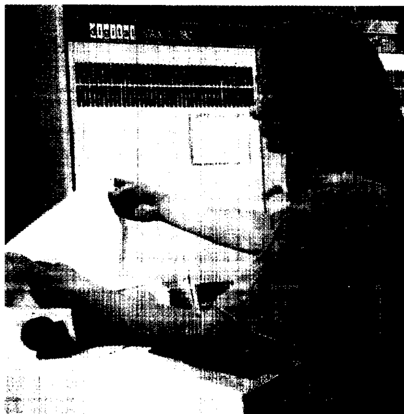
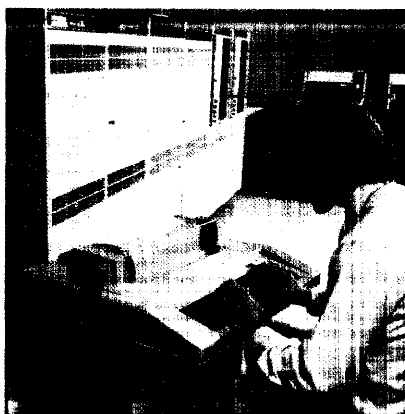
digital

VAX11/780 HARDWARE HANDBOOK 1979-80



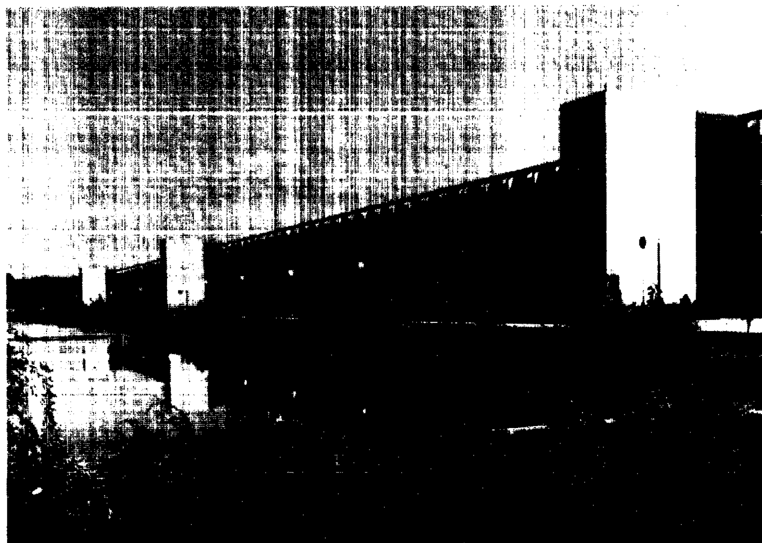
# VAX11 780



# HARDWARE HANDBOOK

digital

PRINTED IN USA EB 17835 18/79 050 02 44



## 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.

# VAX11 780

## HARDWARE HANDBOOK

digital

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 responsibility for any errors that may appear in this manual.

VAX, VMS, SBI, PDP, UNIBUS, MASSBUS
are trademarks of
Digital Equipment Corporation

This handbook was designed, produced and typeset
by DIGITAL's Sales Support Literature Group
using an In-house text-processing system
operating on a DECSYSTEM-20.

Copyright © 1979, by Digital Equipment Corporation

# CONTENTS


**CHAPTER 1 VAX-11/780 HARWARE INTRODUCTION**

SYSTEM INTRODUCTION . . . . . 1
THE VAX-11/780 CENTRAL PROCESSING UNIT . . . . . 4
THE CONSOLE SUBSYSTEM . . . . . 10
THE MEMORY SUBSYSTEM . . . . . 11
THE INPUT/OUTPUT SUBSYTEMS . . . . . 12

**CHAPTER 2 CONSOLE SUBSYSTEM**

INTRODUCTION . . . . . 17
CONSOLE INTERFACE BOARD . . . . . 18
CONSOLE BUS STRUCTURE . . . . . 20
CONSOLE/VAX-11 INTERACTION . . . . . 22
READ ONLY MEMORY (ROM) . . . . . 23
THE CONSOLE COMMAND LANGUAGE . . . . . 23
CONSOLE ERROR MESSAGES . . . . . 27

**CHAPTER 3 CENTRAL PROCESSOR**

INTRODUCTION . . . . . 33
HARDWARE ELEMENTS . . . . . 34
PROCESSOR OPERATION . . . . . 37
USER PROGRAMMING CONCEPTS . . . . . 41
USER PROGRAMMING ENVIRONMENT . . . . . 45
SYSTEM PROGRAMMING CONCEPTS . . . . . 52
SYSTEM PROGRAMMING ENVIRONMENT . . . . . 54

**CHAPTER 4 PROCESS STRUCTURE**

PROCESS DEFINITION . . . . . 67
PROCESS CONTEXT . . . . . 67
ASYNCHRONOUS SYSTEM TRAPS (AST) . . . . . 72
PROCESS STRUCTURE INTERRUPTS . . . . . 72
PROCESS STRUCTURE INSTRUCTIONS . . . . . 73
USAGE EXAMPLE . . . . . 77

**CHAPTER 5 EXCEPTIONS AND INTERRUPTS**

INTRODUCTION . . . . . 81
INTERRUPTS . . . . . 83
SERIOUS SYSTEM FAILURES . . . . . 87
SYSTEM CONTROL BLOCK (SCB) . . . . . 88
STACKS . . . . . 92
SERIALIZATION OF EXCEPTIONS AND
INTERRUPTIONS . . . . . 95
INITIATE EXCEPTION OR INTERRUPT . . . . . 96

**CHAPTER 6 MEMORY MANAGEMENT**

INTRODUCTION ........ 101
VIRTUAL ADDRESS SPACE ........ 102
VIRTUAL ADDRESS ........ 104
ADDRESS TRANSLATION ........ 105
ACCESS CONTROL ........ 107
SYSTEM SPACE ADDRESS TRANSLATION ........ 109
PROCESS SPACE ADDRESS TRANSLATION ........ 111
MEMORY MANAGEMENT CONTROL ........ 115
FAULTS AND PARAMETERS ........ 117
PRIVILEGED SERVICES AND ARGUMENT VALIDATION ........ 118
ISSUES ........ 119

**CHAPTER 7 SYNCHRONOUS BACKPLANE INTERCONNECT**

INTRODUCTION ........ 125
SBI STRUCTURE ........ 126
SYNCHRONOUS BACKPLANE INTERCONNECT THROUGHPUT ........ 145

**CHAPTER 8 MAIN MEMORY SUBSYSTEM**

INTRODUCTION ........ 147
MEMORY CONTROLLER ........ 148
BASIC MEMORY OPERATIONS ........ 149
INTERLOCK CYCLES ........ 152
ERROR CHECKING AND CORRECTION (ECC) ........ 153
MEMORY CONFIGURATION REGISTERS ........ 153
MEMORY INTERLEAVING ........ 159
ROM BOOTSTRAP ........ 160

**CHAPTER 9 UNIBUS SUBSYSTEM**

INTRODUCTION ........ 163
UNIBUS SUMMARY ........ 163
UNIBUS ADAPTER ........ 168
SBI ACCESS TO UNIBUS ADDRESS SPACE ........ 170
UNIBUS ACCESS TO THE SBI ADDRESS SPACE ........ 175
UNIBUS ADAPTER DATA TRANSFER PATHS ........ 178
INTERRUPTS ........ 190
UNIBUS ADAPTER (NEXUS) REGISTER SPACE ........ 194
SBI ADDRESSABLE UNIBUS ADAPTER REGISTERS ........ 196
POWER FAIL AND INITIALIZATION ........ 216

**CHAPTER 10 MASSBUS SUBSYSTEM**

INTRODUCTION ........ 223
MASSBUS ADAPTER OPERATION ........ 227

CONTROL PATH .......... 229
MBA ACCESS .......... 229
INTERNAL REGISTERS .......... 231

**CHAPTER 11 PRIVILEGED REGISTERS**

INTRODUCTION .......... 245
SYSTEM IDENTIFICATION REGISTERS (SID) .......... 245
CONSOLE TERMINAL REGISTERS .......... 246
CLOCK REGISTERS .......... 247
VAX-11/780 ACCELERATOR .......... 250
VAX-11/780 MICRO CONTROL STORE .......... 252

**CHAPTER 12 PRIVILEGE INSTRUCTIONS**

INTRODUCTION .......... 257

**CHAPTER 13 SYSTEM ARCHITECTURAL IMPLICATIONS**

INTRODUCTION .......... 269
DATA SHARING AND SYNCHRONIZATION .......... 269
CACHE .......... 270
RESTARTABILITY .......... 271
INTERRUPTS .......... 272
ERRORS .......... 272
I/O STRUCTURE .......... 272

**CHAPTER 14 RELIABILITY AVAILABILITY MAINTAINABILITY PROGRAM**

INTRODUCTION .......... 275
HARDWARE RAMP FEATURES .......... 275

**APPENDIX A**

COMMONLY USED MNEMONICS .......... 281

**APPENDIX B**

INSTRUCTION INDEX .......... 285

**APPENDIX C**

I/O SPACE RESTRICTIONS .......... 297

**APPENDIX D**

INTERNAL DATA (ID) BUS REGISTERS .......... 299

**APPENDIX E**

ADDRESS VALIDATION RULES .......... 313

**APPENDIX F**

VIRTUAL TO PHYSICAL ADDRESS TRANSLATION ....317

**APPENDIX G**

OPERAND SPECIFIER NOTATION ....................321

**GLOSSARY** ..........................................325

**INDEX** ...............................................353

# PREFACE

VAX-11/780 is DIGITAL's 32 bit extension to its 11 family of minicomputers. VAX-11/780 is a fully integrated computer system featuring state-of-the-art hardware technology coupled with a powerful virtual memory operating system, VAX/VMS (Virtual Address Extension/Virtual Memory system). VAX-11 hardware is characterized by its flexible instruction set, 32 bit capability, byte addressability, stack orientation, and highly efficient page-oriented memory management scheme. VAX/VMS is a high-performance operating system designed to complement the VAX-11 hardware. 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-11/780 is general purpose in nature, with the inherent capability to deal with a multitude of user environments. Designed to optimize throughput, the system enables enormous amounts of data to flow through it swiftly and unobstructed. Data transfers are accomplished via the 32 bit high speed Synchronous Backplane Interconnect (SBI). This hardware mechanism ties the system components together by providing a common point of interface including the communications protocol. The SBI interconnects the central processor, main memory (8 million bytes maximum), the UNIBUS subsystem and a mass storage subsystem comprising a maximum of 4.3 billion bytes. VAX-11/780 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-11 instruction set consists of 243 instructions including general-purpose, special function, commercial, and floating point. An optional high-speed floating point accelerator is available for user applications demanding superior floating point performance.

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-11 handbook documentation set is presented in three books:

- The VAX-11 Architecture Handbook introduces VAX-11 system architecture, addressing modes, and the native mode instruction set.
- The VAX-11 Software Handbook introduces the VAX/VMS virtual memory operating system, it operation, hardware interaction, data structures, features, and capbilities.

