPA <27:7>, also known as Page Frame Number, PFN)

**Function:** The SPA bits contain the SBI page address to which “this UNIBUS page” will be mapped. These bits perform the UNIBUS to SBI page address translation. When an SBI transfer is initiated the con-
tents of SPA<27:7> are concatenated with UNIBUS address bits UA<8:2> to form the 28-bit SBI address.

POWER FAIL AND INITIALIZATION
The UNIBUS adapter controls the UNIBUS power fail, power up, and initialization sequences of the UNIBUS. This section explains the behavior of the UNIBUS subsystem for each of the following:
1. System Power Up
2. System Power Down
3. UNIBUS Power Down
4. Programmed Power Down
5. SBI UNJAM

System Power Up
The UNIBUS remains in a powered down state as long as the UNIBUS adapter is in a powered down state. During System Power Up, the UNIBUS adapter will initiate the UNIBUS power up sequence, provided the UNIBUS has power. Once the power up sequence has been completed, the UNIBUS Initialization Complete bit of the UNIBUS adapter status register is set and an interrupt request is initiated to the CPU. If the UNIBUS power was not on at the time that the system powered up, the power up sequence will not continue until the UNIBUS power has been turned on. The power up sequence will completely initialize all registers and functions of the UNIBUS adapter. The deassertion of power supply AC LO will set the adapter power up bit in the Configuration Register and initiate an interrupt request.

SBI Power Fail
The UNIBUS adapter will initiate a UNIBUS power fail sequence whenever an SBI power failure is detected (SBI Dead asserted). The UNIBUS will remain powered down as long as SBI Dead is asserted. The UBA will initiate the UNIBUS power up sequence when SBI Dead is released.

UNIBUS Power Fail
A power loss on the UNIBUS will initiate a UNIBUS power fail sequence. The UNIBUS power down bit of the status register will be set and the UNIBUS adapter will initiate an interrupt request (providing the CNFIE bit is set). The UNIBUS will remain in a powered down state until UNIBUS power has been restored, at which time a UNIBUS power up sequence is initiated. The UNIBUS initialization complete bit of the status register will be set on a successful power up sequence and the UBPDN bit will be cleared. The UNIBUS power fail lines will not affect the state of the SBI power fail lines.
Programmed UNIBUS Power Fail
The software can induce a power fail sequence on the UNIBUS by first setting and then clearing the UNIBUS power fail bit of the control register. The UNIBUS adapter will initiate a power fail sequence when the UPF is set. Once it has been initiated, the power fail sequence will continue to completion independent of the state of the UPF. On completion of the power down sequence, the UNIBUS adapter will initiate a power up sequence if or when the UPF is cleared, provided power is normal for both the UNIBUS and UNIBUS adapter.

Setting the AD INIT bit will also initiate a power fail and initialization sequence on the UNIBUS as well as completely initialize all registers and functions of the UBA.

SBI UNJAM
The assertion of SBI UNJAM will initiate the UNIBUS power fail and initialization sequence. It will also clear all interrupt enable bits of the UBA control register. It will initialize the UBA SBI logic so that the UBA is available for an SBI Command.

EXAMPLE
Presented is a program to read data from the RK06 disk subsystem into memory. The program is fully documented and is designed to demonstrate the loading of the UBA map registers, the use of a buffered data path (including the purge), access to UNIBUS device registers, and initialization of the UNIBUS. In order to run the program, it must be loaded from the floppy disk by the console into memory. Initially, the program can be assembled and linked under VAX/VMS and then transferred to the floppy disk using the RSX-11M utility program FLX. The file structure of the floppy is RT-11 format.
UNIBUS Subsystem

; PROGRAM TO READ FROM THE RK06 INTO MEMORY
; THIS PROGRAM WILL TRANSFER 3210 BYTES FROM THE RK06
; DISK TO MEMORY STARTING AT ADDRESS 4567. THE TRANSFER
; WILL USE A BLOCK OF MAP REGISTERS STARTING AT
; MAP REGISTER 25 AND WILL USE A BUFFERED DATA PATH (BDP),
; IT WILL READ DATA FROM DRIVE 0, TRACK 2, CYLINDER 4, SECTOR 6,
; IT WILL RING THE BELL OF THE CONSOLE IF NO ERRORS ARE
; DETECTED.

; THE PROGRAM IS DESIGNED TO DEMONSTRATE THE LOADING OF THE
; UBA MAP REGISTERS, USE OF A BUFFERED DATA PATH (INCLUDING
; THE PURGE), ACCESS TO UNIBUS DEVICE REGISTERS, AND
; INITIALIZATION OF THE UNIBUS. NO CLAIM IS MADE FOR
; ELEGANCE OF PROGRAMMING STYLE.

; SYMBOL DEFINITIONS

; UBA RELATED SYMBOLS:
UBA_BASE = "X20040000" ; UBA AT TR = 3
UBA_CNFR = "X0" ; OFFSET TO UBA CONFIGURATION REGISTER
UBA_UCIC = "X10000" ; UNIBUS INITIALIZATION COMPLETE
UBA_CR = "X4" ; OFFSET TO UBA CONTROL REGISTER
UBA_ADINIT = "X1" ; ADAPTOR INIT AND UNIBUS INITIALIZATION
UBA_SR = "X8" ; OFFSET TO UBA STATUS REGISTER
UBA_DP_BNE = "X40" ; OFFSET TO UBA DATA PATH REGISTER 0
UBA_DP_BTE = "X40000000" ; BUFFER NOT EMPTY BIT
UBA_MRO = "X800" ; USED TO PURGE BUFFER DATA
MAP_VALID = "X80000000" ; PATH AT END OF XFER.
BYTE_OFFSET = "X20000000" ; BUFFER ERROR BIT

; RK611 RELATED SYMBOLS:
UNIBUS_BASE = "X20100000" ; BASE ADDRESS FOR UNIBUS ADDRESS 0
DK_BASE_ADD = "0777440" ; UNIBUS BASE ADDRESS OF DK611
DK_CS1 = "00" ; RK611 CONTROL STATUS REGISTER 1
DK_CS2 = "010" ; RK611 CONTROL STATUS REGISTER 2
DK_DS = "012" ; RK611 DRIVE STATUS REGISTER
DK_DC = "020" ; RK611 DESIRED CYLINDER REGISTER
DK_BA = "06" ; RK611 BUS ADDRESS REGISTER
DK_MC = "02" ; RK611 MIRROR COUNT REGISTER
DK_KA = "04" ; RK611 KEEPS REGISTER
SCLR = "040" ; RK611 SUBSYSTEM CLEAR
PKACK = "01" ; RK611 PACK ACKNOWLEDGE AND GO BIT SET
READ = "021" ; RK611 DISK READ AND GO BIT SET.
; MISC SYMBOLS:
BELL = "07" ; ASCII BELL
TXDB = 35 ; CONSOLE TRANSMIT DATA BUFFER

; THIS SECTION LOADS THE UBA_BASE ADDRESS INTO RO AND THEN INITIALIZES
; THE UNIBUS.
BEGIN: MOVL #UBA_BASE, RO ; LOAD UBA'S ADDRESS INTO RO
INIT: MOVL #UBA_ADINIT, UBA_CR(RO) ; INIT UBA AND UNIBUS

; THE UBA AND UNIBUS ARE BEING INITIALIZED. THE PROGRAM CANNOT MAKE
; ANYアクセス TO THE UBA OR THE UNIBUS DURING THIS PERIOD OF TIME.
; BUT THAT'S OK BECAUSE WE HAVE LOTS TO DO IN THE MEAN TIME...

; THIS SECTION WILL SET UP THE MAP REGISTERS TO BE USED FOR THE TRANSFER.
; FIND THE OFFSET OF THE INITIAL MAP REGISTER AND PUT INTO R1.
ASHL #2, MAP_REG, R1 ; MULTIPLY THE MAP REGISTER BY 4
ADDL #UBA_MRO, R1 ; ADD THE BASE ADDRESS OF THE MAP REGISTERS
ADDL #UBA_BASE, R1 ; ADD IN THE UBA BASE ADDRESS AND PUT IN R1.

383
UNIBUS Subsystem

; R1 NOW CONTAINS THE ADDRESS OF THE FIRST MAP REGISTER TO BE LOADED.
; THIS SECTION WILL DETERMINE THE CONTENTS OF THE MAP REGISTERS AND
; STORE IT IN R2.

; THIS SECTION WILL DETERMINE THE PAGE FRAME NUMBER OF THE FIRST
; MEMORY PAGE TO BE ACCESSED. THE PHYSICAL MEMORY ADDRESS IS SHIFTED
; RIGHT NINE BITS TO BECOME THE PAGE FRAME NUMBER.

; THE DATA PATH NUMBER TO BE USED FOR THE TRANSFER WILL THEN BE INSERTED
; INTO THE DATA PATH DESIGNATOR FIELD AND THE VALID BIT IS SET.

ASHL $-9, MEMSAD, R2 ; TURN START ADDRESS INTO THE
; PAGE FRAME NUMBER.
INSV DP_NUM, 21, 4, R2 ; INSERT BITS 0-3 OF DP_NUM INTO
; BITS 21-24 OF R2.
DISL $MAP_VALID, R2 ; SET MAP VALID BIT.

; DETERMINE IF THE BYTE ALIGNMENT BIT OF THE BUFFERED DATA PATHS IS
; REQUIRED. THE RK611 ONLY KNOWS ABOUT WORD ALIGNED TRANSFERS. IF
; THE START MEM ADDRESS IS ODD THEN THE BYTE OFFSET BIT OF THE
; MAP REGISTERS MUST BE SET.

BITL ~X1, MEMSAD ; IS MEM ADDRESS ODD?
BEQL CONTINUE ; IF NOT THEN CONTINUE.
DISL $BYTE_OFST, R2 ; YES -- SET BYTE OFFSET BIT

CONTINUE: NOP ;CONTINUE

; THIS SECTION COMPUTES THE CONTENTS OF THE RK611 BUS ADDRESS REGISTER
; AND THE EXTENDED ADDRESS BITS OF CONTROL STATUS REGISTER 1.

; THE RESULT OF THIS SECTION WILL BE THAT WHEN THE RK611 ASSERTS THE
; ADDRESS ONTO THE UNIBUS, UNIBUS ADDRESS BITS <17:09> WILL POINT TO
; THE MAP REGISTER THAT CONTAINS THE PAGE FRAME NUMBER FOR THE TRANSFER
; AND UNIBUS ADDRESS BITS <8:0> WILL CONTAIN THE BYTE OFFSET WITHIN THE
; PAGE.

; THE CONTENTS OF THE REGISTERS WILL BE AS FOLLOWS:
; DK.CS1 BA <17 : 16> AND
; DK.BA BA <15 : 09> WILL CONTAIN THE POINTER
; TO THE MAP REGISTER THAT CONTAINS THE PAGE FRAME NUMBER
; FOR THE TRANSFER.
; DK.BA BA <08 : 00> WILL CONTAIN THE BYTE OFFSET WITHIN THE PAGE.
; R3 WILL BE USED TO SET UP TO CONTAIN THE INITIAL UNIBUS ADDRESS OF
; THE TRANSFER.

ASHL $9, MAP_REG, R3 ; SHIFT MAP REGISTER NUMBER TO FORM
; MAP REGISTER POINTER.
BICL3 #$'FFFFFFE00, MEMSAD, R4 ; CLEAR ALL BUT BYTE OFFSET WITHIN
; THE PAGE
BISL R4, R3 ; AND COMBINE WITH MAP REGISTER POINTER
; IN R3.

; THIS SECTION WILL DETERMINE THE WORD COUNT FOR THE TRANSFER.
; CONVERT BYTE COUNT TO WORD COUNT FOR THE RK611. IF BYTE COUNT IS ODD
; THEN THE WORD COUNT MUST BE INCREMENTED TO CONTAIN ALL BYTES OF THE
; TRANSFER.

INCL BCOUNT ; INCREMENT BYTE COUNT TO ACCOUNT
; FOR ODD BYTE COUNT.
ASHL $-1, BCOUNT, R4 ; CONVERT TO WORD COUNT AND LOAD
; INTO R4.

; THIS SECTION WILL SET UP DISK ADDRESS TRACK AND SECTOR -- WILL USE R5.

MOVL SECTOR, R5 ; GET SECTOR NUMBER.
INSV TRACK, $08, $3, R5 ; INSERT BITS 0-2 OF TRACK INTO
; BITS 8-10 OF R4.

; AT THIS POINT ALL OF THE VALUES REQUIRED FOR THE TRANSFER HAVE BEEN
; DETERMINED. THE VALUES OF THE REGISTERS ARE:

; R0 = UBA_BASE ADDRESS
; R1 = ADDRESS OF FIRST MAP REGISTER TO BE USED FOR TRANSFER
; R2 = CONTENTS FOR THE INITIAL MAP REGISTER
; R3 = INITIAL UNIBUS ADDRESS FOR TRANSFER
; R4 = WORD COUNT
; R5 = SECTOR AND TRACK FOR DK.BA REGISTER

384
UNIBUS Subsystem

; THE REMAINING SECTIONS INVOLVE ACCESSES TO THE UNIBUS AND THE UBA.
; THE INITIALIZATION SEQUENCE MUST BE COMPLETE BEFORE MAKING ACCESSES
; TO THE UBA (OTHER THAN THE CONFIGURATION REGISTER) OR THE UNIBUS.

1$:  BITL  #UBA_UBIC, UBA_CNFR(R0) ; IS UNIBUS INITI ALIZATION COMPLETE?
    BEQ 1$ ; NO - KEEP TESTING
    BEQ 1$ ; YES - CONTINUE

; THIS SECTION WILL LOAD THE UBA MAP REGISTERS THAT WILL BE USED FOR THE
; TRANSFER. THE MAP REGISTERS USED FOR THE BLOCK TRANSFER MUST BE
; CONTIGUOUS. THIS PROGRAM ASSUMES THAT CONTIGUOUS PHYSICAL MEMORY
; PAGES ARE USED FOR THE TRANSFER. THE CONTENTS OF THE INITIAL MAP
; REGISTER WAS PREVIOUSLY DETERMINED AND STORED IN R2. THE PHYSICAL
; ADDRESS OF THE INITIAL MAP REGISTER WAS DETERMINED ABOVE AND STORED
; IN R1.

2$:  MOV L BCOUNT, R6         ; LOAD BYTE COUNT INTO R6
     MOV L R2, (R1)+          ; LOAD MAP REGISTER WITH CONTENTS
                      ; OF R2.
     INCL R2                 ; INCREMENT PAGE FRAME NUMBER.
                      ; (ASSUMES CONTIGUOUS PAGES OF
                      ; PHYSICAL MEMORY).
     SUB L "X200, R6         ; THERE ARE 200(HEX) BYTES PER PAGE.
                      ; ARE ALL MAPS SET UP?
     BGT R R2, (R1)+         ; NO SET UP NEXT MAP REGISTER.
                      ; SET UP ONE MORE SINCE TRANSFER
                      ; TRANSFER MAY NOT BE PAGE ALIGNED.
                      ; INVALIDATE NEXT MAP REGISTER TO
                      ; STOP THE UBA SHOULD THE TRANSFER
                      ; GO BEYOND ITS EXPECTED LIMIT.

3$:  ADD L #DK_BASE.ADD, #UNIBUS_BASE, R1 ; BASE ADDRESS OF RK611 TO R1
     MOV W #CLR, DK_CS2(R1)   ; ISSUE A RK611 SUBSYSTEM CLEAR
     MOV W #CLR, DK_CS1(R1)   ; SELECT DRIVE NUMBER
     MOV W R5, DK_DA(R1)      ; LOAD DISK ADDRESS SECTOR AND TRACK
                      ; STORED IN R5 FROM ABOVE.
     MOV W #PACKACK, DK_CS1(R1) ; ISSUE PACK ACKNOWLEDGE FUNCTION
     TSB  DK_CS1(R1)          ; WAIT FOR READY
     BOE D 3$                ; NOT READY KEEP WAITING
     MOV W CYLINDER, DK_DS(R1) ; LOAD CYLINDER ADDRESS
     MNEW R4, DK_WC(R1)       ; LOAD 2'S COMPLEMENT OF WORD COUNT
     MOV R3, DK_BA(R1)        ; LOAD LOW ORDER 16 BITS OF UNIBUS
                      ; START ADDRESS INTO DK_BA REG
     ASHL R4, #16, R3, R3    ; SHIFT UNIBUS ADDRESS RIGHT 16 BITS
     MOV L DK_FUNC, R4       ; LOAD FUNCTION INTO R4 AND
     INSV R3, #8, #2, R4     ; INSERT ADDRESS BITS 17 AND 16 FROM
                      ; R3 BITS 110 INTO BITS 910 OF
                      ; R4. EXTENDED UNIBUS ADDRESS
                      ; BITS.
     MOV W R4, DK_CS1(R1)    ; ISSUE FUNCTION AND GO TO RK611
     TSB  DK_CS1(R1)         ; WAIT FOR TRANSFER TO COMPLETE
     BOE D 4$                ; NOT COMPLETE - KEEP WAITING.

4$:  THE RK611 HAS BEEN COMPLETED - THE UBA BUFFERED DATA PATH MUST NOW BE
     PURGED. THE UBA IS REQUIRED TO MOVE ANY DATA REMAINING IN THE UBA
     TO MEMORY AND TO INITIALIZE THE BUFFERED DATA PATH FOR ANY SUBSEQUENT
     TRANSFERS. THE PURGE IS ACCOMPLISHED BY SETTING THE BNE BIT OF THE
     DATA PATH REGISTER USED BY THE TRANSFER.

     COMPUTE OFFSET OF DATA PATH REGISTER USED FOR TRANSFER AND PUT IN R2

     ASHL #2, DP_NUM, R2     ; MULTIPLY DP_NUM BY 4 -> R2
     ADD D UBA_DP_ASSOC, R2  ; ADD IN OFFSET OF DATA PATH REG 0
     MOV L #UBA_DP_ASSOC, UBA_BASE(R2) ; SET BNE BIT OF DATA PATH REGISTER
                      ; USED BY THE TRANSFER.
     BITL #UBA_DP_BNE, UBA_BASE(R2) ; TEST FOR ANY ERRORS THAT MAY HAVE
                      ; OCCURRED WITHIN THE UBA BUFFER
                      ; TRANSFER.
UNIBUS Subsystem

BEOL  5$  ; CONTINUE IF THERE WERE NO ERRORS
HALT  ; HALT FOR ERROR DETECTED BY DATA
       ; PATH REGISTER. UBA STATUS REGISTER
       ; SHOULD CONTAIN ERROR BIT.

5$:  TSTW  DK_CS1(R1)  ; TEST FOR RK611 ERROR
     BGEQ  6$  ; CONTINUE IF THERE WERE NO ERRORS
     HALT  ; HALT FOR ERROR DETECTED IN RK611

6$:  MTBR  $BEL1, $TXDB  ; RING BELL ON CONSOLE IF NO ERRORS

; THE TRANSFER PARAMETERS ARE SPECIFIED BELOW:
MEMSAD:  .LONG  4567  ; MEMORY START ADDRESS
BCOUNT:  .LONG  3210  ; NUMBER OF BYTES OF TRANSFER
MAP_REG:  .LONG  25  ; STARTING MAP REGISTER TO BE USED FOR TRANSFER.
DF_NUM:  .LONG  5  ; UBA DATA PATH TO BE USED FOR TRANSFER.
DRIVE:  .LONG  0  ; DRIVE NUMBER TO BE USED FOR TRANSFER
TRACK:  .LONG  2  ; STARTING TRACK
CYLINDER:  .LONG  4  ; STARTING CYLINDER
SECTOR:  .LONG  6  ; STARTING SECTOR
DK_FUNC:  .LONG  READ  ; DISK FUNCTION

.END BEGIN
CHAPTER 19
MASSBUS SUBSYSTEM

INTRODUCTION
The MASSBUS adapter (MBA) is the hardware interface between the Synchronous Backplane Interconnect and the high speed MASSBUS storage devices. The MASSBUS is the communication path linking the MASSBUS adapter to the mass storage device drives.

The MASSBUS adapter performs the following functions:
- Mapping of addresses from virtual (program) to physical (SBI).
- Data buffering between main memory transfer to the MASSBUS and vice versa.
- Transfer of interrupts from MASSBUS device to the SBI.

The VAX-11/780 will support a maximum of four MASSBUS adapters, each adapter supporting up to eight device controllers. A MASSBUS

Figure 19-1 MASSBUS Subsystem Configuration
adapter will support any combination of mass storage devices. Each magnetic tape controller will support up to eight tape drives. Each disk controller will support a single disk drive. Only one controller can transfer data at any one given time. The data transfer rate is dependent upon the particular mass storage device being accessed. Figure 19-1 illustrates a typical MASSBUS subsystem configuration.

The MASSBUS is comprised of 54 signal lines divided into two independent groups: the asynchronous control path (bus) and the synchronous data path (bus). Table 19-1 describes individual MASSBUS signal line function.

**Table 19-1  MASSBUS Line Descriptions**

<table>
<thead>
<tr>
<th>SIGNAL LINE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTROL BUS</td>
<td></td>
</tr>
<tr>
<td>Control and Status (C00-15)</td>
<td>Transfers 16 parallel control or status bits to or from the drive.</td>
</tr>
<tr>
<td>Control Bus Parity (CPA)</td>
<td>Transfers odd control bus parity to or from the drive. Parity is simultane-</td>
</tr>
<tr>
<td></td>
<td>ously transferred with control bus data.</td>
</tr>
<tr>
<td>Drive Select (DS0-2)</td>
<td>Transfers a 3-bit binary code from the MBA to select a controller. The</td>
</tr>
<tr>
<td></td>
<td>drive responds when the (unit) select switch in the controller corresponds</td>
</tr>
<tr>
<td></td>
<td>to the transmitted binary code.</td>
</tr>
<tr>
<td>Register Select (RSO-4)</td>
<td>Transfers a 5-bit binary code from the MBA to select a particular drive</td>
</tr>
<tr>
<td></td>
<td>register.</td>
</tr>
<tr>
<td>Controller to Drive (CTOD)</td>
<td>Indicates in which direction information is to be transferred on the con-</td>
</tr>
<tr>
<td></td>
<td>trol bus. For a controller-to-drive transfer, the MBA asserts CTOD; for a</td>
</tr>
<tr>
<td></td>
<td>drive-to-controller transfer, the MBA negates CTOD.</td>
</tr>
<tr>
<td>Demand (DEM)</td>
<td>Asserted by the MBA to indicate a transfer is to take place on the control</td>
</tr>
<tr>
<td></td>
<td>bus. For a controller-to-drive transfer, DEM is asserted by the MBA when</td>
</tr>
<tr>
<td></td>
<td>data is present. For a drive-to-controller transfer, DEM is asserted by</td>
</tr>
<tr>
<td></td>
<td>the MBA to request data and is</td>
</tr>
<tr>
<td>SIGNAL LINE</td>
<td>DESCRIPTION</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Transfer (TRA)</td>
<td>Asserted by the drive in response to DEM. For a controller-to-drive transfer, TRA is asserted when the data is strobed and negated when DEM is removed. For a drive-to-controller transfer, TRA is asserted when the data is asserted on the bus and negated when the negation of DEM is received.</td>
</tr>
<tr>
<td>Attention (ATTN)</td>
<td>The drive asserts this line to signal the MBA of any change in drive status or an abnormal condition. ATTN is asserted any time a drive's ATA status bit is set. ATTN is common to all drives and may be asserted by more than one drive at a time.</td>
</tr>
<tr>
<td>Initialize (INIT)</td>
<td>Asserted by the MBA to initialize all drives on the bus. This signal is transmitted whenever the MBA receives an initialize command.</td>
</tr>
<tr>
<td>Fail (FAIL)</td>
<td>When asserted, this line indicates a power fail condition has occurred in the MBA or the MBA is in maintenance mode.</td>
</tr>
</tbody>
</table>

**DATA BUS**

<table>
<thead>
<tr>
<th>DATA BUS</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data (D00-15)</td>
<td>These bidirectional lines transfer 16 parallel data bits between the MBA and drives.</td>
</tr>
<tr>
<td>Data Bus Parity (DPA)</td>
<td>Transfers an odd parity bit to or from the drive. Parity is simultaneously transferred with bits on the data bus.</td>
</tr>
<tr>
<td>Sync Clock (SCLK)</td>
<td>Asserted by the drive during a read operation to indicate when data on the data bus is to be strobed by the MBA. During a write operation SCLK is asserted to the MBA to indicate the rate at which data would be presented by the MBA on the data bus.</td>
</tr>
</tbody>
</table>
### SIGNAL LINE

<table>
<thead>
<tr>
<th>SIGNAL LINE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write Clock (WCLK)</td>
<td>Asserted by the MBA to indicate when data written to the drive is to be strobed.</td>
</tr>
<tr>
<td>Run (RUN)</td>
<td>Asserted by the MBA to initiate data transfer command execution. During a data transfer, the drive samples RUN at the end of each sector. If RUN is still asserted, the drive continues the transfer into the next sector; if RUN is negated, the drive terminates the transfer.</td>
</tr>
<tr>
<td>End-of-Block (EBL)</td>
<td>Asserted by the drive at the end of each sector. For certain error conditions where it is necessary to terminate operations immediately, EBL is asserted prior to the normal time. In this case, the transfer is terminated prior to the end of the sector.</td>
</tr>
<tr>
<td>Exception (EXC)</td>
<td>Asserted by the drive or MBA to indicate an error condition during a data transfer command. EXC remains asserted until the trailing edge of the last EBL pulse.</td>
</tr>
<tr>
<td>Occupied (OCC)</td>
<td>Indicates acceptance of a valid data transfer command.</td>
</tr>
</tbody>
</table>

### MASSBUS ADAPTER OPERATION
The MASSBUS adapter consists of an SBI/MBA interface, internal registers, control paths and data paths. Figure 19-3 is a simplified block diagram of the MBA. A tristate internal bus connects the SBI module to the internal registers, control paths, and data paths, and provides for the passage of data to the various functional blocks.

The MBA accepts and executes commands from the CPU and reports the necessary status changes and fault conditions to the CPU. The MBA can transfer register data or a block of data to or from a MASSBUS device. A 256 X 32-bit (bits <30:21> are not writable) RAM stores the physical page addresses of the block of data to be transferred. The memory data (64 bits) will be sent in words (16 bits) to the MASSBUS drive in the order of the first word (bits 15 to 0), followed by the second word (bits 31 to 16) of a long word. Special diagnostic features are built in the hardware to allow on-line diagnosis of the MBA and MASSBUS drives.
MASSBUS Subsystem

Figure 19-2 illustrates the MASSBUS signal line configuration.

Figure 19-2  MASSBUS Signal Line Configuration

Figure 19-3  MASSBUS Adapter
The MBA is capable of handling a MASSBUS drive with a maximum data transfer rate of 16 bits per \( \mu \)sec via the 16-bit wide MASSBUS data path. The MASSBUS adapter controls data transfers between MASSBUS devices and physical memory. A MASSBUS adapter can transfer 16 bits at a time to a mass storage device or it can receive 16 bits at a time from a MASSBUS drive. The MBA contains a 32-byte buffer used to store data enroute to either main memory or mass storage. Transfers (data only) along the SBI, to or from main memory, occur in 64-bit (8-byte) increments. Therefore, there are four MASSBUS transfers (16 bits each) per SBI transaction. The MASSBUS adapter will accept only aligned longword reads and writes to its external or internal registers. An attempt to address a nonexistent register in the MASSBUS adapter will prompt a no-response confirmation.

**MBA Registers**

There are two sets of registers in the MBA address space: internal and external. The MBA internal registers are the registers which are physically located in the MBA. The external registers are located in the MASSBUS drives and are drive-dependent.

There are eight internal registers and a 256 \( \times \) 32-bit RAM. The internal registers primary function is to monitor MBA and operating status conditions. The internal registers also control certain phases of the data transfers between the SBI and the MASSBUS device such as:

- maintaining a byte count to ensure that all of the data to be transferred has been accounted for
- converting virtual addresses to physical addresses for referencing data in memory

The eight internal registers are:

- MBA Configuration Register (CSR)
- MBA Control Register (CR)
- MBA Status Register (SR)
- MBA Virtual Address Register (VAR)
- MBA Byte Count Register (BCR)
- MBA Diagnostic Register (DR)
- MBA Selected Map Register (SMR)
- MBA Command Address Register (CAR)

**NOTE**

The selected map register and the command address register are read only and are valid only during data transfers.
MASSBUS Subsystem

The MBA contains 256 32-bit map registers which are used to map program virtual addresses into SBI physical addresses. Bits \(<30:21>\) of the map register are reserved and are not writable. The mapping registers allow transfers to or from contiguous or non-contiguous physical memory. Figure 19-4 illustrates the mapping of a virtual address to an SBI address.

![Virtual to SBI Address Translation Diagram](image)

*Figure 19-4  Virtual to SBI Address Translation*

**CONTROL PATH**

The control path handles the transfer of control data to and from the MASSBUS devices. Certain sections of the MBA address space map into registers physically located within MASSBUS devices. The MASSBUS control path is used to communicate with these data path registers.

The data path controls the data transferred to and from the MASSBUS device and the SBI. The 32-bit SBI data word is divided into 16-bit (2-
MASSBUS Subsystem

byte) segments required as data on the MASSBUS. When performing a read from MASSBUS device the data path assembles the two 8-bit bytes from the MASSBUS into the 32-bit SBI format. A silo and input/output data buffer provide the means for smoothing the data transfer rate. The data path also contains a write check circuit which can be used under program control to verify the accuracy of the data transfer function.

MBA ACCESS
Each SBI device (NEXUS) is assigned a 2048, 32-bit longword (8K byte) control address space. This space is accessible as part of the SBI I/O longword address space. The command/address format used to access the MBA registers is illustrated in Figure 19-5.

![Figure 19-5 MASSBUS Adapter Addressing Format](Physical Byte Address)

Bit <29> = 1 I/O Address space
Bits <28:17> All zeros
Bits <16:13> Transfer request number of this MBA
Bits <11:10> MBA internal register
0 0 Bits <9:5> = must be zero
0 1 MBA external register
Bits <9:7> = device select
Bits <6:0> = register select
1 0 MBA MAP
Bits <9:0> = MAP address
1 1 Invalid (No response to an address with these bits on)

INTERNAL REGISTERS
The MBA internal registers are described as follows:

MBA Configuration/Status Register (Byte Offset=0)
**Bit: 31**  **Name:** SBI parity error  
**Function:** Set when an SBI parity error is detected. Cleared by power fail or the deassertion of fault signal. Setting of this bit will cause fault to be asserted on SBI.

**Bit: 30**  **Name:** Write data sequence (WS)  
**Function:** Set when no write data is received (neither tag = write data nor ID) following a write command. Cleared by power fail or the deassertion of fault signal. The setting of this bit will cause the assertion of fault on SBI.

**Bit: 29**  **Name:** Unexpected read data (URD)  
**Function:** Set when read data is received when it is not expected. Cleared by power fail or the deassertion of fault signal. The setting of this bit will cause assertion of fault on SBI.

**Bit: 28**  **Function:** This bit must be zero.

**Bit: 27**  **Name:** Multiple transmitter (MT)  
**Function:** Set when the ID on the SBI does not agree with the ID transmitted by MBA while MBA is transmitting information on the SBI. Cleared by power fail or the de-assertion of fault signal. The setting of this bit will cause the assertion of fault on SBI.

(Fault signal will be asserted at the normal confirmation time for one cycle if MBA detects one of the fault conditions. The negation of the fault signal on the SBI will clear all the fault status bits).

**Bit: 26**  **Name:** XMTFLT  
**Function:** Set when SBI fault is detected at the second cycle after MBA transmits information to the SBI. Cleared by power fail or the deassertion of fault signal.

**Bit: 25**  **Name:** Zeros  
**Function:** Reserved for future use.

**Bit: 23**  **Name:** Adapter power down (PD)  
**Function:** Set when the MBA receives assertion of AC LO. Clear when MBA power goes up. Cleared by assertion of INIT, UNJAM, DC LO, or writing one to this bit. The setting of this bit will cause interrupt to CPU.

**Bit: 22**  **Name:** Adapter power up (PU)  
**Function:** Set when MBA receives the deassertion of AC LO. Reset when MBA power goes down. Cleared by assertion of INIT, UNJAM,
DC LO or writing a one to this bit. The setting of this bit will set IE bit and interrupt CPU.

**Bit: 21  Name: Over temperature (OT)**
**Function:** Zero

**Bit: 20:8  Name: All zeros**
**Function:** Reserved for future use.

**Bit: 7:0  Name:**
**Function:** Each adapter is assigned a unique code identifying it. MBA adapter code is: bits <7:0> =00100000.

**MBA Control Register (Byte Offset =4)**

<table>
<thead>
<tr>
<th>31</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

**Bit: 31:4  Name: All zeros**
**Function:** Reserved for future use.

**Bit: 3  Name: MB Maintenance Mode**
**Function:** The setting of this bit will put MBA in the maintenance mode which will allow the diagnostic programmer to exercise and examine the MASSBUS operations without a MASSBUS device. When this bit is set, MBA will block RUN, DEM, and assert FAIL to MASSBUS so that all the devices on MASSBUS will detach from the MASSBUS. The MBA cannot be put in maintenance mode while a data transfer is in progress.

**Bit: 2  Name: Interrupt Enable**
**Function:** Set by writing a one or power up which allows MBA to interrupt CPU when certain conditions occur. Cleared by writing zero or INIT.

**Bit: 1  Name: ABORT**
**Function:** Abort data transfer. Write one to set. The setting of this bit will initiate the data transfer abort sequences which will stop sending commands, stop address and byte counter.
- Negate Run
- Assert EXEC to MASSBUS
- Wait for EBL
- Set DTABT to one at the trailing edge of EBL
- Interrupt CPU if IE bit is one

This bit will be cleared by writing a zero, INIT or UNJAM.

**Bit: 0  Name: Initialization (INIT)**
**MASSBUS Subsystem**

**Function:** The bit is self-clearing. It will always read as zero. The setting of this bit will:
- Clear status bits in MBA Configurator register
- Clear ABORT and IE in MBA Control register
- Clear MBA Status register
- Clear MBA Byte Count register
- Clear control and status bits of diagnostic registers
- Cancel all pending commands except Read Data Pending
- Abort data transfer
- Assert MASSBUS INIT

**MBA Status Register (Byte Offset=8)**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>DTBUSY</td>
<td>Data transfer busy. Bit is set when a data transfer command is received. It is cleared when data transfer is terminated normally or when a data transfer is aborted.</td>
</tr>
<tr>
<td>30</td>
<td>NRCONF</td>
<td>No response confirmation. This bit is set when the MBA receives a no response confirmation for the read command or write command and write data sent to the SBI. It is cleared by writing a one to the bit or INIT. The setting of this bit will cause retry of the command.</td>
</tr>
<tr>
<td>29</td>
<td>CRD</td>
<td>Corrected read data. This bit is set when TAG of read data received from memory is CRD. It is cleared by writing a one to this bit or by INIT.</td>
</tr>
<tr>
<td>28:20</td>
<td>All zeros.</td>
<td>Reserved for future use.</td>
</tr>
<tr>
<td>19</td>
<td>PGE</td>
<td>The PGE bit is set when one or more of the following conditions exists: Program tries to initiate a Data Transfer when MBA is currently performing one. Program tries to load MAP, VAR, or Byte counter when MBA is currently performing a Data Transfer operation.</td>
</tr>
</tbody>
</table>
MASSBUS Subsystem

Program tries to set MB Maintenance Mode during a Data Transfer operation.

The bit is cleared by writing a one. The setting of this bit will cause an interrupt to the CPU if IE is set.

Bit: 18  Name: NFD  
Function: Nonexisting drive. This bit is set when drive fails to assert TRA within 1.5 \( \mu \text{sec} \) after assertion of DEM. The bit is cleared by writing a one to the bit. The setting of this bit will send zero read data back to the SBI, and interrupt CPU if IE is set.

Bit: 17  Name: MCPE  
Function: MASSBUS control parity error. This bit is set when a MASSBUS control parity error occurs. It is cleared by writing a one to the bit. The setting of this bit will cause an interrupt to CPU if IE is set.

Bit: 16  Name: ATTN  
Function: Attention from MASSBUS. Asserted when the attention line in the MASSBUS is asserted. The assertion of this bit will cause an interrupt to the CPU if IE is set.

Bit: 15:14  Name: All zeros.  
Function: Reserved for future use.

Bit: 13  Name: DT CMP  
Function: Data transfer completed. This bit is set when the data transfer is terminated either due to an error or normal completion. It is cleared by writing a one to this bit or INIT. The setting of this bit will cause an interrupt to the CPU if IE bit in control register is set.

Bit: 12  Name: DTABT  
Function: Data transfer aborted. This bit is set with the trailing edge of the EBL when data transfer has been aborted. It is cleared by writing a one to this bit or INIT. The setting of this bit will cause an interrupt to the CPU if IE bit is set.

Bit: 11  Name: DLT  
Function: Data late. This bit is set when:  
1) For a write data transfer or write check data transfer, data buffer is empty when WCLK is sent to MASSBUS.

or

2) for a read data transfer, the data buffer is full while SCLK is received from MASSBUS. This bit is cleared by writing a one to it or INIT. The setting of this bit will cause the data transfer operation to be aborted.

DLT will most likely be set if the system is in single step operation and if the MBA is not in maintenance mode.
MASSBUS Subsystem

Bit: 10  Name: WCK UP ERR
Function: Write Check Upper Error. This bit is set when a compare error is detected in the upper byte while MBA is performing a write check operation. It is cleared by writing a one to this bit or INIT. The setting of this bit will cause the data transfer operation to be aborted.

Bit: 9   Name: WCK LWR ERR
Function: Write Check Lower Error. This bit is set when a compare error is detected in the lower byte while MBA is performing a write check operation. It is cleared by writing a one to this bit or INIT. The setting of this bit will cause the data transfer operation to be aborted.

Bit: 8   Name: MXF
Function: Miss transfer error. This bit is set when an SCLK or OCC is not received within 500 µsec after data transfer busy is set. It is cleared by writing a one to this bit or INIT. The setting of this bit will cause an interrupt to the CPU if IE bit in control register is set.

Bit: 7   Name: MBEXC
Function: MASSBUS Exception. This bit is set when EXC is received from MASSBUS. It is cleared by writing a one to this bit or INIT. The setting of this bit will cause the data transfer operation to be aborted.

Bit: 6   Name: MDPE
Function: MASSBUS data parity error. This bit is set when the MASSBUS data parity error is detected during a read data transfer operation. It is cleared by writing a one to this bit or INIT. The setting of this bit will cause the data transfer operation to be aborted.

Bit: 5   Name: MAPPE
Function: Page Frame Map Parity Error. This bit is set when a parity error is detected on the page frame number read from the PF map. It is cleared by writing a one to this bit or INIT. The setting of this bit will cause the data transfer operation to be aborted.

Bit: 4   Name: INVMAP
Function: Invalid map. This bit is set when the valid bit of the next page frame number is zero when byte count is not zero. It is cleared by writing a one to this bit or INIT. The setting of this bit will cause the data transfer operation to be aborted.

Bit: 3   Name: ERR CONF
Function: Error Confirmation. This bit is set when the MBA receives an error confirmation for the read command or write command. It is cleared by writing a one to this bit or INIT. The setting of this bit will cause the data transfer operation to be aborted.

Bit: 2   Name: RDS
Function: Read Data Substitute. This bit is set when the tag of the
read data received from memory is read data substitute. It is cleared by writing a one to this bit or INIT. The setting of this bit will cause the data transfer operation to be aborted.

**Bit: 1 Name: IS TIMEOUT**

**Function:** Interface Sequence Timeout. An interface sequence is defined as the time from when arbitration for the SBI is begun until:

1) ACK is received for a command address transfer that specifies read or,
2) ACK is received for a command address transfer that specifies write and ACK is also received for each transmission of write data or,
3) ERR confirmation is received for any command/address transfer. The maximum timeout is 102.4 μsecs. The setting of this bit will cause data transfer abort. Cleared by writing a one to this bit or INIT.

**Bit: 0 Name: RD TIMEOUT**

**Function:** Read Data Timeout is defined as the time from when an interface sequence that specifies a read command is completed to the time that the specified read data is returned to the commander. The maximum time out is 102.4 μsecs. The setting of this bit will cause data transfer abort. Cleared by writing a one to this bit or INIT.

**MBA Virtual Address Register (Byte Offset=12)**

<table>
<thead>
<tr>
<th>31</th>
<th>17</th>
<th>16</th>
<th>9</th>
<th>8</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>MAP SELECT</td>
<td>BYTE OFFSET</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The program must load an initial virtual address (pointing to the first byte to be transferred) into this register before a data transfer is initiated. Bits 9 through 16 select one of 256 map registers. The contents of the selected map register and the values in bits 0 through 8 are used to assemble a physical SBI address to be sent to memory. Bits 0 through 8 indicate the byte offset into the page of the current data byte. Note the MBA virtual address register is incremented by 8 after every memory read or write and will not point to the next byte to be transferred if the transfer does not end on a quadword boundary. (It will point 8 bytes ahead.) Also upon a write check error, the virtual address register will not point to the failing data in memory due to the preloading of the silo data buffer. The virtual address of the bad data may be found by determining the number of bytes actually compared to the
MASSBUS Subsystem

MASSBUS (the difference between bits 16 to 31 of RS04 and their initial value) and adding that difference to the initial virtual address.

MBA Byte Counter (Byte Offset=16)

<table>
<thead>
<tr>
<th>31</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MASSBUS BYTE COUNTER (READ ONLY)</td>
<td>SBI BYTE COUNTER (READ/WRITE)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Program loads the 2's complement of the number of bytes for the data transfer to bits <15:0> of this register. MBA hardware will load these 16 bits into bits <31:16>. Bits <31:16> serve as the byte counter for the number of bytes transferred through the SBI interface. The starting byte count with 16 bits of 0's is the maximum number of bytes of a data transfer.

Diagnostic Register (Byte Offset=20)

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
</table>

The diagnostic register may only be read or written while in maintenance mode. Care must be taken while reading or bit-setting this register to insure that the data path is not loading the silo. If the data path is loading the silo while this register is read, the data may be altered.

Bit: 31  Name: IMDPG  
Function: Invert MASSBUS Data Parity Generator.

Bit: 30  Name: IMCPG  
Function: Invert MASSBUS Control Parity Generator.

Bit: 29  Name: IMAPP  
Function: Invert Map Parity.

Bit: 28  Name: BLKSCOM  
Function: Block Sending Command to SBI. During a data transfer, the setting of this bit will eventually cause a DLT bit set and CPU interrupt.

Bit: 27  Name: SIMSCLK  
Function: Simulate SCLK. When MMM bit is set, writing a one to this bit will simulate the assertion of SCLK, and writing a zero to this bit will simulate the deassertion of SCLK.

Bit: 26  Name: SIMEBL
MASSBUS Subsystem

Function: Simulate EBL. When MMM bit is set, writing a one and writing a zero to this bit will simulate the assertion and deassertion of EBL.

Bit: 25  Name: SIMOCC
Function: Simulate OCC. When MMM bit is set, writing a one and writing a zero to this bit will simulate the assertion and deassertion of OCC.