- This book, the VAX-11/780 Hardware Handbook, introduces the VAX-11 hardware elements, including the high-speed synchronous backplane interconnect, the central processor unit, intelligent console subsystem, MASSBUS and UNIBUS subsystems, main memory, and memory management.

This book describes the first of a series of VAX-11 Processors, therefore, certain chapters pertain to system wide architecture as well as processor specific information.

VAX 11/780

# CHAPTER 1
# VAX-11/780 HARDWARE INTRODUCTION

## SYSTEM INTRODUCTION

VAX-11/780 is a high-performance multiprogramming computer system. The system combines a 32 bit architecture, efficient memory management, and a virtual memory operating system to provide essentially unlimited program address space.

The processor's variable length instruction set and variety of data types, including decimal and character string, promote high bit efficiency. The processor hardware and instruction set specifically implement many high-level language constructs and operating system functions.

VAX-11/780 is a multiuser system for both program development and application system execution. It is a priority-scheduled, event-driven system: the assigned priority and activities of the processes in the system determine the level of service they need. Time-critical jobs receive service according to their priority and ability to execute, while the system manages allocation of CPU time and memory residency for normal executing processes.

VAX-11/780 is a highly reliable system. Built-in protection mechanisms in both the hardware and software ensure data integrity and system availability. On-line diagnostics and error detecting and logging verify system integrity. Many hardware and software features provide rapid diagnosis and automatic recovery should the power, hardware, or software fail.

The system is both flexible and extendable. The virtual memory operating system enables the programmer to write large programs that can execute in both small and large memory configurations without requiring the programmer to define overlays or later modify the program to take advantage of additional memory. The command language enables users to modify or extend their command repertoire easily, and allows applications to present their own command interface to users.

Appendix A contains a table of commonly used VAX-11/780 system mnemonics.

## Architecture Overview

VAX is the architecture for the VAX-11/780. The goals of the VAX architecture were to provide a significant enhancement to the virtual addressing capability of the PDP-11 series consistent with small code size, easy exploitation by higher-level languages, and a high degree of compatibility with the PDP-11 family of minicomputers. While VAX-11 is not strictly binary-compatible with the PDP-11 binary code, it does implement a compatibility mode which executes most PDP-11 instructions.

VAX-11 architecture is characterized by a powerful and complete instruction set of 244 basic instructions, a wide range of data types, an elegant set of addressing modes, full demand paging memory management, and a very large virtual address space of over four billion bytes. The native mode instruction set is found in Appendix B. I/O space restrictions on the use of the native mode instruction set is defined in Appendix C. Arithmetic and logical operations can be performed on byte integers (8 bits), word integers (16 bits), and 32-bit longword integers; plus, some instructions can perform operations on 64-bit quadword integers. 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 which can be performed on variable-length bit fields—a new data type for the 11 family.

Another significant feature of the VAX-11 architecture is that instruction addressing is virtually 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. Thus, operands and instructions can begin on any byte address, odd or even. The result of this flexibility is that higher-level language compilers, such as FORTRAN, can generate code that is optimally smaller in size, very efficient, and easy to manipulate in the compiler's data structures. This results in greater performance and lower memory utilization. The VAX/VMS operating system makes the hardware work together as one unit to provide the VAX-11/780 with its multiuser, multiprogramming, virtual memory capabilities. For further information concerning VAX architecture, refer to the ARCHITECTURE HANDBOOK.

## Software Overview

VAX/VMS is the general-purpose operating system for the VAX-11/780. It 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 the execution of 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-11/780 memory management features to provide swapping, paging, and protection and sharing of both code and data. Memory is allocated dynamically. Applications can control the amount of physical memory allocated to executing processes, the protection of pages, and swapping. 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. A process cannot access another process's private pages. VAX/VMS uses the four processor access modes to read and/or write-protect individual pages within a process. Protection of shared pages of memory, files, 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 programming languages commonly used in developing both scientific and commercial applications: FORTRAN, COBOL, 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 the VAX-11/780 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 ANS 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-11 software and the VAX/VMS operating system, refer to the VAX-11 SOFTWARE HANDBOOK.

**Hardware Overview**

The VAX-11/780 computer system consists of the central processing unit (with integral floating point and decimal and character string instructions), the console subsystem, the main memory subsystem, and the I/O subsystem. The I/O subsystem includes the Synchronous Backplane Interconnect (SBI), the UNIBUS and the MASSBUS subsystem.

The SBI is the internal connection path that links the CPU with its subsystems. The VAX-11/780 hardware configuration is illustrated in Figure 1-1.

## THE VAX-11/780 CENTRAL PROCESSING UNIT

The VAX-11/780 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 gigabytes of virtual address space, and provides a complete and powerful instruction set that includes integral decimal, character string, and floating point instructions. The VAX-11/780 includes an 8K byte cache, integral memory management, sixteen 32-bit general registers, 32 interrupt priority levels, and an intelligent console (LSI-11).

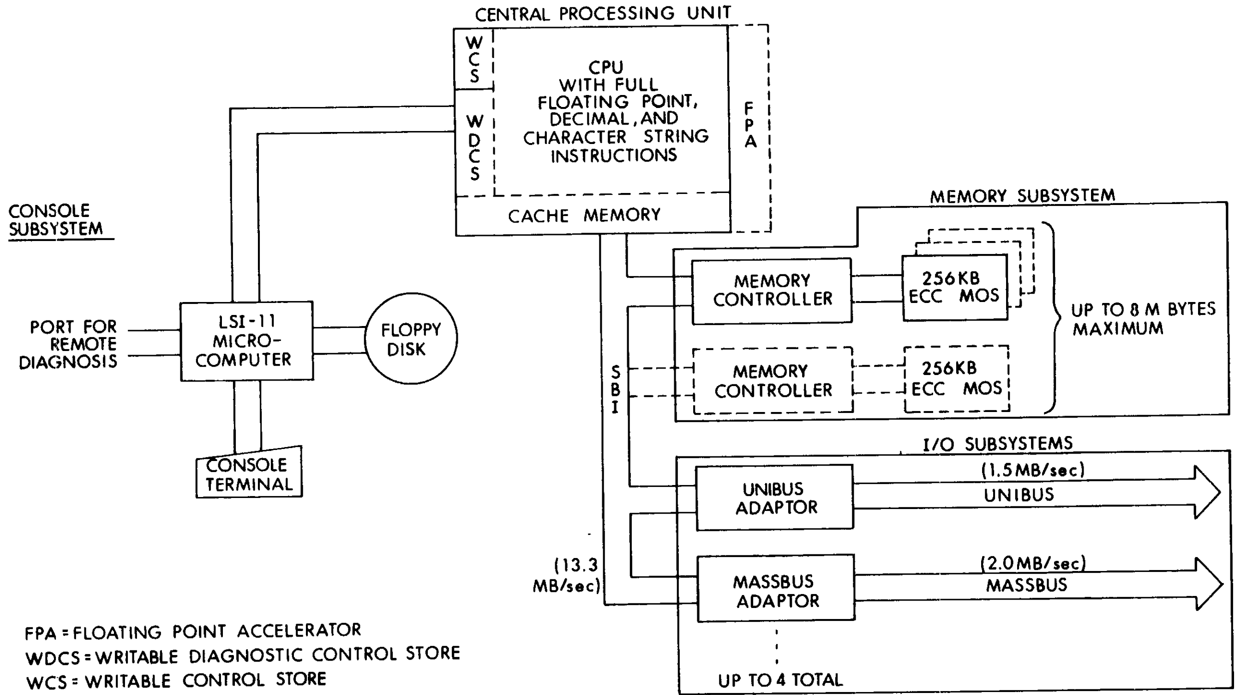



Figure 1-1 VAX-11/780 Hardware Configuration

Figure 1-2 illustrates the elements of the central processing unit.

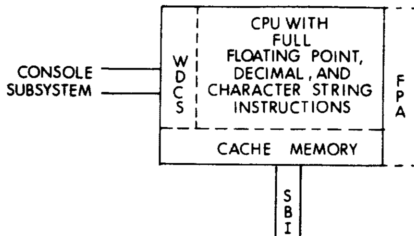



Figure 1-2 Central Processor

**Native Instruction Set** — The VAX-11 instructions are an extension of the PDP-11 instruction set. The VAX-11 instruction set provides 32-bit addressing, 32-bit I/O operations, and 32-bit arithmetic. Instructions can be grouped into related classes based on their function and use:

1) Instructions to manipulate arithmetic and logical data types—These include integer and floating point instructions, packed decimal instructions, character string instructions, and bit and field instructions.

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

| **Data Type** | **Represented As** |
|---|---|
| Integer | byte (8 bits), word (16 bits), longword (32 bits), quadword (64 bits) |
| Floating point | 4-byte floating or 8-byte double floating |
| Packed decimal | string or bytes (up to 31 decimal digits, 2 digits per byte) |
| Character string | string of bytes interpreted as character codes; a numeric string is a character string of codes for decimal numbers (up to 64K bytes) |
| Bits and bit-fields | field length is arbitrary and is defined by the programmer (0 to 32 bits in length) |

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 operating system.

3) Instructions to provide basic program flow control—These include branch, jump, and case instructions, subroutine call instructions, and procedure call instructions.

4) Instructions to quickly perform special operating system functions—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-11 architecture, special attention was given to implementing frequently-used, higher-level language constructs as single VAX-11 instructions. These instructions contribute to decreased program size and increased execution speed. Some of the constructs which have become single instructions on the VAX-11/780 include:

   - the FORTRAN-computed GOTO statement (translates into the VAX-11/780 CASE instruction)
   - the loop construct (e.g., add, compare, and branch translates into the VAX-11/780 ACB instruction)
   - an extensive CALL facility (which aligns the stack on a longword 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-11/780 instructions and data are of variable length. They need not be aligned on longword boundaries in physical memory, but may begin at any byte address (odd or even). Thus, instructions that do not require arguments use only one byte, while other instructions may

take two, three, or up to 30 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.

The VAX-11/780 processor offers nine addressing modes that use the general registers to identify the operand location. Seven of these are essentially the same as for the PDP-11:

| | |
|---|---|
| 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 new addressing modes are:

indexed
literal

Because the instruction set is so flexible, fewer instructions are required to perform any given function. 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-11/780 CPU provides 16 32-bit general registers which can be used for temporary storage, as accumulators, index registers, and base registers. Although all can be used as general-purpose registers, four have special significance depending on the instruction being executed: Register 12 (the CALL argument pointer); Register 13 (the CALL 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-11/780 CPU provides three "cache" systems—the memory cache, an address translation buffer, and an instruction buffer.

MEMORY CACHE
The memory cache (typically 95% hit rate) provides the central processor with high-speed access to main memory. The memory cache reduces main memory read access time from 1800 nanoseconds to an effective 290 nanoseconds, and has a cycle time of 200 nanoseconds. The memory cache also provides 32 bits of lookahead. On a cache miss, 64 bits are read from main memory—32 bits to satisfy the miss and 32 bits of lookahead.

INSTRUCTION BUFFER
The instruction buffer consists of 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, floating point operations, and shifting all at the same time) significantly enhances the VAX-11/780's performance.

TRANSLATION BUFFER
The VAX-11/780 provides an address translation buffer that eliminates extra memory accesses during virtual-to-physical address translations most of the time (typically 97% hit rate). The address translation buffer contains 128 likely-to-be-used virtual-to-physical address translations.

**Clocks** — The standard VAX-11/780 CPU includes two clocks—a programmable real-time clock used by system diagnostics and by the VAX operating system for accounting and scheduling, and a time-of-year clock, which insures the correct time-of-day and date. The time-of-year clock is used by the operating system to enable unattended automatic restart following any service interruption, including a power failure.

**Writable Diagnostic Control Store (WDCS)** — 12K bytes (plus parity) of WDCS are provided to allow the Diagnostic Console Processor to verify crucial parts of the system, (i.e., the CPU, the intelligent console, the SBI, and the memory adapter). In addition, the WDCS can be used to implement updates to the VAX-11/780's microcode.

**Memory Management** — The VAX-11/780 memory management hardware enables the VAX operating system to provide a flexible and efficient virtual memory programming environment. Hardware memory management, in conjunction with the operating system, provides facilities for paging (with user control) and swapping.

In addition, the VAX-11/780 memory management provides four hierarchical modes: kernel, executive, supervisor, and user, with read/write access control for each mode.

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

## THE CONSOLE SUBSYSTEM

The VAX-11/780's integrated console consists of an LSI-11 microcomputer with 16K bytes of read/write memory and 8K bytes of ROM (used to store the LSI diagnostic, the LSI bootstrap, and fundamental console routines), a floppy disk (for the storage of basic diagnostic programs and software updates), a terminal, and an optional remote diagnosis port.

Figure 1-3 illustrates the console subsystem.

The console subsystem serves as a VAX operating system terminal, as the system console, and as a diagnostic console. As a VAX terminal, it is used by authorized system users for normal system operations. As the system console, it is used for operational control (i.e., bootstrapping, initialization, software update). As a diagnostic console, it can access the central processor's major buses and key control points through a special internal diagnostic bus. The console allows operator diagnostic operations through simple keyboard commands.

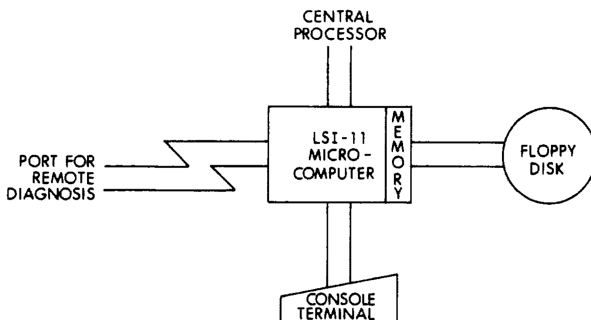



Figure 1-3 Console Subsystem

The floppy disk serves many useful purposes. During system installation, the floppy disk is used as a load device. The hardware bootstrap reads a file from the floppy; this file in turn loads the operating system from the system volume.

In addition, hard core diagnostics (i.e., those that test "crucial" system components) are stored on floppy. Testing of the LSI-11 is performed at power up; microdiagnostics are performed on command.

Because the floppy device is standard on all VAX-11/780 systems, software updates are distributed on this device. Simple commands typed at the console terminal automatically update the system software from the floppy.

## THE MEMORY SUBSYSTEM

The main memory subsystem consists of ECC MOS memory, which is connected to the SBI via the memory controller, as illustrated in Figure 1-4.

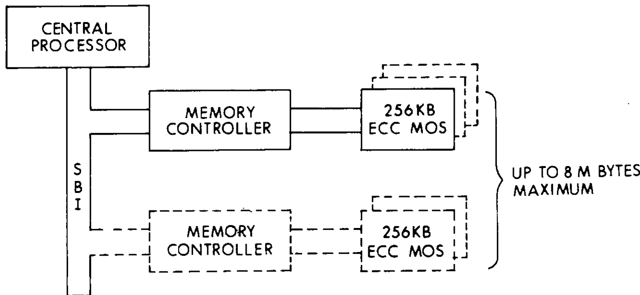



Figure 1-4 Memory Subsystem

The VAX-11/780 physical memory is built using either 4K or 16K MOS RAM chips. Memory is organized in quadwords (64 bits) plus an 8-bit ECC (Error Correcting Code), which allows the correction of all single-bit errors, and the detection of all double-bit errors.

MOS memory may be added in increments of either 128K or 256K byte units to a maximum of either one or four million bytes per controller (depending upon chip capacity). Two memory controllers may be connected to a VAX-11/780 system, for a total of either two or eight million bytes of physical memory. (The minimum memory requirement is 128K bytes utilizing 4K RAM chips and 256K bytes utilizing 16K RAM chips.)

The memory cycle time is 600 nanoseconds. This is equal to the memory access time since MOS memory has nondestructive read-out. Read access time at the central processor (including SBI overhead) is 1800 nanoseconds. This is measured from the time the processor transmits a read request until the processor receives all 64 bits of data. (The central processor always reads 64 bits from memory.) In

spite of the 1800 nanosecond memory access time, the VAX-11/780 processor realizes an effective average operand access time of 290 nanoseconds, due to the large optimized memory cache.

The memory controllers allow the writing of data in full 32 and 64 bit units.

Each memory controller buffers up to four memory access requests. This "request buffer" substantially increases memory throughput and overall system throughput and decreases the need for interleaving for most configurations.

Interleaving is possible with two controllers and equal amounts of memory on each. Interleaving for VAX-11/780 systems should be used when more than two MASSBUS adapters are connected and the MASSBUS and UNIBUS devices are transferring at very high rates, greater than one million bytes/second. Interleaving is enabled/disabled under program control. It is performed at the quadword level (each 64 bits) due to the memory organization. Note that in most cases interleaving will not be required due to the memory controller's request buffer.

## THE INPUT/OUTPUT SUBSYSTEMS

The VAX-11/780's I/O subsystem consists of the SBI, and the UNIBUS and MASSBUS devices connected to the SBI through special buffered interfaces called adapters. As illustrated in Figure 1-5, each VAX-11/780 system has one UNIBUS adapter and can have up to four MASSBUS adapters.

**The Synchronous Backplane Interconnect** — The SBI is the primary control and data transfer path in the VAX-11/780 system. The SBI has a physical address space of one gigabyte (30 bits of address).

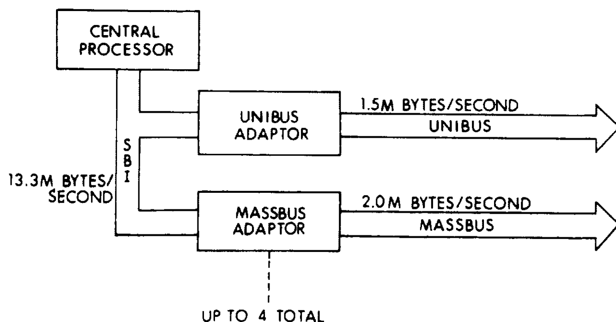



Figure 1-5 I/O Subsystem

Physical address space is all possible memory and I/O addresses that a processor can access. In the VAX-11/780 system, half of the physical address space is for memory addresses and half for I/O addresses, as illustrated in Figure 1-6.

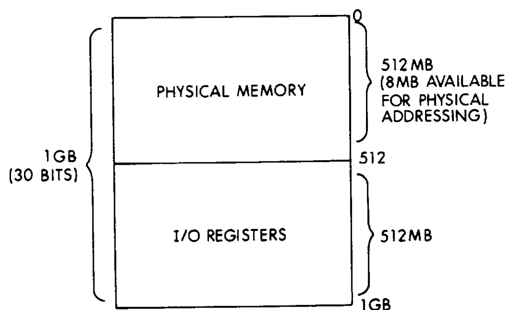



Figure 1-6 SBI Physical Address Space

Presently, VAX-11/780 will support up to either two or eight million bytes of main memory (depending upon storage chip capacity).

Each SBI device (i.e., CPU, MASSBUS adapter, UNIBUS adapter, memory controller) has a unique priority. When a device wants to transmit on the SBI, it asserts a unique request line. At the end of the next 200 nanosecond cycle, each SBI device wanting to use the SBI examines the SBI request lines for higher priority devices. The highest priority device uses the next cycle, while other devices must wait. Whenever possible, an SBI device currently in control of the SBI will free the SBI so that a new transaction may occur on the next cycle. This communication protocol enables:

1) *Distributed arbitration.* Since each device connected to the SBI determines whether or not it will receive the next cycle (rather than a central arbitrator making the decision), signals need travel only one times the length of the SBI, with the advantage of increased speed. Additionally, devices perform a parity check on the control information, assuring that the arbitration is proceeding correctly.
2) *Single 32-bit and two back-to-back 32-bit transfers.* The SBI data path is 32 bits wide. The protocol allows single (32-bit) and double (64-bit) data transfer as transactions. (The I/O adapters always try to transfer data in 64-bit quadwords).

Every transaction on the SBI (i.e., data transfer, address transfer, or command transfer) is parity-checked and confirmed by the receiver.

In addition, substantial protocol checking occurs on every cycle, resulting in high data integrity.

**The UNIBUS** — General-purpose and customer-developed devices are connected to the VAX-11/780 system via the VAX-11/780's UNIBUS. Since the SBI deals in 30-bit addresses (one gigabyte), 18-bit UNIBUS addresses must be translated to 30-bit SBI addresses. This mapping function is performed by the UNIBUS adapter, a special interface between the SBI and the UNIBUS, which translates UNIBUS addresses, data, and interrupt requests to their SBI equivalents, and vice versa.

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

The UNIBUS adapater allows two kinds of data transfer: program interrupt and direct memory access (DMA). To make the most efficient use of the SBI bandwidth, the UNIBUS adapter facilitates high-speed DMA transfers by providing buffered DMA data paths for up to 15 high-speed devices. Each of these channels has a 64 bit buffer (plus byte parity) for holding four 16 bit transfers to and from UNIBUS devices. The result is that only one SBI transfer (64 bits) is required for every four UNIBUS transfers. The maximum aggregate transfer rate through the Buffered Data Paths is 1.5 miillion bytes/second. In addition, on SBI-to-UNIBUS transfers, the UNIBUS adapter anticipates upcoming UNIBUS requests by prefetching the next 64-bit quadword from memory as the last 16-bit word is transferred from the buffer to the UNIBUS. The result is increased performance. By the time the UNIBUS device requests the next word, the UNIBUS adapter has it ready to transfer.

Any number of unbuffered DMA transfers are handled by one direct DMA data path. Every 8 or 16-bit tranfer on the UNIBUS requires a 32-bit tranfer on the SBI (although only 16 bits are used). The maximum transfer rate through the Direct Data Path is 500 thousand bytes/second.

The UNIBUS adapter permits concurrent program interrupt, unbuffered and buffered data transfers. The aggregate throughput rate of the Direct Data Path, plus the 15 Buffered Data Paths, is 1.5 million bytes/second.