Bit: 24  Name: SIMATTN
Function: Simulate ATTN. When MMM bit is set, writing a one and writing a zero to this bit will simulate the assertion and deassertion of ATTN.

Bit: 23  Name: MDIB SEL
Function: Maintenance MASSBUS Data Input Buffer Select. When the bit is set to one, the upper eight bits (B<15:8>) of MDIB will be sent out from B<7:0> of Diagnostic Register if the Diagnostic Register is read. When this bit is zero, the lower eight bits (B<7:0>) of MDIB will be sent out from B<7:0> of Diagnostic Register if a bit is read.

Bit: 22:21  Name: MAINT ONLY
Function: Read/write with no effect. (Used to test writability of these bits).

Bit: 20  Name: MFAIL
Function: MASSBUS Fail (read-only). Fail is asserted when MMM is set.

Bit: 19  Name: MRUN
Function: Maintenance MASSBUS Run (read-only).

Bit: 18  Name: MWCLK
Function: Maintenance MASSBUS WCLK (read-only).

Bit: 17  Name: MFXC
Function: Maintenance MASSBUS FXC (read-only).

Bit: 16  Name: MCTOD
Function: Maintenance MASSBUS MCTOD (read-only).

Bit: 15:13  Name: MDS
Function: Maintenance MASSBUS Device Select (read-only).

Bit: 12:8  Name: MRS
Function: Maintenance MASSBUS Register Select (read-only).

Bit: 7:0  Name: U/L MDIB
Function: Maintenance Upper/Lower MDIB.

Selected Map Register (Byte Offset = 24)
This register is read-only and has the same format as a map register.
but is valid only when DT Busy is set. This is the contents of the map register pointed to by bits 16 through 9 of the virtual address register.

**Command Address Register (Byte Offset=28)**
This register is read-only and valid only when DT Busy is set. It is the value of bits $<31:0>$ of the SBI during the Command/Address part of the MBA's next data transfer cycle.

**MBA External Register (Byte Offset=400 to 7FC)**
External registers are MASSBUS device-dependent. Each device has a maximum of 32 registers.

**MBA Map (Byte Offset=800 to BFC)**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Valid Bit</td>
</tr>
<tr>
<td>30:21</td>
<td>Reserved for future use.</td>
</tr>
<tr>
<td>20:0</td>
<td>Physical Page Frame Number</td>
</tr>
</tbody>
</table>

The MBA contains 256 map registers, each of which may be selected by address bits 0 to 9 when bits $<11:10>$ are 10. Map registers can only be written when there is no data transfer operation in progress. A write to a map register during a data transfer will be ignored and cause the setting of PGE.

**Data Transfer Program Flow**

1) Initialize MASSBUS Adapter.
2) Mount pack into drive.
3) Start drive spinning.
4) Wait for ready light.
5) Issue Pack ACK to drive.
6) Load desired cylinder, sector, track, and registers in drive.
7) Load starting virtual address into MBA's virtual address register.
8) Load 2's complement of number of bytes to be transferred into byte count register in MBA.
9) Load starting map (pointed to by bits $<9:16>$ of VAR) with physical page address.
10) Load successive maps with physical addresses to rest of pages.
11) Issue read/write command to drive.

**EXAMPLE**
Presented is a program to read data from the RP05/RP06 disk subsys-
MASSBUS Subsystem

tem into memory. The program is written in the VAX-11 MACRO assembly language. It is not meant to run with memory management enabled, and will not run under VAX/VMS. This program illustrates the procedures involved in setting up the MASSBUS adapter to transfer bytes of data to memory. In order to run the program, it must be loaded from the floppy disk by the console into memory. Initially, the program can be assembled and linked under VAX/VMS and then transferred to the floppy disk using the RSX-11M utility program FLX.

; PROGRAM TO READ FROM THE RP05/6 INTO MEMORY
;
; SYMBOL DEFINITIONS

MBA_BASE = "X20010000"
MBA_CSR = "X0"
MBA_CR = "X4"
MBA_INIT = "X1"
MBA_SR = "X8"
MBA_DT_BUSY = "X80000000"
MBA_DT_ABORT = "X2000"
MBA_VAR = "XC"
MBA_BCR = "X10"
MBA_MAP0 = "X8000"
MAP_VALID = "X80000000"
RP1_CSR = "X480"
RP_SET_VV_CMD = "X13"
RP_READ_CMD = "X39"
RP1_DA = "X494"
RP1_OF = "X4A4"
RP_16BIT_FORMAT = "X1000"
RP1_DC = "X4A8"

; MBA AT TR=R
; OFFSET TO SBI STATUS REGISTER
; OFFSET TO MBA CONTROL REGISTER
; INITIALIZE BIT
; OFFSET TO MBA STATUS REGISTER
; DTBUSY STATUS BIT
; DTABORT STATUS BIT
; OFFSET TO MBA VIRTUAL ADDRESS REGISTER
; OFFSET TO MBA BYTE COUNT REGISTER
; OFFSET TO FIRST MAP REGISTER
; VALID BIT IN MAP REGISTER
; OFFSET TO CSR ON RP DRIVE 1
; SET VOLUME VALID COMMAND
; READ COMMAND
; OFFSET TO SECTOR AND TRACK REGISTER
; OFFSET TO TRACK OFFSET REGISTER
; 16BIT FORMAT ON DISC SURFACE
; OFFSET TO DESIRED CYLINDER REGISTER

; THIS SECTION LOADS 0 WITH THE ADDRESS OF THE MBA,
; INITIALIZES THE MBA, SET VOLUME VALID AND THE 16BIT
; FORMAT BITS IN THE DRIVE. THESE OPERATIONS (SET VV
; AND 16BIT) ONLY NEED TO BE DONE WHEN A NEW PACK IS
; MOUNTED OR WHEN POWER COMES UP.
BEGIN:

MOVL $MBA_BASE,R0 ; LOAD MBA'S ADDRESS INTO RO
MOVL $MBA_INIT,MBA_CR(R0) ; INITIALIZE MBA
MOVL $RP_SET_VV_CMD,RP1_CSR(R0) ; SET VOLUME VALID
MOVL $RP_16BIT_FORMAT,RP1_OF(R0) ; SET 16BIT FORMAT

; THIS SECTION LOADS THE MAPS. THE STARTING PHYSICAL
; ADDRESS IS SHIFTED RIGHT 9 BITS TO BECOME A PHYSICAL
; PAGE ADDRESS. THE MAP VALID BIT IS OR'ED IN AND MAP
; REGISTER 0 IS LOADED. SUCCESSIVE MAP REGISTERS ARE
; LOADED WITH THE PHYSICAL PAGE ADDRESSES OF THE REST
; OF THE DATA AREA IN MEMORY, THEN THE VIRTUAL ADDRESS
; REGISTER IS LOADED WITH THE OFFSET TO THE FIRST
; BYTE OF THE DATA AREA.

MOVL BCOUNT,R2 ; # OF BYTES TO TRANSFER
MOVL MEMBA,R3 ; STARTING BYTE ADDRESS OF TRANSFER
MOVAL MBA_MAP0(R0),R1 ; R1 HAS ADDRESS OF FIRST MBA MAP
ASHL #-9,R3,R4 ; TURN STARTING ADDRESS INTO PAGE ADDRESS

406
**MASSBUS Subsystem**

```assembly
1$: BISL3  #MAP_VALID,R4,(R1)+ ; SET VALID BIT AND MOVE INTO MAP
INCL  R4  ; BUMP PAGE ADDRESS
SUBL #"X200,R2  ; SET UP ALL MAPS?
BGTR  1$  ; NO, SO SET UP NEXT MAP
BISL3  #MAP_VALID,R4,(R1)+ ; ENABLE ONE MORE MAP SINCE
CLRL  (R1)  ; TRANSFER MAY NOT BE PAGE ALIGNED
CLRL  ; CLEAR VALID BIT IN UNUSED MAP
CLRL  ; THIS WILL STOP THE MBA IF IT SHOULD
CLRL  ; DO SOMETHING WRONG AND TRY TO USE
CLRL  ; THIS MAP ENTRY
BICL  #"XFFFFFFE00,R3  ; LEAVE ONLY BYTE OFFSET INTO PAGE
MOVL  R3,RBA_VAR(R0)  ; LOAD VIRTUAL ADDRESS REGISTER.

THIS SECTION LOADS THE MBA BYTE COUNT REGISTER, AND THE
adder registers in the disc.

MNEBL  BCDOUNT,RBA_BCR(R0)  ; 2'S COMPLEMENT OF # BYTES TO XFER
MOVL  SCYL,RP1_DC(R0)  ; SET DESIRED CYLINDER
MOVL  SECTOR,R2
INSV  STACK,#8,#8,R2  ; CREATE IMAGE OF SECTOR/TRACK REGISTER
MOVL  R2,RP1_DA(R0)  ; LOAD REGISTER IN DRIVE

WE ARE NOW READY TO TRANSFER, THE COMMAND WILL BE WRITTEN
TO THE RP0_CSR, AND THE PROGRAM WILL MONITOR THE DATA TRANSFER
BUSY BIT IN THE MBA STATUS REGISTER. WHEN THE BIT IS CLEARED
THE TRANSFER IS OVER. THE ABORT BIT WILL BE CHECKED TO SEE IF
THERE WAS AN ERROR. IF THE TRANSFER WAS SUCCESSFULL THE ERROR
BIT WILL BE CLEAR.

MOVL  #RP_READ_CMD,RP1_CSR(R0)  ; INITIATE READ OPERATION
BITL  #MBA_DT_BUSY,RBA_SR(R0)  ; DIBUSY STILL SET?
BNED  2$  ; YES -> TRANSFER STILL IN PROGRESS
BITL  #MBA_DT_ABORT,RBA_SR(R0)  ; XFER ABORTED?
BNED  3$  ; YES

HALT  ; NORMAL COMPLETION, SUCESS
HALT  ; ERROR COMPLETION. MBA STATUS REGISTER SHOULD
       ; CONTAIN ERROR INFORMATION.

THE STARTING ADDRESSES ARE SPECIFIED BELOW

MEMSAD:  .LONG  3456  ; MEMORY STARTING ADDRESS
BCOUNT:  .LONG  4321  ; NUMBER OF BYTES TO TRANSFER
SCYL:    .LONG  322  ; STARTING CYLINDER NUMBER
STRACK:  .LONG  4  ; STARTING TRACK
SECTOR:  .LONG  6  ; STARTING SECTOR
```

$
CHAPTER 20
INTERCONNECTS

OVERVIEW
All VAX-11/780 hardware system elements, such as the CPU, main memory, and MASSBUS and UNIBUS Adapters, are connected to the Synchronous Backplane Interconnect (SBI) by devices called interconnects. In the case of the MASSBUS and UNIBUS Adapters, the interconnect also allows the connection of various DIGITAL- and user-designed peripherals to the system via the appropriate bus structure.

The MASSBUS and UNIBUS Adapters interconnects are designed to process data transfers at rates up to the potentials of the compatible peripherals. However, the SBI operates at even faster speeds, up to 13.3 Mb/second, and in 32-bit versus 16-bit increments. To permit VAX users with exceptionally fast I/O requirements to take advantage of the speed and capabilities of the SBI, a special high performance 32-bit parallel interface, the DR780, is available.

Other VAX users with multiprocessor configurations working on complex and intricate problems can enhance the capabilities of their systems with the VAX-11/780 multiport memory option, MA780. The user can utilize shared memory among up to four processors with a throughput rate of up to 11 Mb/second.

This chapter will discuss the features, benefits and operation of these optional, very high performance VAX-11/780 interconnects.

DR780 HIGH PERFORMANCE 32-BIT PARALLEL INTERFACE

<table>
<thead>
<tr>
<th>Features</th>
<th>Benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.67 Mb/second transfer rate</td>
<td>Exceptionally fast, full 32-bit I/O operations</td>
</tr>
<tr>
<td>Command chaining</td>
<td>Allows multiple I/O operations to occur without software intervention</td>
</tr>
<tr>
<td>Dynamic memory mapping performed in hardware</td>
<td>Reduced I/O overhead; applications programs and I/O driver completely independent from memory mapping</td>
</tr>
</tbody>
</table>
**Features**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full VAX/VMS software support</td>
<td>Library of high-level language support routines and complete I/O driver reduce software development time for customer applications</td>
</tr>
<tr>
<td>Symmetric point-to-point interconnect</td>
<td>Enables two VAX-11/780 systems to be connected through two DR780 options in a user written application</td>
</tr>
<tr>
<td>Separate control and data interconnects</td>
<td>Provides for concurrent 32-bit data and 8-bit control transfers</td>
</tr>
<tr>
<td>User generated command packets processed without operating system intervention</td>
<td>Provides a direct link between the DR780 and the user process</td>
</tr>
<tr>
<td>Unrestricted maximum transfer size</td>
<td>Transfers can be of indefinite length, limited only by the size of physical memory</td>
</tr>
</tbody>
</table>

**INTRODUCTION**

The DR780 is a very high performance interface that allows user devices to be connected directly to the VAX-11/780 SBI. It is capable of transferring data to and from memory at speeds of up to 6.67 Mb per second. The DR780 also permits intelligent 32-bit parallel data transfers to a second DR780-equipped VAX-11/780 system, enabling high speed interprocessor communication.

The DR780 is fully software supported by the VAX/VMS operating system with a simple, easy-to-use I/O driver and a library of high level language support routines. This allows the user to avoid much of the work and time associated with writing device handlers.

Data verification is supported on the DR780 by parity checking on both control and data transfers. Other DR780 features that enhance system integrity and maintainability include: error detection and logging, address range checking on data transfers and on-line diagnosis.

Applications that previously required very complex and time-consuming hardware and software development on the part of the user can be much more readily implemented with the DR780.

Figure 20-1 shows a typical VAX-11/780 configuration with a DR780 incorporated in block diagram form.
DR32 DEVICE INTERCONNECT (DDI)
The DR780 is an interface adapter that connects the internal memory bus of the VAX-11/780 processor to a user-accessible bus called the DR32 Device Interconnect (DDI). Two DR780s can be connected to form a VAX-11/780 processor-to-processor link. Figure 20-2 shows the relationship of the DR780 to the VAX-11/780 and the DDI.

As a general purpose data port, the DR780 is a capable of moving continuous streams of data to or from memory at high speed. Data moved from a user device to disk storage must go through an intermediate buffer in main memory.

The DR32 Device Interconnect (DDI) is a bidirectional bus for the transfer of data and control signals. Control signals sent over the DDI are asynchronous and interlocked; data transfers are synchronized with clock signals. Any connection to the DDI is called a DR-device. The DDI provides a point-to-point connection between two DR-devices, one of which must be a VAX processor. The DR-device connected to the external end of the DDI is called the far end DR-device.

The DDI actually consists of two separate and independent interconnects: one for control signals and one for data. The control interconnect is an asynchronous 8-bit bidirectional path for transferring control information to and from a user device. A great deal of flexibility for the design of the user device is available since the width of the
control interconnect (8 bits) enables the DR780 to address up to 256 individual registers. These registers can pass various kinds of information, such as buffer descriptors, commands, or status, to the user process.

The data interconnect is a synchronous 32-bit bidirectional path for data which is synchronized to a single clock. The user can choose to use the internal clock to the DR780 or provide one in the far end device. The transfer rate of the DR780 is selectable under program control in the range of 0.156 to 6.67 Mb/sec. Actual throughput obtained, however, is configuration and application dependent.

**Command Chaining**

Command chaining is the execution of commands without software intervention for each command. Commands are chained in the sense that they follow each other on a queue. After a QIO function starts the DR780, any number of DR780 commands can be executed during that QIO operation. This process continues until the transfer is halted (a command packet is fetched that specifies a Halt command) or an error occurs.

The DR780 can perform **dynamic command chaining**. This means
that commands can be added and even deleted from the command queue while the DR780 is operating. Four features of the DR780 allow command chaining as follows:

- **Fetches commands continuously from main memory**—The DR780 is an intelligent controller whose microprocessor requests command information and transfers data continuously from the virtual memory of the user process without program intervention. This is accomplished through the use of queues.

- **Direct communication between the DR780 and the user process via queues**—This feature provides for greater efficiency and flexibility by allowing the user process to control the DR780 directly. It also means that the I/O driver can be the same for a wide range of applications. Since the user process has the task of building command packets, the I/O driver just performs privileged tasks for the user process, such as supplying the DR780 with the address range of operation, fielding interrupts and aborting the current operation. Also, the time for the user process to supply the DR780 with a command packet or a data buffer is minimized.

- **Dynamic memory mapping**—This is the ability to perform continuous data transfers of arbitrary length without having to reload address translation map registers. Data transfers are specified as virtual addresses which the DR780 dynamically translates to physical address locations. This eliminates the need for mapping registers and allows for very large data transfers. The DR780 buffer size is limited only by the amount of physical memory available (see data chaining). The DR780 uses the same page table entries as the CPU.

- **Concurrent data and command transfers on the DDI**—The DDI’s two separate and independent interconnects allow concurrent operations to occur, such as establishing the next command sequence in parallel with completing the current data transfer. Thus, the DR780 can be ready for the next data transfer immediately upon the completion of the current one.

**Data Chaining**
Command packets can specify data chaining. In data chaining, a number of main memory buffers appear as one large buffer to the far end DR-drive. Data chaining is completely transparent to this device; transfers are seen as a continuous stream of data. Chained buffers can be of arbitrary byte alignment and length. The length of a transfer appears to the far end DR-device to be the total of all the byte counts in the chain, and since chains in the DR780 can be of unlimited length, the device sees the byte count as potentially infinite.
Far End DR-Device Initiated Transfer
The DR780 provides the capability for the far end DR-device to initiate data transfer to the VAX memory, that is, random access mode. This mode is used when two DR780s are connected to form a processor-to-processor link. Random access consists of data transfers to or from the VAX memory without notification of the VAX processor. Random access can be discontinued by either specifying a command packet with random access disabled or by an abort from either the controlling process or the far end DR-device.

Power Failure
If power fails on the DR780, but not on the system, the DR780 driver aborts the active data transfer and returns the status code SS$_POWERFAIL in the I/O status block. If a system power failure occurs, the DR780 driver completes the active data transfer when power is recovered and returns the status code SS$_POWERFAIL.

Interrupts
The DR780 can interrupt the DR780 driver for any of the following reasons:
- An abort has occurred. The QIO is completed.
- A DR780 power down or power up sequence has occurred.
- An unsolicited control message has been sent to the DR780. If the command packet's interrupt control field is properly set up, a packet AST interrupt occurs. The interrupt occurs after the command packet obtained from FREEQ is placed on TERMQ. (Discussion of the various queues and their functions is found in the PROGRAMMING INTERFACE section of this chapter.)
- The DR780 enters the Halt state. The QIO is completed.
- A command packet that specifies an unconditional interrupt has been placed onto TERMQ. This results in a packet AST.
- A command packet with the "interrupt when TERMQ empty" bit set was placed on an empty TERMQ. This results in a packet AST.

PROGRAMMING INTERFACE
The DR780 is supported by a device driver and a high-level language procedure library of support routines.

After the driver initializes, the DR780 application programs communicate directly by inserting command packets onto queues. (Command packets contain commands for the DR780, such as the direction of transfer and/or messages to be sent to the user device.) This direct link between the application program and the DR780 provides faster
communication by avoiding the necessity of going through the I/O driver.

Two interfaces are provided for accessing the DR780: a QIO driver and a set of support routines. The QIO driver requires that the application program build command packets and insert them into the DR780 queues. The set of support routines, on the other hand, provide procedures for building and manipulating command packets and, in addition, perform housekeeping functions, such as maintaining command memory.

The DR780 program interface contains three queues: input, termination, and free. These three queues control the flow of command packets to and from the DR780. The application program builds a command packet and inserts it onto the input queue. The DR780 removes the packet, executes the specified command and then inserts the command packet into the termination queue. Unsolicited input, such as a control message from the user device, is placed in packets removed from the free queue and inserted onto the termination queue for later processing. Figure 20-3 shows the flow of command packets as they are moved to the three queues.

![Diagram of Command Packet Flow](image)

Figure 20-3  Command Packet Flow in the Operation of the DR780

The support routine interface has been designed to be called from high-level languages (such as FORTRAN) where the data manipulation which would be required by the QIO interface might be awkward. Note, however, that the support routine user must be equally as knowledgeable as the user of the QIO interface in terms of knowledge of the DR780 and the meaning of the fields in the command packets.

**Application Program Interface**
The application program interfaces with the DR780 through two mem-
ory areas: the command block and the buffer block. The command block contains the headers for the three queues that provide the communication path between the DR780 and the application program, as well as space for the command packets to be built. The buffer block defines the area of memory that is accessible to the DR780 for the transfer of data between the user device and the DR780 itself.

Initiating Command Sequences
If a command packet is inserted onto an empty INPTQ, the application program must notify the DR780 of this event. This is accomplished by setting bit 0, the GO bit, in a DR780 register. The IO$_STARTDATA QIO returns the GO bit’s address to the application program. After notification by the GO bit that there are command packets on its INPTQ, the DR780 continues to process the packets until INPTQ is empty.

The GO bit can be safely set at any time. While processing command packets, the DR780 ignores the GO bit. If the GO bit is set when the DR780 is idle, the DR780 will attempt to remove a command packet from INPTQ. If INPTQ is empty at this time, the DR780 clears the GO bit and returns to the idle state.

Device-Initiated Command Sequence
If the DR-device that interfaces to the far end of the DDI is capable of transmitting unsolicited control messages, they can be transmitted to the local DR780. These messages are not synchronized to the application program command flow. Therefore, the DR780 uses a third queue, FREEQ, to handle unsolicited messages. Normally, the application program inserts a number of empty command packets into FREEQ to allow the external device to transmit control messages.

If a control message is received from the far end DR-device, the DR780 removes an empty command packet from the head of FREEQ, fills the device message field of this packet with the control message and, when the transmission is completed, inserts the packet onto the tail of TERMQ. (The device message field in this command packet must be large enough for the entire message or a length error will occur.) The application program then removes the packet from TERMQ. If the command packet is from FREEQ, the XF$M_PKT_FREEQPK bit in the DR780 Status Longword is set.

Command Packets
To provide for direct communication between the controlling process and the DR780, the DR780 fetches commands from user-constructed command packets located in main memory. Command packets contain commands for the DR780, such as the direction of transfer, or
Interconnects

messages to be sent to the far end DR-device. The DR780 is simply the conveyor of these messages; it does not examine or add to their content. The controlling process builds command packets, and manipulates the three queues, using the four VAX self-relative queue instructions. Figure 20-4 shows the contents of a DR780 command packet.
PROGRAMMING HINTS
This section contains information on programming considerations relevant to users of the DR780. A complete discussion of the programming aspects of the DR780 can be found in the VAX/VMS I/O Users Guide.

Command Packet Prefetch
The DR780 can prefetch command packets from INPTQ. While executing the command specified in one packet, the DR780 can prefetch the next packet, decode it, and be ready to execute the specified command at the first opportunity. When the command is executed depends on which command is specified. For example, if two read device or write device command packets are on INPTQ, the DR780 fetches the first packet, decodes the command, verifies that the transfer is legal, and starts the data transfer. While the transfer is taking place, the DR780 prefetches the next read device or write device command packet, decodes it, and verifies the transfer legality. The second transfer begins as soon as the first transfer is completed.

On the other hand, if the two command packets on INPTQ are read device (or write device) and write device control message, in that order, the DR780 prefetches the second packet and immediately executes the command, because control messages can be overlapped with data transfers. The DR780 then prefetches the next command packet. In an extreme case, the DR780 can send several control messages over the control portion of the DDI while a single data transfer takes place on the data portion of the DDI.

The prefetch capability and the overlapping of control and data transfers can cause unexpected results when programming the DR780. For instance, if the application program calls for a data transfer to the far end DR-device followed by notification of the far end DR-device that data are present, the program cannot simply insert a write device command packet and then a write control message command packet onto INPTQ—the control message may very likely arrive before the data transfer completes.

A better way to synchronize the data transfer with notification of data arrival is to request an interrupt in the interrupt control field of the data transfer command packet. Then, when the data transfer command packet is removed from TERMQ, the application program can insert a write control message command packet into INPTQ to notify the far end DR-device that the data transfer has completed.

Another consequence of command packet prefetching occurs when, for example, two write device command packets are inserted into
Interconnects

INPTQ while the first data transfer takes place, the second command packet is prefetched and decoded. If an unusual event occurs and the application program must send an immediate control message to the far end DR-device, the application program may insert a write device control message packet onto INPTQ. However, this packet is not sent immediately because the second write device command packet has already been prefetched; the control message is sent after the second data transfer starts.

If the application program needs to send a control message with minimum delay, use one of the following techniques:

- Insert only one data transfer function onto INPTQ at a time. If this is done, a second transfer function will not be prefetched and a control message can be sent at any time.
- Use smaller buffers or a faster data rate to reduce the time necessary to complete a given command packet.
- Issue a $CANCEL system service call followed by another IO$_STARTDATA QIO. This technique will have determinant effect on data throughput.

Action Routines
Action routines provide a useful DR780 programming technique. They can be used in application programs written in either assembly language or a high level language. When a command packet is built, the address of a routine to be executed when the packet is removed from TERMQ is appended to the end of the packet. Then, rather than having to determine what action to perform for a particular packet when it is removed from TERMQ, the specified action routine is called.

Error Checking
Bits <23:0> in the second longword of the I/O status block correspond to the same bits in the DR780 Status Longword (DSL). Although the I/O status block is written only after the QIO function completes, the DSL is stored in every command packet. However, because there is no command packet in which to store a DSL for certain error conditions (for example, FREEQ empty) some errors are reported only in the I/O status block. To check for an error under these conditions, the user should examine the DSL in each packet for success or failure only. Then, if a failure occurs, the specific error can be determined from the I/O status block. The I/O status block should also be checked to verify that the QIO has not completed prior to a wait for the insertion of additional command packets into TERMQ. In this way, the application program can detect asynchronous errors for which there is no command packet available.
Queue Retry Macro
When an interlocked queue instruction is included in the application program, the code should perform a retry if the queue is locked. However, the code should not execute an indefinite number of retries. A programming error can cause the queue header to become corrupted. Consequently, all retry loops should contain a maximum retry count.

Diagnostic Functions
The diagnostic functions listed in Table 20-1 can be used to test the DR780 without the presence of a far end DR-device. For the DR780, the user should perform the following test sequence:
1. Insert a set self-test command packet into INPTQ.
2. Insert a diagnostic write internal command packet that specifies a 128-byte buffer into INPTQ. This packet copies 128 bytes from memory into the DR780 internal data silo.
3. Insert a diagnostic read DDI command packet into INPTQ. This packet transmits the 128 bytes of data from the silo over the DDI and returns it to the silo.
4. Insert a diagnostic read internal command packet that specifies another 128-byte buffer in memory into INPTQ. This packet copies 128 bytes of data from the silo into memory.
5. Compare the two memory buffers for equality. Note that on the DR780, the diagnostic read internal function destroys the first four bytes in the silo before storing the data in memory. Therefore, compare only the last 124 bytes of the two buffers.
6. Insert a clear self-test command packet into INPTQ.

Table 20-1  Device Control Code Descriptions

<table>
<thead>
<tr>
<th>Function</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Device</td>
<td>This function specifies a data transfer from the far end DR-device to the DR780. The control select field describes the information to be transferred prior to the initiation of the data transfer.</td>
</tr>
<tr>
<td>Read Device Chained</td>
<td>This function specifies a data transfer from the far end DR-device to the DR780. The DR780 data chains to the buffer specified in the next command packet in INPTQ. A command packet that specifies Read Device</td>
</tr>
<tr>
<td>Function</td>
<td>Meaning</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Write Device and Write Device Chained</td>
<td>These functions specify data transfers from the DR780 to the far end DR-device. Otherwise, they are similar to Read Device and Read Device Chained.</td>
</tr>
<tr>
<td>Write Device Control Message</td>
<td>This function specifies the transfer of a control message to the far end DR-device. This message is contained in the device message field of this command packet. The Write Device Control Message function directs the controlling DR780 to ignore the byte count and virtual address fields in this command packet.</td>
</tr>
<tr>
<td>Set Self-Test</td>
<td>This function directs the DR780 to set an internal self-test flag and to set a disable signal on the DDI. This signal informs the far end DR-device that the DR780 is in self-test mode. In this condition the DR780 can no longer communicate with the far end DR-device.</td>
</tr>
<tr>
<td>Clear Self-Test</td>
<td>This function directs the DR780 to clear the internal self-test flag set by the Set Self Test function and return to the normal mode of operation.</td>
</tr>
<tr>
<td>No Operation</td>
<td>The NOP function specifically does nothing.</td>
</tr>
</tbody>
</table>
| Diagnostic Read Internal       | This function directs the DR780 to fill the memory buffer, which is described by the virtual address and byte count specified in the current command packet, with the data that are stored in the DR780 data silo. The buffer is filled in a cyclic manner. For example, on the DR780 every 128-byte section of
### Interconnects

<table>
<thead>
<tr>
<th>Function</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Diagnostic Write Internal</strong></td>
<td>the buffer receives the silo data. The amount of data stored in the buffer equals the DDI byte count minus the SBI byte count. The DDI byte count is equal to the original byte count. No data transmission takes place on the DDI for this function. On the DR780, the Diagnostic Read Internal function destroys the first four bytes in the silo before storing the data in the buffer. This function, together with the Diagnostic Read Internal function, is used to test the DR780 read and write capability. The Diagnostic Write Internal function directs the DR780 to store data, which are contained in the memory buffer described by the current command packet, in the DR780 data silo, a FIFO-type buffer. No data transmission takes place on the DDI for this function. The Diagnostic Write Internal function terminates when: 1. The memory buffer is empty (the SBI byte count is 0). 2. An abort has occurred. At the time the function terminates, the amount of data in the silo equals the DDI byte count minus the SBI memory byte count.</td>
</tr>
<tr>
<td><strong>Diagnostic Read DDI</strong></td>
<td>This function tests transmissions over the data portion of the DDI. The DR780 must be in the self-test mode. If not, an abort will occur. On the DR780, the Diagnostic Read DDI function transmits the contents of DR780 data silo locations 0-127 over the DDI and returns the data to the same locations. If data transmission is normal, that is, without errors, the residual memory count is equal to the original byte count, the residual DDI count is 0, and the contents of the silo remain unchanged.</td>
</tr>
<tr>
<td>Function</td>
<td>Meaning</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Diagnostic Write Control Message</td>
<td>This function tests transmissions over the control portion of the DDI. The DR780 must be in self-test mode. If not, an abort will occur. The Diagnostic Write Control Message function directs the DR780 to remove the command packet on FREEQ and check the length of message field. Then the first byte of the message in the command packet on INPTQ is transmitted and read back on the control portion of the DDI. This byte is then written into the message space of the packet from FREEQ. The updated packet from FREEQ is inserted into TERMQ and is followed by the packet from INPTQ.</td>
</tr>
<tr>
<td>Set Random Enable and Clear Random Enable</td>
<td>The Set Random Enable function directs the DR780 to accept read and write commands sent by the far end DR-device. Range checking is performed to verify that all addresses specified by the far end DR-device for access are within the buffer block. Far end DR-device initiated transfers to or from VAX memory are conducted without notification of the VAX processor or the application program. The Clear Random Enable function directs the DR780 to reject far end DR-device initiated transfers. Random access mode must be enabled when the DR780 is used in a processor-to-processor link.</td>
</tr>
<tr>
<td>Set HALT</td>
<td>This function places the DR780 in a halt state. The Set HALT function always generates a packet interrupt regardless of the value in the interrupt control field. If an AST routine was requested on completion of the QIO function, the routine is called after the command packet containing the Set HALT function has been processed by the DR780.</td>
</tr>
</tbody>
</table>
The NOP Command Packet
It is often useful to insert a NOP command packet into INPTQ to test
the state of the DDI disable bit in the DSL. By checking this bit before
initiating a data transfer, an application program can determine if the
far end DR-device is ready to accept data.

Interrupt Control Field
The interrupt control field determines the conditions under which an
interrupt is generated: unconditionally, if TERMQ was empty, or never.
There are three general applications of this field:
1. If a program performs five data transfers, for example, and re-
   quires notification of completion only after all five have comple-
eted, the first four command packets should specify no interrupt
   and the fifth command packet should specify an unconditional
   interrupt.
2. If a program performs a continuous series of data transfers, each
   command packet can specify interrupt only if TERMQ was empty.
   Then, every time an event flag or AST notifies the program that a
   command packet was inserted into TERMQ, the program re-
   moves and processes all packets on TERMQ until it is empty.
3. Command packets that specify no interrupt should never be
   mixed with command packets that specify interrupt if TERMQ was
   empty. If this were done, a command packet that specifies no
   interrupt could be inserted onto TERMQ, followed by a command
   packet that specifies interrupt if TERMQ was empty; the latter
   packet will not interrupt and the program is never notified that
   command packets were inserted onto TERMQ.

PHYSICAL CHARACTERISTICS
The DR780 has the same physical characteristics as both the UNIBUS
Adapter (UBA) and the MASSBUS Adapter (MBA). It requires one
Option Panel Space and can therefore mount in either the central
processor cabinet or the CPU expander cabinet. All necessary cabling
and a power supply are included. The cable used with the DR780 is a
shielded version of the standard round MASSBUS cable for use when
connections from cabinet to cabinet are made. This cable is 25 feet in
length. Note that while the DR780 uses MASSBUS cable, the DDI
electrical characteristics are quite different from those of the MASS-
BUS. All user devices must be mounted in either a free-standing cabi-
net or in an expansion box. When the user device is to be mounted in
an expansion box, a flat MASSBUS cable may be used. This cable is
available in various lengths; a 10 foot flat MASSBUS cable is included
with the DR780 option.
CONFIGURING THE DR780 IN VAX-11/780 SYSTEMS
The DR780 can transfer data at a maximum rate of 6.67 Mb/second. This speed is greater than the maximum rate a single memory controller can handle. It is therefore imperative that the user carefully analyze the application to be performed and determine the maximum rate at which the DR780 will be operating. In many instances, a second memory controller will be required so that the system can take advantage of interleaving memory to achieve the transfer rate desired.

Table 20-2 will enable the user to do a first pass configuration/throughput analysis of a VAX-11/780 system incorporating a DR780. The assumptions used to develop this analysis are as follows:

- The user must identify which SBI devices (MBAs and UBAs) are transferring concurrently with the DR780. Note that this is very different than the number of physical adapters attached to the system.
- The 0 CPU case serves to document the upper limit of DR780 performance. Knowledgeable users can attain the 0 CPU performance with proper programming. For this case, the transfer rates shown require a user-supplied, external clock.
- Generally, users are interested only in the performance of the DR780 with continuous (as opposed to stall) transfers on the DDI. Therefore, only the continuous performance numbers are given.
- Use of small buffers (less than 16 Kb) in command chaining and data chaining will result in some loss of performance.
- UBAs and MBAs have roughly the same effect on DR780 performance, with MBAs with RM03s being the worst case. The configuration rules were obtained by using MBA with RM03 simulations realizing that this is a worst case representation of a mixture of UBAs and MBAs.
- The MA780 multiport memory option is not included in the system configurations. The interaction of these devices is very application dependent and general rules cannot be applied.
- While it is architecturally possible to connect more than one DR780 to a system, properly configuring such a system is very complex since the system's behavior depends highly upon the user application. Consult with your Sales Representative if more than one DR780 is needed.
- The memory read and memory write performance is different enough to specify them separately.
### Interconnects

#### Table 20-2  DR780 Performance

<table>
<thead>
<tr>
<th>Number of Devices (transferring concurrently with the DR780)</th>
<th>DR780 Performance (Mb/second)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Memory</td>
</tr>
<tr>
<td></td>
<td>CPU's(^1) Controllers</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

1 The CPU was simulated as “worst case” CPU, that is, a CPU doing MOVLC5 instructions with fill characters.

2 UBAs + MBAs is the number of adapters transferring concurrently with the DR780. The +1 represents the DR780 in the simulated configuration.

Select the appropriate row which corresponds to the desired configuration and read the applicable DR780 maximum transfer rate. For transfers into memory use the “memory write” column, for transfers from memory use “memory read.” For example, if the desired system consists of one CPU, two memory controllers and two I/O adapters (either MBAs or UBAs or one of each) which are transferring concurrently with the DR780, then the maximum rates for the DR780 are:

- from memory to the DR780 = 3.5 Mb/sec
- from the DR780 into memory = 4.5 Mb/sec

### MA780 Multiport Memory

<table>
<thead>
<tr>
<th><strong>Features</strong></th>
<th><strong>Benefits</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>11 Mb/second maximum total throughput on memory transfers</td>
<td>Extremely fast interprocessor communications</td>
</tr>
<tr>
<td>Two MA780s (2 Mb each) per VAX-11 processor</td>
<td>Expands physical address space up to a system total of 12 Mb</td>
</tr>
</tbody>
</table>
Interconnects

Up to four VAX-11/780 processors can share each MA780

Allows parallel or sequential processing functions for exceptionally fast application processing

Full VAX/VMS operating system support

Access to shared memory is transparent to the user

Invalidation logic enables caching of shared memory data

Eliminates problem of “stale data”

Memory interlock capability

Prevents shared memory from being read by one processor while being updated by another

Individual memory port on/off line switching

Enables selective CPU shutdown permitting high over-all system uptime

INTRODUCTION

The MA780 multiport memory enables up to four VAX-11/780 processors to share a bank of from 256K bytes to 2M bytes of memory. This capability allows VAX-11/780 users to develop multicomputer configurations for very high throughput or enhanced availability applications. The maximum throughput rate for the MA780 is 11M bytes per second.

The VAX/VMS operating system fully supports MA780 configured systems. An application built around multiple cooperating processes can be reconfigured to run on a multi-CPU system with no program modification. Processes in shared memory can be moved from one processor to another with complete transparency to the programs involved. Specifically, VAX/VMS provides support for MA780 configurations in the areas of interprocessor communications (sharing of data regions, VMS mailboxes, common event flags) and the sharing of code among processors.

VAX/VMS itself does not use the shared memory in its dynamic page pool of available memory and no part of the operating system resides in the shared memory. Each CPU in the multiport system operates independently using its own copy of VMS stored in its local memory.

Figure 20-5 illustrates two VAX-11/780 systems incorporating the MA780.

CAPACITY AND EXPANDABILITY

The MA780 comes with 256K bytes of ECC MOS memory as the initial
configuration. Additional amounts of memory can be added in 256K byte increments up to a maximum of 2M bytes. A VAX-11/780 system will support two MA780 subsystems, thus increasing the total addressable physical memory to 12M bytes per system.

Each MA780 option comes equipped with two interface ports to allow communication from one VAX-11/780 processor to a second. Additional interface port options can be added permitting access to a third and fourth CPU. These port options are mounted within the MA780 subsystem.

It is recommended that systems with three or four MA780 I/O ports also incorporate a selective cache invalidate option (see DATA INTEGRITY section).

Physical Layout
Physically, the MA780 is composed of standard VAX-11/780 power supplies, cables and memory array boards, all housed in a standard VAX-11/780 cabinet. Adequate space is allocated within the cabinet to accommodate a second MA780.

The MA780 is compatible with all VAX-11/780 system and therefore can be readily installed on existing systems.

THROUGHPUT
The MA780 has a maximum throughput rate of 11M bytes per second. This rate applies to 64-bit quadword (eight-byte) data transfers. Smaller sized transfers will result in lower throughput. Also, throughput to shared memory as seen by a CPU is very much a function of other factors within that processor, including cache hits, I/O and SBI traffic.

Figure 20-6 illustrates the MA780 cabinet layout.
Port Servicing
The high throughput rate is in part due to the fact that each MA780 port has a buffer for commands and data. Further, each port is serviced on a demand basis, that is, first in-first serviced. No time is wasted in polling inactive ports. A servicing algorithm guarantees that no port waits more than three memory cycles to gain access to shared memory.

Parallel and Sequential Processing
The MA780 can enhance overall system throughput by means of one of two configuration types; parallel or sequential (pipeline) processing.

In parallel processing, two or more CPUs divide a task between them. This allows the CPUs to pool their power and complete the job extremely quickly.

Sequential processing can increase total system throughput by allowing instantaneous data exchange between CPUs that are handling sequential parts of an application. Each processor is dedicated to a specific portion of the total application and upon completion of that portion, passes the results on to the next CPU.

Figure 20-7 illustrates parallel and sequential processing configurations.
DATA INTEGRITY
Since the MA780 utilizes the same memory array cards as those of the VAX-11/780 main memory subsystem, the shared memory also has the built-in error correcting code (ECC) to correct all single-bit errors and detect all double-bit errors on memory reads and writes. Parity error checking is also present on all MA780 internal buses.

Interlock Capability
A problem associated with shared memory is that while one processor is reading a memory location, a second processor can simultaneously be trying to update that location. The MA780 eliminates this problem by locking out the second processor until the first has completed its transaction.

Cache Invalidation Logic
The MA780 has been designed to virtually eliminate the problem of incorrect or stale data in the CPU caches. After a write to a shared memory location, the MA780 sends an invalidate signal to all CPUs. This signal transmits an address to each CPU which then checks to see if its cache contains that address. If so, it marks the contents as invalid and on subsequent reads by the CPU, this invalidation forces the CPU to seek the new information from shared memory. The VAX-11/780 cache therefore retains its transparency to software.

Selective Cache Invalidation
For multicomputer systems where there is considerable traffic on a system's SBI, a selective cache invalidation option is available. It is called an invalidation map and is useful primarily in systems with three or four CPUs. It causes the MA780 to send invalidate signals only to those CPUs that have accessed a given location in shared memory. Its operation is quite straightforward: the invalidation map has one entry for each 64-bit quadword in shared memory. This entry contains four
Interconnects

bits, one for each CPU. Whenever a CPU reads a 64-bit quadword from shared memory, the MA780 controller sets the bit in the correct invalidation map entry. The next time a CPU writes into that 64-bit quadword, the MA780 controller notifies only those CPUs whose bits have been set in the invalidation map. Once that has been done, all the bits are cleared except that of the CPU performing the write operation. In this way, the overall traffic on all SBI's and cache activity is kept to a minimum while data integrity is still maintained.

Battery Backup
Battery backup support is optionally available for the shared memory in system configurations sensitive to power loss and fluctuation. The backup unit resides within the MA780 cabinet and is capable of supporting 2 Mb of memory for a minimum of ten minutes. Smaller amounts of memory are supported for longer periods of time. This level of support is sufficient to protect the memory contents from the vast majority of power interruptions.

FAILSOFT CAPABILITY
Failsoft capability is the capability to maintain a high degree of system uptime by means of a design which incorporates extensive diagnostics to detect faults and allows system functioning to continue even though certain elements of the system may be down for service.

The MA780 supports this failsoft capability through several features. System diagnostics can access the MA780 through any one of its memory access ports and any connected VAX-11/780 system is allowed to read the multiport status register. Each port can be separately switched off-line and the appropriate processor powered off. This permits on-line overall system service even though the switched-off port may be connected to a system element not presently operational.

The MA780 itself generates interrupts to notify ports of power failures and error conditions as soon as they are detected. The VMS operating system logs these detected errors and consequently minimizes the MA780's mean time to repair.