**The MASSBUS(es)** — High-performance mass storage devices, such as the RP series moving head disks, are connected to the VAX-11/780 system using a MASSBUS adapter. The MASSBUS adapter is the

interface between the MASSBUS and the SBI, and performs all control, arbitration, and buffering functions. Address mapping is similar to that performed by the UNIBUS adapter.

There may be a total of four MASSBUS adapters on each VAX-11/780 system. Each adapter can accommodate data transfers of 128K bytes maximum to and from noncontiguous pages in physical memory (scatter/gather). The VAX operating system supports transfers of 64K bytes maximum to be consistent with other devices.

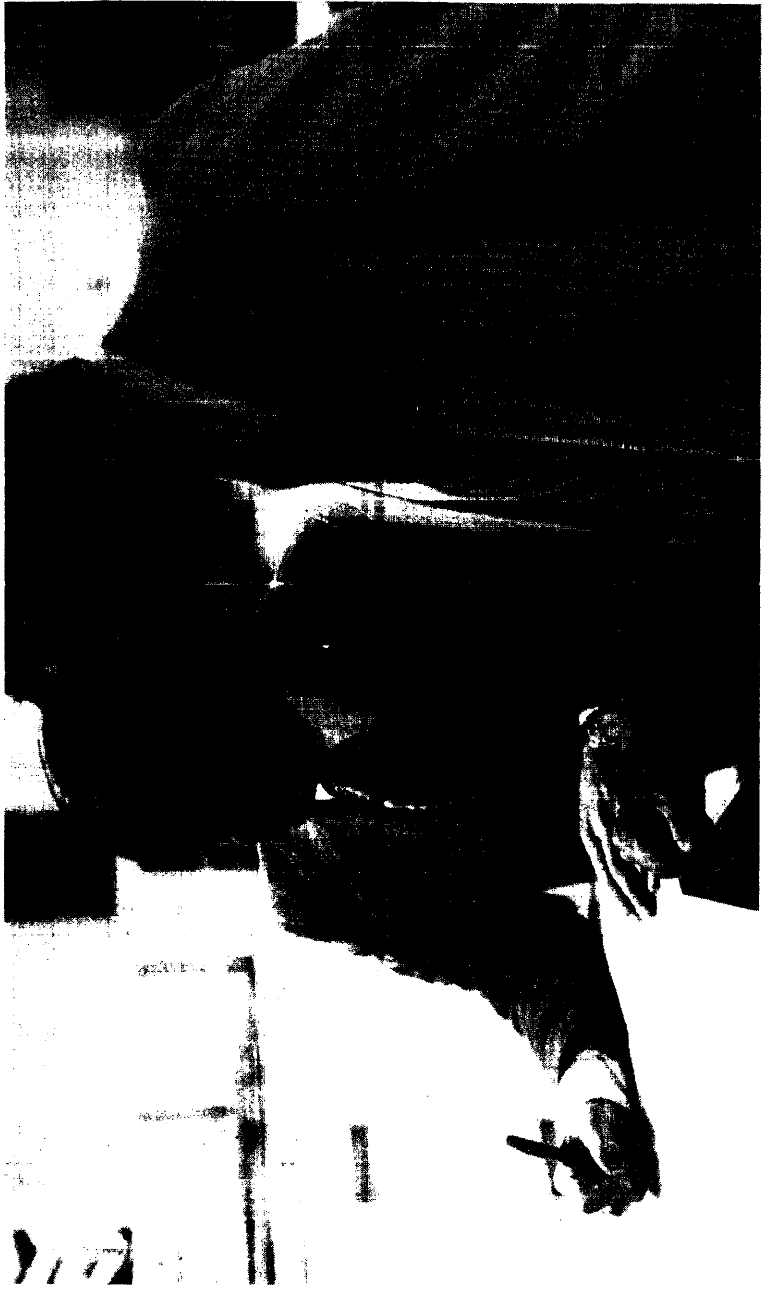
Each MASSBUS adapter uses a 32-byte silo data buffer, which permits transfers at rates up to two million bytes/second to and from physical memory (8M bytes/second with all four). As in the UNIBUS adapter, data is assembled in 64-bit quadwords (plus byte parity) to make maximum efficient use of the SBI bandwidth.

On memory-to-MASSBUS transfers, as on memory-to-UNIBUS transfers, the adapter anticipates upcoming MASSBUS data transfers by prefetching the next 64 bits.

**Optional Hardware Equipment**

Prewired mounting space is available within the VAX-11/780 CPU chassis for mounting the following optional equipment:

1) *A High-Performance Floating Point Accelerator.* The FPA is an independent processor that works in parallel with the base 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) Up to *two million bytes (total) of MOS Memory with ECC.*
3) *Main Memory Battery Backup* for ten minutes for each one million bytes of memory.
4) 12K bytes (plus parity) of user writable control store (WCS). The user can modify or add to the native mode instruction set by programming the WCS.



# CHAPTER 2
# CONSOLE SUBSYSTEM

## 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 (TTA0) 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 2-1 functionally illustrates the console subsystem.

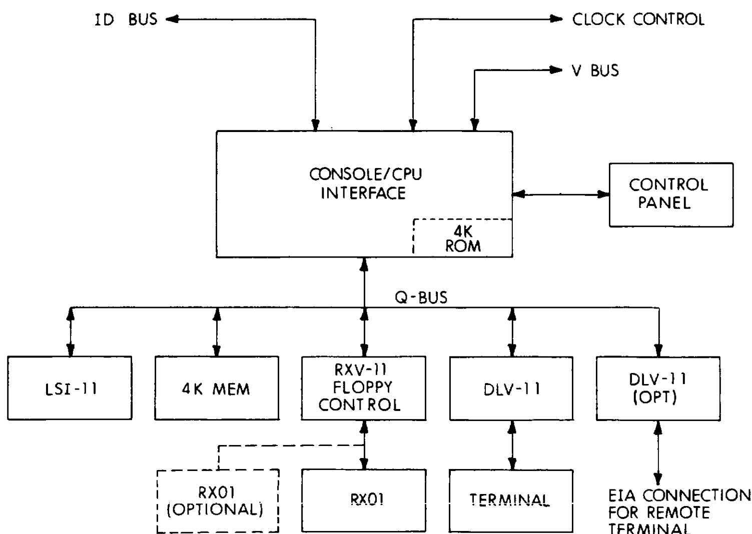



Figure 2-1 Console Subsystem

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 second optional floppy disk drive is also available.
- a system terminal and two serial line interface units (DLV11-E), 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-11 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 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 maintenance 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 2-2 illustrates the VAX-11/780 and LSI-11 interaction and operating mode combinations.

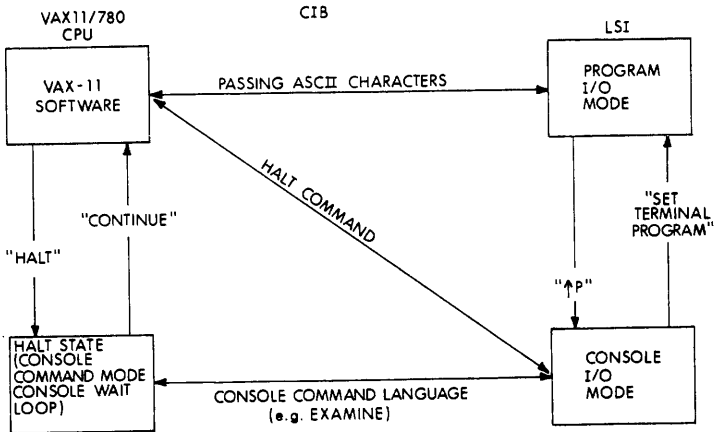



Figure 2-2 VAX-11/780 and LSI-11 Interaction and System Operating States

**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

from program I/O mode to console command mode and then input the HALT command.

The system operator can enter the console command mode from the program I/O mode by typing control-P (↑P). Similarly, the operator can change from console command mode to program I/O mode by typing "Set Terminal Program". While the VAX processor is executing code, only the following subset of commands are permitted:

- SHOW
- SET
- WAIT DONE
- HELP
- EXAMINE /VBUS
- CLEAR

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 V bus interfaces the LSI-11 microprocessor and its peripheral hardware to the VAX-11 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 located in Appendix D.

## 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-11 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-11 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 2-3. The LOAD signal causes the shift register to parallel load from the test points when the VAX-11 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.

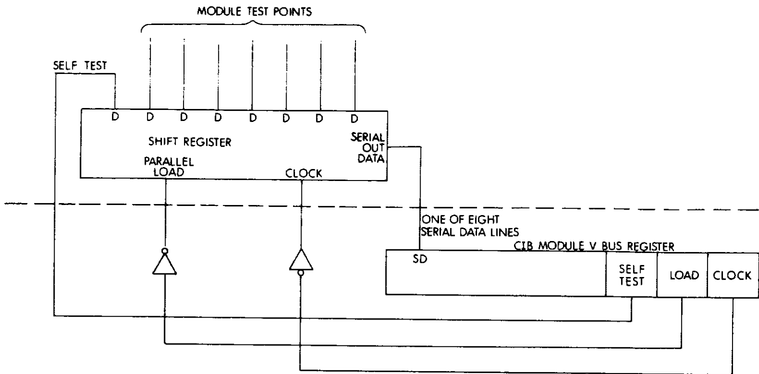



Figure 2-3 V Bus Block Diagram

## CONSOLE/VAX-11 INTERACTION

All data transfer operations between the VAX-11 CPU and the console LSI-11 are routed via the TO and FM ID Registers on the CIB. The LSI-11 Console may look at various points in the VAX-11 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-11 CPU is halted.

The TO and FM internal data registers are illustrated in Figure 2-4.

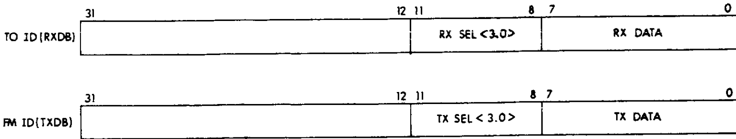



Figure 2-4 TO and FM ID Registers

## 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 9 THRU F (R0 THRU PC) |
| '/VR' | = | VBUS REGISTERS |
| '/IF' | = | 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,
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 ISP |
| 'HELP' | -PRINTS THIS FILE |
| 'INITIALIZE' | -INITIALIZES THE CPU |
| 'LINK' | -CAUSES CONSOLE TO BEGIN COMMAND LINKING. CONSOLE PRINTS REVERSED PROMPT TO INDICATE LINKING. 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 (CLEARS 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 |
| '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 RE- |

| | |
|---|---|
| | CEIVED 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<br>(UNLESS MODE SWITCH IN 'DISABLE') |
| '@<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:<br>1) More than 80 characters in an indirect command line or<br>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 STOPPED,COMMAND ABORTED | A console command that requires the CPU 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:<br><br>Bit 0 = Length violation (bits numbered from right)<br><br>Bit 1 = Fault was on a PTE reference<br><br>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).<br><br>CODE 1-Floppy hardware error. (CRC, Parity, etc.)<br><br>CODE 2-File not found.<br><br>CODE 3-Floppy driver queue overfull.<br><br>CODE 4-Console software requested an illegal sector number. |
| ?FLOPPY NOT READY | The console floppy drive failed to become ready when booting. (Retry) |

| Message | Meaning |
|---|---|
| ?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**

| Message | Meaning |
|---|---|
| ?WARNING-WCS & FPLA VER MISMATCH | The microcode in WCS is not compatible with FPLA. This message is printed on each 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**

| Message | Meaning |
|---|---|
| ?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 3

# 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 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).

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 continously 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.

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 $\mu$sec. The inner loop of the POLY instruction takes approximately one $\mu$sec per degree of polynomial.

**Optional 12K Byte User 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 3-1 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
            Do 100 I = 1, 10
100         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**

| CPU Component | Operation |
|---|---|
| ALU, R | 1000 → PC |
| TB | Translate virtual 1000 to physical 1F600 |
| Cache | Does Cache presently contain address 1F600? (NO) therefore, |

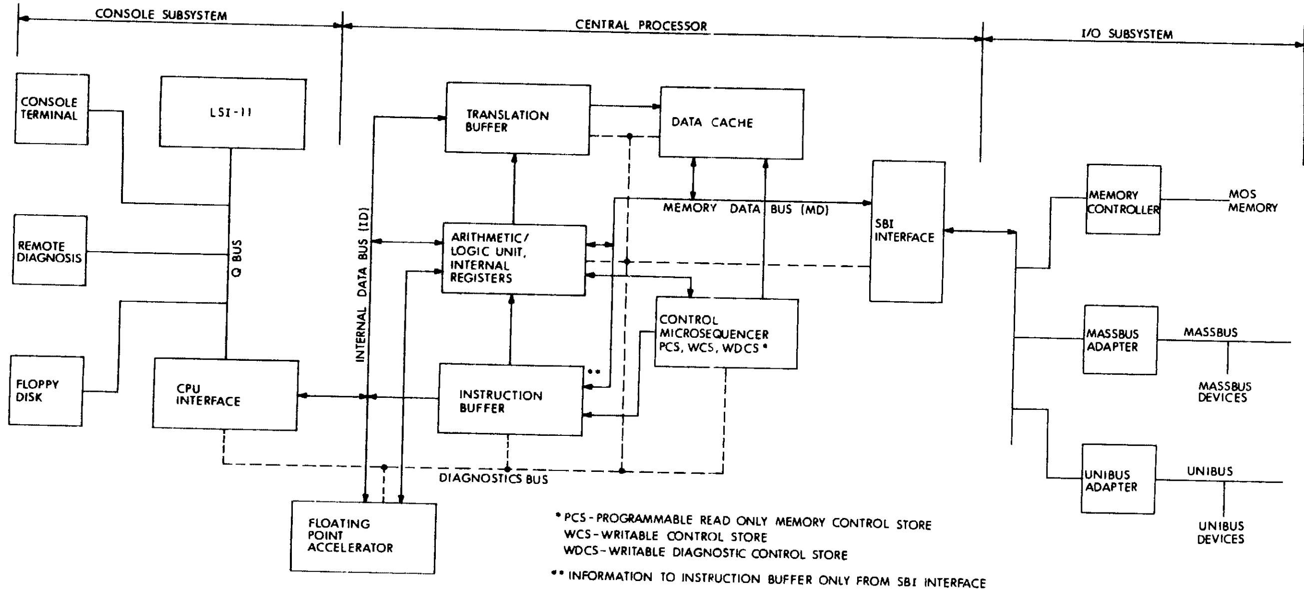



Figure 3-1 The Central Processing Unit

| CPU Component | Operation |
|---|---|
| SBI | Fetch 1F600-1F607 from memory |
| Cache | Store address range 1F600-1F607 and corresponding contents in Data Cache |
| (IB) | Obtain instructions from physical addresses 1F600-1F603 |
| ALU, R | Ask IB for instruction -- (CLRL) |
| ALU, R | Clear R0 |
| (IB asks Cache for more) | IB retrieves physical addresses 1F604-1F607 from Cache |
| ALU, R | Ask IB for next instruction -- (MOVL) |
| ALU, R | Ask IB for destination specifier -- (R1) |
| (IB asks Cache for more) | Cache asks SBI for 1F608 |
| SBI | Asks memory for physical addresses 1F608-1F60F |
| ALU, R | Store 1 in R1 |
| ALU, R | Ask IB for next instruction -- (ADDL2) |
| ALU, R | Calculate base address of N (virtual 1FFC) |
| ALU, R | Adds 4*R1 to address of N to yield virtual address 2000 |
| TB | Look up address of N[1] : physical address A00 |
| Cache | Searches for physical address A00, but finds it not there, therefore, |
| SBI | The SBI enters a wait mode because it is currently completing the fetch operation of physical addresses 1F608-1F60F |
| SBI | Finishes the prefetch operation of physical addresses 1F608-1F60F |
| IB | Grabs 1F608-1F60B |

| CPU Component | Operation |
|---|---|
| Cache | Gets 1F608-1F60F |
| SBI | Starts fetch of physical addresses A00-A07 |
| Cache | Gets A00-A07 |
| ALU, R | A00-A03 |
| ALU, R | Asks IB for destination specifier (R6) |
| (IB asks | |
| for 1F60C) | Cache sends 1F60C-1F60F to IB |
| ALU, R | Add (A00-A03) (i.e., N[1]) to R0 |
| ALU, R | Ask IB for instruction (AOBLEQ) |
| (IB asks for | |
| more data) | Cache asks SBI to get 1F610 from memory |
| ALU, R | Asks IB for next specifier (R1) |
| ALU, R | Add 1 to R1, compare to 10, if less than or equal to 10 then branch |
| ALU, R | Flush (clear) IB, load virtual 1005 into PC |
| IB | Fetch 1005 from cache (resumption of loop) |
| ALU, R | Ask IB for next instruction (ADDL2) |
| . | . |
| . | . |
| . | . |
| SBI | Memory data (1F610) arrives |
| Cache | Takes data, but IB doesn't grab it |
| . | . |
| . | . |
| . | . |
| . | . |
| . | . |
| | on the 11th increment, |
| ALU, R | Add 1 to R1, compare to 10, now however R1 = 11 |

| CPU Component | Operation |
|---|---|
| | and do not branch, but fall through to the next instruction |
| ALU, R | Ask IB for next instruction |

## 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

compatibility mode execution. The native mode instruction set consists of 244 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 3-1 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

**Table 3-1 Data Types**

| DATA TYPE | SIZE | RANGE (decimal) | |
|---|---|---|---|
| Integer | | Signed | Unsigned |
| Byte | 8 bits | -128 to +127 | 0 to 255 |
| Word | 16 bits | -32768 to +32767 | 0 to 65535 |
| Longword | 32 bits | $-2^{31}$ to $+2^{31}-1$ | 0 to $2^{32}-1$ |
| Quadword | 64 bits | $-2^{63}$ to $+2^{63}-1$ | 0 to $2^{64}-1$ |
| Floating Point | | $\pm 2.9 \times 10^{-37}$ to $1.7 \times 10^{38}$ | |
| Floating | 32 bits | approximately seven decimal digits precision | |
| Double Floating | 64 bits | approximately sixteen decimal digits precision | |
| Packed Decimal String | 0 to 16 bytes (31 digits) | numeric, two digits per byte sign in low half of last byte | |
| Character String | 0 to 65535 bytes | one character per byte | |
| Variable-length Bit Field | 0 to 32 bits | dependent on interpretation | |

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 trap to 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 16 32-bit programmable general registers are labelled R0 through R11 (decimal). Registers 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 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 register R7 loads both R7 and R8.

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.

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 3-2 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 3-2 Addressing Modes**

| Mode | Syntax | |
|---|---|---|
| Literal (Immediate) | {S↑ / I↑} *# constant* | |
| Register | **R***n* | |
| Register Deferred | **(R***n***)** | Indexed [**Rx**] |
| Autodecrement | – **(R***n***)** | |
| Autoincrement | **(R***n***)** + | |
| Autoincrement Deferred | @ **(R***n***)** + | |
| (Absolute) | @ *# address* | |
| Displacement | {**B↑** / **W↑** / **L↑**} *displacement* **(R***n***)** / *address* | |
| Displacement Deferred | @ {**B↑** / **W↑** / **L↑**} *displacement* **(R***n***)** / *address* | |

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 procedures. Figure 3-2 illustrates the argument pointer and list. 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 that you want to preserve. The second method is generally used when calling procedures that do not require arguments, or when building an argument list dynamically.

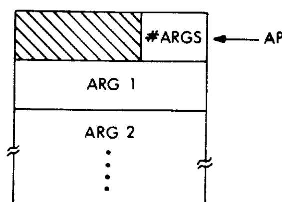



Figure 3-2 Argument Pointer and List

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 3-3 illustrates the Call Frame. Since the previous 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.

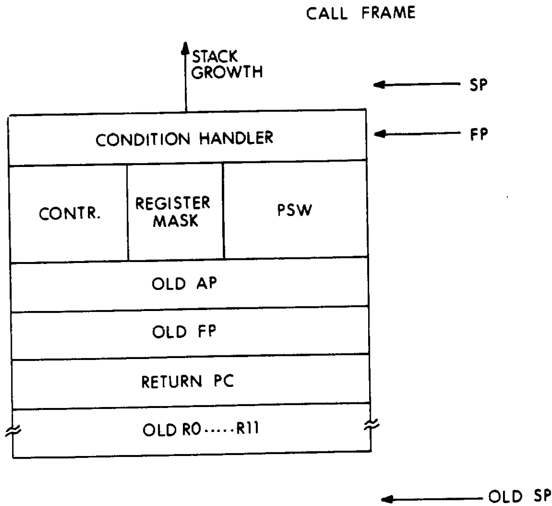



Figure 3-3 Call Frame

**Processor Status Word**

The Processor Status Word (a portion of the Processor Status Longword) 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 3-4, contains two sets of bit fields:

- the condition codes
- the trap enable flags

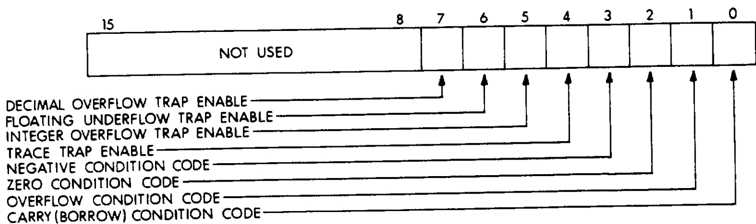



Figure 3-4 Processor Status Word

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 traps that concern the user process: trace traps and arithmetic traps. The trace trap is used for debugging programs. Arithmetic traps 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

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.

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 3-5 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

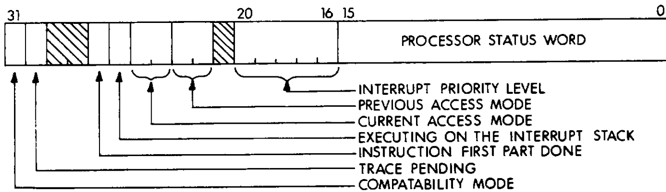



Figure 3-5 Processor Status Longword

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 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, 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 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 instruc-

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 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

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 physical address space is called memory space, and the highest-addressed half I/O space.