Multiprocessor systems maintain their high degree of in-service time through one other important feature. Although each processor is working on the same or related tasks, each processor maintains its own VMS operating system. Thus an operating system failure in one system does not bring down the entire multi-CPU system. The user application, however, must have been programmed to deal with this occurrence.
**USING SHARED MEMORY**
Before an application using multiport memory can execute under VAX/VMS, the system manager must activate the VAX/VMS operating system in processors connected to the multiport memory unit and initialize that memory. The *VAX/VMS System Manager’s Guide* explains the system management responsibilities associated with a multiport memory unit.

**Preparing Multiport Memory for Use**
First, the system manager activates the VAX/VMS operating system in a VAX-11/780 and initializes the multiport memory unit. These actions cause the following to occur:

- The uninitialized shared memory is connected to the VAX/VMS system running in the processor.
- A name is defined that all processes running in all processors can use to refer to the shared memory.
- Limits are set for the following resources in this multiport memory unit:
  - Common event flag clusters: the total number that can be created, by processes running on this processor.
  - Mailboxes: the total number that can be created, and the number that can be created by processes running on this processor.
  - Global sections: the total number that can be created, and the number that can be created by processes running on this processor.

The system manager then activates the VAX/VMS operating system in other processors connected to the multiport memory unit and connects the initialized shared memory to the VAX/VMS system running in each of these processors and sets limits for the number of common event flag clusters, global sections, and mailboxes that processes on each processor can create in the multiport memory.

The system manager can also install global sections in shared memory just as they are installed in local memory. The INSTALL utility can be used to create shared memory global sections for known files. Once the global sections are installed, a process running in any processor connected to the MA780 can map to the section, if the process has the appropriate privilege. The process can gain access to the global section either by using a logical name defined by the system manager or by using the section name specified when the global section was created. In the latter case, the section name must be unique on this processor.
Interconnects

To use facilities in memory shared by multiple processors, all of the user privileges required to use the equivalent facility in local memory must be present. For example, to create a permanent global section, the user must have the PRMGBL privilege, and to create a temporary or permanent mailbox, the user must have the TMPMBX or PRMMBX privilege.

Assigning Logical Names and Logical Name Translation
The user can define a logical name for a shared memory facility with the DEFINE or ASSIGN command or the Create Logical Name ($CRELOG) system service. Application programs can then refer to the facility using the logical name; for example, a process can invoke the Create Mailbox and Assign Channel ($CREMBX) system service specifying the logical name for an existing mailbox to which a channel is to be assigned.

When translating a logical name for a shared memory facility, the VAX/VMS operating system uses a slightly different approach from that used for other logical names. The purpose of this approach is to allow programmers to specify either the complete name (memory name and facility name) or a logical name that the system will translate to the complete name. If logical names are defined properly, a program that uses a given facility in local memory can be run without change to use the facility in shared memory.

Whenever VAX/VMS encounters the name of a common event flag cluster, mailbox, or global section, it performs the following special logical name translation sequence:

1. Inserts one of the following prefixes to the name (or to the part of the name before the colon if a colon is present):
   - CEF$ for common event flag clusters
   - MBX$ for mailboxes
   - LIB$ for global sections
2. Subjects the resultant string to logical name translation. If translation does not succeed (that is, the original name did not use a logical name), passes the original name string to the system service. If translation does succeed, goes to step 3.
3. Appends the part of the original string after the colon (if any) to the translated name.
4. Repeats steps 1 to 3 (up to nine more times, if necessary) until logical name translation fails. When translation fails, passes the string to the system service.
**Interconnects**

**How VAX/VMS Finds Facilities in Shared Memory**

After the VAX/VMS system performs the logical name translation, the final equivalence name must be the name of a facility in either the processor’s local memory or in shared memory. If the equivalence name specifies the name of a shared memory (that is, the name is in the format name:facility-name), VAX/VMS searches for the facility in the appropriate data base of the specified memory.

If the equivalence name specifies a common event flag cluster or mailbox and does not specify a memory name, VAX/VMS searches through the common event flag cluster data base or the mailbox data base until it locates the specified cluster or mailbox. Absence of a memory name as part of a common event flag cluster name or mailbox name indicates that the facility is located in local memory.

If the equivalence name specifies a global section and does not specify a memory name, VAX/VMS looks for the section as follows:

1. First, it searches the global section tables for sections in the processor’s local memory.
2. Then, it searches the global section tables for each initialized shared memory connected to the processor in the order in which they were connected and recognized by the processor.

The result of searching in this order is that global sections in the processor’s local memory take precedence over those in shared memories. Thus, absence of a memory name as part of a global section name is not used as an indication of where the global section is located.

**Using Common Event Flags in Shared Memory**

Under VAX/VMS, any process can associate with up to two common event flag clusters (event flag numbers 64 through 95 and 96 through 127). These clusters can be located in shared memory or in local memory. To create and associate with a common event flag cluster in shared memory and manipulate flags in the cluster, use the same steps as would be used to associate with a common event flag cluster in local memory:

1. Issue the Associate Common Event Flag Cluster ($ASCEFC) system service to create the cluster or to associate with an existing cluster.
2. Issue any of the services that set, clear, and wait for designated event flags, as appropriate.

As with local memory clusters, the first process among cooperating processes to issue the Associate Common Event Flag cluster ($ASCEFC) system service causes the cluster to be created. Any other
Interconnects

process calling this service and specifying the same cluster associates with that cluster. VAX/VMS implicitly qualifies cluster names with the group number of the creator's UIC; therefore, other cooperating processes must belong to the same group.

All of the event flag system services, with the exception of Associate Common Event Flag Cluster and Disassociate Common Event Flag Cluster, function identically regardless of whether they are used with local or shared memory clusters. The only difference with the Associate and Disassociate system services is that to specify a cluster in shared memory, the user must provide the memory name as well as the cluster name. That is, after VAX/VMS performs logical name translation of the name argument, the cluster name must have the following format:

memory-name:cluster-name

Using Mailboxes in Shared Memory
The first process on each processor to refer to a shared memory mailbox must use the Create Mailbox and Assign Channel ($CREMBX) system service to create the mailbox and assign a channel to it. Any $CREMBX system service call referring to a shared memory mailbox must specify a mailbox name that has or translates to the following format:

memory-name:mailbox-name

When the mailbox is created, the $CREMBX system service also creates the mailbox-name portion of the name string as a logical name with an equivalence name in the format MBn. For example, if the complete name string is SHMEM:MAILBOX, the system service will create MAILBOX as a logical name with an equivalence name of, for example, MBB005.

The Assign I/O Channel ($ASSIGN) and Deassign I/O Channel ($DASSIGN) system services require that you specify only the mailbox-name portion of a shared memory mailbox name string. Likewise, any high-level language program statements that open, close, read from, or write to a shared memory mailbox must specify only the mailbox-name portion.

A mailbox in shared memory cannot be used as a process termination mailbox.

Using Global Sections in Shared Memory
Under VAX/VMS, processes can map global sections located in local memory or in shared memory. A global section in shared memory can be mapped to an image file or a data file, just like a global section in
Interconnects

local memory. To create a global section in shared memory, the same steps are performed as to create a global section in local memory:

1. Using VAX-11 RMS, open the file to be mapped.
2. Issue the Create and Map Section ($CRMPSC) system service.

The file to be mapped must reside on a disk device attached to the local processor. Once the section is created, however, processes on all processors attached to the shared memory can map the section.

To map an existing global section in shared memory, the user issues a Map Global Section ($MGBLSC) system service specifying the name of the section. Once the section is mapped, processes gain access to shared memory global sections in the same manner as they do to local memory sections. VAX/VMS thus makes use of the shared memory unit transparent to the process.

VAX/VMS treats the pages of a global section in shared memory differently from pages in local memory. When a process creates a shared-memory global section, VAX/VMS brings all of the pages of the mapped image or data file into memory. Any process mapped to that global section can gain access to those pages without incurring a page fault because the pages are already in physical memory. Unlike process pages in local memory, global section pages in shared memory are not included in the working sets of the processes that map the section.

Because no paging occurs, VAX/VMS never writes the contents of shared memory global section pages back to their disk file. For read/write global sections in which the user wants to maintain an updated file while the application executes, an Update Section File Disk ($UPDSEC) system service must be issued. The process issuing the update request must execute on the same processor as the process that created the global section. The disk file can be updated periodically during execution of the application as a checkpoint precaution. The disk file is automatically updated when the section is deleted.

Each process that has mapped a global section in shared memory can unmap the section in either of the following ways:

- Issue a Delete Virtual Address Space ($DELTVA) system service to delete the process's virtual address space that maps the section.
- Terminate the current image, thereby causing VAX/VMS to unmap the process from the section automatically.

Deleting a global section in shared memory requires an explicit deletion request, because all global sections in shared memory must be permanent sections. The deletion request can be either a Delete Global Section ($DGBLSC) system service issued by the application,
or a deletion request issued by the system manager. In either case, VAX/VMS does not perform the actual deletion until all processes that have mapped the section unmap it.
CHAPTER 21
PRIVILEGED REGISTERS

INTRODUCTION
The processor register space provides access to many types of CPU control and status registers such as the memory management base registers, the PSL, and the multiple stack pointers. These registers are explicitly accessible only by the Move to Processor Register (MTPR) and Move from Processor Register (MFPR) instructions which require kernel mode privileges. This chapter describes those privileged processor registers not found elsewhere in this handbook.

Appendix D contains a description of the ID Bus registers of which the privileged processor registers are a subset. Therefore, those registers containing a processor address are privileged and can be accessed via the MTPR and MFPR instructions only. Chapter 14, Console Subsystem, contains a description of the ID Bus.

SYSTEM IDENTIFICATION REGISTERS (SID)
The system identification register is a read-only constant register that specifies the processor type. The entire SID register is included in the error log and the type field may be used by software to distinguish processor types. Figure 21-1 illustrates the system identification register.

```
<table>
<thead>
<tr>
<th>31</th>
<th>24</th>
<th>23</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE</td>
<td>TYPE SPECIFIC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 21-1  System Identification Register

Type  A unique number assigned by engineering to identify a specific processor:

0      Reserved to DIGITAL (error)
1      VAX-11/780
2      VAX-11/750
3 through 127  Reserved to DIGITAL
128 through 255 Reserved to CSS and customers

Type-specific  Format and content are a function of the value in type. They are intended to include such information as serial number and revision level.
For the VAX-11/780, the type-specific format is:

```
23 15 14 12 11 0
  ECO LEVEL  PLANT  SERIAL NUMBER
```

**CONSOLE TERMINAL REGISTERS**
The console terminal is accessed through four internal registers. Two are associated with receiving from the terminal and two with writing to the terminal. In each direction there is a control/status register and a data buffer register. Figure 21-2 illustrates the console receive control/status register.

![Figure 21-2 Console Receive Control/Status Register (RXCS)](image)

Figure 21-3 illustrates the read-only console receive data buffer register.

![Figure 21-3 Console Receive Data Buffer Register (RXDB)](image)

At bootstrap time, RXCS is initialized to 0. Whenever a datum is received, the read-only bit DONE is set by the console. If IE (interrupt enable) is set by the software, then an interrupt is generated at interrupt priority level (IPL) 20. Similarly, if DONE is already set and the software sets IE, an interrupt is generated (i.e., an interrupt is generated whenever the function (IE AND DONE) changes from 0 to 1). If the received data contained an error such as overrun or loss of connection, then ERR is set in RXDB. The received data appear in DATA. When an MFPR #RXDB,dst is executed, DONE is cleared as is any interrupt request. If ID is 0, then the data are from the console terminal. If ID is not 0, then the entire register is implementation-dependent. In the case of the VAX-11/780, if ID = 1, data are from the floppy disk.
Privileged Registers

At bootstrap time, TXCS is initialized with just the RDY bit set (ready). Whenever the console transmitter is not busy, it sets the read-only bit RDY. If IE (interrupt enable) is set by the software, then an interrupt is generated at IPL 20. Similarly, if RDY is already set and the software sets IE, an interrupt is generated (i.e., an interrupt is generated whenever the function (IE AND RDY) changes from 0 to 1). The software can send a datum by writing it to DATA. When an MTPR src.#TXDB is executed, RDY is cleared as is any interrupt request. If ID is written 0, then the datum is sent to the console terminal. If ID is non-zero, then the entire register is implementation-dependent. In the case of VAX-11/780, if ID = 1, data are sent to the floppy disk. Figure 21-4 illustrates the console transmit control/status register.

Figure 21-4 Console Transmit Control/Status Register (TXCS)

Figure 21-5 illustrates the read-only console transmit data buffer register.

Figure 21-5 Console Transmit Data Buffer Register (TXDB)

CLOCK REGISTERS
The clocks consist of an optional time-of-year clock and a mandatory interval clock. The time-of-year clock is used to measure the duration of power failures and is required by the operating system for unattended restart after a power failure. The interval clock is used for accounting, for time-dependent events, and to maintain the software date and time.

Time-of-Year Clock
The time-of-year clock consists of one longword register. The register forms an unsigned 32-bit binary counter that is driven by a precision clock source with at least .0025% accuracy (approximately 65 seconds per month). The counter has a battery back-up power supply sufficient for at least 100 hours of operation, and the clock does not gain or lose
Privileged Registers

any ticks during transition to or from stand-by power. The battery is recharged automatically. The least significant bit of the counter represents a resolution of 10 milliseconds. Thus, the counter cycles to 0 after approximately 497 days.

If the battery has failed, so that time is not accurate, then the register is cleared upon power up. It then starts counting from 0. Thus, if software initializes this clock to a value corresponding to a large time (e.g., a month), it can check for loss of time after a power restore by checking the clock value. The time-of-year clock register is illustrated in Figure 21-6.

```
31 \[ \text{TIME OF YEAR SINCE SETTING} \]
0
```

Figure 21-6 Time-of-Year Clock Register (TODR)

A value written to TODR with <27:0> non-zero results in an UNPREDICTABLE value in TODR. If the clock is not installed, then the clock always reads out as 0 and ignores writes.

Interval Clock

The interval clock provides an interrupt at IPL 24 at programmed intervals. The counter is incremented at 1 μs intervals, with at least .01% accuracy (8.64 seconds per day). The clock interface consists of three registers in the privileged register space: the read-only interval count register, the write-only next interval count register, and the interval clock control/status register.

Figure 21-7 illustrates the interval count register.

```
31 \[ \text{INTERVAL COUNT} \]
0
```

Figure 21-7 Interval Count Register (ICR)

Figure 21-8 illustrates the next interval count register.