Chapter 6, 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 memory 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 hexadecimal are:

| PAGE | ADDRESS(10) | ADDRESS(16) |
|---|---|---|
| 0 | 0000-0511 | 0000-01FF |
| 1 | 0512-1023 | 0200-03FF |
| 2 | 1024-1535 | 0400-05FF |
| 3 | 1536-2047 | 0600-07FF |
| 4 | 2048-2559 | 0800-09FF |
| 5 | 2560-3071 | 0A00-0BFF |
| 6 | 3072-3583 | 0C00-0DFF |
| 7 | 3584-4095 | 0E00-OFFF |

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 address translation buffer, which is described later, and the translation algorithm itself.

Figure 3-6 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.

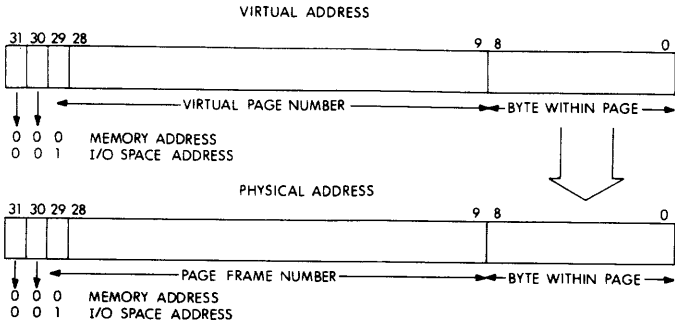



Figure 3-6 Virtual and Physical Address Format

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 3-7 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 3-3 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

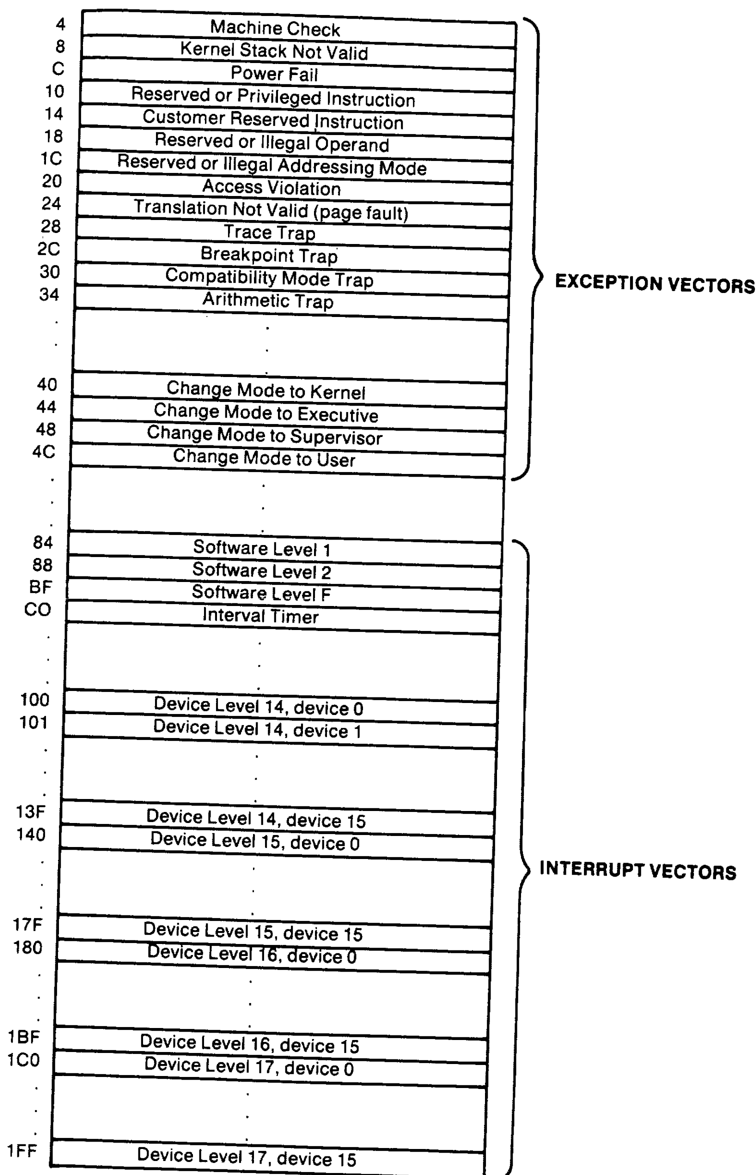



Figure 3-7 System Control Block

**Table 3-3 Interrupt Priority Levels**

<table>
<tr><th colspan="2">PRIORITY</th><th colspan="2" rowspan="2">HARDWARE EVENT</th></tr>
<tr><th>Hex</th><th>Decimal</th></tr>
<tr><td>1F</td><td>31</td><td colspan="2">Machine Check, Kernel Stack Not Valid</td></tr>
<tr><td>1E</td><td>30</td><td colspan="2">Power Fail</td></tr>
<tr><td>1D</td><td>29</td><td></td><td rowspan="5">Processor, Memory, or Bus Error</td></tr>
<tr><td>1C</td><td>28</td><td></td></tr>
<tr><td>1B</td><td>27</td><td></td></tr>
<tr><td>1A</td><td>26</td><td></td></tr>
<tr><td>19</td><td>25</td><td></td></tr>
<tr><td>18</td><td>24</td><td colspan="2">Clock</td></tr>
<tr><td>17</td><td>23</td><td>UNIBUS BR7</td><td rowspan="8">Device Interrupt</td></tr>
<tr><td>16</td><td>22</td><td>UNIBUS BR6</td></tr>
<tr><td>15</td><td>21</td><td>UNIBUS BR6</td></tr>
<tr><td>14</td><td>20</td><td>UNIBUS BR4</td></tr>
<tr><td>13</td><td>19</td><td></td></tr>
<tr><td>12</td><td>18</td><td></td></tr>
<tr><td>11</td><td>17</td><td></td></tr>
<tr><td>10</td><td>16</td><td></td></tr>
<tr><th colspan="2">PRIORITY</th><th colspan="2">SOFTWARE EVENT</th></tr>
<tr><td>0F</td><td>15</td><td></td><td rowspan="4">Reserved for DIGITAL</td></tr>
<tr><td>0E</td><td>14</td><td></td></tr>
<tr><td>0D</td><td>13</td><td></td></tr>
<tr><td>0C</td><td>12</td><td></td></tr>
<tr><td>0B</td><td>11</td><td></td><td rowspan="4">Device Drivers</td></tr>
<tr><td>0A</td><td>10</td><td></td></tr>
<tr><td>09</td><td>09</td><td></td></tr>
<tr><td>08</td><td>08</td><td></td></tr>
<tr><td>07</td><td>07</td><td colspan="2">Timer Process</td></tr>
<tr><td>06</td><td>06</td><td colspan="2">Queue Asynchronous System Trap (AST)</td></tr>
<tr><td>05</td><td>05</td><td colspan="2">Reserved for DIGITAL</td></tr>
<tr><td>04</td><td>04</td><td colspan="2">I/O Post</td></tr>
<tr><td>03</td><td>03</td><td colspan="2">Process Scheduler</td></tr>
<tr><td>02</td><td>02</td><td colspan="2">AST Delivery</td></tr>
<tr><td>01</td><td>01</td><td colspan="2">Reserved for DIGITAL</td></tr>
<tr><td>00</td><td>00</td><td colspan="2">User Process Level</td></tr>
</table>

space, and is $2^{29}$ 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 4
# PROCESS STRUCTURE

## PROCESS DEFINITION

A process is the basic entity schedulabled for execution by the VAX-11/780 processor. A process consists of an address space and both hardware and software context. The hardware context of a process is defined by a Process Control Block (PCB) that 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 4-1.

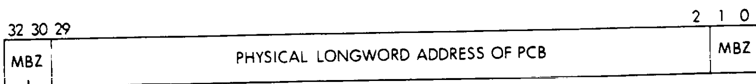



Figure 4-1 Process Control Block Base Register (Read/Write)

## 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 4-2.

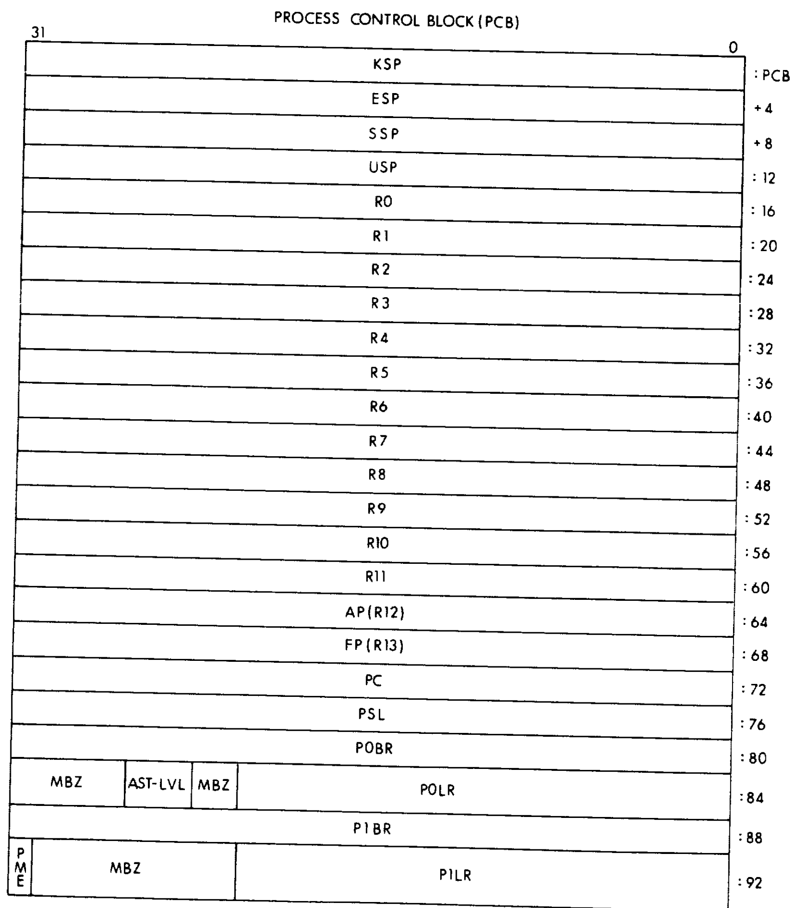



Figure 4-2 Process Control Block

A description of the process control block follows;

| Long word | Bits | Mnemonic | Description |
|---|---|---|---|
| 0 | <31:0> | KSP | 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. |
| 1 | <31:0> | ESP | Executive Stack Pointer. Contains the stack pointer to be used when the current access mode field in the PSL is 1. |
| 2 | <31:0> | SSP | Supervisor Stack Pointer. Contains the stack pointer to be used when the current access mode field in the PSL is 2. |
| 3 | <31:0> | USP | User Stack Pointer. Contains the stack pointer to be used when the current access mode field in the PSL is 3. |
| 4-17 | <31:0> | R0-R11, AP,FP | General registers R0-R11, AP, and FP. |
| 18 | <31:0> | PC | Program Counter. |
| 19 | <31:0> | PSL | Program Status Longword. |
| 20 | <31:0> | P0BR | Base register for page table describing process virtual addresses from zero to $2^{30}-1$. |
| 21 | <21:0> | P0LR | Length register for page table located by P0BR. Describes effective length of page table. |
| 21 | <23:22> | MBZ | Must be zero. |

| | | | |
|---|---|---|---|
| 21 | <26:24> | ASTLVL | 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. |

| ASTLVL | Meaning |
|---|---|
| 0 | AST pending for access mode 0 (kernel) |
| 1 | AST pending for access mode 1 (executive) |
| 2 | AST pending for access mode 2 (supervisor) |
| 3 | AST pending for access mode 3 (user) |
| 4 | No pending AST |
| 5-7 | Reserved to DIGITAL |

| | | | |
|---|---|---|---|
| 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$L_ and the mnemonic shown above. For example, the PCB offset to R3 is PTX$L_R3. The following are also defined:

| | |
|---|---|
| PTX$L_P0LRASTL | Longword 21 |
| PTX$L_P1LRPME | Longword 23 |

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 running. In order to access them, two privileged registers are provided. These are the AST Level register and the PME register. The AST Level register is illustrated in Figure 4-3.