```
31 \[ \text{NEXT INTERVAL COUNT} \]
0
```

Figure 21-8 Next Interval Count Register (NICR)
Privileged Registers

Figure 21-9 illustrates the interval clock control/status register.

<table>
<thead>
<tr>
<th>31 30</th>
<th>8 7 6 5 4 3 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERR</td>
<td>MBZ</td>
</tr>
<tr>
<td>INT</td>
<td>SGL</td>
</tr>
<tr>
<td>X</td>
<td>MBZ</td>
</tr>
<tr>
<td>RUN</td>
<td></td>
</tr>
</tbody>
</table>

Figure 21-9  Interval Clock Control/Status Register (ICCS)

Interval Count Register
The interval register is a read-only register incremented once every microsecond. It is automatically loaded from NICR upon a carry out from bit 31 (overflow) which also interrupts at IPL 24 if the interrupt is enabled.

Next Interval Count Register
The reload register is a write-only register that holds the value to be loaded into ICR when it overflows. The value is retained when ICR is loaded. NICR is capable of being loaded regardless of the current values of ICR and ICCS.

Interval Clock Control/Status Register (ICCS)
The ICCS register contains control and status information for the interval clock.

Bit: 31  Name: ERR
Function: Whenever ICR overflows, if INT is already set, then ERR is set. Thus, ERR indicates a missed clock tick. Attempts to set this bit via MTPR clears ERR.

Bit: 30:8  Name: MBZ
Function: Must be zero.

Bit: 7  Name: INT
Function: Set by hardware every time ICR overflows. If IE is set, then an interrupt is also generated. An attempt to set this bit via MTPR clears INT, thereby re-enabling the clock tick interrupt (if IE is set).

Bit: 6  Name: IE
Function: When set, an interrupt request at IPL 24 is generated every time ICR overflows (INT is set). When clear, no interrupt is requested. Similarly, if INT is already set and the software sets IE, an interrupt is generated (i.e., an interrupt is generated whenever the function (IE AND INT) changes from 0 to 1).

Bit: 5  Name: SGL
Function: A write-only bit. If RUN is clear, each time this bit is set, ICR is incremented by one.
Privileged Registers

Bit: 4  Name: XFR
Function: A write-only bit. Each time this bit is set, NICR is transferred to ICR.

Bit: 3:1  Name: MBZ
Function: Must be zero.

Bit: 0  Name: RUN
Function: When set, ICR increments each microsecond. When clear, ICR does not increment automatically. At bootstrap time, RUN is cleared.

Thus, to set up the interval clock, load the negative of the desired interval into NICR. Then an MTPR #\textasciitilde X51,#ICCS will enable interrupts, reload ICR with the NICR interval and set run. Every “interval count” microseconds will cause INT to be set and an interrupt to be requested. The interrupt routing should execute an MTPR #\textasciitilde XC1,#ICCS to clear the interrupt. If INT has not been cleared (i.e., if the interrupt has not been handled) by the time of the next ICR overflow, the ERR bit will be set.

At bootstrap time, bits \(<6\) and \(<0\) of ICCS are cleared. The rest of ICCS and the contents of NICR and ICR are UNPREDICTABLE.

VAX-11/780 FLOATING POINT ACCELERATOR
The VAX-11/780 processor has an optional accelerator for a subset of the instructions. Two internal registers control the accelerator: ACCS and ACCR.

ACCS is the accelerator control/status register. It indicates whether an accelerator exists, controls whether it is enabled, identifies its type and reports errors and status. At bootstrap time, the type and enable are set; the errors are cleared. Figure 21-10 illustrates the accelerator control/status register.

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERR</td>
<td>R</td>
<td>M</td>
<td>U</td>
<td>O</td>
<td>R</td>
<td>V</td>
<td>MBZ</td>
<td>E</td>
<td>N</td>
<td>B</td>
<td>MBZ</td>
</tr>
<tr>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>R</td>
<td>W</td>
<td></td>
<td></td>
<td>RO</td>
</tr>
</tbody>
</table>

Figure 21-10  Accelerator Control/Status Register (ACCS)

Bit: 31  Name: ERR
Function: Read-only bit specifying that at least one of bits RSV, OVF, and UNF is set. Note that bits \(<31:27\) are normally cleared by the main processor microcode before starting the next macro instruction.

Bit: 30  Name: MBZ
Function: Must be zero.
Privileged Registers

Bit: 29   Name: UNF
Function: Read-only bit specifying that the last operation had an underflow.

Bit: 28   Name: OVF
Function: Read-only bit specifying that the last operation had an overflow.

Bit: 27   Name: RSV
Function: Read-only bit specifying that the last operation had a reserved operand.

Bit: 26:16   Name: MBZ
Function: Must be zero.

Bit: 15   Name: ENB
Function: Read/write field specifying whether the accelerator is enabled. At bootstrap time, this is set if the accelerator is installed and functioning. An attempt to set this is ignored if no accelerator is installed.

Bit: 7:0   Name: TYPE
Function: Read-only field specifying the accelerator type as follows:
0 = No accelerator
1 = Floating point accelerator

Numbers in the range 2 through 127 are reserved to DIGITAL. Numbers in the range 128 through 255 are reserved to CSS/customers.

The accelerator maintenance register (ACCR) controls the accelerator's microprogram counter. At bootstrap time its contents are UNPREDICTABLE. Figure 21-11 illustrates the accelerator maintenance register.

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>24</th>
<th>23</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>9</th>
<th>8</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETL</td>
<td>MBZ</td>
<td>TRAP ADDRESS</td>
<td>E</td>
<td>M</td>
<td>P</td>
<td>M</td>
<td>MBZ</td>
<td>MICRO PC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W</td>
<td>O</td>
<td>RW</td>
<td>W</td>
<td>R</td>
<td>O</td>
<td>O</td>
<td>RW</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 21-11   Accelerator Maintenance Register (ACCR)

Bit: 31   Name: ETL
Function: Enable Trap Address Load. A write-only bit that when set causes <23:16> to be loaded into the accelerator’s trap address register. Subsequently, the main processor’s microcode can force the accelerator to trap to this location by asserting an internal signal.

Bit: 30:24   Name: MBZ
Function: Must Be zero.
Privileged Registers

Bit: 23:16  Name: TRAP
Function: Trap Address. A read/write field used by the main processor to force the accelerator to a specified micro location.

Bit: 15  Name: EML
Function: Enable Micro PC Match Load. A write-only bit that when set causes <8:0> to be loaded into the accelerator's micro PC match register.

Bit: 14  Name: MPM
Function: Micro PC Match. A read-only bit that is set whenever the accelerator's micro PC matches the micro PC match register. This is useful primarily as a scope sync signal.

Bit: 13:9  Name: MBZ
Function: Must be zero.

Bit: 8:0  Name: PC
Function: Next Micro PC on read. This contains the next micro address to be executed.

Match Micro PC on write. If EML is also set, then this updates the micro PC match register.

VAX-11/780 MICRO CONTROL STORE
The VAX-11/780 processor has three registers for microcode control/status. Two are used for writing into any writable control store (WCS) and one is used to control micro breakpoints. Figure 21-12 illustrates the writable control store address register.

![User Control Store Address Register (WCSA)](image)

Figure 21-12  User Control Store Address Register (WCSA)

Reading WCSD indicates which control store addresses are writable. If WCSD<n> is set, then addresses n*1024 through zn*1024+1023 are writable (i.e., that WCSA<n>  EQLU n corresponds to writable control store). n=4 corresponds to WCS that is reserved to DIGITAL for diagnostics and engineering change orders. Other fields correspond to blocks of control that can be used to implement customer- or CSS-specific microcode. Each word of control store contains 96 bits plus 3 parity bits. To write one or more words, initialize WCS ADDR to the address and CTR to 0. Then each MTPR to WCSD will write the
Privileged Registers

Figure 21-13 illustrates the writable control store data register.

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>WCS DATA</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>PRESENT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 21-13  Writable Control Store Data Register (WCSD)

The next 32 bits and automatically increment CTR. When CTR becomes 3, it is automatically cleared and WCS ADDR is incremented. If PIN is set, then any writes to WCSD are done with inverted parity. An attempt to execute a microword with bad parity results in a machine check. At bootstrap time, the contents of WCSA are UNPREDICTABLE. Figure 21-14 illustrates the microprogram breakpoint address register.

<table>
<thead>
<tr>
<th>31</th>
<th>13</th>
<th>12</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MBZ</td>
<td>MICRO PC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 21-14  Microprogram Breakpoint Address Register (MBRK)

Whenever the microprogram PC matches the contents of MBRK, an external signal is asserted. If the console has enabled stop on microbreak, then the processor clock is stopped when this signal is asserted. If the console has not enabled microbreak, then this signal is available as a diagnostic scope point. Many diagnostics use the NOP instruction to trigger this method of giving a scope point. At bootstrap time, the contents of MBRK are UNPREDICTABLE.
CHAPTER 22
RELIABILITY AVAILABILITY
MAINTAINABILITY PROGRAM

INTRODUCTION
A significant factor guiding the development of the VAX-11/780 computer system was an extensive reliability, availability, maintainability program (RAMP). This program affected all aspects of the product, from the design of the basic hardware and software architectures through the final product, the VAX-11/780.

The first goal of the RAMP program was to utilize system design criteria that would effectively increase the mean time between failure rate (MTBF) of the system. Product reliability therefore implies a design that minimizes hardware, software, and system failures.

The second goal of the RAMP program was to incorporate a design that provided fewer and less time consuming maintenance steps, performed with greater ease and better diagnostic tools. System maintainability can be measured in terms of mean time to repair (MTTR). The objective of the RAMP program was to decrease the MTTR through better diagnostic programs and procedures, and packaging that facilitates repairs. The VAX-11/780 computer system was designed and built with integral hardware and software features that continually monitor and verify system integrity.

HARDWARE RAMP FEATURES
A summary of the VAX-11/780 hardware RAMP features contributing to the overall reliability of the system follows:

• Four Hierarchical Access Modes (kernel, executive, supervisor, and user) protect system information and improve system reliability and integrity.

• A Diagnostic Console, consisting of an LSI-11 microcomputer, floppy disk, and console terminal, provides both local and remote diagnosis of system errors and simplifies system bootstrap and software updates. Simple console commands replace lights and switches. The diagnostic console provides faster and easier maintenance procedures and increases availability.
Automatic **Consistency and Error Checking** detects abnormal instruction uses and illegal arithmetic conditions (overflow, underflow, and divide by zero). Continual checking by the hardware (and uniform exception handling by the software) increases data reliability.

**Special Instructions**, such as CALL and RETURN, provide a standard program-calling interface for increased reliability.

**Integral Fault Detection and Maintenance Features**, including:

- **ECC on memory** detects all double-bit errors and corrects all single-bit errors to increase availability and aid in maintenance.

- **ECC on the RP05, RP06, and RK06 disks** detects all errors up to 11 bits and corrects errors in a single error burst of 11 bits.

- An **SBI history silo** maintains a history of the 16 most recent cycles of bus activity and may be examined to aid in problem isolation.

- **Maintenance registers** permit forced error conditions for diagnostic purposes.

- A **high resolution interval timer** permits testing of time-dependent functions.

- Extensive **parity checking** is performed on the SBI, MASSBUS and UNIBUS adapters, memory cache, address translation buffer, microcode, and writable diagnostic control store.

- A **watchdog timer** in the LSI-11 diagnostic console detects hung machine conditions and allows crash/restart recovery actions.

- **Clock margining** provides diagnostic variation of the clock rate and aids in problem isolation.

- **Disabling of the memory management and the cache** aids in isolating hardware problems.

**Fault Tolerance Features**, including:

- **Detection and recording of bad blocks** on disk surfaces increase the reliability of the medium.

- **Write-verify checking hardware** in peripherals is available to verify all input and output disk and tape operations and to ensure data reliability.

- **Track offset retry hardware** enables programmed software recovery from disk transfer errors.

An in-depth description of each of the hardware RAMP features follows:
Hierarchical Access Modes
The memory management hardware defines four hierarchical modes of access privilege: kernel, executive, supervisor, and user. Read and write access to memory is designated separately for each access mode. The VAX operating system is designed so that only the most critical components run in highly privileged access modes (kernel and executive). This “layered” design increases protection, and consequently, data reliability and integrity, for the system and for users.

Diagnostic Console
The diagnostic console is an integral part of the VAX-11/780 processor. It includes an LSI-11 microcomputer, floppy disk, and console terminal and is used for both remote and local diagnosis and system maintenance activities.

The diagnostic console is an integral part of the system. If the LSI-11 console terminal is inoperative, another terminal may be substituted. However, the microcomputer and the floppy diskette are crucial system components; if these units are inoperative, the reliability of the system is seriously impaired. The LSI-11 performs a self-test on power-up.

Consistency and Error Checking
During the execution of many instructions, consistency checks are performed on the operands specified. If these checks fail, an exception is signalled and the current instruction sequence is suspended. The exception handler is entered and the system software or the user’s program provides an appropriate response to the condition. Such checks increase data reliability by preventing various error conditions from propagating through a data base or a system. Some of the checking performed includes:

- **Arithmetic Traps** Traps occur when overflow, underflow, and divide by zero arithmetic conditions are detected. Hardware detection of these error conditions allows checking to be used in high performance software, where software checking would be prohibitively slow. Several of the arithmetic traps, integer overflow, index, and decimal string, are new on the VAX-11/780. Overflow and underflow traps may be enabled or disabled by setting bits in the Processor Status Longword, allowing the arithmetic exception conditions to be ignored, if appropriate.

- **Limit Checking Traps** Decimal string instructions all have length limit checks (0-31 decimal digits) performed on output strings to ensure that instructions do not overwrite adjacent data.
Reserved operand traps “Reserved-to-customer” and “reserved-
to-DIGITAL” fields and opcodes ensure that customer extensions to
the VAX-11/780 architecture (e.g., user-defined instructions or data
structures) do not conflict with future DIGITAL expansions.

Special Instructions
The CALL and RETURN instructions have hardware-implemented reg-
ister save/restore and consistency checking. The use of these instruc-
tions provides a standard interface which is identical on user routine
calls and system calls.

The CRC (Calculate Cyclic Redundancy Check) instruction provides
powerful block checking error code calculations, such as are needed
in communications applications.

Integral Fault Detection and Maintenance Features
These features aid in the diagnosis of hardware errors and in the
efficient maintenance of the system. Specific features include the
following:

- Memory error correcting code (ECC) will correct all single-bit mem-
ory errors, and will detect double-bit memory errors. ECC provides
protection for non-repeatable errors by automatically correcting
data. Detections and corrections are noted in the error log as a
preventive maintenance aid.

- Disk error correcting code detects all errors up to 11 bits and cor-
rects errors in a single error burst of 11 bits. Detections and correc-
tions are noted in the error log as a preventive maintenance aid.

- A System Identifications (SID) hardware register maintains informa-
tion pertinent to the system processor: type and serial number. This
information may be examined (during the software logging process,
for example) to determine the engineering status of the processor.

- A sixteen-level silo monitors SBI activity and contains a history of
the 16 most recent cycles of bus activity. If an error or predeter-
mined special condition occurs, the silo is latched (i.e., the error or
condition can cause the silo contents to “freeze”; see the SBI Silo
Comparator, in the following paragraph) and the contents of the silo
can be examined to help determine the cause of the problem.

- Several maintenance registers contain bus-specific maintenance in-
formation and can be examined at the time of an error to help
determine the cause. These registers are:
  - SBI Fault/Status Register, which detects faults and conditions of
    the SBI
— SBI Silo Comparator, used to lock the SBI silo on predetermined conditions (e.g., specific number of cycles after an event)
— SBI Error Register, which indicates the type of error detected by SBI hardware
— SBI Timeout Address, which contains the physical address that caused a timeout condition on the SBI
— Cache Parity Register, which indicates where parity errors were detected
— SBI Maintenance Register, used to force error conditions in the cache or SBI for diagnostic and simulation purposes (e.g., forced bus timeout or cache miss)
— Translation Buffer Parity Register, which indicates where parity errors were detected

• A high resolution interval timer (1 μsec) is used by diagnostics to test time-dependent functions without requiring machine-specific timing loops in programs.

• Parity and protocol checks are performed on SBI data and address. Parity checks are performed on: MASSBUS data, control and address translation; UNIBUS address translation; memory cache data and address; address translation buffer transaction; microcode; writable user diagnostic control store (1 parity bit for each 32 bits); and CPU internal buses.

• A watchdog timer in the LSI-11 detects hung machine conditions (such as a hang in the microcode or a halt condition). Indicator lights on the front panel show whether the VAX-11/780 CPU is running or in a halt state. If the auto/restart switch on the processor console is set, automatic crash/restart recovery actions are initiated after either a hang condition or a halt.

• Clock margining is provided and causes the SBI clock rate to be varied by console commands to aid the field service engineer in diagnosing intermittent hardware problems.

• Memory management and the cache may be disabled by diagnostics to aid in isolating hardware problems.

Fault Tolerance Features
These features provide the means to continue processing without loss of information, even though hardware errors may be occurring. Specific features include the following:

• The VAX-11/780 performs dynamic bad block handling. Bad blocks may occur when a disk surface becomes worn, or as a result of a failure in the hardware that performed the data transfer. When the VAX-11/780 hardware detects a bad block during a read, the VAX
operating system marks the header of the file in which the error occurred. When the file is eventually de-allocated, the system checks the file header to see if any bad blocks exist in the file. If so, they are designated “permanently in use” and are not allocated for use by other files.

- **Verify checking hardware** for mass storage peripherals is supported by the VAX device drivers. The hardware compares each block for errors immediately after the block is read or written. Checking may be performed on all reads or writes to a file or volume, or specified for a single read or write. This capability increases readability (but also increases the time to complete the read or write operation).

- **Track offset retry hardware** is used (by the operating system) to attempt recovery from disk transfer errors. If an error occurs during a read operation, the error is signalled by the disk hardware and the operation retried. If the retry fails, the disk head may be repositioned slightly (offset) on either side of the normal track location in an attempt to read the data correctly.
PART 4
APPENDICES
GLOSSARY
INDEX
COMMONLY USED MNEMONICS

ACP  Ancillary Control Process
ANS  American National Standard
ASCII American Standard Code for Information Interchange
AST  Asynchronous System Trap
ASTLVL Asynchronous System Trap Level
CCB  Channel Control Block
CM  Compatibility Mode bit in the hardware PSL
CRB  Channel Request Block
CRC  Cyclic Redundancy Check
DAP  Data Access Protocol
DDB  Device Data Block
DDCMP DIGITAL Data Communications Message Protocol
DDT  Driver Data Table
DV  Decimal Overflow trap enable bit in the PSW
ECB  Exit Control Block
ECC  Error Correction Code
ESP  Executive Mode Stack Pointer
ESR  Exception Service Routine
F11ACP Files-11 Ancillary Control Process
FAB  File Access Block
FCA  Fixed Control Area
FCB  File Control Block
FCS  File Control Services
FDT  Function Decision Table
FP  Frame Pointer
FPD  First Part (of an instruction) Done
FU  Floating Underflow trap enable bit in the PSW
GSD  Global Section Descriptor
GST  Global Symbol Table
IDB  Interrupt Dispatch Block
IPL  Interrupt Priority Level
IRP  I/O Request Packet
ISECT Image Section
ISD  Image Section Descriptor
ISP  Interrupt Stack Pointer
IS  Interrupt Stack bit in PSL
ISR  Interrupt Service Routine
IV  Integer Overflow trap enable bit in the PSW
KSP  Kernel Mode Stack Pointer
Appendix A

MBA  MASSBUS Adapter
MBZ  Must Be Zero
MCR  Monitor Console Routine
MFD  Master File Directory
MFPR Move From Process Register instruction
MME  Memory Mapping Enable
MTPR Move To Process Register instruction
MUTEX Mutual Exclusion semaphore
NSP  Network Services Protocol
OPCOM Operator Communication Manager
P0BR Program region Base Register
P0LR Program region Length Register
P0PT Program region Page Table
P1BR Control region Base Register
P1LR Control region Limit Register
P1PT Control region Page Table
PC   Program Counter
PCB  Process Control Block
PCBB Process Control Block Base register
PFN  Page Frame Number
PID  Process Identification Number
PME  Performance Monitor Enable bit in PCB
PSECT Program Section
PSL  Processor Status Longword
PSW  Processor Status Word
PTE  Page Table Entry
QIO  Queue Input/Output Request system service
RAB  Record Access Block
RFA  Record's File Address
RMS  Record Management Services
RWED Read, Write, Execute, Delete
SBI  Synchronous Backplane Interconnect
SBR  System Base Register
SCB  System Control Block
SCBB System Control Block Base register
SLR  System Length Register
SP   Stack Pointer
SPT  System Page Table
SSP  Supervisor Mode Stack Pointer
SVA  System Virtual Address
TP   Trace trap Pending bit in PSL
UBA  UNIBUS Adapter
UCB  Unit Control Block
UCS  User Control Store
UETP User Environment Test Package
UFD  User File Directory
UIC  User Identification Code
<table>
<thead>
<tr>
<th><strong>Abbreviation</strong></th>
<th><strong>Definition</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>USP</td>
<td>User mode Stack Pointer</td>
</tr>
<tr>
<td>VCB</td>
<td>Volume Control Block</td>
</tr>
<tr>
<td>VPN</td>
<td>Virtual Page Number</td>
</tr>
<tr>
<td>WCB</td>
<td>Window Control Block</td>
</tr>
<tr>
<td>WCS</td>
<td>Writeable Control Store</td>
</tr>
<tr>
<td>WDCS</td>
<td>Writeable Diagnostic Control Store</td>
</tr>
</tbody>
</table>
### APPENDIX B

### INSTRUCTION INDEX

#### MNEMONIC LISTING

<table>
<thead>
<tr>
<th>MNEMONIC</th>
<th>INSTRUCTION</th>
<th>_OPCODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACBB</td>
<td>Add compare and branch byte</td>
<td>9D</td>
</tr>
<tr>
<td>ACBD</td>
<td>Add compare and branch D_floating</td>
<td>6F</td>
</tr>
<tr>
<td>ACBF</td>
<td>Add compare and branch F_floating</td>
<td>4F</td>
</tr>
<tr>
<td>ACBL</td>
<td>Add compare and branch long</td>
<td>F1</td>
</tr>
<tr>
<td>ACBW</td>
<td>Add compare and branch word</td>
<td>3D</td>
</tr>
<tr>
<td>ADAWI</td>
<td>Add aligned word interlocked</td>
<td>58</td>
</tr>
<tr>
<td>ADDB2</td>
<td>Add byte 2 operand</td>
<td>80</td>
</tr>
<tr>
<td>ADDB3</td>
<td>Add byte 3 operand</td>
<td>81</td>
</tr>
<tr>
<td>ADDD2</td>
<td>Add D_floating 2 operand</td>
<td>60</td>
</tr>
<tr>
<td>ADDD3</td>
<td>Add D_floating 3 operand</td>
<td>61</td>
</tr>
<tr>
<td>ADDF2</td>
<td>Add F_floating 2 operand</td>
<td>40</td>
</tr>
<tr>
<td>ADDF3</td>
<td>Add F_floating 3 operand</td>
<td>41</td>
</tr>
<tr>
<td>ADDL2</td>
<td>Add long 2 operand</td>
<td>C0</td>
</tr>
<tr>
<td>ADDL3</td>
<td>Add long 3 operand</td>
<td>C1</td>
</tr>
<tr>
<td>ADDP4</td>
<td>Add packed 4 operand</td>
<td>20</td>
</tr>
<tr>
<td>ADDP6</td>
<td>Add packed 6 operand</td>
<td>21</td>
</tr>
<tr>
<td>ADDW2</td>
<td>Add word 2 operand</td>
<td>A0</td>
</tr>
<tr>
<td>ADDW3</td>
<td>Add word 3 operand</td>
<td>A1</td>
</tr>
<tr>
<td>ADWC</td>
<td>Add with carry</td>
<td>D8</td>
</tr>
<tr>
<td>AOBLEQ</td>
<td>Add one and branch on less or equal</td>
<td>F3</td>
</tr>
<tr>
<td>AOBLSS</td>
<td>Add one and branch on less</td>
<td>F2</td>
</tr>
<tr>
<td>ASHL</td>
<td>Arithmetic shift long</td>
<td>78</td>
</tr>
<tr>
<td>ASHP</td>
<td>Arithmetic shift and round packed</td>
<td>F8</td>
</tr>
<tr>
<td>ASHQ</td>
<td>Arithmetic shift quad</td>
<td>79</td>
</tr>
<tr>
<td>BBC</td>
<td>Branch on bit clear</td>
<td>E1</td>
</tr>
<tr>
<td>BBCC</td>
<td>Branch on bit clear and clear</td>
<td>E5</td>
</tr>
<tr>
<td>BBCCI</td>
<td>Branch on bit clear and clear interlocked</td>
<td>E7</td>
</tr>
<tr>
<td>BBCS</td>
<td>Branch on bit clear and set</td>
<td>E3</td>
</tr>
<tr>
<td>BBS</td>
<td>Branch on bit set</td>
<td>E0</td>
</tr>
<tr>
<td>BBSC</td>
<td>Branch on bit set and clear</td>
<td>E4</td>
</tr>
<tr>
<td>BBSS</td>
<td>Branch on bit set and set</td>
<td>E2</td>
</tr>
<tr>
<td>BBSSI</td>
<td>Branch on bit set and set interlocked</td>
<td>E6</td>
</tr>
</tbody>
</table>
### Appendix B

<table>
<thead>
<tr>
<th>MNEMONIC</th>
<th>INSTRUCTION</th>
<th>OPCODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCC</td>
<td>Branch on carry clear</td>
<td>1E</td>
</tr>
<tr>
<td>BCS</td>
<td>Branch on carry set</td>
<td>1F</td>
</tr>
<tr>
<td>BEQL</td>
<td>Branch on equal</td>
<td>13</td>
</tr>
<tr>
<td>BEQLU</td>
<td>Branch on equal unsigned</td>
<td>13</td>
</tr>
<tr>
<td>BGEQ</td>
<td>Branch on greater or equal</td>
<td>18</td>
</tr>
<tr>
<td>BGEQU</td>
<td>Branch on greater or equal unsigned</td>
<td>1E</td>
</tr>
<tr>
<td>BGTR</td>
<td>Branch on greater</td>
<td>14</td>
</tr>
<tr>
<td>BGTRU</td>
<td>Branch on greater unsigned</td>
<td>1A</td>
</tr>
<tr>
<td>BICB2</td>
<td>Bit clear byte 2 operand</td>
<td>8A</td>
</tr>
<tr>
<td>BICB3</td>
<td>Bit clear byte 3 operand</td>
<td>8B</td>
</tr>
<tr>
<td>BICL2</td>
<td>Bit clear long 2 operand</td>
<td>CA</td>
</tr>
<tr>
<td>BICL3</td>
<td>Bit clear long 3 operand</td>
<td>CB</td>
</tr>
<tr>
<td>BICPSW</td>
<td>Bit clear program status word</td>
<td>B9</td>
</tr>
<tr>
<td>BICW2</td>
<td>Bit clear word 2 operand</td>
<td>AA</td>
</tr>
<tr>
<td>BICW3</td>
<td>Bit clear word 3 operand</td>
<td>AB</td>
</tr>
<tr>
<td>BISB2</td>
<td>Bit set byte 2 operand</td>
<td>88</td>
</tr>
<tr>
<td>BISB3</td>
<td>Bit set byte 3 operand</td>
<td>89</td>
</tr>
<tr>
<td>BISL2</td>
<td>Bit set long 2 operand</td>
<td>C8</td>
</tr>
<tr>
<td>BISL3</td>
<td>Bit set long 3 operand</td>
<td>C9</td>
</tr>
<tr>
<td>BISPSW</td>
<td>Bit set program status word</td>
<td>B8</td>
</tr>
<tr>
<td>BISW2</td>
<td>Bit set word 2 operand</td>
<td>A8</td>
</tr>
<tr>
<td>BISW3</td>
<td>Bit set word 3 operand</td>
<td>A9</td>
</tr>
<tr>
<td>BITB</td>
<td>Bit test byte</td>
<td>93</td>
</tr>
<tr>
<td>BITL</td>
<td>Bit test long</td>
<td>D3</td>
</tr>
<tr>
<td>BITW</td>
<td>Bit test word</td>
<td>B3</td>
</tr>
<tr>
<td>BLBC</td>
<td>Branch on low bit clear</td>
<td>E9</td>
</tr>
<tr>
<td>BLBS</td>
<td>Branch on low bit set</td>
<td>E8</td>
</tr>
<tr>
<td>BLEQ</td>
<td>Branch on less or equal</td>
<td>15</td>
</tr>
<tr>
<td>BLEQU</td>
<td>Branch on less or equal unsigned</td>
<td>1B</td>
</tr>
<tr>
<td>BLSS</td>
<td>Branch on less</td>
<td>19</td>
</tr>
<tr>
<td>BLSSU</td>
<td>Branch on less unsigned</td>
<td>1F</td>
</tr>
<tr>
<td>BNEQ</td>
<td>Branch on not equal</td>
<td>12</td>
</tr>
<tr>
<td>BNEQU</td>
<td>Branch on not equal unsigned</td>
<td>12</td>
</tr>
<tr>
<td>BPT</td>
<td>Break point fault</td>
<td>03</td>
</tr>
<tr>
<td>BRB</td>
<td>Branch with byte displacement</td>
<td>11</td>
</tr>
<tr>
<td>BRW</td>
<td>Branch with word displacement</td>
<td>31</td>
</tr>
</tbody>
</table>

462
<table>
<thead>
<tr>
<th>MNEMONIC</th>
<th>INSTRUCTION</th>
<th>_OPCODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSBB</td>
<td>Branch to subroutine with byte displacement</td>
<td>10</td>
</tr>
<tr>
<td>BSBW</td>
<td>Branch to subroutine with word displacement</td>
<td>30</td>
</tr>
<tr>
<td>BVC</td>
<td>Branch on overflow clear</td>
<td>1C</td>
</tr>
<tr>
<td>BVS</td>
<td>Branch on overflow set</td>
<td>1D</td>
</tr>
<tr>
<td>CALLG</td>
<td>Call with general argument list</td>
<td>FA</td>
</tr>
<tr>
<td>CALLS</td>
<td>Call with stack</td>
<td>FB</td>
</tr>
<tr>
<td>CASEB</td>
<td>Case byte</td>
<td>8F</td>
</tr>
<tr>
<td>CASEL</td>
<td>Case long</td>
<td>CF</td>
</tr>
<tr>
<td>CASEW</td>
<td>Case word</td>
<td>AF</td>
</tr>
<tr>
<td>CHME</td>
<td>Change mode to executive</td>
<td>BD</td>
</tr>
<tr>
<td>CHMK</td>
<td>Change mode to kernel</td>
<td>BC</td>
</tr>
<tr>
<td>CHMS</td>
<td>Change mode to supervisor</td>
<td>BE</td>
</tr>
<tr>
<td>CHMU</td>
<td>Change mode to user</td>
<td>BF</td>
</tr>
<tr>
<td>CLRBB</td>
<td>Clear byte</td>
<td>94</td>
</tr>
<tr>
<td>CLRD</td>
<td>Clear D_floating</td>
<td>7C</td>
</tr>
<tr>
<td>CLRF</td>
<td>Clear F_floating</td>
<td>7C</td>
</tr>
<tr>
<td>CLRL</td>
<td>Clear long</td>
<td>D4</td>
</tr>
<tr>
<td>CLRQ</td>
<td>Clear quad</td>
<td>7C</td>
</tr>
<tr>
<td>CLRW</td>
<td>Clear word</td>
<td>B4</td>
</tr>
<tr>
<td>CMPB</td>
<td>Compare byte</td>
<td>91</td>
</tr>
<tr>
<td>CMPC3</td>
<td>Compare character 3 operand</td>
<td>29</td>
</tr>
<tr>
<td>CMPC5</td>
<td>Compare character 5 operand</td>
<td>2D</td>
</tr>
<tr>
<td>CMPD</td>
<td>Compare D_floating</td>
<td>71</td>
</tr>
<tr>
<td>CMPF</td>
<td>Compare F_floating</td>
<td>51</td>
</tr>
<tr>
<td>CMPL</td>
<td>Compare long</td>
<td>D1</td>
</tr>
<tr>
<td>CMPP3</td>
<td>Compare packed 3 operand</td>
<td>35</td>
</tr>
<tr>
<td>CMPP4</td>
<td>Compare packed 4 operand</td>
<td>37</td>
</tr>
<tr>
<td>CMPV</td>
<td>Compare field</td>
<td>EC</td>
</tr>
<tr>
<td>CMPW</td>
<td>Compare word</td>
<td>B1</td>
</tr>
<tr>
<td>CMPZV</td>
<td>Compare zero-extended field</td>
<td>ED</td>
</tr>
<tr>
<td>CRC</td>
<td>Calculate cyclic redundancy check</td>
<td>0B</td>
</tr>
<tr>
<td>CVTBD</td>
<td>Convert byte to D_floating</td>
<td>6C</td>
</tr>
<tr>
<td>CVTBF</td>
<td>Convert byte to F_floating</td>
<td>4C</td>
</tr>
<tr>
<td>CVTBL</td>
<td>Convert byte to long</td>
<td>98</td>
</tr>
</tbody>
</table>
### Appendix B

<table>
<thead>
<tr>
<th>MNEMONIC</th>
<th>INSTRUCTION</th>
<th>OPCODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVTROF</td>
<td>Convert byte to word</td>
<td>99</td>
</tr>
<tr>
<td>CVTDB</td>
<td>Convert double to byte</td>
<td>68</td>
</tr>
<tr>
<td>CVTDF</td>
<td>Convert D_floating to F_floating</td>
<td>76</td>
</tr>
<tr>
<td>CVTDL</td>
<td>Convert D_floating to long</td>
<td>6A</td>
</tr>
<tr>
<td>CVTDW</td>
<td>Convert D_floating to word</td>
<td>69</td>
</tr>
<tr>
<td>CVTFB</td>
<td>Convert F_floating to byte</td>
<td>48</td>
</tr>
<tr>
<td>CVTFA</td>
<td>Convert F_floating to D_floating</td>
<td>55</td>
</tr>
<tr>
<td>CVTFL</td>
<td>Convert F_floating to long</td>
<td>4A</td>
</tr>
<tr>
<td>CVTFW</td>
<td>Convert F_floating to word</td>
<td>49</td>
</tr>
<tr>
<td>CVTLD</td>
<td>Convert long to D_floating</td>
<td>6E</td>
</tr>
<tr>
<td>CVTFLF</td>
<td>Convert long to F_floating</td>
<td>4E</td>
</tr>
<tr>
<td>CVTPL</td>
<td>Convert long to packed</td>
<td>F9</td>
</tr>
<tr>
<td>CVTLW</td>
<td>Convert long to word</td>
<td>F7</td>
</tr>
<tr>
<td>CVTSL</td>
<td>Convert packed to long</td>
<td>36</td>
</tr>
<tr>
<td>CVTTP</td>
<td>Convert trailing numeric to packed</td>
<td>26</td>
</tr>
<tr>
<td>CVTPT</td>
<td>Convert packed to trailing numeric</td>
<td>24</td>
</tr>
<tr>
<td>CVTSP</td>
<td>Convert packed to leading separate numeric</td>
<td>08</td>
</tr>
<tr>
<td>CVTRDL</td>
<td>Convert rounded D_floating to long</td>
<td>6B</td>
</tr>
<tr>
<td>CVTRFL</td>
<td>Convert rounded F_floating to long</td>
<td>4B</td>
</tr>
<tr>
<td>CVTSP</td>
<td>Convert leading separate numeric to packed</td>
<td>09</td>
</tr>
<tr>
<td>CVTWB</td>
<td>Convert word to byte</td>
<td>33</td>
</tr>
<tr>
<td>CVTWD</td>
<td>Convert word to D_floating</td>
<td>6D</td>
</tr>
<tr>
<td>CVTWF</td>
<td>Convert word to F_floating</td>
<td>4D</td>
</tr>
<tr>
<td>CVTWL</td>
<td>Convert word to long</td>
<td>32</td>
</tr>
<tr>
<td>DECB</td>
<td>Decrement byte</td>
<td>97</td>
</tr>
<tr>
<td>DECL</td>
<td>Decrement long</td>
<td>D7</td>
</tr>
<tr>
<td>DECW</td>
<td>Decrement word</td>
<td>B7</td>
</tr>
<tr>
<td>DIVB2</td>
<td>Divide byte 2 operand</td>
<td>86</td>
</tr>
<tr>
<td>DIVB3</td>
<td>Divide byte 3 operand</td>
<td>87</td>
</tr>
<tr>
<td>DIVD2</td>
<td>Divide D_floating 2 operand</td>
<td>66</td>
</tr>
<tr>
<td>DIVD3</td>
<td>Divide D_floating 3 operand</td>
<td>67</td>
</tr>
<tr>
<td>DIVF2</td>
<td>Divide F_floating 2 operand</td>
<td>46</td>
</tr>
<tr>
<td>MNEMONIC</td>
<td>INSTRUCTION</td>
<td>OPCODE</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>DIVF3</td>
<td>Divide F_floating 3 operand</td>
<td>47</td>
</tr>
<tr>
<td>DIVL2</td>
<td>Divide long 2 operand</td>
<td>C6</td>
</tr>
<tr>
<td>DIVL3</td>
<td>Divide long 3 operand</td>
<td>C7</td>
</tr>
<tr>
<td>DIVP</td>
<td>Divide packed</td>
<td>27</td>
</tr>
<tr>
<td>DIVW2</td>
<td>Divide word 2 operand</td>
<td>A6</td>
</tr>
<tr>
<td>DIVW3</td>
<td>Divide word 3 operand</td>
<td>A7</td>
</tr>
<tr>
<td>EDITPC</td>
<td>Edit packed to character</td>
<td>38</td>
</tr>
<tr>
<td>EDIV</td>
<td>Extended divide</td>
<td>7B</td>
</tr>
<tr>
<td>EMODD</td>
<td>Extended modulus D_floating</td>
<td>74</td>
</tr>
<tr>
<td>EMODF</td>
<td>Extended modulus F_floating</td>
<td>54</td>
</tr>
<tr>
<td>EMUL</td>
<td>Extended multiply</td>
<td>7A</td>
</tr>
<tr>
<td>EXTV</td>
<td>Extract field</td>
<td>EE</td>
</tr>
<tr>
<td>EXTZV</td>
<td>Extract zero-extended field</td>
<td>EF</td>
</tr>
<tr>
<td>FFC</td>
<td>Find first clear bit</td>
<td>EB</td>
</tr>
<tr>
<td>FFS</td>
<td>Find first set bit</td>
<td>EA</td>
</tr>
<tr>
<td>HALT</td>
<td>Halt</td>
<td>00</td>
</tr>
<tr>
<td>INCB</td>
<td>Increment byte</td>
<td>96</td>
</tr>
<tr>
<td>INCL</td>
<td>Increment long</td>
<td>D6</td>
</tr>
<tr>
<td>INCW</td>
<td>Increment word</td>
<td>B6</td>
</tr>
<tr>
<td>INDEX</td>
<td>Compute index</td>
<td>0A</td>
</tr>
<tr>
<td>INSQHI</td>
<td>Insert into queue at head, interlocked</td>
<td>5C</td>
</tr>
<tr>
<td>INSQTI</td>
<td>Insert into queue at tail, interlocked</td>
<td>5D</td>
</tr>
<tr>
<td>INSQUE</td>
<td>Insert into queue</td>
<td>0E</td>
</tr>
<tr>
<td>INSV</td>
<td>Insert field</td>
<td>F0</td>
</tr>
<tr>
<td>JMP</td>
<td>Jump</td>
<td>17</td>
</tr>
<tr>
<td>JSB</td>
<td>Jump to subroutine</td>
<td>16</td>
</tr>
<tr>
<td>LDPCTX</td>
<td>Load process context</td>
<td>16</td>
</tr>
<tr>
<td>LOCC</td>
<td>Locate character</td>
<td>3A</td>
</tr>
<tr>
<td>MATCHC</td>
<td>Match characters</td>
<td>39</td>
</tr>
<tr>
<td>MCOMB</td>
<td>Move complemented byte</td>
<td>92</td>
</tr>
<tr>
<td>MCOMML</td>
<td>Move complemented long</td>
<td>D2</td>
</tr>
<tr>
<td>MCOMMW</td>
<td>Move complemented word</td>
<td>B2</td>
</tr>
<tr>
<td>MFPR</td>
<td>Move from privileged register</td>
<td>DB</td>
</tr>
<tr>
<td>MNEG</td>
<td>Move negated byte</td>
<td>8E</td>
</tr>
<tr>
<td>MNNEG</td>
<td>Move negated D_floating</td>
<td>72</td>
</tr>
</tbody>
</table>
### Appendix B

<table>
<thead>
<tr>
<th>MNEMONIC</th>
<th>INSTRUCTION</th>
<th>OPCODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNEG</td>
<td>Move negated F_floating</td>
<td>52</td>
</tr>
<tr>
<td>MNEGL</td>
<td>Move negated long</td>
<td>CE</td>
</tr>
<tr>
<td>MNEGW</td>
<td>Move negated word</td>
<td>AE</td>
</tr>
<tr>
<td>MOVAB</td>
<td>Move address of byte</td>
<td>9E</td>
</tr>
<tr>
<td>MOVAD</td>
<td>Move address of D_floating</td>
<td>7E</td>
</tr>
<tr>
<td>MOVAF</td>
<td>Move address of F_floating</td>
<td>DE</td>
</tr>
<tr>
<td>MOVAL</td>
<td>Move address of long</td>
<td>DE</td>
</tr>
<tr>
<td>MOVAQ</td>
<td>Move address of quad</td>
<td>7E</td>
</tr>
<tr>
<td>MOVAW</td>
<td>Move address of word</td>
<td>3E</td>
</tr>
<tr>
<td>MOVB</td>
<td>Move byte</td>
<td>90</td>
</tr>
<tr>
<td>MOVC3</td>
<td>Move character 3 operand</td>
<td>28</td>
</tr>
<tr>
<td>MOVC5</td>
<td>Move character 5 operand</td>
<td>2C</td>
</tr>
<tr>
<td>MOVD</td>
<td>Move D_floating</td>
<td>70</td>
</tr>
<tr>
<td>MOVF</td>
<td>Move F_floating</td>
<td>50</td>
</tr>
<tr>
<td>MOVL</td>
<td>Move long</td>
<td>D0</td>
</tr>
<tr>
<td>MOVAP</td>
<td>Move packed</td>
<td>34</td>
</tr>
<tr>
<td>MOVPSL</td>
<td>Move processor status longword</td>
<td>DC</td>
</tr>
<tr>
<td>MOVQ</td>
<td>Move quad</td>
<td>7D</td>
</tr>
<tr>
<td>MOVTC</td>
<td>Move translated characters</td>
<td>2E</td>
</tr>
<tr>
<td>MOVTUC</td>
<td>Move translated until character</td>
<td>2F</td>
</tr>
<tr>
<td>MOVW</td>
<td>Move word</td>
<td>B0</td>
</tr>
<tr>
<td>MOVZBL</td>
<td>Move zero-extended byte to long</td>
<td>9A</td>
</tr>
<tr>
<td>MOVZBW</td>
<td>Move zero-extended byte to word</td>
<td>9B</td>
</tr>
<tr>
<td>MOVZWL</td>
<td>Move zero-extended word to long</td>
<td>3C</td>
</tr>
<tr>
<td>MTPR</td>
<td>Move to privileged register</td>
<td>DA</td>
</tr>
<tr>
<td>MULB2</td>
<td>Multiply byte 2 operand</td>
<td>84</td>
</tr>
<tr>
<td>MULB3</td>
<td>Multiply byte 3 operand</td>
<td>85</td>
</tr>
<tr>
<td>MULD2</td>
<td>Multiply D_floating 2 operand</td>
<td>64</td>
</tr>
<tr>
<td>MULD3</td>
<td>Multiply D_floating 3 operand</td>
<td>65</td>
</tr>
<tr>
<td>MULF2</td>
<td>Multiply F_floating 2 operand</td>
<td>44</td>
</tr>
<tr>
<td>MULF3</td>
<td>Multiply F_floating 3 operand</td>
<td>45</td>
</tr>
<tr>
<td>MULL2</td>
<td>Multiply long 2 operand</td>
<td>C4</td>
</tr>
<tr>
<td>MULL3</td>
<td>Multiply long 3 operand</td>
<td>C5</td>
</tr>
<tr>
<td>MULP</td>
<td>Multiply packed</td>
<td>25</td>
</tr>
<tr>
<td>MULW2</td>
<td>Multiply word 2 operand</td>
<td>A4</td>
</tr>
<tr>
<td>MNEMONIC</td>
<td>INSTRUCTION</td>
<td>OPCODE</td>
</tr>
<tr>
<td>------------</td>
<td>---------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>MULW3</td>
<td>Multiply word 3 operand</td>
<td>A5</td>
</tr>
<tr>
<td>NOP</td>
<td>No operation</td>
<td>01</td>
</tr>
<tr>
<td>POLYD</td>
<td>Evaluate polynomial D_floating</td>
<td>75</td>
</tr>
<tr>
<td>POLYF</td>
<td>Evaluate polynomial F_floating</td>
<td>55</td>
</tr>
<tr>
<td>POPR</td>
<td>Pop registers</td>
<td>BA</td>
</tr>
<tr>
<td>PROBER</td>
<td>Probe read access</td>
<td>0C</td>
</tr>
<tr>
<td>PROBEW</td>
<td>Probe write access</td>
<td>0D</td>
</tr>
<tr>
<td>PUSHAB</td>
<td>Push address byte</td>
<td>9F</td>
</tr>
<tr>
<td>PUSHAD</td>
<td>Push address of D_floating</td>
<td>7F</td>
</tr>
<tr>
<td>PUSHAF</td>
<td>Push address of F_floating</td>
<td>DF</td>
</tr>
<tr>
<td>PUSHAL</td>
<td>Push address of long</td>
<td>DF</td>
</tr>
<tr>
<td>PUSHAQ</td>
<td>Push address of quad</td>
<td>7F</td>
</tr>
<tr>
<td>PUSHAW</td>
<td>Push address of word</td>
<td>3F</td>
</tr>
<tr>
<td>PUSHL</td>
<td>Push long</td>
<td>DD</td>
</tr>
<tr>
<td>PUSHR</td>
<td>Push registers</td>
<td>BB</td>
</tr>
<tr>
<td>REI</td>
<td>Return from exception or interrupt</td>
<td>02</td>
</tr>
<tr>
<td>REMQHI</td>
<td>Remove from queue head, interlocked</td>
<td>5E</td>
</tr>
<tr>
<td>REMQTI</td>
<td>Remove from queue tail, interlocked</td>
<td>5F</td>
</tr>
<tr>
<td>REMQUE</td>
<td>Remove from queue</td>
<td>0F</td>
</tr>
<tr>
<td>RET</td>
<td>Return from called procedure</td>
<td>04</td>
</tr>
<tr>
<td>ROTL</td>
<td>Rotate long</td>
<td>9C</td>
</tr>
<tr>
<td>RSB</td>
<td>Return from subroutine</td>
<td>05</td>
</tr>
<tr>
<td>SBWC</td>
<td>Subtract with carry</td>
<td>D9</td>
</tr>
<tr>
<td>SCANC</td>
<td>Scan for character</td>
<td>2A</td>
</tr>
<tr>
<td>SKPC</td>
<td>Skip character</td>
<td>3B</td>
</tr>
<tr>
<td>SOBGEQ</td>
<td>Subtract one and branch on greater or equal</td>
<td>F4</td>
</tr>
<tr>
<td>SOBGTR</td>
<td>Subtract one and branch on greater</td>
<td>F5</td>
</tr>
<tr>
<td>SPANC</td>
<td>Span characters</td>
<td>2B</td>
</tr>
<tr>
<td>SUBB2</td>
<td>Subtract byte 2 operand</td>
<td>82</td>
</tr>
<tr>
<td>SUBB3</td>
<td>Subtract byte 3 operand</td>
<td>83</td>
</tr>
<tr>
<td>SUBD2</td>
<td>Subtract double 2 operand</td>
<td>62</td>
</tr>
<tr>
<td>SUBD3</td>
<td>Subtract double 3 operand</td>
<td>63</td>
</tr>
<tr>
<td>SUBF2</td>
<td>Subtract floating 2 operand</td>
<td>42</td>
</tr>
<tr>
<td>MNEMONIC</td>
<td>INSTRUCTION</td>
<td>OPCODE</td>
</tr>
<tr>
<td>----------</td>
<td>------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>SUBF3</td>
<td>Subtract floating 3 operand</td>
<td>43</td>
</tr>
<tr>
<td>SUBL2</td>
<td>Subtract long 2 operand</td>
<td>C2</td>
</tr>
<tr>
<td>SUBL3</td>
<td>Subtract long 3 operand</td>
<td>C3</td>
</tr>
<tr>
<td>SUBP4</td>
<td>Subtract packed 4 operand</td>
<td>22</td>
</tr>
<tr>
<td>SUBP6</td>
<td>Subtract packed 6 operand</td>
<td>23</td>
</tr>
<tr>
<td>SUBW2</td>
<td>Subtract word 2 operand</td>
<td>A2</td>
</tr>
<tr>
<td>SUBW3</td>
<td>Subtract word 3 operand</td>
<td>A3</td>
</tr>
<tr>
<td>SVPCTX</td>
<td>Save process context</td>
<td>07</td>
</tr>
<tr>
<td>TSTB</td>
<td>Test byte</td>
<td>95</td>
</tr>
<tr>
<td>TSTD</td>
<td>Test D_floating</td>
<td>73</td>
</tr>
<tr>
<td>TSTF</td>
<td>Test F_floating</td>
<td>53</td>
</tr>
<tr>
<td>TSTL</td>
<td>Test long</td>
<td>D5</td>
</tr>
<tr>
<td>TSTW</td>
<td>Test word</td>
<td>B5</td>
</tr>
<tr>
<td>XFC</td>
<td>Extended function call</td>
<td>FC</td>
</tr>
<tr>
<td>XORB2</td>
<td>Exclusive OR byte 2 operand</td>
<td>8C</td>
</tr>
<tr>
<td>XORB3</td>
<td>Exclusive OR byte 3 operand</td>
<td>8D</td>
</tr>
<tr>
<td>XORL2</td>
<td>Exclusive OR long 2 operand</td>
<td>CC</td>
</tr>
<tr>
<td>XORL3</td>
<td>Exclusive OR long 3 operand</td>
<td>CD</td>
</tr>
<tr>
<td>XORW2</td>
<td>Exclusive OR word 2 operand</td>
<td>TC</td>
</tr>
<tr>
<td>XORW3</td>
<td>Exclusive OR word 3 operand</td>
<td>AD</td>
</tr>
<tr>
<td></td>
<td><em>Reserved to DIGITAL</em></td>
<td>57</td>
</tr>
<tr>
<td></td>
<td><em>Reserved to DIGITAL</em></td>
<td>59</td>
</tr>
<tr>
<td></td>
<td><em>Reserved to DIGITAL</em></td>
<td>5A</td>
</tr>
<tr>
<td></td>
<td><em>Reserved to DIGITAL</em></td>
<td>5B</td>
</tr>
<tr>
<td></td>
<td><em>Reserved to DIGITAL</em></td>
<td>77</td>
</tr>
<tr>
<td>ESCD</td>
<td><em>Reserved to DIGITAL</em></td>
<td>FD</td>
</tr>
<tr>
<td>ESCE</td>
<td><em>Reserved to DIGITAL</em></td>
<td>FE</td>
</tr>
<tr>
<td>ESCF</td>
<td><em>Reserved to DIGITAL</em></td>
<td>FF</td>
</tr>
</tbody>
</table>

**_OPCODE LISTING**

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>MNEMONIC</th>
<th>INSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>HALT</td>
<td>Halt</td>
</tr>
<tr>
<td>01</td>
<td>NOP</td>
<td>No operation</td>
</tr>
<tr>
<td>02</td>
<td>REI</td>
<td>Return from exception or interrupt</td>
</tr>
<tr>
<td>03</td>
<td>BPT</td>
<td>Break point fault</td>
</tr>
<tr>
<td>04</td>
<td>RET</td>
<td>Return from called procedure</td>
</tr>
<tr>
<td>05</td>
<td>RSB</td>
<td>Return from subroutine</td>
</tr>
<tr>
<td>06</td>
<td>LDPCTX</td>
<td>Load process context</td>
</tr>
<tr>
<td>07</td>
<td>SVPCTX</td>
<td>Save process context</td>
</tr>
</tbody>
</table>

468
# Appendix B

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>MNEMONIC</th>
<th>INSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>08</td>
<td>CVTPS</td>
<td>Convert packed to leading separate numeric</td>
</tr>
<tr>
<td>09</td>
<td>CVTSP</td>
<td>Convert leading separate numeric to packed</td>
</tr>
<tr>
<td>OA</td>
<td>INDEX</td>
<td>Compute index</td>
</tr>
<tr>
<td>OB</td>
<td>CRC</td>
<td>Calculate cyclic redundancy check</td>
</tr>
<tr>
<td>OC</td>
<td>PROBER</td>
<td>Probe read access</td>
</tr>
<tr>
<td>OD</td>
<td>PROBEW</td>
<td>Prove write access</td>
</tr>
<tr>
<td>OE</td>
<td>INSQUE</td>
<td>Insert into queue</td>
</tr>
<tr>
<td>OF</td>
<td>REMQUE</td>
<td>Remove from queue</td>
</tr>
<tr>
<td>10</td>
<td>BSBB</td>
<td>Branch to subroutine with byte displacement</td>
</tr>
<tr>
<td>11</td>
<td>BRB</td>
<td>Branch with byte displacement</td>
</tr>
<tr>
<td>12</td>
<td>BNEQ, BNEQU</td>
<td>Branch on not equal, Branch on not equal unsigned</td>
</tr>
<tr>
<td>13</td>
<td>BEQL, BEQLU</td>
<td>Branch on equal, Branch on equal unsigned</td>
</tr>
<tr>
<td>14</td>
<td>BGTR</td>
<td>Branch on greater</td>
</tr>
<tr>
<td>15</td>
<td>BLEQ</td>
<td>Branch on less or equal</td>
</tr>
<tr>
<td>16</td>
<td>JSB</td>
<td>Jump to subroutine</td>
</tr>
<tr>
<td>17</td>
<td>JMP</td>
<td>Jump</td>
</tr>
<tr>
<td>18</td>
<td>BGEQ</td>
<td>Branch on greater or equal</td>
</tr>
<tr>
<td>19</td>
<td>BLSS</td>
<td>Branch on less</td>
</tr>
<tr>
<td>1A</td>
<td>BGTRU</td>
<td>Branch on greater unsigned</td>
</tr>
<tr>
<td>1B</td>
<td>BLEQU</td>
<td>Branch on less or equal unsigned</td>
</tr>
<tr>
<td>1C</td>
<td>BVC</td>
<td>Branch on overflow clear</td>
</tr>
<tr>
<td>1D</td>
<td>BVS</td>
<td>Branch on overflow set</td>
</tr>
<tr>
<td>1E</td>
<td>BGEQU, BCC</td>
<td>Branch on greater or equal unsigned, Branch on carry clear</td>
</tr>
<tr>
<td>1F</td>
<td>BLSSU, BCS</td>
<td>Branch on less unsigned, Branch on carry set</td>
</tr>
<tr>
<td>20</td>
<td>ADDP4</td>
<td>Add packed 4 operand</td>
</tr>
<tr>
<td>21</td>
<td>ADDP6</td>
<td>Add packed 6 operand</td>
</tr>
<tr>
<td>22</td>
<td>SUBP4</td>
<td>Subtract packed 4 operand</td>
</tr>
<tr>
<td>23</td>
<td>SUBP6</td>
<td>Subtract packed 6 operand</td>
</tr>
<tr>
<td>24</td>
<td>CVTPT</td>
<td>Convert packed to trailing numeric</td>
</tr>
<tr>
<td>25</td>
<td>MULP</td>
<td>Multiply packed</td>
</tr>
<tr>
<td>26</td>
<td>CVTTP</td>
<td>Convert trailing numeric to packed</td>
</tr>
<tr>
<td>27</td>
<td>DIVP</td>
<td>Divide packed</td>
</tr>
<tr>
<td>OPCODE</td>
<td>MNEMONIC</td>
<td>INSTRUCTION</td>
</tr>
<tr>
<td>-------</td>
<td>------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>28</td>
<td>MOV C3</td>
<td>Move character 3 operand</td>
</tr>
<tr>
<td>29</td>
<td>CMPC3</td>
<td>Compare character 3 operand</td>
</tr>
<tr>
<td>2A</td>
<td>SCANC</td>
<td>Scan for character</td>
</tr>
<tr>
<td>2B</td>
<td>SPANC</td>
<td>Span characters</td>
</tr>
<tr>
<td>2C</td>
<td>MOV C5</td>
<td>Move character 5 operand</td>
</tr>
<tr>
<td>2D</td>
<td>CMPC5</td>
<td>Compare character 5 operand</td>
</tr>
<tr>
<td>2E</td>
<td>MOVT C</td>
<td>Move translated characters</td>
</tr>
<tr>
<td>2F</td>
<td>MOVT UC</td>
<td>Move translated until character</td>
</tr>
<tr>
<td>30</td>
<td>BSBW</td>
<td>Branch to subroutine with word displacement</td>
</tr>
<tr>
<td>31</td>
<td>BRW</td>
<td>Branch with word displacement</td>
</tr>
<tr>
<td>32</td>
<td>CVTW L</td>
<td>Convert word to long</td>
</tr>
<tr>
<td>33</td>
<td>CVTW B</td>
<td>Convert word to byte</td>
</tr>
<tr>
<td>34</td>
<td>MOV P</td>
<td>Move packed</td>
</tr>
<tr>
<td>35</td>
<td>CMPP3</td>
<td>Compare packed 3 operand</td>
</tr>
<tr>
<td>36</td>
<td>CVTPL</td>
<td>Convert packed to long</td>
</tr>
<tr>
<td>37</td>
<td>CMPP4</td>
<td>Compare packed 4 operand</td>
</tr>
<tr>
<td>38</td>
<td>EDIT PC</td>
<td>Edit packed to character</td>
</tr>
<tr>
<td>39</td>
<td>MATCHC</td>
<td>Match characters</td>
</tr>
<tr>
<td>3A</td>
<td>LOCC</td>
<td>Locate character</td>
</tr>
<tr>
<td>3B</td>
<td>SKPC</td>
<td>Skip character</td>
</tr>
<tr>
<td>3C</td>
<td>MOVZWL</td>
<td>Move zero-extended word to long</td>
</tr>
<tr>
<td>3D</td>
<td>ACBW</td>
<td>Add compare and branch word</td>
</tr>
<tr>
<td>3E</td>
<td>MOVAW</td>
<td>Move address of word</td>
</tr>
<tr>
<td>3F</td>
<td>PUSH AW</td>
<td>Push address of word</td>
</tr>
<tr>
<td>40</td>
<td>ADD F2</td>
<td>Add F_floating 2 operand</td>
</tr>
<tr>
<td>41</td>
<td>ADD F3</td>
<td>Add F_floating 3 operand</td>
</tr>
<tr>
<td>42</td>
<td>SUB F2</td>
<td>Subtract F_floating 2 operand</td>
</tr>
<tr>
<td>43</td>
<td>SUB F3</td>
<td>Subtract F_floating 3 operand</td>
</tr>
<tr>
<td>44</td>
<td>MUL F2</td>
<td>Multiply F_floating 2 operand</td>
</tr>
<tr>
<td>45</td>
<td>MUL F3</td>
<td>Multiply F_floating 3 operand</td>
</tr>
<tr>
<td>46</td>
<td>DIV F2</td>
<td>Divide F_floating 2 operand</td>
</tr>
<tr>
<td>47</td>
<td>DIV F3</td>
<td>Divide F_floating 3 operand</td>
</tr>
<tr>
<td>48</td>
<td>CVTF B</td>
<td>Convert F_floating to byte</td>
</tr>
<tr>
<td>49</td>
<td>CVTF W</td>
<td>Convert F_floating to word</td>
</tr>
<tr>
<td>4A</td>
<td>CVTF L</td>
<td>Convert F_floating to long</td>
</tr>
<tr>
<td>4B</td>
<td>CVTR FL</td>
<td>Convert rounded F_floating to long</td>
</tr>
<tr>
<td>4C</td>
<td>CVT BF</td>
<td>Convert byte to F_floating</td>
</tr>
</tbody>
</table>
### Appendix B

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>MNEMONIC</th>
<th>INSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>4D</td>
<td>CVTWCF</td>
<td>Convert word to F-floating</td>
</tr>
<tr>
<td>4E</td>
<td>CVTLF</td>
<td>Convert long to F-floating</td>
</tr>
<tr>
<td>4F</td>
<td>ACBF</td>
<td>Add compare and branch F-floating</td>
</tr>
<tr>
<td>50</td>
<td>MOVF</td>
<td>Move F-floating</td>
</tr>
<tr>
<td>51</td>
<td>CMPF</td>
<td>Compare F-floating</td>
</tr>
<tr>
<td>52</td>
<td>MNEGF</td>
<td>Move negated F-floating</td>
</tr>
<tr>
<td>53</td>
<td>TSTF</td>
<td>Test F-floating</td>
</tr>
<tr>
<td>54</td>
<td>EMODF</td>
<td>Extended modulus F-floating</td>
</tr>
<tr>
<td>55</td>
<td>POLYF</td>
<td>Evaluate polynomial F-floating</td>
</tr>
<tr>
<td>56</td>