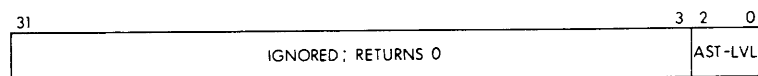



Figure 4-3 AST Level Register (Read/Write)

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

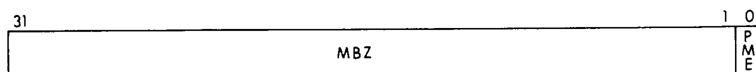



Figure 4-4 Performance Monitor Enable Register (Read/Write)

At bootstrap time, PME is cleared.

## ASYNCHRONOUS SYSTEM TRAPS (AST)

Asynchronous System Traps are a technique for notifying a process of events that are not synchronized with its execution and initiating 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 VAX-11 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 protected access mode. Since outward access mode transitions 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.

General Software Flow for AST processing:

1. An event associated with an AST causes software enqueuing of an AST control block to the software PCB, and the software 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. The (IPL2) interrupt service routine should compute the correct new value for ASTLVL that prevents additional AST delivery interrupts while in kernel mode, and move 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.
3. The (IPL2) interrupt service routine should compute the correct new value for ASTLVL that prevents additional AST delivery interrupts while in kernel mode and move 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.

They are:

(IPL2) — AST delivery interrupt.
This interrupt is triggered by an REI that detects PSL<current mode> GEQU ASTLVL and indicates that a pending AST may now be delivered for the currently executing process.

(IPL3) — Process scheduling interrupt.
This interrupt is only triggered by software to allow the software running at IPL 3 to cause the currently executing process to be blocked and the highest priority executable process to be scheduled.

**PROCESS STRUCTURE INSTRUCTIONS**

Process scheduling software must execute on the interrupt stack (PSL<IS> set) in order to have a noncontext-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 store process context instructions, the following notation conventions are used:

| Notation | Meaning |
|---|---|
| tmp | tmp1 and tmp2 are pseudo registers which are not normally implemented in hardware |
| ! | indicates comment statement |
| ← | 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 |
| ( ) | this indicates the contents of the address specified by the included expression |
| <N:M> | this notation indicates the field consisting of bits N thru M of the immediately preceding value |
| { } | indicate an exception |
| [ ] | used to group terms for clarity, and usually appear in logical expressions |

**LDPCTX Load Process Context**

Purpose: restore register and memory management context

Format: Opcode

Operation:

```
if PSL<current mode> NEQU 0 then
    {privileged instruction fault};
    {invalidate per-process translation buffer entries};
    !PCB 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);
    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 instruction};
    P1BR←tmp1;
    if (PCB +92)<30:22> NEQU 0 then
    {reserved operand fault};
    P1LR←(PCB+92)<21:0>;
```

```
   PME←(PCB+92)<31>;
   if (PCB+92)<30:22> NEQU 0 then {reserved operand abort};
   if PSL <IS> EQLU 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
reserved 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.

**NOTE**

> Some processors keep a copy of each of the per-process stack pointers in internal registers. In those processors that do, LDPCTX loads the internal registers from the PCB. Those processors that do not keep a copy of all four per-process stack pointers in internal registers keep only the current access mode register in an internal register and switch this with the PCB contents whenever the current access mode field changes.

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

Condition Codes:

```
N←N;
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:

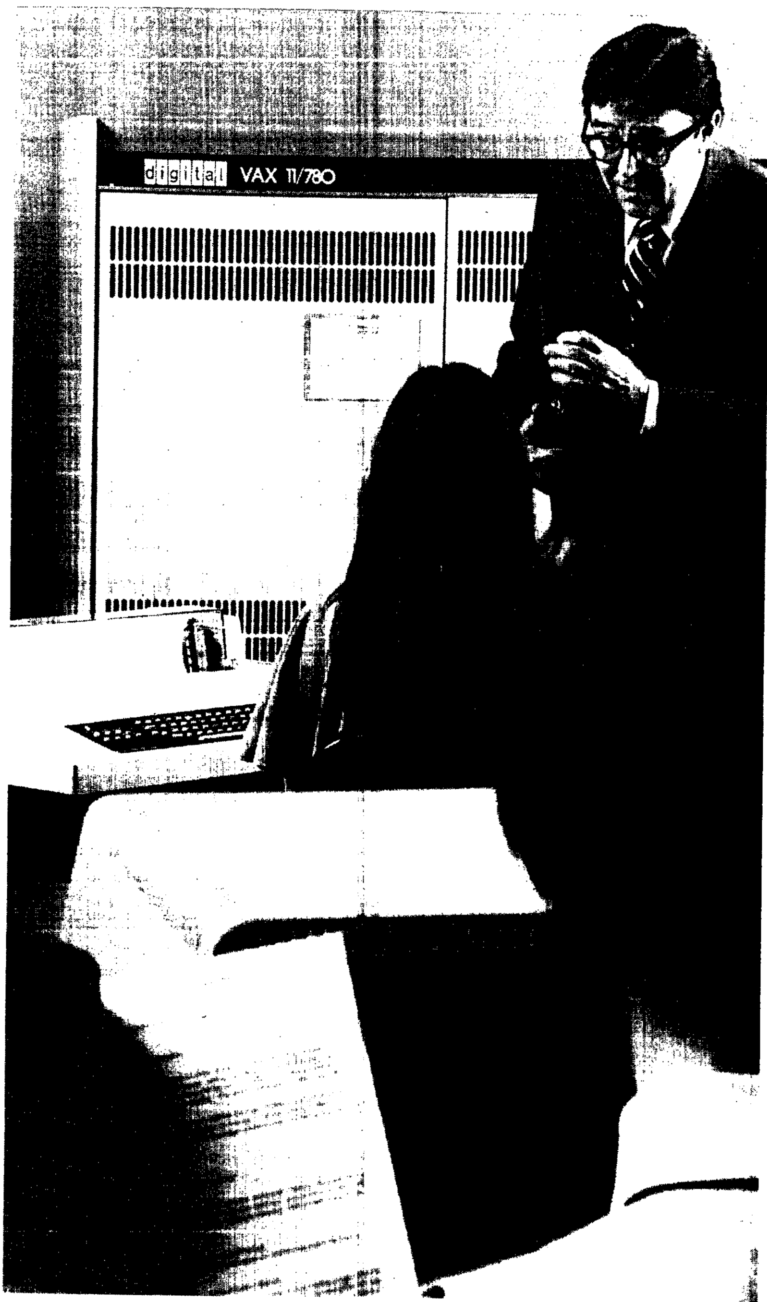
1. The map, ASTLVL, and PME contents of the PCB are not saved because they are rarely changed. Thus, not writing them saves overhead.
2. Some processors keep a copy of each of the per-process stack pointers in internal registers. In those processors that do, SVPCTX stores the internal registers into the PCB. Those processors that do not keep a copy of all four per-process stack pointers in internal registers, keep only the current access mode register in an internal register and switch this with the PCB contents whenever the current access mode field changes.
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.