<td>CVTFD</td>
<td>Convert F-floating to D-floating</td>
</tr>
<tr>
<td>57</td>
<td>RESERVED</td>
<td>RESERVED to DIGITAL</td>
</tr>
<tr>
<td>58</td>
<td>ADAWI</td>
<td>Add aligned word interlocked</td>
</tr>
<tr>
<td>59</td>
<td>RESERVED</td>
<td>RESERVED to DIGITAL</td>
</tr>
<tr>
<td>5A</td>
<td>RESERVED</td>
<td>RESERVED to DIGITAL</td>
</tr>
<tr>
<td>5B</td>
<td>RESERVED</td>
<td>RESERVED to DIGITAL</td>
</tr>
<tr>
<td>5C</td>
<td>INSQHI</td>
<td>Insert into queue at head, interlocked</td>
</tr>
<tr>
<td>5D</td>
<td>INSQHI</td>
<td>Insert into queue at tail, interlocked</td>
</tr>
<tr>
<td>5E</td>
<td>REMQHI</td>
<td>Remove from queue head, interlocked</td>
</tr>
<tr>
<td>5F</td>
<td>REMQHI</td>
<td>Remove from queue tail, interlocked</td>
</tr>
<tr>
<td>60</td>
<td>ADDD2</td>
<td>Add D-floating 2 operand</td>
</tr>
<tr>
<td>61</td>
<td>ADDD3</td>
<td>Add D-floating 3 operand</td>
</tr>
<tr>
<td>62</td>
<td>SUBD2</td>
<td>Subtract D-floating 2 operand</td>
</tr>
<tr>
<td>63</td>
<td>SUBD3</td>
<td>Subtract D-floating 3 operand</td>
</tr>
<tr>
<td>64</td>
<td>MULD2</td>
<td>Multiply D-floating 2 operand</td>
</tr>
<tr>
<td>65</td>
<td>MULD3</td>
<td>Multiply D-floating 3 operand</td>
</tr>
<tr>
<td>66</td>
<td>DIVD2</td>
<td>Divide D-floating 2 operand</td>
</tr>
<tr>
<td>67</td>
<td>DIVD3</td>
<td>Divide D-floating 3 operand</td>
</tr>
<tr>
<td>68</td>
<td>CVTDB</td>
<td>Convert D-floating to byte</td>
</tr>
<tr>
<td>69</td>
<td>CVTDB</td>
<td>Convert D-floating to word</td>
</tr>
<tr>
<td>6A</td>
<td>CVTDL</td>
<td>Convert D-floating to long</td>
</tr>
<tr>
<td>6B</td>
<td>CVTRDL</td>
<td>Convert rounded D-floating to long</td>
</tr>
<tr>
<td>6C</td>
<td>CVTBD</td>
<td>Convert byte to D-floating</td>
</tr>
<tr>
<td>6D</td>
<td>CVTWD</td>
<td>Convert word to D-floating</td>
</tr>
<tr>
<td>6E</td>
<td>CVTLD</td>
<td>Convert long to D-floating</td>
</tr>
<tr>
<td>6F</td>
<td>ACBD</td>
<td>Add compare and branch D-floating</td>
</tr>
<tr>
<td>70</td>
<td>MOVD</td>
<td>Move D-floating</td>
</tr>
<tr>
<td>OPCODE</td>
<td>MNEMONIC</td>
<td>INSTRUCTION</td>
</tr>
<tr>
<td>--------</td>
<td>-----------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>71</td>
<td>CMPD</td>
<td>Compare D_floating</td>
</tr>
<tr>
<td>72</td>
<td>MNEGD</td>
<td>Move negated D_floating</td>
</tr>
<tr>
<td>73</td>
<td>TSTD</td>
<td>Test D_floating</td>
</tr>
<tr>
<td>74</td>
<td>EMODD</td>
<td>Extended modulus D_floating</td>
</tr>
<tr>
<td>75</td>
<td>POLYD</td>
<td>Evaluate polynomial D_floating</td>
</tr>
<tr>
<td>76</td>
<td>CVTDF</td>
<td>Convert D_floating to F_floating</td>
</tr>
<tr>
<td>77</td>
<td></td>
<td>RESERVED to DIGITAL</td>
</tr>
<tr>
<td>78</td>
<td>ASHL</td>
<td>Arithmetic shift long</td>
</tr>
<tr>
<td>79</td>
<td>ASHQ</td>
<td>Arithmetic shift quad</td>
</tr>
<tr>
<td>7A</td>
<td>EMUL</td>
<td>Extended multiply</td>
</tr>
<tr>
<td>7B</td>
<td>EDIV</td>
<td>Extended divide</td>
</tr>
<tr>
<td>7C</td>
<td>CLRQ, CLRQD</td>
<td>Clear quad, Clear D_floating</td>
</tr>
<tr>
<td>7D</td>
<td>MOVQ</td>
<td>Move quad</td>
</tr>
<tr>
<td>7E</td>
<td>MOVAQ, MOVAD</td>
<td>Move address of quad, Move</td>
</tr>
<tr>
<td></td>
<td></td>
<td>address of D_floating</td>
</tr>
<tr>
<td>7F</td>
<td>PUSHAQ, PUSHAD</td>
<td>Push address of quad, Push</td>
</tr>
<tr>
<td></td>
<td></td>
<td>address of D_floating</td>
</tr>
<tr>
<td>80</td>
<td>ADDB2</td>
<td>Add byte 2 operand</td>
</tr>
<tr>
<td>81</td>
<td>ADDB3</td>
<td>Add byte 3 operand</td>
</tr>
<tr>
<td>82</td>
<td>SUBB2</td>
<td>Subtract byte 2 operand</td>
</tr>
<tr>
<td>83</td>
<td>SUBB3</td>
<td>Subtract byte 3 operand</td>
</tr>
<tr>
<td>84</td>
<td>MULB2</td>
<td>Multiply byte 2 operand</td>
</tr>
<tr>
<td>85</td>
<td>MULB3</td>
<td>Multiply byte 3 operand</td>
</tr>
<tr>
<td>86</td>
<td>DIVB2</td>
<td>Divide byte 2 operand</td>
</tr>
<tr>
<td>87</td>
<td>DIVB3</td>
<td>Divide byte 3 operand</td>
</tr>
<tr>
<td>88</td>
<td>BISB2</td>
<td>Bit set byte 2 operand</td>
</tr>
<tr>
<td>89</td>
<td>BISB3</td>
<td>Bit set byte 3 operand</td>
</tr>
<tr>
<td>8A</td>
<td>BICB2</td>
<td>Bit clear byte 2 operand</td>
</tr>
<tr>
<td>8B</td>
<td>BICB3</td>
<td>Bit clear byte 3 operand</td>
</tr>
<tr>
<td>8C</td>
<td>XORB2</td>
<td>Exclusive OR byte 2 operand</td>
</tr>
<tr>
<td>8D</td>
<td>XORB3</td>
<td>Exclusive OR byte 3 operand</td>
</tr>
<tr>
<td>8E</td>
<td>MNEGB</td>
<td>Move negated byte</td>
</tr>
<tr>
<td>8F</td>
<td>CASEB</td>
<td>Case byte</td>
</tr>
<tr>
<td>90</td>
<td>MOVB</td>
<td>Move byte</td>
</tr>
<tr>
<td>91</td>
<td>CMPB</td>
<td>Compare byte</td>
</tr>
<tr>
<td>92</td>
<td>MCOMB</td>
<td>Move complemented byte</td>
</tr>
<tr>
<td>93</td>
<td>BITB</td>
<td>Bit test byte</td>
</tr>
<tr>
<td>94</td>
<td>CLRFB</td>
<td>Clear byte</td>
</tr>
<tr>
<td>95</td>
<td>TSTB</td>
<td>Test byte</td>
</tr>
</tbody>
</table>

472
### Appendix B

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>MNEMONIC</th>
<th>INSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>96</td>
<td>INCB</td>
<td>Increment byte</td>
</tr>
<tr>
<td>97</td>
<td>DECB</td>
<td>Decrement byte</td>
</tr>
<tr>
<td>98</td>
<td>CVTBL</td>
<td>Convert byte long</td>
</tr>
<tr>
<td>99</td>
<td>CVTBW</td>
<td>Convert byte to word</td>
</tr>
<tr>
<td>9A</td>
<td>MOVZBL</td>
<td>Move zero-extended byte to long</td>
</tr>
<tr>
<td>9B</td>
<td>MOVZBW</td>
<td>Move zero-extended byte to word</td>
</tr>
<tr>
<td>9C</td>
<td>ROTL</td>
<td>Rotate long</td>
</tr>
<tr>
<td>9D</td>
<td>ACBB</td>
<td>Add compare and branch byte</td>
</tr>
<tr>
<td>9E</td>
<td>MOVAB</td>
<td>Move address of byte</td>
</tr>
<tr>
<td>9F</td>
<td>PUSHAB</td>
<td>Push address of byte</td>
</tr>
<tr>
<td>A0</td>
<td>ADDW2</td>
<td>Add word 2 operand</td>
</tr>
<tr>
<td>A1</td>
<td>ADDW3</td>
<td>Add word 3 operand</td>
</tr>
<tr>
<td>A2</td>
<td>SUBW2</td>
<td>Subtract word 2 operand</td>
</tr>
<tr>
<td>A3</td>
<td>SUBW3</td>
<td>Subtract word 3 operand</td>
</tr>
<tr>
<td>A4</td>
<td>MULW2</td>
<td>Multiply word 2 operand</td>
</tr>
<tr>
<td>A5</td>
<td>MULW3</td>
<td>Multiply word 3 operand</td>
</tr>
<tr>
<td>A6</td>
<td>DIVW2</td>
<td>Divide word 2 operand</td>
</tr>
<tr>
<td>A7</td>
<td>DIVW3</td>
<td>Divide word 3 operand</td>
</tr>
<tr>
<td>A8</td>
<td>BISW2</td>
<td>Bit set word 2 operand</td>
</tr>
<tr>
<td>A9</td>
<td>BISW3</td>
<td>Bit set word 3 operand</td>
</tr>
<tr>
<td>AA</td>
<td>BICW2</td>
<td>Bit clear word 2 operand</td>
</tr>
<tr>
<td>AB</td>
<td>BICW3</td>
<td>Bit clear word 3 operand</td>
</tr>
<tr>
<td>AC</td>
<td>XORW2</td>
<td>Exclusive OR word 2 operand</td>
</tr>
<tr>
<td>AD</td>
<td>XORW3</td>
<td>Exclusive OR word 3 operand</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>MNEMONIC</th>
<th>INSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>AE</td>
<td>MNEGW</td>
<td>Move negated word</td>
</tr>
<tr>
<td>AF</td>
<td>CASEW</td>
<td>Case word</td>
</tr>
<tr>
<td>B0</td>
<td>MOVW</td>
<td>Move word</td>
</tr>
<tr>
<td>B1</td>
<td>CMPW</td>
<td>Compare word</td>
</tr>
<tr>
<td>B2</td>
<td>MCOMW</td>
<td>Move complemented word</td>
</tr>
<tr>
<td>B3</td>
<td>BITW</td>
<td>Bit test word</td>
</tr>
<tr>
<td>B4</td>
<td>CLRW</td>
<td>Clear word</td>
</tr>
<tr>
<td>B5</td>
<td>TSTW</td>
<td>Test word</td>
</tr>
<tr>
<td>B6</td>
<td>INCW</td>
<td>Increment word</td>
</tr>
<tr>
<td>B7</td>
<td>DECW</td>
<td>Decrement word</td>
</tr>
<tr>
<td>B8</td>
<td>BISPSW</td>
<td>Bit set processor status word</td>
</tr>
</tbody>
</table>

473
<table>
<thead>
<tr>
<th>OPCODE</th>
<th>MNEMONIC</th>
<th>INSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>B9</td>
<td>BICPSW</td>
<td>Bit clear processor status word</td>
</tr>
<tr>
<td>BA</td>
<td>POPR</td>
<td>Pop register</td>
</tr>
<tr>
<td>BB</td>
<td>PUSHR</td>
<td>Push register</td>
</tr>
<tr>
<td>BC</td>
<td>CHMK</td>
<td>Change mode to kernel</td>
</tr>
<tr>
<td>BD</td>
<td>CHME</td>
<td>Change mode to executive</td>
</tr>
<tr>
<td>BE</td>
<td>CHMS</td>
<td>Change mode to supervisor</td>
</tr>
<tr>
<td>BF</td>
<td>CHMU</td>
<td>Change mode to user</td>
</tr>
<tr>
<td>C0</td>
<td>ADDL2</td>
<td>Add long 2 operand</td>
</tr>
<tr>
<td>C1</td>
<td>ADDL3</td>
<td>Add long 3 operand</td>
</tr>
<tr>
<td>C2</td>
<td>SUBL2</td>
<td>Subtract long 2 operand</td>
</tr>
<tr>
<td>C3</td>
<td>SUBL3</td>
<td>Subtract long 3 operand</td>
</tr>
<tr>
<td>C4</td>
<td>MULL2</td>
<td>Multiply long 2 operand</td>
</tr>
<tr>
<td>C5</td>
<td>MULL3</td>
<td>Multiply long 3 operand</td>
</tr>
<tr>
<td>C6</td>
<td>DIVL2</td>
<td>Divide long 2 operand</td>
</tr>
<tr>
<td>C7</td>
<td>DIVL3</td>
<td>Divide long 3 operand</td>
</tr>
<tr>
<td>C8</td>
<td>BISL2</td>
<td>Bit set long 2 operand</td>
</tr>
<tr>
<td>C9</td>
<td>BISL3</td>
<td>Bit set long 3 operand</td>
</tr>
<tr>
<td>CA</td>
<td>BICL2</td>
<td>Bit clear long 2 operand</td>
</tr>
<tr>
<td>CB</td>
<td>BICL3</td>
<td>Bit clear long 3 operand</td>
</tr>
<tr>
<td>CC</td>
<td>XORL2</td>
<td>Exclusive OR long 2 operand</td>
</tr>
<tr>
<td>CD</td>
<td>XORL3</td>
<td>Exclusive OR long 3 operand</td>
</tr>
<tr>
<td>CE</td>
<td>MNEGL</td>
<td>Move negated long</td>
</tr>
<tr>
<td>CF</td>
<td>CASEL</td>
<td>Case long</td>
</tr>
<tr>
<td>D0</td>
<td>MOVVL</td>
<td>Move long</td>
</tr>
<tr>
<td>D1</td>
<td>CMPL</td>
<td>Compare long</td>
</tr>
<tr>
<td>D2</td>
<td>MCOMML</td>
<td>Move complemented long</td>
</tr>
<tr>
<td>D3</td>
<td>BITL</td>
<td>Bit test long</td>
</tr>
<tr>
<td>D4</td>
<td>CLRL, CLRF</td>
<td>Clear long, Clear F_float</td>
</tr>
<tr>
<td>D5</td>
<td>TSTL</td>
<td>Test long</td>
</tr>
<tr>
<td>D6</td>
<td>INCL</td>
<td>Increment long</td>
</tr>
<tr>
<td>D7</td>
<td>DECL</td>
<td>Decrement long</td>
</tr>
<tr>
<td>D8</td>
<td>ADWC</td>
<td>Add with carry</td>
</tr>
<tr>
<td>D9</td>
<td>SBWC</td>
<td>Subtract with carry</td>
</tr>
<tr>
<td>DA</td>
<td>MTPR</td>
<td>Move to processor register</td>
</tr>
<tr>
<td>DB</td>
<td>MFPR</td>
<td>Move from processor register</td>
</tr>
<tr>
<td>DC</td>
<td>MOVPSL</td>
<td>Move processor status longword</td>
</tr>
<tr>
<td>DD</td>
<td>PUSHL</td>
<td>Push long</td>
</tr>
<tr>
<td>OPCODE</td>
<td>MNEMONIC</td>
<td>INSTRUCTION</td>
</tr>
<tr>
<td>--------</td>
<td>----------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>DE</td>
<td>MOVAL, MOVAF</td>
<td>Move address of long, Move address of F_floating</td>
</tr>
<tr>
<td>DF</td>
<td>PUSHL, PUSHAF</td>
<td>Push address of long, Push address of F_floating</td>
</tr>
<tr>
<td>E0</td>
<td>BBS</td>
<td>Branch on bit set</td>
</tr>
<tr>
<td>E1</td>
<td>BBC</td>
<td>Branch on bit clear</td>
</tr>
<tr>
<td>E2</td>
<td>BBSS</td>
<td>Branch on bit set and set</td>
</tr>
<tr>
<td>E3</td>
<td>BBCS</td>
<td>Branch on bit clear and set</td>
</tr>
<tr>
<td>E4</td>
<td>BBSC</td>
<td>Branch on bit set and clear</td>
</tr>
<tr>
<td>E5</td>
<td>BBCC</td>
<td>Branch on bit clear and clear</td>
</tr>
<tr>
<td>E6</td>
<td>BBSSI</td>
<td>Branch on bit set and set interlocked</td>
</tr>
<tr>
<td>E7</td>
<td>BBCCI</td>
<td>Branch on bit clear and clear interlocked</td>
</tr>
<tr>
<td>E8</td>
<td>BLBS</td>
<td>Branch on low bit set</td>
</tr>
<tr>
<td>E9</td>
<td>BLBC</td>
<td>Branch on low bit clear</td>
</tr>
<tr>
<td>EA</td>
<td>FFS</td>
<td>Find first set bit</td>
</tr>
<tr>
<td>EB</td>
<td>FFC</td>
<td>Find first clear bit</td>
</tr>
<tr>
<td>EC</td>
<td>CMPV</td>
<td>Compare field</td>
</tr>
<tr>
<td>ED</td>
<td>CMPZV</td>
<td>Compare zero-extended field</td>
</tr>
<tr>
<td>EE</td>
<td>EXTV</td>
<td>Extract field</td>
</tr>
<tr>
<td>EF</td>
<td>EXTZV</td>
<td>Extract zero-extended field</td>
</tr>
<tr>
<td>F0</td>
<td>INSV</td>
<td>Insert field</td>
</tr>
<tr>
<td>F1</td>
<td>ACBL</td>
<td>Add compare and branch long</td>
</tr>
<tr>
<td>F2</td>
<td>AOBLSS</td>
<td>Add one and branch on less</td>
</tr>
<tr>
<td>F3</td>
<td>AOBLEQ</td>
<td>Add one and branch on less or equal</td>
</tr>
<tr>
<td>F4</td>
<td>SOBGEQ</td>
<td>Subtract one and branch on greater or equal</td>
</tr>
<tr>
<td>F5</td>
<td>SOBGTR</td>
<td>Subtract one and branch on greater</td>
</tr>
<tr>
<td>F6</td>
<td>CVTLB</td>
<td>Convert long to byte</td>
</tr>
<tr>
<td>F7</td>
<td>CVTLW</td>
<td>Convert long to word</td>
</tr>
<tr>
<td>F8</td>
<td>ASHP</td>
<td>Arithmetic shift and round packed</td>
</tr>
<tr>
<td>F9</td>
<td>CVTLP</td>
<td>Convert long to packed</td>
</tr>
<tr>
<td>FA</td>
<td>CALLG</td>
<td>Call with general argument list</td>
</tr>
<tr>
<td>FB</td>
<td>CALLS</td>
<td>Call with stack</td>
</tr>
<tr>
<td>FC</td>
<td>XFC</td>
<td>Extended function call</td>
</tr>
<tr>
<td>FD</td>
<td>ESCD</td>
<td>Reserved to DIGITAL</td>
</tr>
<tr>
<td>FE</td>
<td>ESCE</td>
<td>Reserved to DIGITAL</td>
</tr>
<tr>
<td>FF</td>
<td>ESCF</td>
<td>Reserved to DIGITAL</td>
</tr>
</tbody>
</table>
APPENDIX C
ADDRESS VALIDATION RULES

The memory management system described in Chapter 4 separates validation from the access of arguments. It is necessary to adopt certain coding conventions to prohibit unauthorized user access to sensitive data. Specifically it must not be possible for a user to call an inner access mode in such a way that will corrupt system integrity (e.g., cause supervisory code to write over itself) or incorrectly allow access to data that would otherwise have been inaccessible (e.g., the reading of a password table).

The following discussion sets forth operating system requirements that must be adhered to when accessing arguments from an inner access mode to avoid a breach of security.

The following requirements are made concerning operating system software:

1. Operating system software (kernel and executive mode) is trustworthy and does not maliciously attempt to break down the protection mechanisms (e.g., change the mapping or protection of pages at arbitrary times).

2. The protection of a shared page may not be changed unless the share count (a software construct) is one and the process attempting the change is that sharer. Share count = a software maintained record of the number of processes sharing a page.

3. The protection of a page with a nonzero I/O pending count (a software construct) may not be changed until the count goes to zero.

4. Operating system software will not deliver ASTs to outer access modes while the process is executing in an inner access mode.

5. Arguments passed to an inner access mode can be maliciously destroyed asynchronously by another process (e.g., shared data) or by an I/O transfer, but not by a less privileged mode of the executing process itself.

6. Kernel and executive stacks are never allocated in shared memory or accessible to other than their respective access modes.

The following summarizes related aspects of the VAX hardware:

1. Four access modes are provided and there is a stack per-process per-access mode.

2. Protection is hierarchical with the innermost access mode being the least restricted and the outermost the most restricted.
Appendix C

3. Four instructions are provided to change the processor mode to the four access modes (CHMU, CHMS, CHME, and CHMK); furthermore, when a process is executing a change mode instruction the access mode can only be decreased (changed to a more privileged mode) or left the same.

4. Two instructions are provided to validate the accessibility of arguments: Probe Read (PROBER) and Probe Write (PROBEW). These instructions validate the accessibility of arguments using the maximization of the Previous Mode field of PSL and a specified access mode. Thus only current and more restricted access modes can be probed.

5. The Return from Interrupt instruction (REI) insures that the current mode field of the restored PSL is greater than or equal to the current mode field of the current PSL and that the previous mode field of the restored PSL is greater than or equal to the current mode field of the restored PSL.

Given the previous operating system requirements, the following rules guarantee that less privileged modes cannot pass erroneous addresses to more privileged modes.

1. All addresses (including indirect addresses) passed as arguments to an inner access mode must be copied (preferably to a register, but in any case to an area of memory that is not modifiable by less privileged modes) before the accessibility of the actual argument is validated. In some programs such an address will later be used to asynchronously post information back to an outer access mode. In such cases, the least privileged access mode that can perform the specified read or write operation must be copied from the corresponding page table entry and stored with the argument address.

NOTE

Using least privileged does not work properly when the data structure resides in pages with different protection and the first page has a lesser protection value than the others. When checking the accessibility of such a structure in the context of the serial execution of the process, the check will succeed, but later when the accessibility is checked again during the asynchronous posting of information, the check will fail. This situation is considered to be an operating system bug (may cause the generation of a bug check) and merely causes no information to be posted.
2. The synchronous validation of argument addresses (i.e., as the result of serial program execution) must be explicitly coded using Probe instructions specifying an access mode of zero (i.e., cause maximization to previous access mode).

3. The asynchronous validation of argument addresses (i.e., as the result of software interrupts) must be explicitly coded using Probe instructions specifying the least privileged access mode stored when the argument address was saved (see 1) and with a previous access mode field equal to or greater than that of the current mode field of PSL (i.e., cause maximization to least privileged access mode).

4. All arguments to be written must be PROBEWed before they are written (otherwise there would be a potential protection violation).

5. All arguments to be read must be PROBERed before they are read to defend against arguments mapped to I/O space and thereby causing an I/O side effect.

6. All addresses passed from an outer access mode to an inner access mode must be copied and validated before being passed as arguments in a call to a more inner access mode. This insures the integrity of intermediate modes.

This discussion is centered on the validation of argument addresses. There are other arguments that also deserve similar handling. Such arguments are typically address modifiers (e.g., a buffer length) and in most cases must also be copied to insure system integrity.
APPENDIX D
VIRTUAL TO PHYSICAL ADDRESS TRANSLATION
The address translation example shown determines a 32-bit physical address on a VAX-11/780 system. Address translation on the VAX-11/750 follows an identical pattern, except that the translation results in a 24-bit physical address.

4. FETCH SPTE

5. CALCULATE PHYSICAL ADDRESS OF PIPTE
6. **FETCH PTE**

Assume contents of location (000380A0)

![Diagram of PTE structure]

**PROTECTION** + C which is user read; supervisor write

**NOT USED BY HARDWARE**

**PFN**

Bits 20:0 of PTE = PFN, these form the high order bits of physical address 29:9

7. **CALCULATE PHYSICAL ADDRESS OF OPERAND**

![Diagram of bit manipulation]
APPENDIX E

VAX-11/780 INTERNAL DATA (ID) BUS REGISTERS

Instruction Buffer Register

ID Address 00
Processor Address —

Time of Day Register (TODR)

ID Address 01
Processor Address 1B

Reserved Register

ID Address 02
Processor Address —
Appendix E

System Identification Register (SID)

ID Address 03
Processor Address 3E

Console Receive Control/Status Register (RXCS)

ID Address 04
Processor Address 20

Console Receive Data Buffer Register (RXDB)

ID Address 05
Processor Address 21

As defined in the software, bits <7:0> define the data field, bits <11:8> define the ID field and bit <15> is the error bit. However, the hardware is not restricted to this convention.
Appendix E

Console Transmit Control/Status Register (TXCS)

ID Address 06
Processor Address 22

As defined in the software, bits <7:0> define the data field, and bits <11:8> define the ID field. However, the hardware is not restricted to this convention.

DQ Register

ID Address 08
Processor Address —

Next Interval Count Register (NICR)

ID Address 09
Processor Address 19
Appendix E

Interval Clock Control/Status Register (ICCS)

ID Address 0A
Processor Address 18

Interval Counter Register (ICR)

ID Address 0B
Processor Address 1A

ID Address 0C
Processor Address 13 (Accesses AST level bits <2:0>(ASTR))
Processor Address 3D (Accesses performance monitor Enable bit <3>)(PMR)

Vector Register

ID Address 0D
Processor Address —
Appendix E

Software Interrupt Summary Register (SISR)

ID Address 0E
Processor Address 15

Processor Status Longword Register (PSL)

ID Address 0F
Processor Address 12

Translation Buffer Data Register (TB)

ID Address 10
Processor Address —

Reserved Register

ID Address 11
Processor Address —
Appendix E

Translation Buffer Control Register 0

ID Address 12
Processor Address —

Translation Buffer Control Register 1

ID Address 13
Processor Address —

Accelerator Control Register 0

ID Address 14
Processor Address —
Appendix E

Accelerator Control Register 1
ID Address 15
Processor Address —

Accelerator Maintenance Register
ID Address 16
Processor Address —

Accelerator Control/Status Register (ACCS)
ID Address 17
Processor Address 28

SBI Silo Register
ID Address 18
Processor Address 31
Appendix E

SBI Silo Register

ID Address 19
Processor Address 34

![Diagram of SBI Silo Register]

- Read Data Substitute Interrupt Enable
- Corrected Read Data
- Read Data Substitute
- Central Processor Time Out
- Central Processor Time Out Status 1
- Central Processor Time Out Status 0
- Central Processor Error Confirm Action
- Instruction Buffer Read Data Substitute
- Instruction Buffer Time Out
- Instruction Buffer Time Out Status 1
- Instruction Buffer Time Out Status 0
- Instruction Buffer Error Confirmation
- Double Bus Error
- SBI Not Busy

SBI Time Out Address Register

ID Address 1A
Processor Address 35

![Diagram of SBI Time Out Address Register]

- Mode 1
- Mode 0
- Protection Check
- Physical Address <29:02>

SBI Fault Signal Register

ID Address 1B
Processor Address 30

![Diagram of SBI Fault Signal Register]

- Parity Fault
- Unexpected Read Data Fault
- Multiple Transmitter Fault
- Transmitting During Fault
- Fault Latch
- Fault Interrupt Enable
- Fault Signal
- Fault Lock

492
Appendix E

SBI Silo Comparator Register
ID Address 1C
Processor Address 32

SBI/Cache Maintenance Register
ID Address 1D
Processor Address 33

Cache Parity Register
ID Address 1E
Processor Address —
Appendix E

Reserved Register
ID Address 1F
Processor Address —

Micro Stack Register
ID Address 20
Processor Address —

Micro Match Register
ID Address 21
Processor Address 3C

Writable Control Store Address Register
ID Address 22
Processor Address 2C

Writable Control Store Data Register
ID Address 23
Processor Address 2D

DATA TO WRITEABLE CONTROL STORE
Appendix E

P0 Base Register (P0BR)
ID Address 24
Processor Address 08

P1 Base Register (P1BR)
ID Address 25
Processor Address 0A

System Base Register (SBR)
ID Address 26
Processor Address 0C

Reserved Register
ID Address 27
Processor Address —

Kernal Stack Pointer Register (KSP)
ID Address 28
Processor Address 00
Executive Stack Pointer Register (ESP)
ID Address 29
Processor Address 01

Supervisor Stack Pointer Register (SSP)
ID Address 2A
Processor Address 02

User Stack Pointer Register (USP)
ID Address 2B
Processor Address 03

Interrupt Stack Pointer Register (ISP)
ID Address 2C
Processor Address 04

First Part Done Address Register
ID Address 2D
Processor Address —
Appendix E

D Save Register
ID Address 2E
Processor Address —

Q Save Register
ID Address 2F
Processor Address —

Temp 0 to Temp 9 Registers
ID Address (30 to 39)
Processor Address —

Process Control Block Base Register (PCBB)
ID Address 3A
Processor Address 10

System Control Block Base Register (SCBB)
ID Address 3B
Processor Address 11

PHYSICAL ADDRESS OF PROCESS CONTROL BLOCK

PHYSICAL PAGE ADDRESS OF THE SYSTEM CONTROL BLOCK
Appendix E

P0 Length Register (P0LR)
ID Address 3C
Processor Address 09

P1 Length Register (P1LR)
ID Address 3D
Processor Address 0B

System Length Register (SLR)
ID Address 3E
Processor Address 0D

Reserved Register
ID Address 3F
Processor Address —
APPENDIX F

VAX-11/750 INTERNAL PROCESSOR REGISTERS

Kernel Stack Pointer Register (KSP)
Processor Address 00

Executive Stack Pointer Register (ESP)
Processor Address 01

Supervisor Stack Pointer Register (SSP)
Processor Address 02

User Stack Pointer Register (USP)
Processor Address 03

Interrupt Stack Pointer Register (ISP)
Processor Address 04
Appendix F

P0 Base Register (P0BR)
Processor Address 08

P0 Length Register (P0LR)
Processor Address 09

P1 Base Register (P1BR)
Processor Address 0A

P1 Length Register (P1LR)
Processor Address 0B
Appendix F

System Base Register (SBR)
Processor Address 0C

System Length Register (SLR)
Processor Address 0D

Process Control Block Base Register (PCBB)
Processor Address 10

System Control Block Base Register (SCBB)
Processor Address 11
Appendix F

Processor Status Longword Register (PSL)
Interrupt Priority Level (IPL) Bits <16:20>
Processor Address 12

Processor Address 13 (Accesses AST level bits <2:0>) (ASTR)
Processor Address 3D (Accesses Performance Monitor Enable bit <3>) (PMR)

Software Interrupt Request Register (SIRR)
Processor Address 14

502
Appendix F

Software Interrupt Register (SISR)
Processor Address 15

[Diagram of SISR]

Machine Check Status Register (MCSR)
Processor Address 17

[Diagram of MCSR]

Interval Clock Control/Status Register (ICCS)
Processor Address 18

[Diagram of ICCS]

Next Interval Count Register (NICR)
Processor Address 19

[Diagram of NICR]
Appendix F

Interval Counter Register (ICR)
Processor Address 1A

Time of Day Register (TODR)
Processor Address 1B

Console Storage Receive Status Register (CSRS)
Processor Address 1C

Console Storage Receive Data Register (CSRDR)
Processor Address 1D
Appendix F

Console Storage Transmit Status Register (CSTS)
Processor Address 1E

![Diagram of CSTS Register]

Console Storage Transit Data Register (CSTD)
Processor Address 1F

![Diagram of CSTD Register]

Console Receive Control/Status Register (RXCS)
Processor Address 20

![Diagram of RXCS Register]

Console Receive Data Buffer Register (RXDB)
Processor Address 21

![Diagram of RXDB Register]
Appendix F

Console Transmit Control/Status Register (TXCS)
Processor Address 22

![Diagram of TXCS Register]

Console Transmit Data Buffer Register (TXDB)
Processor Address 23

![Diagram of TXDB Register]

Translation Buffer Disable Register (TBDR)
Processor Address 24

![Diagram of TBDR Register]

Cache Disable Register (CADR)
Processor Address 25

![Diagram of CADR Register]
Appendix F

Machine Check Error Summary Register (MCESR)
Processor Address 26

Cache Error Register (CAER)
Processor Address 27

Initialize UNIBUS Register
Processor Address 37

Memory Management Enable Register (MME)
Processor Address 38
Appendix F

Translation Buffer Invalidate All Register (TBIA)
Processor Address 39

Translation Buffer Invalidate Single Register (TBIS)
Processor Address 3A

Translation Buffer Register (TB)
Processor Address 3B

System Identification Register (SID)
Processor Address 3E

508
OPERAND SPECIFIERS
Operand specifiers are described in the following way:

<name><access type><data type>

where:

Name is a suggestive name for the operand in the context of the instruction. The name is often abbreviated.

Access type is a letter denoting the operand specifier access type:

a Calculate the effective address of the specified operand. Address is returned in a longword which is the actual instruction operand. Context of address calculation is given by <data type>.

b No operand reference. Operand specifier is a branch displacement. Size of branch displacement is given by <data type>.

m Operand is read, potentially modified and written. Note that this is NOT an indivisible memory operation. Also note that if the operand is not actually modified, it may not be written back. However, modify type operands are always checked for both read and write accessibility.

r Operand is read-only.

v Calculate the effective address of the specified operand. If the effective address is in memory, the address is returned in a longword which is the actual instruction operand. Context of address calculation is given by <data type>.

If the effective address is R[n], then the operand actually appears in R[n], or in R[n+1]’R[n].

w Operand is written only.

Data type is a letter denoting the data type of the operand:

b byte
d D_floating
f F_floating
Appendix G

\[ g \] G\_floating
\[ h \] H\_floating
\[ l \] longword
\[ q \] quadword
\[ w \] word
\[ x \] first data type specified by instruction
\[ y \] second data type specified by instruction

OPERATION DESCRIPTION NOTATION

The operation of each instruction is given as a sequence of control and assignment statements in an ALGOL-like syntax. No attempt is made to define the syntax formally; it is assumed to be familiar to the reader.

\[ + \] addition
\[ - \] subtraction, unary minus
\[ \ast \] multiplication
\[ / \] division (quotient only)
\[ ** \] exponentiation
\[ ' \] concatenation
\[ \leftarrow \] is replaced by
\[ = \] is defined as

\[ R_n \text{ or } R[n] \] contents of register \( R_n \)
\[ PC, SP, FP, \text{ or } AP \] the contents of register \( R_{15}, R_{14}, R_{13}, \text{ or } R_{12} \) respectively
\[ PSW \] the contents of the processor status word
\[ PSL \] the contents of the processor status longword
\[ (x) \] contents of memory location whose address is \( x \)
\[ (x)+ \] contents of memory location whose address is \( x \); \( x \) incremented by the size of operand referenced at \( x \)
\[ -(x) \] \( x \) decremented by size of operand to be referenced at \( x \); contents of memory location whose address is \( x \)
\[ <x:y> \] a modifier which delimits an extent from bit position \( x \) to bit position \( y \) inclusive
\[ <x_1,x_2,...,x_n> \] a modifier which enumerates bits \( x_1,x_2,...,x_n \)
\[ x...y \] \( x \) through \( y \) inclusive

510
Appendix G

{} arithmetic parentheses used to indicate precedence
AND logical AND
OR logical OR
XOR logical XOR
NOT logical (1's) complement
LSS less than signed
LSSU less than unsigned
LEQ less than or equal signed
LEQU less than or equal unsigned
EQL equal signed
EQLU equal unsigned
NEQ not equal signed
NEQU not equal unsigned
GEQ greater than or equal signed
GEQU greater than or equal unsigned
GTR greater than signed
GTRU greater than unsigned
SEXT (x) is sign-extended to size of operand needed
ZEXT (x) is zero-extended to size of operand needed
REM (x, y) remainder of x divided by y
MINU (x, y) minimum unsigned of x and y

The following conventions are used:

- Other than that caused by ( ) +, or −( ), and the advancement of PC, only operands or portions of operands appearing on the left side of assignment statements are affected.
- No operator precedence is assumed, other than that replacement (↔) has the lowest precedence. Precedence is indicated explicitly by { }.
- All arithmetic, logical, and relational operators are defined in the context of their operand. For example "+" applied to floating operands means a floating add while "+" applied to byte operands is an integer byte add. Similarly, "LSS" is a floating comparison when applied to floating operands while "LSS" is an integer byte comparison when applied to byte operands.
Appendix G

- Instruction operands are evaluated according to the operand specifier conventions. The order in which operands appear in the instruction description has no effect on the order of evaluation.
- Condition codes are in general affected on the value of actual stored results, not on "true" results (which might be generated internally to greater precision). Thus, for example, two positive integers can be added together and the sum stored, because of overflow, as a negative value. The condition codes will indicate a negative value even though the "true" result is clearly positive.
APPENDIX H
I/O SPACE RESTRICTIONS

A subset of native mode instructions is not used to reference I/O space. The reasons are:

1. String instructions are restartable via PSL<FPD>.
2. The PC, SP, or PCBB cannot point to I/O space.
3. I/O space does not support operand types of quad, F_floating, D_floating, G_floating, H_floating, field, or queue; nor can the position, size, length, or base of them be from I/O space.
4. The instruction may be interruptable because it is potentially a slow instruction in some implementations.
5. Only instructions with a maximum of one modify or write destination can be used. The destination must be the last operand.

For any memory reference to I/O space, the programmer must use an instruction from the following lists and must ensure that no interrupts or faults will occur, including page faults, after the first I/O space reference. To ensure no interrupts, the programmer must avoid operand specifier addressing modes 9, 11, 13, and 15, and these modes indexed. (Symbolically, these are @(Rn)+, @B↑D(Rn), @W↑D(Rn), and @L↑D(Rn), and these indexed.) The hardware may allow interrupts for these modes in order to minimize interrupt latency. For the instructions in the following lists, the hardware ensures that no other interrupts will occur after the first I/O space access.

Since these instructions are not interruptable after I/O space accesses (except for the addressing modes above), their execution will extend the interrupt latency. The programmer should make some effort to keep them short by minimizing the number of memory references. Use R0 through R13 instead, for example.

Instructions for which any explicit operands can be in I/O space are:

MOV[B,W,L]   PUSHL
ADD[B,W,L]3   ADAWI
INC[B,W,L]   ADWC
DEC[B,W,L]   SBWC
Appendix H

BIS{B,W,L}3
BIC{B,W,L}2
BIC{B,W,L}3
XOR{B,W,L}2
XOR{B,W,L}3
MOVA{B,W,L}
MOVAQ
PUSHA{B,W,L}
PUSHAQ
CASE{B,W,L}
MOVPSEL
BISPSW
BICPSW
CHM{K,E,S,U}
PROBE{R,W}
MTPR, MFPR

Instructions for which all operands except the branch displacement can be in I/O space are:

BLB {S,C}

Instructions for which some operands can be in I/O space are:

XFC        (depending on implementation)
REMQUE      addr (destination)

In spite of the above rules, it is possible for a specific hardware implementation to execute macro code from the I/O space and/or to allow the stack or PCB to be in I/O space. This might be used as part of the bootstrap process, for example. If this is done, then it is valid for software to transfer to this code.
APPENDIX I

TECHNICAL SPECIFICATIONS FOR VAX-11/750 PROCESSOR

Processor Type
- Microprogrammed 80-bit control store word

Micro-control store instruction time
- 320 nanoseconds

Control store size
- 6K words (80-bit words), read-only memory

Internal data path
- 32 bits

Maximum system I/O rate
- 5 Mb/s

Instruction buffer size
- 8-byte lookahead

CPU Cache Memory

Size
- 4 Kbytes direct mapped

Effective main memory cycle time
- 400 nanoseconds/32 bits

Typical hit ratio
- 90%

Typical cache cycle time
- 320 nanoseconds

CPU Address Translation Buffer

Size
- 512 address translations

Typical hit ratio
- 98-99%

CPU Clocks

Realtime clock
- Crystal controlled, 01% accuracy
  1 microsecond resolution

Time-of-year clock
- Includes recharging battery backup for over 100 hours

VAX Instruction Set
- 16 32-bit registers
- 248 basic operations
- 56 optional instructions
- 32 priority interrupt levels
- PDP-11 compatibility mode instructions.
Appendix I

Multiple data types: Integer, floating point, packed decimal, character string, variable bit fields, numeric strings, queues

Addressing modes: 9

Other Standard Features
- Power-fail automatic restart
- Hardware bootstrap load for up to four different devices
- Single serial line ASCII console interface
- Eight-line communications multiplexer (DZ11-A)
- TU58 cartridge tape unit
- Microverify diagnostic run automatically on power-up
- Extensive reliability and maintainability features
- Virtual console commands from DECwriter terminal

Main Memory
Virtual address space: 4 billion bytes
Physical address lines: 16 megabytes (24 bits)
Physical expansion: 2 megabytes in 256 Kbyte increments
Parity: 7-bit error correcting code (ECC) per 32-bit long-word
Technology: 16K-bit dynamic RAMs (200 nanosecond access time)
Cycle times: 800 nanoseconds per 32-bit read, 640 nanoseconds per 32-bit write
Power failure protection: Optional battery backup

I/O UNIBUS Adapter
Maximum UNIBUS I/O rate: 1.5 Mb/s through buffered data paths
Buffered data paths: 3 total, 4-byte buffer in each
Interrupts: Directly vectored
Appendix I

VAX-11/750 OPTIONS

MASSBUS Adapters (up to 3 per VAX-11/750)
Maximum bandwidth 2.0 Mb/s per MBA (to a system maximum of 5.0 Mb/s)
Devices Up to eight high speed disks or tapes
Buffer size 32 bytes per MBA

User Control Store
Size 1 Kword (80-bit words)
Technology Writeable memory (RAM), 70 nanosecond access time

Memory Battery Backup
Minimum 10 minutes with 2 Mb of MS750-AA backup time

Mechanical (VAX-11/750-AA/AB cabinet as installed)
Weight 182 kg (400 lbs.) Max.
Height 106.2 cm (41.8 in.)
Width 73.7 cm (29.0 in.)
Depth 76.3 cm (30.0 in.)

Electrical Power Requirements

Maximum VAX-11/750-AA VAX-11/750-AB
AC line voltage tolerance 90-128V 180-256V
Frequency tolerance 47-63Hz 47-63Hz
Phases 1 1
Steady state current 25 @ 90 Vrms 12.5 @ 180 Vrms
Surge current 100A 100A
Surge duration (exponential decay) 8 cycles 8 cycles
Plug Type L5-30P (Varies by country)

AC cable length 3 m (9.84 ft.)
Maximum heat dissipation 1460 kcal/hr (5800 BTU/Hr)
Maximum AC power consumption 1700 watts
Appendix I

Nominal voltage: 120 volts at 18 amps

**Typical AC Utilization by Option***

<table>
<thead>
<tr>
<th>Module Description</th>
<th>AC Utilization (Volt Amps)</th>
<th>AC Utilization (Watts)</th>
<th>Heat Dissipation (BTU/Hour)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic CPU, 256 Kb TU58, DZ11-A</td>
<td>725</td>
<td>505</td>
<td>1723</td>
</tr>
<tr>
<td>Each additional 256 Kb memory card</td>
<td>19</td>
<td>14</td>
<td>47</td>
</tr>
<tr>
<td>Memory battery backup</td>
<td>137</td>
<td>50</td>
<td>40</td>
</tr>
<tr>
<td>Remote diagnosis option</td>
<td>328</td>
<td>137</td>
<td>96</td>
</tr>
<tr>
<td>Each MASSBUS adapter</td>
<td>407</td>
<td>180</td>
<td>125</td>
</tr>
</tbody>
</table>

* UNIBUS peripheral controllers plugged into the remaining eight slots in the VAX-11/750 DD11-DK backplane must also be included for an accurate total.

**Environment***

Operating:
- Radiated acoustic noise level: 64dB at 1 Meter (3.3 feet) in front at 1.5 meters height
- Temperature: 10° to 40° C (50° to 104° F)
- Relative humidity: 10 to 90%
- Maximum altitude: 2.4 km (8000 ft.)

Storage:
- Temperature: -40° to 66° C (-40° to 151° F)
- Relative humidity: 10 to 95%

**Expansion Slots**

- UNIBUS: 2 quad slots, 6 hex slots (in addition to DZ11-A), Second distribution box for DZ11 lines
- CPU Options: 7 for memory array cards (256 Kb)
Appendix I

1 each for User Control Store, Remote Diagnosis Option, and general I/O adaptor slots for up to 3 MASSBUS adaptors

Max. UNIBUS D.C. Power in VAX-11/750 Cabinet
+ 15 Volts 2 amps
− 15 Volts 3.5 amps
+ 5 Volts 25.0 amps

* Environmental and power figures are given for the VAX-11/750-AA/AB only. Consult DIGITAL Field Service for the recommended value for an entire system.

Specification for TU58 in VAX-11/750 Console
Capacity per cartridge 256 Kbytes
(512 records by 512 bytes)

Capacity per track 128 Kbytes
Tape length 140 feet
Data transfer rate 1.92 Kbytes/s
Average access time 9.3 seconds
Maximum access time 28 seconds
Read/write tape speed 30 in/s
Search tape speed 60 in/s
Bit density 800 bpi
Number of passes per cartridge to read/write cartridge 4 (2 tracks interleaved)
Best case read/write time for entire cartridge 10 minutes

Custom Technology Specifications for the VAX-11/750
Implementation technique Gate arrays
Circuit technology Low power bipolar Schottky
Circuit density Large scale integration (LSI)
Die size .215 inches × .244 inches
Power utilized per die 2 watts maximum
Package size 144 sq. in. (2.4 inches × 0.6 inches)
Number of pins/package 48
Appendix I

I/O circuits/die
Logic gates
Voltages used
Speed per gate

44 I/O transceiver gates
400 identical 4-input NAND gates
+2.5 volts, +5 volts
5-10 nanoseconds

Front View of VAX-11/750 System Cabinet
Rear View of VAX-11/750 System Cabinet
## TECHNICAL SPECIFICATIONS FOR VAX-11/780 PROCESSOR

<table>
<thead>
<tr>
<th>Description</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Type</td>
<td>Microprogrammed, 99-bit control store word</td>
</tr>
<tr>
<td></td>
<td>Microcontrol store instruction time—200 nanoseconds</td>
</tr>
<tr>
<td></td>
<td>Control store size—5K words (99-bit words), 4K words</td>
</tr>
<tr>
<td></td>
<td>WDCS</td>
</tr>
<tr>
<td>Internal data path</td>
<td>32 bits</td>
</tr>
<tr>
<td><strong>CPU Cache Memory</strong></td>
<td></td>
</tr>
<tr>
<td>Size</td>
<td>8K bytes, 2-way set associative</td>
</tr>
<tr>
<td>Effective main memory cycle time</td>
<td>1800 nanoseconds/64 bits</td>
</tr>
<tr>
<td>Typical hit ratio</td>
<td>95%</td>
</tr>
<tr>
<td>Typical cache cycle time</td>
<td>290 nanoseconds</td>
</tr>
<tr>
<td><strong>CPU Address Translation Buffer</strong></td>
<td></td>
</tr>
<tr>
<td>Size</td>
<td>128 address translations</td>
</tr>
<tr>
<td>Typical hit ratio</td>
<td>97%</td>
</tr>
<tr>
<td><strong>CPU Clocks</strong></td>
<td></td>
</tr>
<tr>
<td>Real-time clock</td>
<td>Crystal controlled, .01% accuracy</td>
</tr>
<tr>
<td></td>
<td>1-microsecond resolution</td>
</tr>
<tr>
<td>Time-of-year clock</td>
<td>Includes recharging battery backup for over 100 hours</td>
</tr>
<tr>
<td><strong>VAX Instruction Set</strong></td>
<td></td>
</tr>
<tr>
<td>16 32-bit registers</td>
<td></td>
</tr>
<tr>
<td>248 basic operations</td>
<td></td>
</tr>
<tr>
<td>32 priority interrupt levels</td>
<td></td>
</tr>
<tr>
<td>Multiple data types</td>
<td>Integer, floating point, packed decimal, character string, variable bit fields, and numeric strings</td>
</tr>
<tr>
<td></td>
<td>PDP-11 compatibility mode instructions</td>
</tr>
<tr>
<td>Addressing modes</td>
<td>9</td>
</tr>
</tbody>
</table>
Other standard features

- Power fail/automatic restart
- Single serial line ASCII console interface
- 8-line communications multiplexer (DZ11-A)
- RX01 floppy disk drive
- Writable diagnostic control store (WDCS)
- Extensive reliability and maintainability features
- Virtual console commands from DECwriter terminal

Main Memory

<table>
<thead>
<tr>
<th>Feature</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual address space</td>
<td>4 billion bytes</td>
</tr>
<tr>
<td>Physical address lines</td>
<td>1 billion bytes (30 bits)</td>
</tr>
<tr>
<td>Physical expansion</td>
<td>8 megabytes in 256K-byte increments</td>
</tr>
<tr>
<td>Parity</td>
<td>8-bit error correcting code (ECC) per 64-bit quadword</td>
</tr>
<tr>
<td>Technology</td>
<td>16K-bit dynamic RAMs (200 nanosecond access time)</td>
</tr>
<tr>
<td>Cycle times</td>
<td>800 nanoseconds per 64-bit read (1300 nanoseconds with single-bit errors)</td>
</tr>
<tr>
<td></td>
<td>1400 nanoseconds per 64-bit write</td>
</tr>
<tr>
<td>Power failure protection</td>
<td>Optional battery backup</td>
</tr>
</tbody>
</table>

I/O UNIBUS Adapter (1 standard, up to 4 total)

- Maximum UNIBUS I/O rate: 1.5 Mb/sec through buffered data paths
- Buffered data paths: 15 total, 8-byte buffer in each
- Maximum number of bus loads: 18 without a repeater
- Interrupts: Directly vectored via UNIBUS adapter

VAX-11/780 Options MASSBUS Adapters (up to 4)

- Devices: Up to 8 high-speed disks or tapes
- Buffer size: 32 bytes per MBA
Appendix J

User Control Store
Size 1K word (99-bit words)
Technology Writable memory (RAM), 70 nanosecond access time

Memory Battery Backup
Minimum backup time 10 minutes with 4 Mb of MS780-D memory

Floating Point Accelerator
Enhances performance of all floating point instructions (single & double precision) including polynomial evaluation, integer/floating conversions, 8-, 16-, and 32-bit integer multiply.

Mechanical (VAX-11/780 cabinet as installed)
Weight 498 kg (1100 lbs.)
Height 153.7 cm (60.5 in.)
Width 118.1 cm (46.5 in.)
Depth 76.2 cm (30 in.)

Electrical Power Requirements
AC line voltage 120/208V
Frequency tolerance 59-61 Hz
Phases 3 phase
  phase A: 11.2 A max. continous
  phase B: 9.9 A max. continous
  phase C: 13.1 A max. continous
  neutral: 14.4 A max. continous
Also available 220/380V 50 Hz and 240/415V 50 Hz
Plug type L5-30P (varies by country)
AC cable length 3 m (9.84 ft.) from back of cabinet
Maximum heat dissipation 5350 kcal/hr. (21,230 BTU/hr.)
Maximum ac power consumption 6225 watts

525
### Appendix J

#### Typical Power Requirements and Thermal Dissipation

<table>
<thead>
<tr>
<th>Option</th>
<th>Watts</th>
<th>BTU/hr.</th>
<th>kcal/hr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fully-configured CPU cabinet</td>
<td>6,225</td>
<td>21,230</td>
<td>5,350</td>
</tr>
<tr>
<td>Fully-configured CPU expansion cabinet, H9602-HA(HB)</td>
<td>2,000</td>
<td>6,820</td>
<td>1,720</td>
</tr>
<tr>
<td>Fully-configured UNIBUS options cabinet H9602-DF(DH)</td>
<td>2,000</td>
<td>6,820</td>
<td>1,720</td>
</tr>
<tr>
<td>MBA</td>
<td>150</td>
<td>512</td>
<td>130</td>
</tr>
<tr>
<td>UBA, DW780-AA(AB)</td>
<td>300</td>
<td>1024</td>
<td>260</td>
</tr>
<tr>
<td>512 Kb memory, control &amp; power supply</td>
<td>350</td>
<td>1,195</td>
<td>302</td>
</tr>
<tr>
<td>Fully configured multiport memory (2 MA780 subsystems, 4 Mb Memory, 8 ports)</td>
<td>1800</td>
<td>6140</td>
<td>1550</td>
</tr>
<tr>
<td>Multiport memory with 1 MA780 subsystem, 2 Mb memory, 4 ports</td>
<td>1000</td>
<td>3410</td>
<td>860</td>
</tr>
<tr>
<td>DR780</td>
<td>226</td>
<td>771</td>
<td>194</td>
</tr>
<tr>
<td>FP780-AA(AB)</td>
<td>300</td>
<td>1,025</td>
<td>260</td>
</tr>
<tr>
<td>KU780 (WCS)</td>
<td>100</td>
<td>341</td>
<td>86</td>
</tr>
<tr>
<td>H7112-A(B)</td>
<td>25</td>
<td>85</td>
<td>22</td>
</tr>
</tbody>
</table>

#### NOTE

UNIBUS peripheral controllers plugged into the remaining eight slots in the VAX-11/780 DD11-DK backplane must also be included for an accurate total.
Appendix J

Environment
Operating:
Radiated acoustic noise level (at 1 meter height) front: 74 dB rear: 65 dB
Temperature 15° to 32°C (59° to 90°F)
Relative humidity 20% to 80%
Maximum altitude 2.4 km (8000 ft.)
Nonoperating:
Temperature −40° to 66°C (−40° to 151°F)
Relative humidity 0 to 95%

VAX-11/780 Cabinet Space Allocation
APPENDIX K
SYSTEM THROUGHPUT CONSIDERATIONS

For the majority of applications, the standard VAX packaged systems provide more than ample I/O bus capacity. And system throughput is determined by the computing requirements and the time waiting for the disk to seek or spin. This appendix is intended to provide guidance in configuring those remaining applications that are characterized by very intensive I/O.

Bus and Memory Bandwidths
The bandwidth or speed capacity of the SBI is 13.3 million bytes per second. That is not to say that 13.3 Mb/s will be achieved in any specific configuration or application. In practice, the flow of data is usually much slower because of contention between devices, because of irregularities in the rate of requests, and because the devices may not be able to capitalize on available bandwidth. Often all of these conditions are true. The 13.3 Mb/s number could be called the protocol limit to the bus bandwidth.

The focal point of traffic on the bus is the memory subsystem, since most transfers are between a memory and one of the other types of connections, either the CPU or an I/O bus adapter. The memory is 72 bits wide (64 bits of data plus 8 ECC bits). This width contributes to the capacity of a fully configured large system to handle an enormous amount of I/O traffic, inasmuch as a single memory can accept a 64-bit write every 0.8 μs. However, it takes 1.4 μs to handle writes of less than 64 bits since the memory must do an internal update rather than a simple overwrite.

Of the other connections to the SBI, the CPU represents the most intensive traffic load on the memory subsystem unless the configuration includes a DR780. The CPU can and often will request bursts of repeated 32-bit writes to memory every 1.2 μs and less often request short bursts of 64-bit reads from memory. Although the processor in computing will request data much more often than it will generate data, the effect of the very large, write-through cache is to reverse these statistics as seen by the memory subsystem. The cache portion of the processor in typical applications will do 32-bit writes to memory more than ten times as often as it will do 64-bit reads. Frequently, these 32-bit writes will occur in bursts, as for example when the operating system clears a page of memory or when a COBOL application moves data into a buffer. This would mean that despite its low priority, the processor could easily dominate a single memory controller; if the
request buffer of the memory controller were enabled fully, the processor could have a write in process, four writes pending, and another write waiting to be requested. This would result in long delays as seen by the I/O bus adapters, and high-speed disk transfers would be marginal at best. For this reason, the memory request buffer is by design limited to a depth of one command unless interleaving is enabled. Even with the request buffer limited, the processor presents sufficient contention to limit the effective MASSBUS speed capacity in single memory configurations to about 1.3 to 1.5 Mb/s, a figure below the MASSBUS protocol bandwidth but sufficient to support disk products.

If there is more than one MASSBUS channel transferring data concurrently, then interchannel contention for a single memory controller can contribute to data-lates and overruns. Except in certain time-critical or data acquisition applications, an occasional data-late is harmless (VAX/VMS will simply retry the operation), but an excessive percentage of data-lates can interfere with productive work. A system based on interleaved memory will almost never experience even a harmless data-late.

MASSBUS Arrangements
The VAX/VMS system offers flexibility in physically configuring the arrangement of disks and tapes over one or more MASSBUS channels. Typically tapes are installed on one channel, and all disks are on another, even if there are several disks. This is not always the best arrangement; sometimes there should be disks on the tape channel; sometimes extra channels should be included even if the total number of disks is below the maximum allowed on one channel. The best arrangement depends on the objectives of the installation and the characteristics of the application.

MASSBUS products support overlapped control operations. Once a control function such as a disk seek or a tape rewind has been initiated, the MASSBUS is free to start another function, control or transfer. This is supported in VAX/VMS, so overlapped seeking, for example, happens whether the disks are distributed over multiple channels or concentrated on one. However, once a data transfer is physically started, then it must complete or abort before another data transfer operation on that same channel can be started or a seek completion can be recognized. It is for this reason that tape drives are advisable on a disk channel only if use of the two is essentially mutually exclusive. Except for memory contention, the MASSBUS channels are quite independent, and VAX/VMS supports concurrent operations on multiple buses.
Appendix K

Tape drives connect to a MASSBUS differently than do disks. Disks interface to a MASSBUS directly through interface electronics associated with and replicated in each individual disk. Tape drives are cabled to a common interface called a formatter which in turn connects to the MASSBUS. The formatter is included as one of the components of a master tape drive. Typically all of the tapes on a system share, and contend for, the services of the same formatter. They need not, and this is a consideration for applications with intense tape traffic.

Because VAX/VMS supports concurrent multiple bus traffic, the I/O capacity of a fully configured VAX can be very large, but only if the memory subsystem is matched to the I/O subsystem with consideration for the processor as an essential element. If the memory subsystem is the single controller minimum, then the restriction that transfers on the MASSBUS are mutually exclusive can be a beneficial restriction in that the software cannot initiate a transfer until the previous one is complete. Unproductive contention for the single memory can be partially or completely avoided by putting the application-required number of disks on the physically minimum possible number of buses. If contention would still be a problem, there is no sensible alternative other than a second memory controller. The immediate need is to predict the threat of unproductive memory contention; the second need is to estimate the impact of the resulting data-lates and overruns.

Unproductive memory contention will occur if the aggregate set of DMA buses cannot all together sustain their individual peak instantaneous rates. The emphasis is on the peak rates, not the average rates; the ratio of peak to average determines the frequency of contention and the impact but not the eventual certainty. As a guideline, the maximum allowable peak rates for a single memory controller system are two channels at 1.3 Mb/s or three channels at 0.806 Mb/s.

Another general guideline to follow is: if transfers are relatively short and infrequent, then memory contention is not likely to be a problem because there will be adequate capacity for a retry. However, if the transfers are long and occur frequently, then unproductive memory contention will be much more bothersome because of the high likelihood that the retries will also fail.

Two memory controllers interleaved can easily support a full complement of I/O channels at full speed. By including the second memory and extra MASSBUS channels, the application can benefit by spreading the disk traffic over multiple channels. This benefit can be significant if the physical transfers are long and negligible if the transfers are typically only a few blocks. Swapping traffic and some data processing
applications are characterized by transfers of more than half a disk track. Transfers of only a few blocks tend to be dominated by the rotational latency.

Impact on the Processor
Whenever there is sufficiently intense I/O traffic, the effective speed of the processor is slowed by the contention for memory. This is rarely important, but if the success of the application is critically dependent on a prediction of execution time, then this effect should be considered. The major I/O parameter here is the total of the average I/O rates. In predicting the average I/O rate, remember that the relationship between peak and average depends upon the transfer length, the dead time between transfers, and any gaps within the transfer. With two memory controllers the processor will be slowed roughly 4% per averaged megabyte per second of I/O traffic; with only a single memory the impact is about two to four times greater.

For example, consider a small system with RP06 disk drives. The average I/O load consists of 15 transfers per second with each transfer to be 5 Kb long. Although the peak transfer rate of an RP06 is 0.806 Mb/s, the average rate would be 15 multiplied by 0.005, or 0.075 Mb/s. With the previously mentioned 4% per averaged Mb/s slowdown factor, the throughput slowdown in the system being considered here is 0.3%, or virtually nil. If, instead, the I/O load were a multibuffered disk-to-disk copy from one channel to another, then ignoring overhead and allowing 10 ms for an adjacent track seek, 7 ms for rotational latency after each seek, and 214,016 bytes to be transferred in the next 19 revolutions, the calculated average rate per disk is 0.64 Mb/s. If the seeks and transfers of the other disk were completely overlapped with the first, then the total average I/O rate during the copy would be 1.28 Mb/s, corresponding to a slowdown of 5% incorporating the 4% per Mb/s factor. This is still within the speed tolerance of the basic processor specification, but the example and methodology shown here should be helpful in applications with really intense I/O traffic.

Even in the absence of I/O traffic, the computing rate of a system with a single memory controller will be as much as 15% slower than an interleaved memory configuration. The slowdown depends on the frequency of consecutive writes to memory. In most cases it is imperceptible.

UNIBUS Configurations
The total throughput of any UNIBUS system is a function of both the bus master and the bus slave. The characteristic that comes into play in determining system throughput is the delay from the time MSYN is
Appendix K

received on the bus until SSYN is asserted for all of the different types of functions that the UNIBUS adapter can perform. The actual measured times on a system can vary as a function of other system activity. If the memory or the SBI is in use when the UNIBUS adapter makes a reference, then the UNIBUS adapter will be stalled until that activity is completed and the MSYN to SSYN time will be lengthened accordingly.

DMC-11 Configurations
The DMC-11 Network Link is a high performance interconnection of UNIBUS structured systems for use where the computers are located within the same facility. The DMC-11 can be configured for operation at up to 1 megabaud (1 million bits per second). The DMC-11 can also be used with non-UNIBUS computers, such as the VAX-11/780, through the UNIBUS adapter. However, certain care must be exercised in the operation.

In the case of the VAX-11/780, the DMC-11 cannot take advantage of the buffered data path, but must use the direct data path, which is slower. The resultant NPR rate for the DMC-11 is not fast enough to support the one megabit per second rate in both transmit and receive lines simultaneously. Therefore, simultaneous transmit and receive at one megabit can occur only as long as the silos in the DMC-11 provide some slack.

Whenever the transmit silo is empty and the receive silo is full, the transmit line will suffer underruns and experience CRC errors. The message rejected by the CRC error would be corrected on an automatic retry unless the line is so busy that the retry also fails.

The performance on the line is therefore dependent on the amount of line usage or the message length, and since the amount of usage is not usually controlled, the message length is the only parameter that can be used to prevent link problems from occurring. A message length of 100 bytes or fewer will not cause problems on a VAX-11/780 processor, but larger message lengths could cause problems when link usage is high. The guideline is therefore to keep message lengths below 100 bytes at the one megabaud rate.

This throughput issue is a major concern when the DMC-11 is configured in VAX-11/780 systems. The use of DECnet software and its inherent protocol costs will allow the DMC-11 to operate at one megabit speeds, although the average output will be much lower.

KMC-11 Configurations
The KMC-11 is an auxiliary processor, complete with memory, that can be used to off-load from the CPU such tasks as data communica-
tions, and analog input/output. The KMC-11 operates in parallel with the main CPU, or in the case of VAX systems, in parallel with the UNIBUS adapter.

The KMC-11 interface to the UNIBUS is directly controlled by the microcode in the KMC-11. Through this microcode, it is possible to gain control of the UNIBUS, hold it, and disrupt the normal operation of other units, including the VAX processor. It is important that VAX systems incorporating KMC-11s under the direction of user-developed software do not hold the UNIBUS for longer than 20 microseconds.

The KMC-11 can affect UNIBUS throughput in another manner also. As an auxiliary processor, the KMC-11 is continuously asserting itself as bus master to determine if there is work for it to do, such as processing data from a DZ11 multiplexer. If the rate at which the KMC-11 asserts its bus mastership is great enough, UNIBUS throughput can be significantly reduced. Careful use of the KMC-11, or the use of additional UNIBUS adapters, can help avoid this situation.

Three reference sources useful to those interfacing devices to VAX systems through the UNIBUS adapter are the PDP-11 UNIBUS Design Description shown in the PDP-11 UNIBUS Handbook, Order No. EB-17525-20, DW780 UNIBUS Adapter Technical Description, Order No. EK-DW780-TD, and the I/O User's Guide, VMS Documentation Set, Volume III.

Impact of the DR780
VAX systems which incorporate a DR780 high performance interconnect also have configuration and application dependent throughput considerations. A detailed discussion of these issues and a chart showing typical throughput rates as a function of system configuration and workload can be found in Chapter 20. The DR780 General Purpose Interface Technical Description, Order No. EK-DR780-TD and the DR780 User's Guide, Order No. EK-DR780-UG are also available reference sources.

Impact of the MA780
VAX systems using the MA780 multiport memory option also have unique throughput considerations depending on the total system configuration and application. A description of the MA780 can be found in Chapter 20 of this handbook and a detailed technical description of its operation is available in the MA780 Multiport Memory Technical Description, Order No. EK-MA780-TD.
GLOSSARY

abort  An exception that occurs in the middle of an instruction and potentially leaves the registers and memory in an indeterminate state, such that the instruction cannot necessarily be restarted.

absolute indexed mode  An indexed addressing mode in which the base operand specifier is addressed in absolute mode.

absolute mode  In absolute mode addressing, the PC is used as the register in autoincrement deferred mode. The PC contains the address of the location containing the actual operand.

absolute time  Time values expressing a specific date (month, day, and year) and time of day. Absolute time values are always expressed in the system as positive numbers.

access mode  1. Any of the four processor access modes in which software executes. Processor access modes are, in order from most to least privileged and protected: kernel (mode 0), executive (mode 1), supervisor (mode 2), and user (mode 3). When the processor is in kernel mode, the executing software has complete control of, and responsibility for, the system. When the processor is in any other mode, the processor is inhibited from executing privileged instructions. The Processor Status Longword contains the current access mode field. The operating system uses access modes to define protection levels for software executing in the context of a process. For example, the Executive runs in kernel and executive mode and is most protected. The command interpreter is less protected and runs in supervisor mode. The debugger runs in user mode and is no more protected than normal user programs. 2. See record access mode.

access type  1. The way in which the processor accesses instruction operands. Access types are: read, write, modify, address, and branch. 2. The way in which a procedure accesses its arguments. 3. See also record access type.

access violation  An attempt to reference an address that is not mapped into virtual memory or an attempt to reference an address that is not accessible by the current access mode.

account name  A string that identifies a particular account used to accumulate data on a job's resource use. This name is the user's accounting charge number, not the user's UIC.

address  A number used by the operating system and user software to identify a storage location. See also virtual address and physical address.
address access type  The specified operand of an instruction is not directly accessed by the instruction. The address of the specified operand is the actual instruction operand. The context of the address calculation is given by the data type of the operand.

addressing mode  The way in which an operand is specified; for example, the way in which the effective address of an instruction operand is calculated using the general registers. The basic general register addressing modes are called: register, register deferred, autoincrement, autoincrement deferred, autodecrement, displacement, and displacement deferred. In addition, there are six indexed addressing modes using two general registers, and literal mode addressing. The PC addressing modes are called: immediate (for register deferred mode using the PC), absolute (for autoincrement deferred mode using the PC), and branch.

address space  The set of all possible addresses available to a process. Virtual address space refers to the set of all possible virtual addresses. Physical address space refers to the set of all possible physical addresses sent out on the SBI.

allocate a device  To reserve a particular device unit for exclusive use. A user process can allocate a device only when that device is not allocated by any other process.

alphanumeric character  An upper or lower case letter (A-Z, a-z), a dollar sign ($), an underscore (_), or a decimal digit (0-9).

alternate key  An optional key within the data records in an indexed file; used by VAX-11 RMS to build an alternate index. See key (indexed files) and primary key.

American Standard Code for Information Interchange (ASCII)  A set of 8-bit binary numbers representing the alphabet, punctuation, numerals, and other special symbols used in text representation and communications protocol.

Ancillary Control Process (ACP)  A process that acts as an interface between user software and an I/O driver. An ACP provides functions supplemental to those performed in the driver, such as file and directory management. Three examples of ACPs are: the Files-11 ACP (F11ACP), the magnetic tape ACP (MTACP), and the networks ACP (NETACP).

area  Areas are VAX-11 RMS maintained regions of an indexed file. They allow a user to specify placement and/or specific bucket sizes for particular portions of a file. An area consists of any number of buckets, and there may be from 1 to 255 areas in a file.
Glossary

**Argument Pointer**  General register 12 (R12). By convention, AP contains the address of the base of the argument list for procedures initiated using the CALL instructions.

**assign a channel**  To establish the necessary software linkage between a user process and a device unit before a user process can transfer any data to or from that device. A user process requests the system to assign a channel and the system returns a channel number.

**asynchronous record operation**  A mode of record processing in which a user program can continue to execute after issuing a record retrieval or storage request without having to wait for the request to be fulfilled.

**Asynchronous System Trap**  A software-simulated interrupt to a user-defined service routine. ASTs enable a user process to be notified asynchronously with respect to its execution of the occurrence of a specific event. If a user process has defined an AST routine for an event, the system interrupts the process and executes the AST routine when that event occurs. When the AST routine exits, the system resumes the process at the point where it was interrupted.

**Asynchronous System Trap level (ASTLVL)**  A value kept in an internal processor register that is the highest access mode for which an AST is pending. The AST does not occur until the current access mode drops in priority (raises in numeric value) to a value greater than or equal to ASTLVL. Thus, an AST for an access mode will not be serviced while the processor is executing in a higher priority access mode.

**authorization file**  See user authorization file.

**autodecrement indexed mode**  An indexed addressing mode in which the base operand specifier uses autodecrement mode addressing.

**autodecrement mode**  In autodecrement mode addressing, the contents of the selected register are decremented, and the result is used as the address of the actual operand for the instruction. The contents of the register are decremented according to the data type context of the register: 1 for byte, 2 for word, 4 for longword and floating, 8 for quadword and double floating.

**autoincrement deferred indexed mode**  An indexed addressing mode in which the base operand specifier uses autoincrement deferred mode addressing.

**autoincrement deferred mode**  In autoincrement deferred mode addressing, the specified register contains the address of a longword
Glossary

which contains the address of the actual operand. The contents of the register are incremented by 4 (the number of bytes in a longword). If the PC is used as the register, this mode is called absolute mode.

autoincrement indexed mode An indexed addressing mode in which the base operand specifier uses autoincrement mode addressing.

autoincrement mode In autoincrement mode addressing, the contents of the specified register are used as the address of the operand, then the contents of the register are incremented by the size of the operand.

balance set The set of all process working sets currently resident in physical memory. The processes whose working sets are in the balance set have memory requirements that balance with available memory. The balance set is maintained by the system swapper process.

base operand address The address of the base of a table or array referenced by index mode addressing.

base operand specifier The register used to calculate the base operand address of a table or array referenced by index mode addressing.

base priority The process priority that the system assigns a process when it is created. The scheduler never schedules a process below its base priority. The base priority can be modified only by the system manager or the process itself.

base register A general register used to contain the address of the first entry in a list, table, array, or other data structure.

binding See linking.

bit string See variable-length bit field.

block 1. The smallest addressable unit of data that the specified device can transfer in an I/O operation (512 contiguous bytes for most disk devices). 2. An arbitrary number of contiguous bytes used to store logically related status, control, or other processing information.

block I/O The set of VAX-11 RMS procedures that allow you direct access to the blocks of a file regardless of file organization.

bootstrap block A block in the index file on a system disk that contains a program that can load the operating system into memory and start its execution.

branch access type An instruction attribute which indicates that the processor does not reference an operand address, but that the oper-
and is a branch displacement. The size of the branch displacement is given by the data type of the operand.

**branch mode** In branch addressing mode, the instruction operand specifier is a signed byte or word displacement. The displacement is added to the contents of the updated PC (which is the address of the first byte beyond the displacement), and the result is the branch address.

**bucket** A storage structure, consisting of from 1 to 32 blocks, used for building and processing relative and indexed files. A bucket contains one or more records or record cells. Buckets are the unit of contiguous transfer between VAX-11 RMS buffers and the disk.

**buffered I/O** See system buffered I/O.

**bug check** The operating system's internal diagnostic check. The system logs the failure and crashes the system.

**byte** A byte is eight contiguous bits starting on an addressable byte boundary. Bits are numbered from the right, 0 through 7, with bit 0 the low-order bit. When interpreted arithmetically, a byte is a 2's complement integer with significance increasing from bits 0 through 6. Bit 7 is the sign bit. The value of the signed integer is in the range $-128$ to 127 decimal. When interpreted as an unsigned integer, significance increases from bits 0 through 7 and the value of the unsigned integer is in the range 0 to 255 decimal. A byte can be used to store one ASCII character.

**cache memory** A small, high-speed memory placed between slower main memory and the processor. A cache increases effective memory transfer rates and processor speed. It contains copies of data recently used by the processor, and fetches several bytes of data from memory in anticipation that the processor will access the next sequential series of bytes.

**call frame** See stack frame.

**call instructions** The processor instructions CALLG (Call Procedure with General Argument List) and CALLS (Call Procedure with Stack Argument List).

**call stack** The stack, and conventional stack structure, used during a procedure call. Each access mode of each process context has one call stack, and interrupt service context has one call stack.

**channel** A logical path connecting a user process to a physical device unit. A user process requests the operating system to assign a channel to a device so the process can transfer data to or from that device.
Glossary

character  A symbol represented by an ASCII code. See also alphanumeric character.

character string  A contiguous set of bytes. A character string is identified by two attributes: an address and a length. Its address is the address of the byte containing the first character of the string. Subsequent characters are stored in bytes of increasing addresses. The length is the number of characters in the string.

character string descriptor  A quadword data structure used for passing character data (strings). The first word of the quadword contains the length of the character string. The second word can contain type information. The remaining longword contains the address of the string.

cluster  1. A set of contiguous blocks that is the basic unit of space allocation on a Files-11 disk volume. 2. A set of pages brought into memory in one paging operation. 3. An event flag cluster.

command  An instruction, generally an English word, typed by the user at a terminal or included in a command file which requests the software monitoring a terminal or reading a command file to perform some well-defined activity. For example, typing the COPY command requests the system to copy the contents of one file into another file.

command file  A file containing command strings. See also command procedure.

command interpreter  Procedure-based system code that executes in supervisor mode in the context of a process to receive, syntax check, and parse commands typed by the user at a terminal or submitted in a command file.

command parameter  The positional operand of a command delimited by spaces, such as a file specification, option, or constant.

command procedure  A file containing commands and data that the command interpreter can accept in lieu of the user typing the commands individually on a terminal.

command string  A line (or set of continued lines), normally terminated by typing the carriage return key, containing a command and, optionally, information modifying the command. A complete command string consists of a command, its qualifiers, if any, and its parameters (file specifications, for example), if any, and their qualifiers, if any.

common  A FORTRAN term for a program section that contains only data.

common event flag cluster  A set of 32 event flags that enables
cooperating processes to post event notification to each other. Common event flag clusters are created as they are needed. A process can associate with up to two common event flag clusters.

**compatibility mode**  A mode of execution that enables the central processor to execute non-privileged PDP-11 instructions. The operating system supports compatibility mode execution by providing an RSX-11M programming environment for an RSX-11M task image. The operating system compatibility mode procedures reside in the control region of the process executing a compatibility mode image. The procedures intercept calls to the RSX-11M Executive and convert them to the appropriate operating system functions.

**condition**  An exception condition detected and declared by software. For example, see failure exception mode.

**condition codes**  Four bits in the Processor Status Word that indicate the results of previously executed instructions.

**condition handler**  A procedure that a process wants the system to execute when an exception condition occurs. When an exception condition occurs, the operating system searches for a condition handler and, if found, initiates the handler immediately. The condition handler may perform some action to change the situation that caused the exception condition and continue execution for the process that incurred the exception condition. Condition handlers execute in the context of the process at the access mode of the code that incurred the exception condition.

**condition value**  A 32-bit quantity that uniquely identifies an exception condition.

**context**  The environment of an activity. See also process context, hardware context, and software context.

**context indexing**  The ability to index through a data structure automatically because the size of the data type is known and used to determine the offset factor.

**context switching**  Interrupting the activity in progress and switching to another activity. Context switching occurs as one process after another is scheduled for execution. The operating system saves the interrupted process' hardware context in its hardware process control block (PCB) using the Save Process Context instruction, and loads another process' hardware PCB into the hardware context using the Load Process Context instruction, scheduling that process for execution.
continuation character  A hyphen at the end of a command line signifying that the command string continues on to the next command line.

console  The manual control unit integrated into the central processor. The console includes an LSI-11 microprocessor and a serial line interface connected to a hard copy terminal. It enables the operator to start and stop the system, monitor system operation, and run diagnostics.

console terminal  The hard copy terminal connected to the central processor console.

control region  The highest-addressed half of per-process space (the P1 region). Control region virtual addresses refer to the process-related information used by the system to control the process, such as: the kernel, executive, and supervisor stacks, the permanent I/O channels, exception vectors, and dynamically used system procedures (such as the command interpreter and RSX-11M programming environment compatibility mode procedures). The user stack is also normally found in the control region, although it can be relocated elsewhere.

Control Region Base Register (P1BR)  The processor register, or its equivalent in a hardware process control block, that contains the base virtual address of a process control region page table.

Control Region Length Register (P1LR)  The processor register, or its equivalent in a hardware process control block, that contains the number of non-existent page table entries for virtual pages in a process control region.

copy-on-reference  A method used in memory management for sharing data until a process accesses it, in which case it is copied before the access. Copy-on-reference allows sharing of the initial values of a global section whose pages have read/write access but contain pre-initialized data available to many processes.

counted string  A data structure consisting of a byte-sized length followed by the string. Although a counted string is not used as a procedure argument, it is a convenient representation in memory.

current access mode  The processor access mode of the currently executing software. The Current Mode field of the Processor Status Longword indicates the access mode of the currently executing software.

cylinder  The tracks at the same radius on all recording surfaces of a disk.
**Glossary**

**D floating (point) datum** A floating point datum consisting of 8 contiguous bytes (64 bits) starting on an arbitrary byte boundary. The value of the D floating datum is in the approximate range (+ or −) \(0.29 \times 10^{-38}\) to \(1.7 \times 10^{38}\). The precision is approximately one part in \(2^{55}\) or typically sixteen decimal digits.

**data base** 1. All the occurrences of data described by a data base management system. 2. A collection of related data structures.

**data structure** Any table, list, array, queue, or tree whose format and access conventions are well-defined for reference by one or more images.

**data type** In general, the way in which bits are grouped and interpreted. In reference to the processor instructions, the data type of an operand identifies the size of the operand and the significance of the bits in the operand. Operand data types include: byte, word, long-word and quadword integer, floating and double floating, character string, packed decimal string, and variable-length bit field.

**deferred echo** Refers to the fact that terminal echoing does not occur until a process is ready to accept input entered by type ahead.

**delta time** A time value expressing an offset from the current date and time. Delta times are always expressed in the system as negative numbers whose absolute value is used as an offset from the current time.

**demand zero page** A page, typically of an image stack or buffer area, that is initialized to contain all zeros when dynamically created in memory as a result of a page fault. This feature eliminates the waste of disk space that would otherwise be required to store blocks (pages) that contain only zeros.

**descriptor** A data structure used in calling sequences for passing argument types, addresses and other optional information. See character string descriptor.

**detached process** A process that has no owner. The parent process of a tree of subprocesses. Detached processes are created by the job controller when a user logs on the system or when a batch job is initiated. The job controller does not own the user processes it creates; these processes are therefore detached.

**device** The general name for any physical terminus or link connected to the processor that is capable of receiving, storing, or transmitting data. Card readers, line printers, and terminals are examples of record-oriented devices. Magnetic tape devices and disk devices are examples of mass storage devices. Terminal line interfaces and interprocessor links are examples of communications devices.
Glossary

device interrupt  An interrupt received on interrupt priority level 16 through 23. Device interrupts can be requested only by devices, controllers, and memories.

device name  The field in a file specification that identifies the device unit on which a file is stored. Device names also include the mnemonics that identify an I/O peripheral device in a data transfer request. A device name consists of a mnemonic followed by a controller identification letter (if applicable), followed by a unit number (if applicable), and ends with a colon (:).

device queue  See spool queue.

device register  A location in device controller logic used to request device functions (such as I/O transfers) and/or report status.

device unit  One drive, and its controlling logic, of a mass storage device system. A mass storage system can have several drives connected to it.

diagnostic  A program that tests logic and reports any faults it detects.

direct I/O  An I/O operation in which the system locks the pages containing the associated buffer in memory for the duration of the I/O operation. The I/O transfer takes place directly from the process buffer. Contrast with system buffered I/O.

direct mapping cache  A cache organization in which only one address comparison is needed to locate any data in the cache because any block of main memory data can be placed in only one possible position in the cache. Contrast with fully associative cache.

directory  A file used to locate files on a volume that contains a list of file names (including type and version number) and their unique internal identifications.

directory name  The field in a file specification that identifies the directory file in which a file is listed. The directory name begins with a left bracket ([ or <) and ends with a right bracket (] or >).

displacement deferred indexed mode  An indexed addressing mode in which the base operand specifier uses displacement deferred mode addressing.

displacement deferred mode  In displacement deferred mode addressing, the specifier extension is a byte, word, or longword displacement. The displacement is sign extended to 32 bits and added to a base address obtained from the specified register. The result is the address of a longword which contains the address of the actual operand. If the PC is used as the register, the updated contents of the PC
are used as the base address. The base address is the address of the first byte beyond the specifier extension.

**displacement indexed mode**  An indexed addressing mode in which the base operand specifier uses displacement mode addressing.

**displacement mode**  In displacement mode addressing, the specifier extension is a byte, word, or longword displacement. The displacement is sign extended to 32 bits and added to a base address obtained from the specified register. The result is the address of the actual operand. If the PC is used as the register, the updated contents of the PC are used as the base address. The base address is the address of the first byte beyond the specifier extension.

**drive**  The electro-mechanical unit of a mass storage device system on which a recording medium (disk cartridge, disk pack, or magnetic tape reel) is mounted.

**driver**  The set of code that handles physical I/O to a device.

**dynamic access**  A technique in which a program switches from one record access mode to another while processing a file.

**echo**  A terminal handling characteristic in which the characters typed by the terminal user on the keyboard are also displayed on the screen or printer.

**effective address**  The address obtained after indirect or indexing modifications are calculated.

**entry mask**  A word whose bits represent the registers to be saved or restored on a subroutine or procedure call using the call and return instructions.

**entry point**  A location that can be specified as the object of a call. It contains an entry mask and exception enables known as the entry point mask.

**equivalence name**  The string associated with a logical name in a logical name table. An equivalence name can be, for example, a device name, another logical name, or a logical name concatenated with a portion of a file specification.

**error logger**  A system process that empties the error log buffers and writes the error messages into the error file. Errors logged by the system include memory system errors, device errors and timeouts, and interrupts with invalid vector addresses.

**escape sequence**  An escape is a transition from the normal mode of operation to a mode outside the normal mode. An escape character
is the code that indicates the transition from normal to escape mode. An escape sequence refers to the set of character combinations starting with an escape character that the terminal transmits without interpretation to the software set up to handle escape sequences.

**event** A change in process status or an indication of the occurrence of some activity that concerns an individual process or cooperating processes. An incident reported to the scheduler that affects a process' ability to execute. Events can be synchronous with the process' execution (a wait request), or they can be asynchronous (I/O completion). Some other events include: swapping; wake request; page fault.

**event flag** A bit in an event flag cluster that can be set or cleared to indicate the occurrence of the event associated with that flag. Event flags are used to synchronize activities in a process or among many processes.

**event flag cluster** A set of 32 event flags which are used for event posting. Four clusters are defined for each process: two process-local clusters, and two common event flag clusters. Of the process-local flags, eight are reserved for system use.

**exception** An event detected by the hardware (other than an interrupt or jump, branch, case, or call instruction) that changes the normal flow of instruction execution. An exception is always caused by the execution of an instruction or set of instructions (whereas an interrupt is caused by an activity in the system independent of the current instruction). There are three types of hardware exceptions: traps, faults, and aborts. Examples are: attempts to execute a privileged or reserved instruction, trace faults, compatibility mode faults, breakpoint instruction execution, and arithmetic faults such as floating point overflow, underflow, and divide by zero.

**exception condition** A hardware- or software-detected event other than an interrupt or jump, branch, case, or call instruction that changes the normal flow of instruction execution.

**exception dispatcher** An operating system procedure that searches for a condition handler when an exception condition occurs. If no exception handler is found for the exception or condition, the image that incurred the exception is terminated.

**exception enables** See trap enables.

**exception vector** See vector.

**executable image** An image that is capable of being run in a process. When run, an executable image is read from a file for execution in a process.
Glossary

executive  The generic name for the collection of procedures included in the operating system software that provides the basic control and monitoring functions of the operating system.

executive mode  The second most privileged processor access mode (mode 1). The record management services (RMS) and many of the operating system's programmed service procedures execute in executive mode.

exit  An image exit is a rundown activity that occurs when image execution terminates either normally or abnormally. Image rundown activities include deassigning I/O channels and disassociation of common event flag clusters. Any user- or system-specified exit handlers are called.

exit handler  A procedure executed when an image exits. An exit handler enables a procedure that is not on the call stack to gain control and clean up procedure-own data bases before the actual image exit occurs.

extended attribute block (XAB)  An RMS user data structure that contains additional file attributes beyond those expressed in the file access block (FAB), such as boundary types (aligned on cylinder, logical block number, virtual block number) and file protection information.

extension  The amount of space to allocate at the end of a file each time a sequential write exceeds the allocated length of the file.

extent  The contiguous area on a disk containing a file or a portion of a file. Consists of one or more clusters.

F-floating (point) datum  A floating point datum consisting of 4 contiguous bytes (32 bits) starting on an arbitrary byte boundary. The value of the F-floating datum is in the approximate range (+ or −) 0.29 × 10^{-38} to 1.7 × 10^{38}. The precision is approximately one part in 2^{23} or typically seven decimal digits.

failure exception mode  A mode of execution selected by a process indicating that it wants an exception condition declared if an error occurs as the result of a system service call. The normal mode is for the system service to return an error status code for which the process must test.

fault  A hardware exception condition that occurs in the middle of an instruction and that leaves the registers and memory in a consistent state, such that elimination of the fault and restarting the instruction will give correct results.
field  1. See variable-length bit field. 2. A set of contiguous bytes in a logical record.

file  An organized collection of related items (records) maintained in an accessible storage area, such as disk or tape.

file access block (FAB)  An RMS user data structure that represents a request for data operations related to the file as a whole, such as OPEN, CLOSE, or CREATE.

file header  A block in the index file describing a file on a Files-11 disk structure. The file header identifies the locations of the file’s extents. There is a file header for every file on the disk.

file name  The field preceding a file type in a file specification that contains a 1- to 9-character logical name for a file.

filename extension  See file type.

file organization  The physical arrangement of data in the file. You select the specific organization from those offered by VAX-11 RMS, based on your individual needs for efficient data storage and retrieval. See indexed file organization, relative file organization, and sequential file organization.

Files-11  The name of the on-disk structure used by the RSX-11, IAS and VAX/VMS operating systems. Volumes created under this structure are transportable between these operating systems.

file specification  A unique name for a file on a mass storage medium. It identifies the node, the device, the directory name, the file name, the file type, and the version number under which a file is stored.

file structure  The way in which the blocks forming a file are distributed on a disk or magnetic tape to provide a physical accessing technique suitable for the way in which the data in the file is processed.

file system  A method of recording, cataloging, and accessing files on a volume.

file type  The field in a file specification that is preceded by a period or dot (.) and consists of a zero- to three-character type identification. By convention, the type identifies a generic class of files that have the same use or characteristics, such as ASCII text files, binary object files, etc.

fixed control area  An area associated with a variable length record available for controlling or assisting record access operations. Typical uses include line numbers and printer format control information.

fixed-length record format  Property of a file in which all records are
Glossary

of the same size. This format provides simplicity in determining the exact location of a record in the file and eliminates the need to prefix a record size field to each record.

**floating (point) datum** A numeric data type in which the number is represented by a fraction (less than 1 and greater than or equal to ½) multiplied by 2 raised to a power. There are four floating point data types: F_floating (4 bytes), D_floating (8 bytes), G_floating (8 bytes), and H_floating (16 bytes)

**foreign volume** Any volume other than a Files-11 formatted volume which may or may not be file structured.

**fork process** A dynamically created system process such as a process that executes device driver code or the timer process. Fork processes have minimal context. Fork processes are scheduled by the hardware rather than by the software. The timer process is dispatched directly by software interrupt. I/O driver processes are dispatched by a fork dispatcher. Fork processes execute at software interrupt levels and are dispatched for execution immediately. Fork processes remain resident until they terminate.

**frame pointer** General register 13 (R13). By convention, FP contains the base address of the most recent call frame on the stack.

**fully associative cache** A cache organization in which any block of data from main memory can be placed anywhere in the cache. Address comparision must take place against each block in the cache to find any particular block. Contrast with direct mapping cache.

**G_floating (point) datum** A floating point datum consisting of 8 contiguous bytes (64 bits) starting on an arbitrary byte boundary. The value of the G_floating datum is in the approximate range (+ or −) 0.56 × 10^{308} to 0.9 × 10^{308}. The precision is approximately one part in 2^{52} or typically fifteen decimal digits.

**general register** Any of the sixteen 32-bit registers used as the primary operands of the native mode instructions. The general registers include 12 general purpose registers which can be used as accumulators, as counters, and as pointers to locations in main memory, and the Frame Pointer (FP), Argument Pointer (AP), Stack Pointer (SP), and Program Counter (PC) registers.

**generic device name** A device name that identifies the type of device but not a particular unit; a device name in which the specific controller and/or unit number is omitted.

**giga** Metric term used to represent the number 1 followed by nine zeros.
**Glossary**

**global page table**  The page table containing the master page table entries for global sections.

**global section**  A data structure (e.g., FORTRAN global common) or shareable image section potentially available to all processes in the system. Access is protected by privilege and/or group number of the UIC.

**global symbol**  A symbol defined in a module that is potentially available for reference by another module. The linker resolves (matches references with definitions) global symbols. Contrast with local symbol.

**global symbol table (GST)**  In a library, an index of strongly defined global symbols used to access the modules defining the global symbols. The linker will also put global symbol tables into an image. For example, the linker appends a global symbol table to executable images that are intended to run under the symbolic debugger, and it appends a global symbol table to all shareable images.

**group**  1. A set of users who have special access privileges to each other’s directories and files within those directories (unless protected otherwise), as in the context “system, owner, group, world,” where group refers to all members of a particular owner’s group. 2. A set of jobs (processes and their subprocesses) who have access privileges to a group’s common event flags and logical name tables, and may have mutual process controlling privileges, such as scheduling, hibernation, etc.

**group number**  The first number in a User Identification Code (UIC).

**H_floating (point) datum**  A floating point datum consisting of 16 contiguous bytes (128 bits) starting on an arbitrary byte boundary. The value of the H_floating datum is in the approximate range (+ or −) $0.84 \times 10^{4932}$ to $0.59 \times 10^{4932}$. The precision is approximately one part in $2^{112}$ or typically 33 decimal digits.

**hardware context**  The values contained in the following registers while a process is executing: the Program Counter (PC); the Processor Status Longword (PSL); the 14 general registers (R0 through R13); the four processor registers (P0BR, P0LR, P1BR and P1LR) that describe the process virtual address space; the Stack Pointer (SP) for the current access mode in which the processor is executing; plus the contents to be loaded in the Stack Pointer for every access mode other than the current access mode. While a process is executing, its hardware context is continually being updated by the processor. While a process is not executing, its hardware context is stored in its hardware PCB.
Glossary

hardware process control block (PCB)  A data structure known to the processor that contains the hardware context when a process is not executing. A process' hardware PCB resides in its process header.

hibernation  A state in which a process is inactive, but known to the system with all of its current status. A hibernating process becomes active again when a wake request is issued. It can schedule a wake request before hibernating, or another process can issue its wake request. A hibernating process also becomes active for the time sufficient to service any AST it may receive while it is hibernating. Contrast with suspension.

home block  A block in the index file that contains the volume identification, such as volume label and protection.

image  An image consists of procedures and data that have been bound together by the linker. There are three types of images: executable, shareable, and system.

image activator  A set of system procedures that prepare an image for execution. The image activator establishes the memory management data structures required both to map the image's virtual pages to physical pages and to perform paging.

image exit  See exit.

image I/O segment  That portion of the control region that contains the RMS internal file access blocks (IFAB) and I/O buffers for the image currently being executed by a process.

image name  The file name of the file in which an image is stored.

image privileges  The privileges assigned to an image when it is linked. See process privileges.

image section (isect)  A group of program sections (psects) with the same attributes (such as read-only access, read/write access, absolute, relocatable, etc.) that is the unit of virtual memory allocation for an image.

immediate mode  In immediate mode addressing, the PC is used as the register in autoincrement mode addressing.

index  The structure which allows retrieval of records in an indexed file by key value. See key (indexed files).

index file  The file on a Files-11 volume that contains the access information for all files on the volume and enables the operating system to identify and access the volume.

index file bit map  A table in the index file of a Files-11 volume that indicates which file headers are in use.
Glossary

index register A register used to contain an address offset.

indexed addressing mode In indexed mode addressing, two registers are used to determine the actual instruction operand: an index register and a base operand specifier. The contents of the index register are used as an index (offset) into a table or array. The base operand specifier supplies the base address of the array (the base operand address or BOA). The address of the actual operand is calculated by multiplying the contents of the index register by the size (in bytes) of the actual operand and adding the result to the base operand address. The addressing modes resulting from index mode addressing are formed by adding the suffix “indexed” to the addressing mode of the base operand specifier: register deferred indexed, autoincrement indexed, autoincrement deferred indexed (or absolute indexed), autodecrement indexed, displacement indexed, and displacement deferred indexed.

indexed file organization A file organization which allows random retrieval of records by key values and sequential retrieval of records in sorted order by key value. See key (indexed files).

indirect command file See command procedure.

input stream The source of commands and data. One of: the user's terminal, the batch stream, or an indirect command file.

instruction buffer A buffer in the processor used to contain bytes of the instruction currently being decoded and to pre-fetch instructions in the instruction stream. The control logic continuously fetches data from memory to keep the buffer full.

interleaving Assigning consecutive physical memory addresses alternately between two memory controllers.

interlocked The property of a read followed by a write to the same datum with no possibility of an intervening reference by a second processor or I/O device. Examples are the Branch on Bit Interlocked and Add Aligned Word Interlocked instructions.

interprocess communication facility A common event flag, mailbox, or global section used to pass information between two or more processes.

interrecord gap A blank space deliberately placed between data records on the recording surface of a magnetic tape.

interrupt An event other than an exception or branch, jump, case, or call instruction that changes the normal flow of instruction execution. Interrupts are generally external to the process executing when the interrupt occurs. See also device interrupt, software interrupt, and urgent interrupt.
**Glossary**

**interrupt priority level (IPL)** The interrupt level at which the processor executes when an interrupt is generated. There are 31 possible interrupt priority levels. IPL 1 is lowest, 31 highest. The levels arbitrate contention for processor service. For example, a device cannot interrupt the processor if the processor is currently executing at an interrupt priority level greater than the interrupt priority level of the device's interrupt service routine.

**interrupt service routine** The routine executed when a device interrupt occurs.

**interrupt stack** The system-wide stack used when executing in interrupt service context. At any time, the processor is either in a process context executing in user, supervisor, executive or kernel mode, or in system-wide interrupt service context operating with kernel privileges, as indicated by the interrupt stack and current mode bits in the PSL. The interrupt stack is not context switched.

**interrupt stack pointer** The stack pointer for the interrupt stack. Unlike the stack pointers for process context stacks, which are stored in the hardware PCB, the interrupt stack pointer is stored in an internal register.

**interrupt vector** See vector.

**I/O driver** See driver.

**I/O function** An I/O operation that is interpreted by the operating system and typically results in one or more physical I/O operations.

**I/O function code** A 6-bit value specified in a Queue I/O Request system service that describes the particular I/O operation to be performed (e.g., read, write, rewind).

**I/O function modifier** A 10-bit value specified in a Queue I/O Request system service that modifies an I/O function code (e.g., read terminal input no echo).

**I/O lockdown** The state of a page such that it cannot be paged or swapped out of memory until any I/O in progress to that page is completed.

**I/O rundown** An operating system function in which the system cleans up any I/O in progress when an image exits.

**I/O space** The region of physical address space that contains the configuration registers, and device control/status and data registers.

**I/O status block** A data structure associated with the Queue I/O Request system service. This service optionally returns a status code, number of bytes transferred, and device- and function-dependent in-
formation in an I/O status block. It is not returned from the service call, but filled in when the I/O request completes.

**job**  1. A job is the accounting unit equivalent to a process and the collection of all the subprocesses, if any, that it and its subprocesses create. Jobs are classified as batch and interactive. For example, the job controller creates an interactive job to handle a user's requests when the user logs onto the system and it creates a batch job when the symbiont manager passes a command input file to it. 2. A print job.

**job controller**  The system process that establishes a job's process context, starts a process running the LOGIN image for the job, maintains the accounting record for the job, manages symbionts, and terminates a process and its subprocesses.

**job queue**  A list of files that a process has supplied for processing by a specific device, for example, a line printer.

**kernel mode**  The most privileged processor access mode (mode 0). The operating system's most privileged services, such as I/O drivers and the pager, run in kernel mode.

**key**

*indexed files:* A character string, a packed decimal number, a 2- or 4-byte unsigned binary number, or a 2- or 4-byte signed integer within each data record in an indexed file. You define the length and location within the records; VAX-11 RMS uses the key to build an index. See primary key, alternate key, and random access by key value.

*relative files:* The relative record number of each data record in a data file; VAX-11 RMS uses the relative record numbers to identify and access data records in a relative file in random access mode. See relative record number.

**lexical function**  A command language construct that the command interpreter evaluates and substitutes before it performs expression analysis on a command string. Lexical functions return information about the current process, such as UIC or default directory; and about character strings, such as length or substring locations.

**librarian**  A program that allows the user to create, update, modify, list, and maintain object library, image library, and assembler macro library files.

**library file**  A direct access file containing one or more modules of the same module type.

**limit**  The size or number of given items requiring system resources (such as mailboxes, locked pages, I/O requests, open files, etc.) that a job is allowed to have at any one time during execution, as specified
Glossary

by the system manager in the user authorization file. See also quota.

**line number**  A number used to identify a line of text in a file processed by a text editor.

**linker**  A program that reads one or more object files created by language processors and produces an executable image file, a shareable image file, or a system image file.

**linking**  The resolution of external references between object modules used to create an image, the acquisition of referenced library routines, service entry points, and data for the image, and the assignment of virtual addresses to components of an image.

**literal mode**  In literal mode addressing, the instruction operand is a constant whose value is expressed in a 6-bit field of the instruction. If the operand data type is byte, word, longword, quadword, or octoword, the operand is zero-extended and can express values in the range 0 through 63 (decimal). If the operand data type is F_, D_, G_, or H_floating, the 6-bit field is composed of two 3-bit fields, one for the exponent and the other for the fraction. The operand is extended to F_, D_, G_, or H_floating format.

**locality**  See program locality.

**local symbol**  A symbol meaningful only to the module that defines it. Symbols not identified to a language processor as global symbols are considered to be local symbols. A language processor resolves (matches references with definitions) local symbols. They are not known to the linker and cannot be made available to another object module. They can, however, be passed through the linker to the symbolic debugger. Contrast with global symbol.

**locate mode**  Technique used for a record input operation in which the data records are not copied from the I/O buffer. See move mode.

**locking a page in memory**  Making a page in an image ineligible for either paging or swapping. A page stays locked in memory until it is specifically unlocked.

**locking a page in the working set**  Making a page in an image ineligible for paging out of the working set for the image. The page can be swapped when the process is swapped. A page stays locked in a working set until it is specifically unlocked.

**logical block number**  A number used to identify a block on a mass storage device. The number is a volume-relative address rather than its physical (device-oriented) address or its virtual (file-relative) address. The blocks that constitute the volume are labeled sequentially starting with logical block 0.
Glossary

**logical I/O function** A set of I/O operations (e.g., read and write logical block) that allow restricted direct access to device level I/O operations using logical block addresses.

**logical name** A user-specified name for any portion or all of a file specification. For example, the logical name INPUT can be assigned to a terminal device from which a program reads data entered by a user. Logical name assignments are maintained in logical name tables for each process, each group, and the system. A logical name can be created and assigned a value permanently or dynamically.

**logical name table** A table that contains a set of logical names and their equivalence names for a particular process, a particular group, or the system.

**logical I/O functions** A set of I/O functions that allow restricted direct access to device level I/O operations.

**logical record** A group of related fields treated as a unit.

**longword** Four contiguous bytes (32 bits) starting on an addressable byte boundary. Bits are numbered from right to left, 0 through 31. The address of the longword is the address of the byte containing bit 0. When interpreted arithmetically, a longword is a 2's complement integer with significance increasing from bit 0 to bit 30. When interpreted as a signed integer, bit 31 is the sign bit. The value of the signed integer is in the range $-2,147,483,648$ to $2,147,483,647$. When interpreted as an unsigned integer, significance increases from bit 0 to bit 31. The value of the unsigned integer is in the range $0$ through $4,294,967,295$.

**macro** A statement that requests a language processor to generate a predefined set of instructions.

**mailbox** A software data structure that is treated as a record-oriented device for general interprocess communication. Communication using a mailbox is similar to other forms of device-independent I/O. Senders perform a write to a mailbox, the receiver performs a read from that mailbox. Some system-wide mailboxes are defined: the error logger and OPCOM read from system-wide mailboxes.

**main memory** See physical memory.

**mapping window** A subset of the retrieval information for a file that is used to translate virtual block numbers to logical block numbers.

**mass storage device** A device capable of reading and writing data on mass storage media such as a disk pack or a magnetic tape reel.

**member number** The second number in a user identification code that uniquely identifies that code.
memory management  The system functions that include the hardware's page mapping and protection and the operating system's image activator and pager.

Memory Mapping Enable (MME)  A bit in a processor register that governs address translation.

modify access type  The specified operand of an instruction or procedure is read, and is potentially modified and written, during that instruction's or procedure's execution.

module  1. A portion of a program or program library, as in a source module, object module, or image module. 2. A board, usually made of plastic covered with an electrical conductor, on which logic devices (such as transistors, resistors, and memory chips) are mounted, and circuits connecting these devices are etched, as in a logic module.

Monitor Console Routine (MCR)  The command interpreter in an RSX-11 system.

mount a volume  1. To logically associate a volume with the physical unit on which it is loaded (an activity accomplished by system software at the request of an operator). 2. To load or place a magnetic tape or disk pack on a drive and place the drive online (an activity accomplished by a system operator).

move mode  Technique used for a record transfer in which the data records are copied between the I/O buffer and your program buffer for calculations or operations on the record. See locate mode.

mutex  A semaphore that is used to control exclusive access to a region of code that can share a data structure or other resource. The mutex (mutual exclusion) semaphore ensures that only one process at a time has access to the region of code.

name block (NAM)  An RMS user data structure that contains supplementary information used in parsing file specifications.

native image  An image whose instructions are executed in native mode.

native mode  The processor's primary execution mode in which the programmed instructions are interpreted as byte-aligned, variable-length instructions that operate on byte, word, longword, quadword, and octaword integer, F_, D_, G_ and H_floating format, character string, packed decimal, and variable-length bit field data. The instruction execution mode other than compatibility mode.

network  A collection of interconnected individual computer systems.
Glossary

nibble  The low-order or high-order four bits of a byte.

node  An individual computer system in a network.

null process  A small system process that is the lowest priority process in the system and takes one entire priority class. One function of the null process is to accumulate idle processor time.

numeric string  A contiguous sequence of bytes representing up to 31 decimal digits (one per byte) and possibly a sign. The numeric string is specified by its lowest addressed location, its length, and its sign representation.

object module  The binary output of a language processor such as the assembler or a compiler, which is used as input to the linker.

object time system (OTS)  See Run Time Procedure Library.

octaword  Sixteen contiguous bytes (128 bits) starting on an addressable byte boundary. Bits are numbered from right to left, 0 to 127. An octaword is identified by the address of the byte containing the low-order bit (bit 0).

offset  A fixed displacement from the beginning of a data structure. System offsets for items within a data structure normally have an associated symbolic name used instead of the numeric displacement. Where symbols are defined, programmers always reference the symbolic names for items in a data structure instead of using the numeric displacement.

opcode  The pattern of bits within an instruction that specify the operation to be performed.

operand specifier  The pattern of bits in an instruction that indicate the addressing mode, a register and/or displacement, which, taken together, identify an instruction operand.

operand specifier type  The access type and data type of an instruction's operand(s). For example, the test instructions are of read access type, since they only read the value of the operand. The operand can be of byte, word, or longword data type, depending on whether the opcode is for the TSTB (test byte), TSTW (test word), or TSTL (test longword) instruction.

Operator Communication Manager (OPCOM)  A system process that is always active. OPCOM receives input from a process that wants to inform an operator of a particular status or condition, passes a message to the operator, and tracks the message.

operator's console  Any terminal identified as a terminal attended by a system operator.
owner  In the context "system, owner, group, world," an owner is the particular member (of a group) to which a file, global section, mailbox, or event flag cluster belongs.

owner process  The process (with the exception of the job controller) or subprocess that created a subprocess.

packed decimal  A method of representing a decimal number by storing a pair of decimal digits in one byte, taking advantage of the fact that only four bits are required to represent the numbers 0 through 9.

packed decimal string  A contiguous sequence of up to 16 bytes interpreted as a string of nibbles. Each nibble represents a digit except the low-order nibble of the highest addressed byte, which represents the sign. The packed decimal string is specified by its lowest addressed location and the number of digits.

page  1. A set of 512 contiguous byte locations used as the unit of memory mapping and protection. 2. The data between the beginning of file and a page marker, between two markers, or between a marker and the end of a file.

page fault  An exception generated by a reference to a page which is not mapped into a working set.

page fault cluster size  The number of pages read in on a page fault.

page frame number (PFN)  The address of the first byte of a page in physical memory. The high-order 21 bits of the physical address of the base of a page.

page marker  A character or characters (generally a form feed) that separates pages in a file that is processed by a text editor.

pager  A set of kernel mode procedures that executes as the result of a page fault. The pager makes the page for which the fault occurred available in physical memory so that the image can continue execution. The pager and the image activator provide the operating system's memory management functions.

page table entry (PTE)  The data structure that identifies the location and status of a page of virtual address space. When a virtual page is in memory, the PTE contains the page frame number needed to map the virtual page to a physical page. When it is not in memory, the page table entry contains the information needed to locate the page on secondary storage (disk).

paging  The action of bringing pages of an executing process into physical memory when referenced. When a process executes, all of its pages are said to reside in virtual memory. Only the actively used pages, however, need to reside in physical memory. The remaining
pages can reside on disk until they are needed in physical memory. In this system, a process is paged only when it references more pages than it is allowed to have in its working set. When the process refers to a page not in its working set, a page fault occurs. This causes the operating system’s pager to read in the referenced page if it is on disk (and, optionally, other related pages depending on a cluster factor), replacing the least recently faulted pages as needed. A process pages only against itself.

**parameter**  See command parameter.

**per-process address space**  See process address space.

**physical address**  The address used by hardware to identify a location in physical memory or on directly addressable secondary storage devices such as a disk. A physical memory address consists of a page frame number and the number of a byte within the page. A physical disk block address consists of a cylinder or track and sector number.

**physical address space**  The set of all possible 30-bit physical addresses that can be used to refer to locations in memory (memory space) or device registers (I/O space).

**physical block**  A block on a mass storage device referred to by its physical (device-oriented) address rather than a logical (volume-relative) or virtual (file-relative) address.

**physical I/O functions**  A set of I/O functions that allow access to all device level I/O operations except maintenance mode.

**physical memory**  The memory modules connected to the SBI that are used to store: 1) instructions that the processor can directly fetch and execute, and 2) any other data that a processor is instructed to manipulate. Also called main memory.

**position-dependent code**  Code that can execute properly only in the locations in virtual address space that are assigned to it by the linker.

**position-independent code**  Code that can execute properly without modification wherever it is located in virtual address space, even if its location is changed after it has been linked. Generally, this code uses addressing modes that form an effective address relative to the PC.

**primary key**  The mandatory key within the data records of an indexed file; used by VAX-11 RMS to determine the placement of records within the file and to build the primary index. See key (indexed files) and alternate key.
Glossary

primary vector  A location that contains the starting address of a condition handler to be executed when an exception condition occurs. If a primary vector is declared, that condition handler is the first handler to be executed.

private section  An image section of a process that is not shareable among processes. See also global section.

privilege  See process privilege, user privilege, and image privilege.

privileged instructions  In general, any instructions intended for use by the operating system or privileged system programs. In particular, instructions that the processor will not execute unless the current access mode is kernel mode (e.g., HALT, SVPCTX, LDPCTX, MTPR, and MFPR).

procedure  1. A routine entered via a Call instruction. 2. See command procedure.

process  The basic entity scheduled by the system software that provides the context in which an image executes. A process consists of an address space and both hardware and software context.

process address space  See process space.

process context  The hardware and software contexts of a process.

process control block (PCB)  A data structure used to contain process context. The hardware PCB contains the hardware context. The software PCB contains the software context, which includes a pointer to the hardware PCB.

process header  A data structure that contains the hardware PCB, accounting and quota information, process section table, working set list, and the page tables defining the virtual layout of the process.

process header slots  That portion of the system address space in which the system stores the process headers for the processes in the balance set. The number of process header slots in the system determines the number of processes that can be in the balance set at any one time.

process identification (PID)  The operating system’s unique 32-bit binary value assigned to a process.

process I/O segment  That portion of a process control region that contains the process permanent RMS internal file access block for each open file, and the I/O buffers, including the command interpreter’s command buffer and command descriptors.

process name  A 1- to 15-character ASCII string that can be used to identify processes executing under the same group number.
Glossary

**processor register**  A part of the processor used by the operating system software to control the execution states of the computer system. They include the system base and length registers, the program and control region base and length registers, the system control block base register, the software interrupt request register, and many more.

**Processor Status Longword (PSL)**  A system programmed processor register consisting of a word of privileged processor status and the PSW. The privileged processor status information includes: the current IPL (interrupt priority level), the previous access mode, the current access mode, the interrupt stack bit, the trace fault pending bit, and the compatibility mode bit.

**Processor Status Word (PSW)**  The low-order word of the Processor Status Longword. Processor status information includes: the condition codes (carry, overflow, zero, negative), the arithmetic trap enable bits (integer overflow, decimal overflow, floating underflow), and the trace enable bit.

**process page tables**  The page tables used to describe process virtual memory.

**process priority**  The priority assigned to a process for scheduling purposes. The operating system recognizes 32 levels of process priority, where 0 is low and 31 high. Levels 16 through 31 are used for time-critical processes. The system does not modify the priority of a time-critical process (although the system manager or process itself may). Levels 0 through 15 are used for normal processes. The system may temporarily increase the priority of a normal process based on the activity of the process.

**process privileges**  The privileges granted to a process by the system, which are a combination of user privileges and image privileges. They include, for example, the privilege to: affect other processes associated with the same group as the user's group, affect any process in the system regardless of UIC, set process swap mode, create permanent event flag clusters, create another process, create a mailbox, and perform direct I/O to a file-structured device.

**process section**  See private section.

**process space**  The lowest-addressed half of virtual address space, where per-process instructions and data reside. Process space is divided into a program region and a control region.

**Program Counter (PC)**  General register 15 (R15). At the beginning of an instruction's execution, the PC normally contains the address of
a location in memory from which the processor will fetch the next instruction it will execute.

**program locality** A characteristic of a program that indicates how close or far apart the references to locations in virtual memory are over time. A program with a high degree of locality does not refer to many widely scattered virtual addresses in a short period of time.

**programmer number** See member number.

**program region** The lowest-addressed half of process address space (P0 space). The program region contains the image currently being executed by the process and other user code called by the image.

**Program region Base Register (P0BR)** The processor register, or its equivalent in a hardware process control block, that contains the base virtual address of the page table entry for virtual page number 0 in a process program region.

**Program region Length Register (P0LR)** The processor register, or its equivalent in a hardware process control block, that contains the number of entries in the page table for a process program region.

**program section (psect)** A portion of a program with a given protection and set of storage management attributes. Program sections that have the same attributes are gathered together by the linker to form an image section.

**project number** See group number or account number.

**pure code** See re-entrant code.

**quadword** Eight contiguous bytes (64 bits) starting on an addressable byte boundary. Bits are numbered from right to left, 0 to 63. A quadword is identified by the address of the byte containing the low-order bit (bit 0). When interpreted arithmetically, a quadword is a 2's complement integer with significance increasing from bit 0 to bit 62. Bit 63 is used as the sign bit. The value of the integer is in the range $-2^{63}$ to $2^{63} - 1$.

**qualifier** A portion of a command string that modifies a command verb or command parameter by selecting one of several options. A qualifier, if present, follows the command verb or parameter to which it applies and is in the format: "/qualifier:option." For example, in the command string "PRINT filename/COPIES:3," the COPIES qualifier indicates that the user wants three copies of a given file printed.

**queue** 1. n. A circular, doubly-linked list. See system queues. v. To make an entry in a list or table, perhaps using the INSQUE instruction. 2. See job queue.
queue priority The priority assigned to a job placed in a spooler queue or a batch queue.

quota The total amount of a system resource, such as CPU time, that a job is allowed to use in an accounting period, as specified by the system manager in the user authorization file. See also limit.

random access by key Indexed files only: Retrieval of a data record in an indexed file by either a primary or alternate key within the data record. See key (indexed files).

random access by record's file address The retrieval of a record by its unique address, which is provided to the program by RMS. This method of access is the only means of randomly accessing a sequentially organized file containing variable length records.

random access by relative record number Retrieval of a record by its relative record number. See relative record number. For relative files, random access by relative record number is synonymous with random access by key. See random access by key (relative files only).

read access type An instruction or procedure operand attribute indicating that the specified operand is only read during instruction or procedure execution.

record A set of related data that your program treats as a unit.

record access block (RAB) An RMS user data structure that represents a request for a record access stream. A RAB relates to operations on the records within a file, such as UPDATE, DELETE, or GET.

record access mode The method used in RMS for retrieving and storing records in a file. One of three methods: sequential, random, and record's file address.

record blocking The technique of grouping multiple records into a single block. On magnetic tape, an IRG is placed after the block rather than after each record. This technique reduces the number of I/O transfers required to read or write the data; and, in addition (for magnetic tape), increases the amount of usable storage area. Record blocking also applies to disk files.

record cell A fixed-length area in a relative file that can contain a record. The concept of fixed-length record cells lets VAX-11 RMS directly calculate the record's actual position in the file.

record format The way a record physically appears on the recording surface of the storage medium. The record format defines the method for determining record length.
Glossary

**record length**  The size of a record; that is, the number of bytes in a record.

**record locking**  A facility that prevents access to a record by more than one record stream or process until the initiating record stream or process releases the record.

**Record Management Services**  A set of operating system procedures that are called by programs to process files and records within files. RMS allows programs to issue READ and WRITE requests at the record level (record I/O) as well as read and write blocks (block I/O). RMS is an integral part of the system software. RMS procedures run in executive mode.

**record-oriented device**  A device such as a terminal, line printer, or card reader, on which the largest unit of data a program can access in one I/O operation is the device's physical record.

**record's file address**  The unique address of a record in a file, which is returned by RMS whenever a record is accessed, that allows records in disk files to be accessed randomly regardless of file organization. This address is valid only for the life of the file. If an indexed file is reorganized, then the RFA of each record will typically change.

**re-entrant code**  Code that is never modified during execution. It is possible to let many users share the same copy of a procedure or program written as re-entrant code.

**register**  A storage location in hardware logic other than main memory. See also general register, processor register, and device register.

**register deferred indexed mode**  An indexed addressing mode in which the base operand specifier uses register deferred mode addressing.

**register deferred mode**  In register deferred mode addressing, the contents of the specified register are used as the address of the actual instruction operand.

**register mode**  In register mode addressing, the contents of the specified register are used as the actual instruction operand.

**relative file organization**  The arrangement of records in a file where each record occupies a cell of equal length within a bucket. Each cell is assigned a successive number, called a relative record number, which represents the cell's position relative to the beginning of the file.

**relative record number**  An identification number used to specify the position of a record cell relative to the beginning of the file; used as the key during random access by key mode to relative files.
resource  A physical part of the computer system such as a device or memory, or an interlocked data structure such as a mutex. Quotas and limits control the use of physical resources.

resource wait mode  An execution state in which a process indicates that it will wait until a system resource becomes available when it issues a service request requiring a resource. If a process wants notification when a resource is not available, it can disable resource wait mode during program execution.

return status code  See status code.

RMS-11  A set of routines which is linked with compatibility mode programs, and provides similar functional capabilities to VAX-11 RMS. The file organizations and record formats used by RMS-11 are identical to those of VAX-11 RMS.

Run Time Procedure Library  The collection of procedures available to native mode images at run time. These library procedures (such as trigonometric functions, etc.) are common to all native mode images, regardless of the language processor used to compile or assemble the program.

scatter/gather  The ability to transfer in one I/O operation data from discontiguous pages in memory to contiguous blocks on disk, or data from contiguous blocks on disk to discontiguous pages in memory.

secondary storage  Random access mass storage.

secondary vector  A location that identifies the starting address of a condition handler to be executed when a condition occurs and the primary vector contains zero or the handler to which the primary vector points chooses not to handle the condition.

section  A portion of process virtual memory that has common memory management attributes (protection, access, cluster factor, etc.). It is created from an image section, a disk file, or as the result of a Create Virtual Address Space system service. See global section, private section, image section, and program section.

self-relative queue  A circularly linked list whose forward and backward links use the address of the entry in which they occur as the base address for the link displacement to the linked entry. Contrast with absolute addresses used to link a queue.

sequential file organization  A file organization in which records appear in the order in which they were originally written. The records can be fixed length or variable length.

sequential record access mode  Record storage or retrieval which starts at a designated point in the file and continues in one-after-the-
Glossary

other fashion through the file. That is, records are accessed in the order in which they physically appear in the file.

shareable image  An image that has all of its internal references resolved, but which must be linked with an object module(s) to produce an executable image. A sharable image cannot be executed. A shareable image file can be used to contain a library of routines. A shareable image can be used to create a global section by the system manager.

shell process  A predefined process that the job initiator copies to create the minimum context necessary to establish a process.

signal  1. An electrical impulse conveying information. 2. The software mechanism used to indicate that an exception condition was detected.

slave terminal  A terminal from which it is not possible to issue commands to the command interpreter. A terminal assigned to application software.

small process  A system process that has no control region in its virtual address space and has an abbreviated context. Examples are the working set swapper and the null process. A small process is scheduled in the same manner as user processes, but must remain resident during its execution.

software context  The context maintained by the operating system that describes a process. See software process control block (PCB).

software interrupt  An interrupt generated on interrupt priority level 1 through 15, which can be requested only by software.

software process control block (PCB)  The data structure used to contain a process’ software context. The operating system defines a software PCB for every process when the process is created. The software PCB includes the following kinds of information about the process: current state; storage address if it is swapped out of memory; unique identification of the process, and address of the process header (which contains the hardware PCB). The software PCB resides in system region virtual address space. It is not swapped with a process.

software priority  See process priority and queue priority.

spooling  output spooling: The method by which output to a low-speed peripheral device (such as a line printer) is placed into queues maintained on a high-speed device (such as disk) to await transmission to the low-speed device. Input spooling: the method by which input from a low-speed peripheral (such as the card reader) is placed into queues maintained on a high-speed device (such as disk) to await transmission to a job processing that input.
Glossary

spool queue  The list of files supplied by processes that are to be processed by a symbiont. For example, a line printer queue is a list of files to be printed on the line printer.

stack  An area of memory set aside for temporary storage, or for procedure and interrupt service linkages. A stack uses the last-in, first-out concept. As items are added to ("pushed on") the stack, the stack pointer decrements. As items are retrieved from ("popped off") the stack, the stack pointer increments.

stack frame  A standard data structure built on the stack during a procedure call, starting from the location addressed by the FP to lower addresses, and popped off during a return from procedure. Also called call frame.

stack pointer  General register 14 (R14). SP contains the address of the top (lowest address) of the processor-defined stack. Reference to SP will access one of the five possible stack pointers (kernel, executive, supervisor, user, or interrupt) depending on the value in the current mode and interrupt stack bits in the Processor Status Longword (PSL).

state queue  A list of processes in a particular processing state. The scheduler uses state queues to keep track of processes’ eligibility to execute. They include: processes waiting for a common event flag, suspended processes, and executable processes.

status code  A longword value that indicates the success or failure of a specific function. For example, system services always return a status code in R0 upon completion.

store through  See write through.

strong definition  Definition of a global symbol that is explicitly available for reference by modules linked with the module in which the definition occurs. The linker always lists a global symbol with a strong definition in the symbol portion of the map. The librarian always includes a global symbol with a strong definition in the global symbol table of a library.

strong reference  A reference to a global symbol in an object module that requests the linker to report an error if it does not find a definition for the symbol during linking. If a library contains the definition, the linker incorporates the library module defining the global symbol into the image containing the strong reference.

subprocess  A subsidiary process created by another process. The process that creates a subprocess is its owner. A subprocess receives resource quotas and limits from its owner. When an owner process is
removed from the system, all its subprocesses (and their subprocesses) are also removed.

supervisor mode  The third most privileged processor access mode (mode 2). The operating system's command interpreter runs in supervisor mode.

suspension  A state in which a process is inactive, but known to the system. A suspended process becomes active again only when another process requests the operating system to resume it. Contrast with hibernation.

swap mode  A process execution state that determines the eligibility of a process to be swapped out of the balance set. If process swap mode is disabled, the process working set is locked in the balance set.

swapping  The method for sharing memory resources among several processes by writing an entire working set to secondary storage (swap out) and reading another working set into memory (swap in). For example, a process' working set can be written to secondary storage while the process is waiting for I/O completion on a slow device. It is brought back into the balance set when I/O completes. Contrast with paging.

switch  See (command) qualifier.

symbiont  A full process that transfers record-oriented data to or from a mass storage device. For example, an input symbiont transfers data from card readers to disks. An output symbiont transfers data from disks to line printers.

symbiont manager  The function (in the system process called the job controller) that maintains spool queues, and dynamically creates symbiont processes to perform the necessary I/O operations.

symbol  See local symbol, global symbol, and universal global symbol.

Synchronous Backplane Interconnect (SBI)  The part of the hardware that interconnects the processor, memory controllers, MASSBUS adapters, and the UNIBUS adapter.

synchronous record operation  A mode of record processing in which a user program issues a record read or write request and then waits until that request is fulfilled before continuing to execute.

system  In the context "system, owner, group, world," the system refers to the group numbers that are used by operating system and its controlling users, the system operators and system manager.

system address space  See system space and system region.
Glossary

System Base Register (SBR)  A processor register containing the physical address of the base of the system page table.

system buffered I/O  An I/O operation, such as terminal or mailbox I/O, in which an intermediate buffer from the system buffer pool is used instead of a process-specified buffer. Contrast with direct I/O.

System Control Block (SCB)  The data structure in system space that contains all the interrupt and exception vectors known to the system.

System Control Block Base register (SCBB)  A processor register containing the base address of the system control block.

system device  The random access mass storage device unit on which the volume containing the operating system software resides.

system dynamic memory  Memory reserved for the operating system to allocate as needed for temporary storage. For example, when an image issues an I/O request, system dynamic memory is used to contain the I/O request packet. Each process has a limit on the amount of system dynamic memory that can be allocated for its use at one time.

System Identification Register  A processor register which contains the processor type and serial number.

system image  The image that is read into memory from secondary storage when the system is started up.

System Length Register (SLR)  A processor register containing the length of the system page table in longwords, that is, the number of page table entries in the system region page table.

System Page Table (SPT)  The data structure that maps the system region virtual addresses, including the addresses used to refer to the process page tables. The System Page Table (SPT) contains one Page Table Entry (PTE) for each page of system region virtual memory. The physical base address of the SPT is contained in a register called the SBR.

system process  A process that provides system-level functions. Any process that is part of the operating system. See also small process, fork process.

system programmer  A person who designs and/or writes operating systems, or who designs and writes procedures or programs that provide general purpose services for an application system.

system queue  A queue used and maintained by operating system procedures. See also state queues.
Glossary

**system region**  The third quarter of virtual address space. The lowest-addressed half of system space. Virtual addresses in the system region are shareable between processes. Some of the data structures mapped by system region virtual addresses are: system entry vectors, the System Control Block (SCB), the System Page Table (SPT), and process page tables.

**system services**  Procedures provided by the operating system that can be called by user processes.

**system space**  The highest-addressed half of virtual address space. See also system region.

**system virtual address**  A virtual address identifying a location mapped by an address in system space.

**system virtual space**  See system space.

**task**  An RSX-11/IAS term for a process and image bound together.

**terminal**  The general name for those peripheral devices that have keyboards and video screens or printers. Under program control, a terminal enables people to type commands and data on the keyboard and receive messages on the video screen or printer. Examples of terminals are the LA36 DECwriter hard-copy terminal and VT100 video display terminal.

**time-critical process**  A process assigned to a software priority level between 16 and 31, inclusive. The scheduling priority assigned to a time-critical process is never modified by the scheduler, although it can be modified by the system manager or process itself.

**timer**  A system fork process that maintains the time of day and the date. It also scans for device timeouts and performs time-dependent scheduling upon request.

**track**  A collection of blocks at a single radius on one recording surface of a disk.

**transfer address**  The address of the location containing a program entry point (the first instruction to execute).

**translation buffer**  An internal processor cache containing translations for recently used virtual addresses.

**trap**  An exception condition that occurs at the end of the instruction that caused the exception. The PC saved on the stack is the address of the next instruction that would normally have been executed. All software can enable and disable some of the trap conditions with a single instruction.
Glossary

**trap enables**  Three bits in the Processor Status Word that control the processor's action on certain arithmetic exceptions.

**two's complement**  A binary representation for integers in which a negative number is one greater than the bit complement of the positive number.

**two-way associative cache**  A cache organization which has two groups of directly mapped blocks. Each group contains several blocks for each index position in the cache. A block of data from main memory can go into any group at its proper index position. A two-way associative cache is a compromise between the extremes of fully associative and direct mapping cache organizations that takes advantage of the features of both.

**type ahead**  A terminal handling technique in which the user can enter commands and data while the software is processing a previously entered command. The commands typed ahead are not echoed on the terminal until the command processor is ready to process them. They are held in a type ahead buffer.

**unit record device**  A device such as a card reader or line printer.

**universal global symbol**  A global symbol in a shareable image that can be used by modules linked with that shareable image. Universal global symbols are typically a subset of all the global symbols in a shareable image. When creating a shareable image, the linker ensures that universal global symbols remain available for reference after symbols have been resolved.

**unwind the call stack**  To remove call frames from the stack by tracing back through nested procedure calls using the current contents of the FP register and the FP register contents stored on the stack for each call frame.

**urgent interrupt**  An interrupt received on interrupt priority levels 24 through 31. These can be generated only by the processor for the interval clock, serious errors, and power fail.

**user authorization file**  A file containing an entry for every user that the system manager authorizes to gain access to the system. Each entry identifies the user name, password, default account, User Identification Code (UIC), quotas, limits, and privileges assigned to individuals who use the system.

**user environment test package (UETP)**  A collection of routines that verify that the hardware and software systems are complete, properly installed, and ready to be used.

**User File Directory (UFD)**  See directory.
Glossary

User Identification Code (UIC) The pair of numbers assigned to users and to files, global sections, common event flag clusters, and mailboxes that specifies the type of access (read and/or write access, and in the case of files, execute and/or delete access) available to the owners, group, world, and system. It consists of a group number and a member number separated by a comma.

user mode The least privileged processor access mode (mode 3). User processes and the Run Time Library procedures run in user mode.

user name The name that a person types on a terminal to log on to the system.

user number See member number.

user privileges The privileges granted a user by the system manager. See process privileges.

utility A program that provides a set of related general purpose functions, such as a program development utility (an editor, a linker, etc.), a file management utility (file copy or file format translation program), or operations management utility (disk backup/restore, diagnostic program, etc.).

value return registers The general registers R0 and R1 used by convention to return function values. These registers are not preserved by any called procedures. They are available as temporary registers to any called procedure. All other registers (R2, R3,...,R11, AP, FP, SP, PC) are preserved across procedure calls.

variable-length bit field A set of 0 to 32 contiguous bits located arbitrarily with respect to byte boundaries. A variable bit field is specified by four attributes: 1) the address A of a byte, 2) the bit position P of the starting location of the bit field with respect to bit 0 of the byte at address A, 3) the size, in bits, of the bit field, and 4) whether the field is signed or unsigned.

variable-length record format A file format in which records are not necessarily the same length.

variable with fixed-length control record format Property of a file in which records of variable-length contain an additional fixed control area capable of storing data that may have no bearing on the other contents of the record. Variable with fixed-length control record format is not applicable to indexed files.

VAX-11 Record Management Services (VAX-11 RMS) The file and record access subsystem of the VAX/VMS operating system for VAX. VAX-11 RMS helps your application program process records within
files, thereby allowing interaction between your application program and its data.

**vector** 1. A interrupt or exception vector is a storage location known to the system that contains the starting address of a procedure to be executed when a given interrupt or exception occurs. The system defines separate vectors for each interrupting device controller and for classes of exceptions. Each system vector is a longword. 2. For exception handling, users can declare up to two software exception vectors (primary and secondary) for each of the four access modes. Each vector contains the address of a condition handler. 3. A one-dimensional array.

**version number** 1. The field following the file type in a file specification. It begins with a period (.) and is followed by a number which generally identifies it as the latest file created of all files having the identical file specification but for version number. 2. The number used to identify the revision level of program.

**virtual address** A 32-bit integer identifying a byte "location" in virtual address space. The memory management hardware translates a virtual address to a physical address. The term "virtual address" may also refer to the address used to identify a virtual block on a mass storage device.

**virtual address space** The set of all possible virtual addresses that an image executing in the context of a process can use to identify the location of an instruction or data. The virtual address space seen by the programmer is a linear array of 4,294,967,296 \(2^{32}\) byte addresses.

**virtual block** A block on a mass storage device referred to by its file-relative address rather than its logical (volume-oriented) or physical (device-oriented) address. The first block in a file is always virtual block 1.

**virtual I/O functions** A set of I/O functions that must be interpreted by an ancillary control process.

**virtual memory** The set of storage locations in physical memory and on disk that are referred to by virtual addresses. From the programmer's viewpoint, the secondary storage locations appear to be locations in physical memory. The size of virtual memory in any system depends on the amount of physical memory available and the amount of disk storage used for non-resident virtual memory.

**virtual page number** The virtual address of a page of virtual memory.
Glossary

**volume**
*Disks:* An ordered set of 512-byte blocks. The basic medium that carries a Files-11 structure.

*Magnetic tape:* A reel of magnetic tape, which may contain a part of a file, a complete file, or more than one file.

**volume set**  A collection of related volumes.

**wait**  To become inactive. A process enters a process wait state when the process suspends itself, hibernates, or declares that it needs to wait for an event, resource, mutex, etc.

**wake**  To activate a hibernating process. A hibernating process can be awakened by another process or by the timer process, if the hibernating process or another process scheduled a wake-up call.

**weak definition**  Definition of a global symbol that is not explicitly available for reference by modules linked with the module in which the definition occurs. The librarian does not include a global symbol with a weak definition in the global symbol table of a library. Weak definitions are often used when creating libraries to identify those global symbols that are needed only if the module containing them is otherwise linked with a program.

**weak reference**  A reference to a global symbol that requests the linker not to report an error or to search the default library's global symbol table to resolve the reference if the definition is not in the modules explicitly supplied to the linker. Weak references are often used when creating object modules to identify those global symbols that may not be needed at run time.

**wild card**  A symbol, such as an asterisk, that is used in place of a file name, file type, directory name, or version number in a file specification to indicate "all" for the given field.

**window**  See mapping window.

**word**  Two contiguous bytes (16 bits) starting on an addressable byte boundary. Bits are numbered from the right, 0 through 15. A word is identified by the address of the byte containing bit 0. When interpreted arithmetically, a word is a 2's complement integer with significance increasing from bit 0 to bit 14. If interpreted as a signed integer, bit 15 is the sign bit. The value of the integer is in the range $-32,768$ to 32,767. When interpreted as an unsigned integer, significance increases from bit 0 through bit 15 and the value of the unsigned integer is in the range 0 through 65,535.

**working set**  The set of pages in process space to which an executing process can refer without incurring a page fault. The working set
must be resident in memory for the process to execute. The remaining pages of that process, if any, are either in memory and not in the process working set or they are on secondary storage.

**working set swapper**  A system process that brings process working sets into the balance set and removes them from the balance set.

**world**  In the context “system, owner, group, world,” world refers to all users, including the system operators, the system manager, and users both in an owner’s group and in any other group.

**write access type**  The specified operand of an instruction or procedure is written only during that instruction’s or procedure’s execution.

**write allocate**  A cache management technique in which cache is allocated on a write miss as well as on the usual read miss.

**write back**  A cache management technique in which data from a write operation to cache are copied into main memory only when the data in cache must be overwritten. This results in temporary inconsistencies between cache and main memory. Contrast with write through.

**write through**  A cache management technique in which data from a write operation are copied in both cache and main memory. Cache and main memory data are always consistent. Contrast with write back.
INDEX

aborts 144

absolute (autoincrement deferred) mode 141, 270

ACB instruction 10

accelerator control/status register (ACCS) 444 to 445

accelerator maintenance register (ACCR) 445 to 446

access
  address space, in VAX-11/750 149 to 151
  on MASSBUS adapter 179 to 189
  memory, privileges for 59
to shared data 84

access control 63 to 65

access control violation faults 65, 75

access modes 63, 76 to 77
  changing 88
  hierarchical 226, 449, 451
  processor 277 to 279

ACCR (accelerator maintenance register) 445 to 446

ACCS (accelerator control/status register) 444 to 445

action routines 419

adapters (buffered interfaces) 15

see also MASSBUS adapter; UNIBUS adapter

ADAWI instruction 84

addressing modes 11
  in VAX-11/750 systems 136 to 138
  in VAX-11/780 systems 264 to 266, 269 to 270

address register (on UNIBUS) 168

address space
  memory cycles for access to, in VAX-11/750 149 to 151
  SBI, UNIBUS access to 339 to 341
  UNIBUS 164, 334 to 339
  virtual 60 to 63

address translation 65 to 67
  process space 69 to 74
  system space 67 to 68
  UNIBUS to SBI 340 to 341
  virtual to physical 481 to 483

address translation buffers 12
  in VAX-11/750 CPU 133
  in VAX-11/780 CPU 257 to 258

address translation maps 16, 340

address validation rules 477 to 479

ALERT line 303

ALPCTL field 215

ALU (arithmetic logic unit)
  microarchitecture and 213, 214
  microprogramming and 218

ALUCI field 214

ALUSHF field 215

AP, see Argument Pointer

application kernels 211

application program interfaces 415 to 416

arbitration lines 291 to 292

architecture of VAX systems 5, 83 to 87
  microprogramming and 212 to 216

argument lists 142

Argument Pointer (AP; R12) 11
  on VAX-11/750 CPU 139, 141 to 142
  on VAX-11/780 CPU 264, 269 to 272

argument validation 76 to 77

arithmetic exceptions 41 to 44
  on VAX-11/750 CPU 145

arithmetic logic unit, see ALU

arithmetic traps 42 to 44, 227, 451

array cards 147 to 148, 323

ASSIGN command 433

577
Assign I/O Channel ($ASSIGN) system service 435
Associate Common Event Flag Cluster ($ASCEFC) 434, 435
AST (asynchronous system traps) 25 to 26, 89, 280, 286
ASTLVL (AST Level Register) 23 to 26
asynchronous control path (bus) 390
asynchronous system traps (AST) 25 to 26, 89, 280, 286
autodecrement mode 141, 270
autoincrement deferred (absolute) mode 141, 270
autoincrement (immediate) mode 141, 270
automatic recoveries 4
automatic restarts 225, 235
availability
of VAX-11/750 systems 223 to 236
of VAX-11/780 systems 449 to 454
bad blocks 229
base registers 268
battery backup 18, 149
automatic restart and 235
for MA780 431
for processor clocks 133, 199, 441
for VAX-11/780 main memory subsystem 324
BBCCI instruction 84
BBSSI instruction 84
BCR (MBA Byte Counter) 186, 403
BDP (buffered data paths) 168 to 171, 340, 342 to 347, 350, 354
BINARY LOAD command 110, 118 to 119
BINARY UNLOAD command 110, 118 to 119
bit data 9
bit-field data 9, 136, 264
bootblocks 121
BOOT command 110, 112 to 113
BOOT DEVICE switch 108
bootstrapping (initializing)
privileged registers during 25 ROMs for 148 to 149, 323 to 324
TU58 tape drives and 119
of VAX-11/750 systems 120 to 126
VAX-11/780 UNIBUS and 381 to 382
BR Receive Vector Registers (BRRVR) 354 to 357, 374 to 375
BRs (bus requests) 161, 328 to 329
BRSVR (Buffer Selection Verification Registers) 373 to 374
BTE (Buffer Transfer Error) bit 354
buffer block, on DR780 416
buffered data paths (BDP) 168 to 171, 340, 342 to 347, 350, 354
buffers
address translation 12
in data chaining 413
instruction 133
in MASSBUS 176, 177
translation 74 to 75
in UNIBUS data paths 165, 168 to 171, 340, 342 to 347, 350, 354
in VAX-11/780 CPU 257 to 258
Buffer Selection Verification Registers (BRSVR) 373 to 374
Buffer Transfer Error (BTE) bit 354
bus cycles 160
buses
control 175
DR32 411 to 414
internal, in VAX-11/750 systems 213 to 214
Internal Data, in VAX-11/780...
Index

systems, registers for 485 to 498
Synchronous Backplane
Interconnect and 289
synchronous data path 390
VAX-11/780 console subsystem
and 242 to 243
see also MASSBUS; UNIBUS
BUS field 216
bus request levels 328 to 329
bus requests (BRs) 161, 328 to 329
BUT/IRD1 217 to 219
BUT/RETURN micro-order 219
BUT/WX.EQ.0 micro-order 216
Byte Counter (BCR) 186, 403
byte integers 5
byte offset data transfers 347 to 349
BYTE WRITE cycle 150 to 151

Cache Disable Register (CADR) 205
Cache Error Register (CAER) 205 to
206, 228
cache invalidation logic 430 to 431
Cache Parity Register 453
caches 11 to 12, 84 to 85
disabling of 450, 453
in VAX-11/750 CPU 132 to 133
in VAX-11/780 CPU 257
CADR (Cache Disable Register) 205
CAER (Cache Error Register) 205 to
206, 228
Calculate Cyclic Redundancy Check
(CRC) instruction 227, 452
call (stack) frames 140, 142, 267
CALL instructions
for privileged services 88
on VAX-11/750 140
on VAX-11/780 266, 272, 452
CAR (Command Address
Register) 188 to 189, 405
CCL, see console command
language
central processing unit, see CPU
chaining 329
command 412 to 413
Change Mode (CHM)
instructions 76, 88 to 91, 279
character string data 9, 136, 264
check bits (parity bits) 317
CHME instruction 76
CHM (Change Mode)
instructions 76, 88 to 91, 279
CHMK instruction 76
CHMS instruction 76
CHMU instruction 76
CIB (Console Interface Board) 240
to 243, 245
CLK field 216
clock functions 303
clock margining 450, 453
clocks 12
interval 228
processor 133, 258
registers for 199 to 201, 441 to
444
CNF field 302
CNFGR (configuration register) 360
to 363
Command Address Register
(CAR) 188 to 189, 405
command/address tag 295 to 297
command block, on DR780 416
command chaining 412 to 413
command code 304 to 309
command generated error
messages 249 to 250
commands
in console command
language 109 to 110, 112 to 119
DR780 and 416 to 417
MASSBUS 176
common event flags 433 to 435

579
Index

communications
  console terminal 105
  DR780 for 413
  MASSBUS for 175, 389
  UNIBUS for 158 to 160, 327, 328
compatibility
  of VAX software 2, 6
  of VAX systems with PDP-11 systems 5
compatibility mode 135, 263
condition codes 144, 267
condition handlers 145, 274
configuration register (CNFGR) 360 to 363
Configuration/Status Register (CSR) 396 to 398
consistency
  in VAX-11/750 systems 227
  in VAX-11/780 systems 450 to 452
console command language (CCL) 104 to 105, 109 to 119, 226 to 227, 241, 245 to 249
console command (I/O) mode 105, 241, 242
console error messages 249 to 252
Console Interface Board (CIB) 240 to 243, 245
console modes 104 to 106
console receive control/status register (RXCS) 194, 440
console receive data buffer register (RXDB) 194 to 195, 440
console storage receive data register (CSRD) 197
console storage receive status register (CSRS) 196 to 197
console storage transmit data register (CSTD) 198
console storage transmit status register (CSTS) 197 to 198
Console subsystem 12 to 13
  on VAX-11/750 systems 103 to 126, 226 to 227
  on VAX-11/780 systems 239 to 252
console terminal registers 194 to 196, 440 to 441
console transmit control/status register (TXCS) 195 to 196, 441
console transmit data buffer register (TXDB) 441
context switching 21, 274 to 275
CONTINUE command 110, 113
control bus 175
control characters, in console command language 112
control interconnects 411 to 412
controllers, memory
  for DR780 425
  on VAX-11/750 systems 147 to 149
  on VAX-11/780 systems 311 to 313
control lines 303 to 304
control path 395 to 396
control region 135, 268
control registers
  in VAX-11/750 main memory subsystem 151 to 155
  in VAX-11/750 MASSBUS adapter 180 to 182
  in VAX-11/750 UNIBUS 170 to 171
  in VAX-11/780 MASSBUS adapter 398 to 399
  in VAX-11/780 UNIBUS 363 to 366, 370 to 372
control signals 411
control store
  on VAX-11/750 CPU 132
  on VAX-11/780 CPU 256 to 257
CPU (central processing unit) 7 to 8
  console command language
  for 105, 110
Index

interrupt priority levels on 33, 34
interrupts on 35
MASSBUS and 176
MASSBUS adapter and 392
priority on 6
UNIBUS access of 163 to 164
UNIBUS priority of 160, 162, 329
VAX-11/750 129 to 145
VAX-11/750, technical
specifications for 515 to 521
VAX-11/780 255 to 286
VAX-11/780, Console Interface
Board and 240 to 242
VAX-11/780, LSI-11 and 243 to
244
VAX-11/780, messages for errors
generated by 250 to 251
VAX-11/780, technical
specifications for 523 to 527
CPU state indicator lights 108 to
109
CR (MBA Control Register) 180 to
182, 398 to 399
CRC (Calculate Cyclic Redundancy
Check) instruction 227, 452
Create Logical Name ($CRELOG)
system service 433
Create Mailbox and Assign Channel
($CREMBX) system service 435
critical data structure
information 234
CSA microinstruction 216
CSR (MBA Configuration/Status
Register) 396 to 398
CSR 0 (register) 151 to 152
CSR 1 (register) 152 to 154
CSR 2 (register) 154 to 155
CSRD (console storage receive data
register) 197
CSRS (console storage receive status
register) 196 to 197
CSTD (console storage transmit data
register) 198
CSTS (console storage transmit
status register) 197 to 198
customer writable control store
(WCS) 259, 446
data
handled by VAX-11/750 CPU 131
to 132
MASSBUS transfer of 189 to 190
sharing of 83 to 84
types of 8 to 9, 136, 264
UNIBUS transactions of 162 to
171
see also data transfers
data bus 175
data chaining 413
data integrity, in MA780 430 to 431
data interconnects 412
data management 7
Data Path Number 166 to 168
data path registers (DPR) 350, 375
to 378
data paths
MASSBUS 178, 394
in microprogramming 214, 218
in Synchronous Backplane
Interconnect 289 to 309
in VAX-11/750 UNIBUS 132
in VAX-11/780 CPU 257
in VAX-11/780 UNIBUS
adapter 342 to 354
data transfer program flow
in VAX-11/750 MASSBUS 189 to
190
in VAX-11/780 MASSBUS 405
data transfers 16
using DR32 411
using DR780 interface 410, 425
using MA780 428 to 429
data verification 410
DATI data transactions 162, 168 to
170, 347
DATI function 351
DATIP data transactions 162, 168 to
Index

170 DATIP-DATO/DATOB functions 84, 350 to 354
DATOB data transactions 162, 169 to 170
DATO data transactions 162, 169, 170
DCL (DIGITAL Command Language) 2
DCR (Diagnostic Control Register) 370 to 372
DDC (DIGITAL Diagnostic Center) 3, 13, 18, 230 to 232
DDI (DR32 Device Interconnect) 17, 411 to 414
DDP (direct data path; on UNIBUS) 165 to 168, 342 to 344
Dead signal function 303 to 304
Deassign I/O Channel ($DASSGN) system service 435
decimal string overflow trap 43
DEFINE command 433
Delete Global Section ($DGBLSC) 436 to 437
Delete Virtual Address Space ($DELTVA) system service 436
DEPOSIT command 110, 113 to 116
device controllers 177, 283
device interrupts 36
device registers 334
device ROM code 122 to 124
$DGBLSC (Delete Global Section) 436 to 437
diagnostic console 449, 451
Diagnostic Control Register (DCR) 370 to 372
diagnostic register (DR) 186 to 188, 403 to 405
diagnostics 3 to 4
in DR780 420 to 423
in MA780 431
maintainability and 225, 449
MASSBUS 177, 392
Remote Diagnosis Module for 13, 18, 106, 109, 226 to 227
TU58 tape drives and 119
in VAX-11/750 systems 230 to 234
writable diagnostic control store for 258
DIGITAL Command Language (DCL) 2
DIGITAL Diagnostic Center (DDC) 3, 13, 18, 230 to 232
direct data path (DDP; on UNIBUS) 165 to 168, 342 to 344
direct memory access (DMA) 16, 110
disabling of memory management and cache features 450
disk error correcting code 228, 452
displacement deferred mode 141, 270
displacement mode 141, 270
DMA (direct memory access) 16, 110
DOSERVICE hardware routine 217
DPR (data path registers) 350, 375 to 378
DQ field 214
DR (diagnostic register) 186 to 188, 403 to 405
DR32 Device Interconnect (DDI) 17, 411 to 414
DR780 (general purpose parallel interface) 15, 17, 409 to 426
DR780 Status Longword (DSL) 419
dynamic bad block handling 229, 453 to 454
dynamic command chaining 412 to 413
dynamic memory mapping 413
ECC, see error-correcting code
emulation  210, 211
  on VAX-11/750 systems  217 to 218
error checking
  in DR780  419
  in VAX-11/750 systems  227
  in VAX-11/780 systems  450 to 452
error-correcting code (ECC)  14
  in MA780  430
  in VAX-11/750 systems  227 to 228, 234
  in VAX-11/780 systems  311 to 312, 317, 450, 452
error logging  234 to 235
error messages  249 to 252
errors
  console command  119
  interrupts and  86
  in machine check exceptions  40
  non-existent memory (NXM)  164
  in READ cycles  150
typing, in console command language  111 to 112
  in WRITE cycles  151
ESP (Executive Stack Pointer)  23
events, interrupts caused by  35 to 36
EXAMINE command  109, 113 to 116
exceptions  33 to 57
  on VAX-11/750 CPU  144 to 145
  on VAX-11/780 CPU  267, 274, 275, 282 to 283
exception service routines  34, 35
executive mode  63, 226, 278
Executive Stack Pointer (ESP)  23
extended function instruction (XFC)  89, 99
extended read command  307 to 308
Extended Read cycles  313, 315
extended write masked command  309
Extended Write Masked cycles  303 to 314, 316
external registers  189, 405
Failed Map Entry Register (FMER)  372
Failed UNIBUS Address Registers (FUBAR)  373
FAIL function  304
failsoft capability  431
fault detection, integral  227 to 228
FAULT field  302
fault-isolation diagnostics  233
Fault Parameter Word  75
faults  144
  access control  65, 75 to 76
  arithmetic  44
  fault tolerance features  228 to 229, 450, 453 to 454
  FC opcode  219
FIND FIRST instruction  10
flags
  common event  433 to 435
  microprogramming and  218 to 219
  on UNIBUS  168
floating/decimal string divide by zero trap  43
floating divide by zero fault  44
floating overflow fault  44
floating overflow trap  43
floating point accelerator (FPA)  17
to 18, 259, 444 to 446
floating point data  9, 136, 264
floating underflow fault  44
floating underflow trap  43
floppy disks, messages for errors generated by  251
Index

FMER (Failed Map Entry Register) 372
FM Internal Data register 240, 243, 244
FORTRAN (language) 2
  arbitrary instruction addressing for 5
FP, see Frame Pointer
FPA (floating point accelerator) 17
  to 18, 259, 444 to 446
Frame Pointer (FP; R13) 11
  on VAX-11/750 CPU 139, 141 to 142
  on VAX-11/780 CPU 264, 269 to 272, 274
free queue (FREEQ) 416
front panel, on VAX-11/750 104, 107 to 109
FUBAR (Failed UNIBUS Address Registers) 373

gate array technology 131, 226
general registers 11, 268 to 270
global sections 435 to 437
GOTO statement 10

HALT command 110, 117
hardware 7 to 18
  RAMP features on 449 to 454
  reliability, availability, maintainability features and 226 to 229
  on VAX-11/750 CPU 130 to 133
  on VAX-11/780 CPU 256 to 259
hardware process control block 286
HELP commands 3
high performance floating point accelerator (FPA) 17 to 18, 259, 444 to 446
high resolution interval timer 450, 453
ICCS (Interval Clock Control/Status Register) 200 to 201, 443 to 444
ICR (Interval Count Register) 200, 442, 443
ID (Internal Data) Bus 242, 243
  registers for 485 to 498
illegal characters, in console command language 111 to 112
images 134, 263
immediate (autoincrement) mode 141, 270
INDEX instruction 43
index registers
  in VAX-11/750 CPU 136 to 139
  in VAX-11/780 CPU 268, 269
information lines 292 to 301
INITIALIZE command 110, 117
initializing, see bootstrapping
input/output, see I/O
input queue (INPTQ) 416, 418, 419, 424
INSQHI instruction 84
INSQTI instruction 84
installation 3 to 4
INSTALL utility 432
instruction addressing 5
instruction buffers 12, 133
instructions 461 to 475
  in console command language 112 to 119
  native instruction set 8 to 11, 263 to 264
  privileged 88 to 99, 278 to 280
  process structure 26 to 31
  special 227, 450, 452
  suspended 51 to 52
instruction set extensions 210 to 211
instruction sets
  on VAX-11/750 135 to 138
  on VAX-11/780 263 to 264

584
index

integer data 9, 136, 264
integer divide by zero trap 42 to 43
integer overflow trap 42
integral fault detection 227 to 228, 450, 452 to 453
interconnects
  in VAX-11/780 systems 409 to 437
  see also Synchronous Backplane Interconnect
interfaces
  for console terminal communications 105
  DR780 409 to 414
  I/O 15 to 17
  MASSBUS adapter 175, 389
  programming 414 to 417
  Synchronous Backplane Interconnect 289 to 309
  VAX-11/750 UNIBUS 157 to 172
  VAX-11/780 UNIBUS 327 to 386
interleaving 323
interlock cycles 316 to 317
interlock read cycle 316 to 317
interlock read masked command 307
interlock read masked cycle 316
interlocks
  MA780 and 430
  synchronization and 84
interlock write cycle 317
interlock write masked command 307
interlock write masked cycle 317
Internal Data (ID) Bus 242, 243
  registers for 485 to 498
internal processor (VAX-11/750) registers 499 to 508
internal registers (VAX-11/780 MASSBUS) 396 to 405
interrupt control field 424
interrupt priority level register (IPLR) 38
interrupt priority levels (IPL) 33 to 35, 39, 40, 86, 283
  UNIBUS and 160, 161, 171
interrupt request lines 303
interrupts 33 to 57
  on DR780 414, 424
  emulation and 218
  process structure 26
  system architecture and 86
  VAX-11/750 UNIBUS and 161
  on VAX-11/780 systems 275 to 276, 282 to 283
  VAX-11/780 UNIBUS and 354 to 357
interrupt service routine (ISR) 33, 283
interrupt stack (IS) 33, 48 to 50, 283
interrupt stack not valid halt 40
interrupt summary tag 299 to 300
Interval Clock Control/Status Register (ICCS) 200 to 201, 443 to 444
interval clocks 199, 228, 258, 441 to 444
Interval Count Register (ICR) 200, 442, 443
I/O
  subsystems for 15 to 18
  system architecture for 87
I/O processing 283 to 286
I/O space 283 to 286
  restrictions on 513 to 514
IPL, see interrupt priority levels
IPL 2 interrupts 25, 26
IPL 3 interrupts 26
IPLR (interrupt priority level register) 38
IS (interrupt stack) 33, 48 to 50, 283
ISR (interrupt service routine) 33, 283
JSR/PUSH micro-order 219
kernel executive 193
kernel mode 63, 226, 278
kernel stack 49
kernel stack not valid abort 39 to 40
Kernel Stack Pointer (KSP) 22

languages 7
LDPCTX (Load Process Context) instruction 27 to 29, 89, 95 to 96, 280, 286
translation buffers and 74
length violations 64 to 65
limit checking traps 227, 451
LIT field 216
LIT/LONLIT micro-order 215 to 216
Load Process Context instruction see LDPCTX instruction
logical names 433, 434
longword access enable (LWAE) bit 350
longword-aligned 32-bit random access mode 350 to 351
longword integers 5
LONGWORD WRITE cycles 150
LONLIT (Long Literal) register 214 to 216
loops 10
LSI-11 bus (Q bus) 242 to 243
LSI-11 microprocessor 240 to 244
LSI Schottky logic 1
LWAE (longword access enable) bit 350

MA780 multiport memory 409, 426 to 437
Machine Check Error Summary Register (MCESR) 202, 228
machine check exceptions 40
Machine Check Status Register (MCSR) 202 to 204, 228

mailboxes 433, 435
main memory subsystems 14
in VAX-11/750 systems 147 to 155
in VAX-11/780 systems 311 to 324
maintainability features
of VAX-11/750 systems 223 to 236
of VAX-11/780 systems 449 to 454
maintenance 3 to 4
maintenance features
of VAX-11/750 systems 227 to 228, 236
of VAX-11/780 systems 450, 452 to 453
maintenance registers 228, 450, 452 to 453
Map Enable Register (MAPEN) 74, 206
Map Global Section ($MGBLSC) 436
mapping 59 to 60
virtual to physical page 281 to 282
map registers
on VAX-11/750 MASSBUS 189
on VAX-11/780 CPU 276
on VAX-11/780 MASSBUS 405
on VAX-11/780 UNIBUS 341, 347, 349, 378 to 381
mask field 301
MASSBUS 15 to 17
on VAX-11/750 systems 175 to 190
on VAX-11/780 389 to 407
MASSBUS adapter (MBA) interconnects and 409
on VAX-11/750 systems 175 to 189
on VAX-11/780 systems 389 to 390, 392 to 405
master/slave relationships 159, 328
MBA, see MASSBUS adapter
<table>
<thead>
<tr>
<th>Index</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MBA Byte Counter (BCR)</td>
<td>186, 403</td>
</tr>
<tr>
<td>MBA Command Address Register (CAR)</td>
<td>188 to 189, 405</td>
</tr>
<tr>
<td>MBA Configuration/Status Register (CSR)</td>
<td>396 to 398</td>
</tr>
<tr>
<td>MBA Control Register (CR)</td>
<td>180 to 182, 398 to 399</td>
</tr>
<tr>
<td>MBA Diagnostic Register (DR)</td>
<td>186 to 188, 403 to 405</td>
</tr>
<tr>
<td>MBA External Registers</td>
<td>189, 405</td>
</tr>
<tr>
<td>MBA Map Registers</td>
<td>189, 405</td>
</tr>
<tr>
<td>MBA Status Register (SR)</td>
<td>182 to 185, 399 to 402</td>
</tr>
<tr>
<td>MBA Virtual Address Register (VAR)</td>
<td>185 to 186, 402 to 403</td>
</tr>
<tr>
<td>MBRK (Microprogram Breakpoint Address Register)</td>
<td>447</td>
</tr>
<tr>
<td>MBUS</td>
<td>213 to 215</td>
</tr>
<tr>
<td>MCESR (Machine Check Error Summary Register)</td>
<td>202, 228</td>
</tr>
<tr>
<td>MCSR (Machine Check Status Register)</td>
<td>202 to 204, 228</td>
</tr>
<tr>
<td>mean time between failure rate (MTBF)</td>
<td>449</td>
</tr>
<tr>
<td>mean time to repair (MTTR)</td>
<td>449</td>
</tr>
<tr>
<td>memory</td>
<td>14</td>
</tr>
<tr>
<td>main subsystems for</td>
<td>18</td>
</tr>
<tr>
<td>optional</td>
<td>18</td>
</tr>
<tr>
<td>shared, MA780 and</td>
<td>432 to 437</td>
</tr>
<tr>
<td>shared data in</td>
<td>78 to 80, 83 to 84</td>
</tr>
<tr>
<td>in VAX-11/750 systems</td>
<td>147 to 155</td>
</tr>
<tr>
<td>in VAX-11/780 systems</td>
<td>311 to 324</td>
</tr>
<tr>
<td>VAX-11/780 UNIBUS data transfers to and from</td>
<td>345 to 347</td>
</tr>
<tr>
<td>on VAX/VMS</td>
<td>6</td>
</tr>
<tr>
<td>virtual</td>
<td>276</td>
</tr>
<tr>
<td>memory caches</td>
<td>11 to 12, 84 to 85</td>
</tr>
<tr>
<td>disabling of</td>
<td>450, 453</td>
</tr>
<tr>
<td>in VAX-11/750 CPU</td>
<td>132 to 133</td>
</tr>
<tr>
<td>in VAX-11/780 CPU</td>
<td>257</td>
</tr>
<tr>
<td>memory configuration register</td>
<td>317 to 319</td>
</tr>
<tr>
<td>memory management</td>
<td>12, 59 to 80</td>
</tr>
<tr>
<td>disabling of</td>
<td>450, 453</td>
</tr>
<tr>
<td>in VAX-11/780 systems</td>
<td>280 to 281</td>
</tr>
<tr>
<td>memory management control</td>
<td>74 to 75</td>
</tr>
<tr>
<td>Memory Mapping Enable (MME)</td>
<td>65</td>
</tr>
<tr>
<td>metal oxide semiconductor (MOS)</td>
<td>311, 313</td>
</tr>
<tr>
<td>memory</td>
<td>65</td>
</tr>
<tr>
<td>MFPR, see Move from Processor Register instruction</td>
<td>436</td>
</tr>
<tr>
<td>$MGBLSC (Map Global Section)</td>
<td>436</td>
</tr>
<tr>
<td>microarchitecture of VAX-11/750 systems</td>
<td>213 to 216</td>
</tr>
<tr>
<td>micro control store</td>
<td>446 to 447</td>
</tr>
<tr>
<td>microinstructions</td>
<td>210</td>
</tr>
<tr>
<td>Microprogram Breakpoint Address Register (MBRK)</td>
<td>447</td>
</tr>
<tr>
<td>microprogramming</td>
<td>209 to 220</td>
</tr>
<tr>
<td>microprograms</td>
<td>210</td>
</tr>
<tr>
<td>micro-routine error messages</td>
<td>250</td>
</tr>
<tr>
<td>microsequencers</td>
<td>216</td>
</tr>
<tr>
<td>microverify code</td>
<td>228</td>
</tr>
<tr>
<td>microverify routines</td>
<td>233 to 234</td>
</tr>
<tr>
<td>MME (Memory Mapping Enable)</td>
<td>65</td>
</tr>
<tr>
<td>modes, access</td>
<td>63, 76 to 77</td>
</tr>
<tr>
<td>changing</td>
<td>88</td>
</tr>
<tr>
<td>hierarchical</td>
<td>226, 449, 451</td>
</tr>
<tr>
<td>processor</td>
<td>277 to 278</td>
</tr>
</tbody>
</table>
modes, addressing 11
  in VAX-11/750 systems 136 to 138
  in VAX-11/780 systems 264 to 266, 269 to 270
modes, console 104 to 106
MOS (metal oxide semiconductor) memory 311, 313
Move from Processor Register (MFPR) instruction 89, 96 to 99, 193, 280, 439
Move to Processor Register (MTPR) instruction 89, 96 to 99, 193, 280, 439
MTBF (mean time between failure rate) 449
MTPR, see Move to Processor Register instruction
MTTR (mean time to repair) 449
multiprocessor systems, interrupt priority levels on 33
multiprogramming, on VAX-11/780 systems 274 to 275
multiuser systems, VAX-11/750 in 1
MUX field 214
native mode 135 to 136, 263
native mode instruction set 8 to 11, 263 to 264
NEXT command 117
next interval count register (NICR) 200, 442, 443
NEXUSs 289 to 292, 333
  UNIBUS adapter register space and 358 to 360
NICR (next interval count register) 200, 442, 443
non-existent memory (NXM) bus errors 164
Non-Processor Grant (NPG) line 328, 329
non-processor request transfers (NPR) 160, 161, 163, 328, 329
NOP command packet 424
NPG (Non-Processor Grant) line 328, 329
NPR (non-processor request transfers) 160, 161, 163, 328, 329
NXM (non-existent memory) bus errors 164
OFF-LOCAL-REMOTE switch 104, 107
Offset Bit 166 to 168
on-line diagnostics
  on VAX-11/750 systems 232 to 233
  see also diagnostics
operand specifier notation 509 to 512
  for process structure instructions 26 to 27
operation code (opcode) 264
operating states 104
operating system consistency checks 234
operating systems, VAX/VMS 2 to 4, 6 to 7, 434, 436
options
  battery backup 149, 324
  customer writable control store 259
  floating point accelerator 258 to 259
  hardware 17 to 18
  installation of 3 to 4
  MA780 409, 426 to 437
  microprogramming 209
  Remote Diagnosis Module 13, 106, 109, 226 to 227, 230 to 233
organization of VAX systems, microprogramming and 212 to 213
P0 Base Register (POBR) 23, 70
P0 Length Register (POLR) 23, 70
Index

P0 page table (P0PT) 70
P0 space 70 to 72
P1 Base Register (P1BR) 24, 72 to 73
P1 Length Register (P1LR) 24, 72
P1 page table (P1PT) 72
P1 space 72 to 74
packaging, for VAX-11/750 systems 229 to 230
packed decimal data 9, 136, 264
Page Frame Number (PFN) 164, 166
pages 60
mapping of, virtual to physical 281 to 282
protection for 62 to 64, 67
page table entries (PTE) 66 to 67
memory management faults and 76
process space and 69, 70, 72
protection codes in 64
shared sections in process space and 78 to 79
system page tables and 68
virtual to physical page mapping and 282
parallel processing 429
parameters, memory management 75 to 76
parity bits (check bits) 317
parity checks 228, 450, 453
parity field 293
PC, see Program Counter
PCB (Process Control Block) 21 to 24, 26
PCBB (Process Control Block Base Register) 21, 286
PDP-11 Compatibility Mode 3, 4
PDP-11 systems
UNIBUS on 157
VAX compatibility with 5
Performance Monitor Enable (PME) 24, 25
peripheral devices
interconnects for 409
UNIBUS for 157 to 160, 327 to 328
per-process space 268
PFN (Page Frame Number) 164, 166
physical addresses 276, 280
translation of virtual addresses to 481 to 483
physical address space 280 to 281
in VAX-11/780 main memory subsystem 313
physical pages, mapping of virtual pages to 281 to 282
pipeline (sequential) processing 429
PME (Performance Monitor Enable) 24, 25
position independent code 270
power-fail recovery code 225
power failures
automatic recoveries after 3, 4
battery backups for 149
 caches and 85
Dead signal function for 303 to 304
in DR780 414
interrupts for 36
POWER ON ACTION switch and 104
VAX-11/780 UNIBUS and 381 to 382
POWER ON ACTION switch 104, 108
prefetch instruction buffers 258
priorities
of bus request levels 328
CPU 6
for interrupts 33 to 35, 39, 40, 86, 283
in UNIBUS 159 to 162, 329
priority dispatching 275 to 276
privileged instructions 88 to 99, 278 to 280
privileged registers 24 to 25
    in VAX-11/750 systems 193 to 206
    in VAX-11/780 systems 439 to 447
privileges
    for memory access 59
    modes of 63, 76 to 77, 226
PROBE instructions 77, 88, 91 to 93, 279
PROBER instruction 88, 279
PROBEW instruction 88, 279
procedures
    VAX-11/750 139 to 140
    VAX-11/780 266 to 267
process address space 60
process context 286
process context stack pointers 49
Process Control Block (PCB) 21 to 24, 26
Process Control Block Base (PCBB) Register 21, 286
process control services 6
processes 6, 60
    definition of 134, 275
    structure of 21 to 31
process level 34
processor clocks
    VAX-11/750 133
    VAX-11/780 258
processors, see CPU
processor states 211 to 212
Processor Status Longword (PSL) 21, 23
    arithmetic traps and 227
    interrupts and exceptions on 33 to 35, 38, 50
    mode changes and 88
    on VAX-11/780 CPU 276, 277
Processor Status Word on VAX-11/750 CPU 134, 143 to 144
    on VAX-11/780 CPU 273 to 274, 276
process page tables 69 to 70
process space 61
    address translation for 69 to 74
    shared sections in 78 to 79
    on VAX-11/750 134 to 135
process structure instructions 26 to 31
process structure interrupts 26
process virtual address space 263, 268
Program Counter (PC; R15) 11, 21, 23, 105
    interrupts and exceptions on 34
    on VAX-11/750 CPU 139 to 141
    on VAX-11/780 CPU 264, 269, 270
program development 6 to 7
program I/O (system terminal) mode 104, 105, 240 to 242
programming
    DR780 and 418 to 424
    microprogramming, on VAX-11/750 systems 209 to 220
    on VAX-11/750 systems 134
    on VAX-11/780 systems 263 to 286
programming interfaces 414 to 417
program region 135, 268
programs 134, 263
prompt symbol (>>>) 105
protected instructions 278 to 280
protection 6
    for pages, in memory management 60, 62, 67
protection code 63 to 64
protocol checks 453
PSL, see Processor Status Longword
Index

PTE, see page table entries 350, 354
purge operations 350, 354
Q bus (LSI-11 bus) 242 to 243
QIO driver 415
quadword integers 5
queue retry macro 420

RAM (random access memory) 311
RAMP (reliability, availability, maintainability program) 449 to 454
random access memory (RAM) 311
RBUS 213 to 215
RDM, see Remote Diagnosis Module
RDM console mode 105, 106
READ cycles 149 to 150, 314 to 315
read data tag 294
Read Mask Function 304 to 305
read-only memories, see ROMs
realization 212
real-time (interval) clocks 199, 228, 258, 441 to 444
reconfiguration, on VAX-11/750 systems 235 to 236
Record Management Services (RMS) 7
recoveries, automatic 4
register deferred mode 141, 270
registers 11
during bootstrapping 122
for Internal Data Bus 485 to 498
for interrupts 36 to 38
I/O, constraints on 87
maintenance 228, 450, 452 to 453
memory configuration 317 to 323
for memory management 74
in microprogramming processor state 212
privileged 24 to 25
privileged, in VAX-11/750 systems 193 to 206
privileged instructions and 89, 96 to 99
in Process Control Block 23 to 24
in process structure 21
during restarts 85
scratch pad 214 to 217
stack, accessing 50 to 51
in VAX-11/750 CPU 139 to 142
for VAX-11/750 internal processor 499 to 508
in VAX-11/750 main memory subsystem 151 to 155
in VAX-11/750 MASSBUS adapter 177 to 189
in VAX-11/750 UNIBUS 170 to 171
in VAX-11/780 CPU 255, 264 to 266, 268 to 274
in VAX-11/780 Internal Data (ID) bus 485 to 498
in VAX-11/780 MASSBUS adapter 394 to 405
in VAX-11/780 UNIBUS 329 to 330
in VAX-11/780 UNIBUS adapter 360 to 381
REI (Return from Exception or Interrupt) instruction 25, 26, 55 to 57, 93 to 94, 279 to 280, 283
privileged instructions and 88 to 89
reliability of VAX-11/750 systems 223 to 236
of VAX-11/780 systems 449 to 454
reliability, availability, maintainability program (RAMP) 449 to 454
Remote Diagnosis Module (RDM) 13, 18, 106, 226 to 227, 230 to 233
indicators for 109
REMQH1 instruction 84
<table>
<thead>
<tr>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>REMQTI instruction  84</td>
</tr>
<tr>
<td>request (REQ) lines  303</td>
</tr>
<tr>
<td>reserved operand traps  227, 452</td>
</tr>
<tr>
<td>reserved tags  300</td>
</tr>
<tr>
<td>RESET switch  105, 108</td>
</tr>
<tr>
<td>response lines  302</td>
</tr>
<tr>
<td>restart parameter block (RPB)  124 to 125</td>
</tr>
<tr>
<td>restarts  85 to 86</td>
</tr>
<tr>
<td>POWER ON ACTION switch  for  104</td>
</tr>
<tr>
<td>Return from Exception or Interrupt instruction, see REI instruction</td>
</tr>
<tr>
<td>RETURN instruction  452</td>
</tr>
<tr>
<td>RMS (Record Management Services)  7</td>
</tr>
<tr>
<td>ROMs (read-only memories)  for bootstrapping  148 to 149, 323 to 324</td>
</tr>
<tr>
<td>microprogramming and  209, 210 in VAX-11/780 Console Interface Board  245</td>
</tr>
<tr>
<td>RPB (Restart Parameter Block)  124 to 125</td>
</tr>
<tr>
<td>RSRC  216</td>
</tr>
<tr>
<td>RX01 floppy disk systems  4, 13</td>
</tr>
<tr>
<td>RXCS (console receive control/status register)  194, 440</td>
</tr>
<tr>
<td>RXDB (console receive data buffer register)  194 to 195, 440</td>
</tr>
<tr>
<td>SBI UNJAM  382</td>
</tr>
<tr>
<td>SBR (System Base Register)  68</td>
</tr>
<tr>
<td>SCB, see System Control Block</td>
</tr>
<tr>
<td>SCBB, see System Control Block Base</td>
</tr>
<tr>
<td>scheduling, on CPU  6</td>
</tr>
<tr>
<td>Schottky logic  1</td>
</tr>
<tr>
<td>scratch pad registers  214 to 217</td>
</tr>
<tr>
<td>selective cache invalidation  430 to 431</td>
</tr>
<tr>
<td>SELF TEST command  110</td>
</tr>
<tr>
<td>semantics, of console command language  111</td>
</tr>
<tr>
<td>sequential (pipeline) processing  429</td>
</tr>
<tr>
<td>serious system failures  39 to 40</td>
</tr>
<tr>
<td>shared memory, MA780 and  430, 432 to 437</td>
</tr>
<tr>
<td>sharing  78 to 80 of data  83 to 84</td>
</tr>
<tr>
<td>SID (system identification registers)  193, 228, 439 to 440, 452</td>
</tr>
<tr>
<td>SINGLE STEP command  110</td>
</tr>
<tr>
<td>SIRR (Software Interrupt Request Register)  34, 37 to 38</td>
</tr>
<tr>
<td>SISR (Software Interrupt Summary Register)  36 to 37</td>
</tr>
<tr>
<td>SLR (System Length Register)  68</td>
</tr>
<tr>
<td>software  6 to 7 compatibility of  2 development of  4</td>
</tr>
<tr>
<td>interrupts generated by  36 for memory management  60 to 61 processor interrupt priority levels of  34</td>
</tr>
<tr>
<td>reliability, availability and maintainability features of  234 to 236</td>
</tr>
<tr>
<td>Software Interrupt Request Register (SIRR)  34, 37 to 38</td>
</tr>
<tr>
<td>Software Interrupt Summary Register (SISR)  36 to 37</td>
</tr>
</tbody>
</table>

592
Index

software process control blocks 286
source/destination identity field 301
SP, see Stack Pointer
special instructions 227, 450, 452
SPT, see system page tables
SR (MBA Status Register) 182 to 185, 399 to 402
SSP (Supervisor Stack Pointer) 23
stack (call) frames 140, 142, 267
Stack Pointer (SP; R14) 11, 48
on VAX-11/750 CPU 139 to 142
on VAX-11/780 CPU 264, 266, 269 to 272
stack registers 50 to 51
stacks 11, 48 to 49
in VAX-11/750 CPU 139 to 140
in VAX-11/780 CPU 266 to 267
START command 110, 117
status registers
in VAX-11/750 MASSBUS adapter 182 to 185
in VAX-11/750 main memory subsystem 151 to 155
in VAX-11/750 UNIBUS 170 to 171
in VAX-11/780 MASSBUS adapter 399 to 402
in VAX-11/780 UNIBUS 366 to 370
storage media 4
subroutines
VAX-11/750 139 to 140
VAX-11/780 266 to 267
subscript range trap 43
supervisor mode 63, 226, 278
Supervisor Stack Pointer (SSP) 23
support routine interfaces 415
SVPCTX (Save Process Context) instruction 26, 29 to 30, 89, 95 to 96, 280
symbols, in console command language 111
synchronization, data sharing and 83 to 84
Synchronous Backplane Interconnect (SBI) 15, 289 to 309
access to UNIBUS Address Space and 334 to 339
address space of, UNIBUS access to 339 to 341
interconnects and 409
MASSBUS adapter and 389, 396
UNIBUS and 328
UNIBUS adapter and 332 to 333
UNIBUS adapter registers addressable by 360 to 381
VAX-11/780 memory controller and 311
synchronous data path (bus) 390
syntactic error messages 249
syntax of console command language 111, 245 to 246
system address space 60
System Base Register (SBR) 68
System Control Block (SCB) 44 to 48, 57
arithmetic exceptions in 41
interrupt vectors in 36, 283
memory management faults in 75
UNIBUS and 171
System Control Block Base (SCBB) 44 to 45
UNIBUS and 171
System Control Block Base 44, 282 to 283
system failures 39 to 40
see also power failures
system identification registers (SID) 193, 228, 439 to 440, 452
System Length Register (SLR) 68
system management functions 225
system managers 432
System Page Table Length Registers (SLR) 70
system page tables (SPT) 68
size of 77 to 78
593
Index

system services 6
system space 61 to 62
  address translation for 67 to 68
  shared sections in 79 to 80
  on VAX-11/750 134, 135
  on VAX-11/780 268
system terminal (program I/O)
  mode 104, 105
system verification 232
system-wide context 283

tag field 293 to 300
tape drives, see TU58 tape cartridges
TB (Translation Buffer) Register 206, 217
TBDR (Translation Buffer Group Disable Register) 204
TBIA (Translation Buffer Invalidate All Register) 75
TBIS (Translation Buffer Invalidate Single Register) 75
terminals
  VAX-11/750 console subsystem as 12 to 13, 104 to 105
  VAX-11/780 Console Interface Board as 241
termination queue (TERMQ) 416, 418, 424
TEST command 118
time-of-year clock register (TODR) 442
time-of-years clocks 12, 133, 199, 258, 441 to 442
TODR (time-of-year clock register) 442
T0 Internal Data register 240, 243, 244
trace faults 145
track offset retry hardware 229, 450, 454
translation buffer 74 to 75
Translation Buffer Group Disable Register (TBDR) 204
Translation Buffer Invalidate All Register (TBIA) 75
Translation Buffer Invalidate Single Register (TBIS) 75
Translation Buffer Parity Register 453
Translation Buffer Register (TB) 206, 217
translation not valid faults 75
traps 144, 227
  arithmetic 42 to 44, 451
  reserved operand 452
TU58 tape cartridges 4, 13, 103 to 104, 119, 226
  boot ROM for 148
  registers for 196 to 198
two-way set associative memory
  caches 257
TXCS (console transmit control/status register) 195 to 196, 441
TXDB (console transmit data buffer register) 441
UACR (VAX-11/780 UNIBUS control register) 363 to 366
UBA (UNIBUS Address Space) 164, 334 to 336
UBASR (UNIBUS adapter interrupt service routine) 357
UCS, see user control store
UETP (User Environment Test Package) 232
UNIBUS 15 to 16
  on VAX-11/750 systems 157 to 172
  on VAX-11/780 systems 327 to 386
UNIBUS adapter
  interconnects and 409
Index

on VAX-11/750 158, 159, 163 to 167
on VAX-11/780 327, 332 to 333, 336, 339 to 340, 342 to 354, 381
UNIBUS adapter interrupt service routine (UBASR) 357
UNIBUS adapter registers 360 to 381
UNIBUS adapter (NEXUS) register space 358 to 360
UNIBUS Address Space (UBA) 164, 334 to 336
UNIBUS Map 164 to 168
UNIBUS signal line definitions 330 to 332
UNIBUS space 87, 286
uniform exception handling 234
UNJAM function 304
Update Section File Disk ($UPDSEC) system service 436
urgent interrupts 36
USAR (VAX-11/780 UNIBUS status register) 366 to 370
user control store (UCS) 18
microprogramming and 209, 216, 219 to 220
on VAX-11/750 CPU 133
user control store address register (WCSA) 446, 447
User Environment Test Package (UETP) 232
user identification codes 6
user microprogramming 218 to 219
user mode 63, 226, 278
User Stack Pointer (USP) 23
user stacks 140
user writable control store (WCS) 259, 446
USP (User Stack Pointer) 23

Valid Bit 166, 167

VAR (MBA Virtual Address Register) 185 to 186, 402 to 403
variable bit field data 264
variable-length bit fields 5
VAX systems 1 to 5
architecture of 83 to 97
hardware of 7 to 18
registers on 97 to 99
software on 6 to 7
see also VAX-11/750 systems;
VAX-11/780 systems
VAX-11/750 MASSBUS adapter (MBA) 175 to 189
VAX-11/750 systems 1, 2, 4
Console subsystem on 103 to 126
CPU in 129 to 145
CPU technical specifications for 515 to 521
internal processor registers for 499 to 508
I/O subsystem on 15
main memory subsystem on 14, 147 to 155
MASSBUS on 175 to 190
microprogramming on 209 to 220
privileged registers on 193 to 206
process space address translation on 72, 74
registers specific to 98 to 99
reliability, availability, and maintainability features of 223 to 236
system space address translation on 68
UNIBUS on 157 to 172
VAX-11/750 UNIBUS adapter 158, 159, 163 to 167
VAX-11/780 floating point accelerator (FPA) 17 to 18, 259, 444 to 446
VAX-11/780 MASSBUS adapter (MBA) 389 to 390, 392 to 405
interconnects and 409
VAX-11/780 micro control store 446 to 447
VAX-11/780 systems 1, 2, 4

595
Index

Console subsystem on 239 to 252
CPU in 255 to 286
CPU technical specifications for 523 to 527
interconnects in 409 to 437
Internal Data Bus for, registers on 485 to 498
I/O subsystem on 15
main memory subsystem on 14, 311 to 324
MASSBUS on 389 to 407
privileged registers in 439 to 447
process space address translation on 71, 73, 74
registers specific to reliability, availability, and maintainability features of 449 to 454
Synchronous Backplane Interconnect on 289 to 309
system space address translation on 68
UNIBUS on 327 to 386
VAX-11/780 UNIBUS adapter 327, 332 to 333, 336, 339 to 340
data transfer paths in 342 to 354
interconnects and power up sequence and 381
VAX Console Subsystem 3 to 4
on VAX-11/750 systems 103 to 126, 226 to 227
on VAX-11/780 systems 239 to 252
VAX/VMS operating system 2 to 4, 6 to 7
shared memories in 434, 436
V bus 243
vectors 144, 282 to 283
in System Control Block 45 to 48
verify checking hardware 229, 450, 454
violations
access control 75
length 64 to 65
virtual addresses 59 to 60, 62 to 63, 255, 276
translation of physical addresses to 481 to 483
Virtual Address Register (VAR) 185 to 186, 402 to 403
virtual address space 60 to 63
on VAX-11/750 systems 130, 134 to 135
on VAX-11/780 systems 263, 268, 280 to 282
virtual memory 276
virtual pages 60
mapping of physical pages to 281 to 282
volumes 7
warm starts 124 to 126
watchdog timer 450, 453
WBUS 213
WCS (customer writable control store) 259, 446
WCSA (writable control store address register) 446, 447
WCSD (writable control store data register) 446, 447
WCTRL field 216
WDCS (writable diagnostic control store) 258
Wilkes, M.V. 209
word integers 5
WORD WRITE cycle 150 to 151
writable control store (WCS) 259, 446
writable control store address register (WCSA) 446, 447
writable control store data register (WCSD) 446, 447
writable diagnostic control store (WDCS) 258
WRITE cycles 150 to 151
write data tag 299
write masked cycles 313, 315
Index

write masked function 305  XB execution buffer 217
write-verify checking hardware 229, XFC (extended function
450, 454 instruction) 89, 99
VAX HARDWARE HANDBOOK
1980-81

Your comments and suggestions will help us in our continuous effort to improve the quality and usefulness of our handbooks.

What is your general reaction to this handbook? (format, accuracy, completeness, organization, etc.) ____________________________________________

____________________________________________________________________________________

____________________________________________________________________________________

____________________________________________________________________________________

What features are most useful? __________________________________________________________

____________________________________________________________________________________

____________________________________________________________________________________

____________________________________________________________________________________

Does the publication satisfy your needs? ____________________________________________________________________________________________

____________________________________________________________________________________

____________________________________________________________________________________

What errors have you found? ____________________________________________________________

____________________________________________________________________________________

____________________________________________________________________________________

____________________________________________________________________________________

Additional comments ________________________________________________________________

____________________________________________________________________________________

____________________________________________________________________________________

____________________________________________________________________________________

Name

____________________________________________________________________________________

Title

____________________________________________________________________________________

Company __________________________________________ Dept. __________________________

Address _________________________________________________________________

City __________________________ State ____________ Zip ____________

(staple here)
digital

HANDBOOK SERIES

Microcomputer Processor
Microcomputers and Memories
PDP-11 Processor
PDP-11 Software
Peripherals
Terminals and Communications
VAX Architecture
VAX Software
VAX Hardware