**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 @#PHYSPCB,PCBB          ;Set physical PCB
                             address in PCBB
LDPCTX                       ;Load context from
                             PCB
                             ;For new process
REI                          ;Place process in
                             execution
```


digital VAX 11/780

CHAPTER 5

# EXCEPTIONS AND INTERRUPTS

## INTRODUCTION

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

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

Other events are relevant to other processes, or to the system as a whole, and are therefore 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). Further, some interrupts are of such urgency that they require high priority service, while others must be synchronized with independent events. 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) will the processor raise the IPL and service the interrupt request. The interrupt service routine is entered at the IPL of the interrupt request and will 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, 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, 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 1F (hex) do so on only one processor. Thus, special software action is required to stop other processors in a multiprocessor system.

Most exception service routines execute at IPL 0 in response to exception conditions caused by the software. A variation from this is serious system failures, which raise IPL to the highest level (1F, hex) to minimize processor interruption until the problem is corrected. Exception service routines are usually coded to avoid exceptions; however, nested exceptions can occur.

**Processor Interrupt Priority Levels (IPLs)**

The processor has 31 interrupt priority levels (IPLs), divided into 15 software levels (numbered, in hex, 01 to 0F), and 16 hardware levels (10 to 1F, hex). User applications, system calls, and system services all run at process level, which may be thought of as IPL 0. Higher numbered interrupt levels have higher priority; that is to say, any requests at an interrupt level higher than the processor's current IPL will interrupt immediately, but requests at a lower or equal level are deferred.

Interrupt levels 01 through 0F (hex) 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 will be cleared by the hardware when the interrupt is taken.

Interrupt levels 10 to 17 (hex) are for use by devices and controllers, including UNIBUS devices. UNIBUS levels BR4 to BR7 correspond to VAX-11 interrupt levels 14 to 17 (hex).

Interrupt levels 18 to 1F (hex) are for use 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 (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 may be independent of the current instruction.
2. An exception condition is usually serviced in the context of the process that produced the exception condition, while an interrupt is serviced independently from the currently running process.
3. The IPL of the processor is usually not changed when the processor initiates an exception, while the IPL is always raised when an interrupt is initiated.

4. Exception service routines usually execute on a per-process stack, while interrupt service routines normally execute on a per-CPU stack.
5. Enabled exceptions are initiated immediately no matter what the processor IPL is, while interrupts are held off 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 overflow is disabled, or the processor is at the same or higher IPL, the condition will eventually initiate 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 of the exception.

**INTERRUPTS**

The processor services interrupt requests between instructions. 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, in order 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:

1. Device completion (IPL 10-17 hex)
2. Device error (IPL 10-17 hex)
3. Device alert (IPL 10-17 hex)
4. Device memory error (IPL 10-17 hex)
5. Console terminal transmit and receive (IPL 14 hex)
6. Interval timer (IPL 18 hex)
7. Recovered memory or bus or processor errors (implementation-specific, IPL 18 to 1D hex). The VAX-11/780 processor interrupts at 1B on memory errors.
8. Unrecovered memory or bus or processor errors (implementation-specific, IPL 18 to 1D hex)
9. Power fail (IPL 1E hex)
10. Software interrupt invoked by MTPR #SIRR (IPL 01 to 0F hex)
11. AST delivery when REI restores a PSL with IS clear and mode greater than or equal to ASTLVL.

Each device controller has a separate set of interrupt vector locations in the system control block (SCB), thus eliminating the need to poll controllers in order 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 (Hex)**
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 (hex) is reserved for power fail. Interrupt level 1F (hex) is reserved 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 might also be used to allow a kernel mode debugger to gain control on any condition.

**Device Interrupts—Levels 10-17 (Hex)**
The processor provides eight priority levels for use by peripheral devices. Any given implementation may or may not implement all eight levels of interrupts. The minimal implementation is levels 14-17 (hex) that correspond to the UNIBUS levels BR4 to BR7 if the system has a UNIBUS.

**Software-Generated Interrupts—Levels 01-0F (Hex)**
The processor provides 15 priority interrupt levels for use by software.

**Software Interrupt Summary Register**
Pending software interrupts are recorded in the Software Interrupt Summary Register (SISR). The SISR contains 1's in the bit positions correponding to levels on which software interrupts are pending. All such levels, of course, must be lower than the current processor IPL, or the processor would have taken the requested interrupt. Figure 5-1 illustrates the software interrupt summary register.

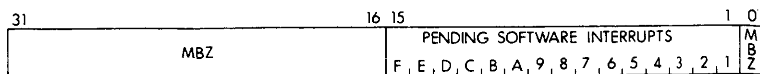



Figure 5-1 Software Interrupt Summary Register (Read/Write)

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 its state after a power fail, for example. |

**Software Interrupt Request Register**

The software interrupt request register (SIRR) is a write-only four-bit privileged register used for making software interrupt requests. Figure 5-2 illustrates the software interrupt request register.

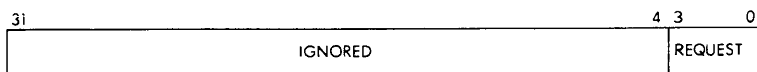



Figure 5-2 Software Interrupt Request Register (Write Only)

Executing MTPR src,#SIRR requests an interrupt at the level specified by src<3:0>. Once a software interrupt request is made, it will be cleared by the hardware 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 will be deferred until the IPL is lowered to less than src<3:0>, with no higher interrupt level pending. This lowering of IPL is by either REI or by MTPR x,#IPL. If src<3:0> is 0, no interrupt will occur or be requested.

Note that no indication is given if there is already a request at the selected level. Therefore, the service routine must NOT assume that there is 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 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 to remove a control block from the queue of service requests. If REMQUE returns failure (nothing in the queue), the service routine exits with REI.
4. If REMQUE 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 IPL with the MTPR instruction will load the processor priority field in the Program Status Longword (PSL), that is, PSL<20:16> is loaded from IPL<4:0>. Reading from IPL with the MFPR instruction will read the processor priority field from the PSL. On writing IPL, bits <31:5> are ignored, and on reading IPL, bits <31:5> are returned zero. Figure 5-3 illustrates the interrupt priority level register.

Figure 5-3 Interrupt Priority Level Register (Read/Write)

At bootstrap time, IPL is initialized to 31 (1F, hex).

Interrupt service routines must follow the discipline of not lowering 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 IPL5; it then sets IPL to 8, and then posts software requests at IPL3, IPL7, and IPL9. Then a device interrupt arrives at IPL11 (hex). Finally IPL is set back to IPL5. The sequence of execution is:

| event | state after contents of IPL (hex) | event SISR (hex) | IPL in PSL on stack |
|---|---|---|---|
| (initial) | 5 | 0 | 0 |
| MTPR #8,#IPL | 8 | 0 | 0 |
| MTPR #3,#SIRR | 8 | 8 | 0 |
| MTPR #7,#SIRR | 8 | 88 | 0 |
| MTPR #9, #SIRR interrupts to | 9 | 88 | 8,0 |
| device interrupts to | 11 | 88 | 9,8,0 |
| device service routine REI | 9 | 88 | 8,0 |
| IPL9 service routine REI | 8 | 88 | 0 |
| MTPR #5,#IPL changes IPL to 5 and the request for 7 is granted immediately | 7 | 8 | 5,0 |
| IPL7 service routine REI | 5 | 8 | 0 |
| initial IPL5 service routine REI back to IPL0 and the request for 3 is granted immediately | 3 | 0 | 0 |
| IPL3 service routine REI | 0 | 0 | — |

## SERIOUS SYSTEM FAILURES

Although serious system failures are exceptions, they are discussed here rather than in the 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 kernel stack during the initiation of an exception or interrupt. Usually this is an indication of a stack overflow or other executive software error. The attempted exception is transformed into an abort that uses the interrupt stack. No extra information is pushed on the interrupt stack in addition to PSL and PC. IPL is raised to 1F (hex). 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 normal memory management fault is initiated. If the exception vector <1:0> for Kernel Stack Not Valid is 0 or 3, the behavior of the processor is UNDEFINED.

### Interrupt Stack Not Valid Halt

An interrupt stack not valid halt is an exception that indicates that the interrupt stack was not valid or that a memory error occurred while the processor was pushing information onto the interrupt 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. As usual for exceptions, this exception is taken independently of IPL. IPL is raised to 1F (hex). Implementation-specific information is pushed on the stack as longwords. The processor specifies the number of bytes pushed by placing the number of bytes pushed as the last longword pushed (0 if none, 4 if one,...). This count excludes the PC, PSL, and count longwords. Software can decide, 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 anywhere, 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 0 or 3, the behavior of the processor is UNDEFINED.

## 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.

### 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 5-4 illustrates the system control block base register.

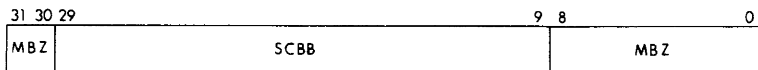



Figure 5-4 System Control Block Base Register (Read-Only)

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. Each vector is interpreted as follows by the hardware. 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.

1 Service this event on the interrupt stack. If this event is an exception, the IPL is raised to 1F (hex).

2 Service this event in writable control store, passing bits <15:2> to the installation-specific microcode there. If writable control store does not exist or is not loaded, the operation is UNDEFINED. On the VAX-11/780 processor, the operation in this case is a HALT.

3 Operation UNDEFINED. Reserved to DIGITAL. On the VAX-11/780 processor, the operation is a HALT.

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. Bits <1:0> in the CHMx vectors must be zero or the operation is UNDEFINED. On the VAX-11/780 processor, these bits are ignored in the CHMx vectors.

**System Control Block (exception and interrupt vectors)**

| Vector (hex) | Name | Notes |
|---|---|---|
| 00 | Unused | Reserved to DIGITAL. |
| 04 | Machine Check | Processor-and error-specific information is pushed on the stack, if possible. Restartability is processor-specific. Vector<1:0> must be 1 for meaningful operation. IPL is raised to 1F (hex). The number of bytes of parameters is pushed on the stack and is implementation-dependent. This vector causes an abort/fault/trap. |

| Vector (hex) | Name | Notes |
| --- | --- | --- |
| 08 | Kernel Stack Not Valid | Vector<1:0> must be 1 for meaningful operation. IPL is raised to 1F (hex). This vector causes an abort. There are zero parameters. |
| 0C | Power Fail | IPL is raised to 1E (hex). This vector causes an interrupt. There are zero parameters. |
| 10 | Reserved/ Privileged Instruction | Opcodes reserved to DIGITAL and privileged instructions. This vector causes a fault. There are zero parameters. |
| 14 | Customer Reserved Instruction | XFC instruction. This vector causes a fault. There are zero parameters. |
| 18 | Reserved Operand | This vector causes a fault/abort. There are zero parameters. |
| 1C | Reserved Addressing Mode | This vector causes a fault. There are zero parameters. For greater detail refer to the Architecture Handbook. |
| 20 | Access Control Violation | Virtual address, etc., causing fault is pushed onto stack. This vector results in a fault. There are two parameters. |
| 24 | Translation Not Valid | Virtual address, etc., causing fault is pushed onto stack. This vector results in a fault. There are two parameters. |
| 28 | Trace Pending (TP) | This vector results in a fault. There are zero parameters. For greater detail refer to the Architecture Handbook. |

| Vector (hex) | Name | Notes |
|---|---|---|
| 2C | Breakpoint Instruction | This vector results in a fault. There are two parameters. For greater detail refer to the Architecture Handbook. |
| 30 | Compatibility | A type code is pushed onto the stack. This vector results in a fault/abort. There is one parameter. |
| 34 | Arithmetic | A type code is pushed onto the stack. This vector results in a trap. There is one parameter. |
| 38-3C | Unused | Reserved to DIGITAL. |
| 40 | CHMK | The operand word is sign-extended and pushed onto the stack. Vector <1:0> MBZ. This vector results in a trap. There is one parameter. |
| 44 | CHME | The operand word is sign-extended and pushed onto the stack. Vector <4:0> MBZ. This vector results in a trap. There is one parameter. |
| 48 | CHMS | The operand word is sign-extended and pushed onto the stack. Vector <1:0> MBZ. This vector results in a trap. There is one parameter. |
| 4C | CHMU | The operand word is sign-extended and pushed onto the stack. Vector <1:0> MBZ. This vector results in a trap. There is one parameter. |
| 50-80 | Unused | Reserved to DIGITAL. |
| 84 | Software Level 1 | This vector results in an interrupt. There are zero parameters. |

| Vector (hex) | Name | Notes |
|---|---|---|
| 88 | Software Level 2 | Ordinarily used for AST delivery. This vector results in an interrupt. There are zero parameters. |
| 8C-BC | Software Levels 3-F | This vector results in an interrupt. There are zero parameters. |
| C0 | Interval Timer | IPL is 18 (hex). This vector results in an interrupt. There are zero parameters. |
| C4-F4 | Unused | Reserved to DIGITAL. |
| F8 | Console Terminal Receive | IPL is 14 (hex). This vector results in an interrupt. There are zero parameters. |
| FC | Console Terminal Transmit | IPL is 14 (hex). This vector results in an interrupt. There are zero parameters. |
| 100-1FC | Device Vectors | This vector results in an interrupt. There are zero parameters. |

In the VAX-11/780 processor, only hardware levels 14 to 17 (hex) 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 (hex) NEXUS 0-15
140-17C IPL 15 (hex) NEXUS 0-15
180-1BC IPL 16 (hex) NEXUS 0-15
1C0-1FC IPL 17 (hex) NEXUS 0-15

**STACKS**

At any time, the processor is either in a process context (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 associated with each of these five states, and any time the processor changes from one of these states to another, 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 PCB, although some hardware implementations may keep them in privileged registers. The interrupt stack pointer (ISP) is in a privileged register.

**Stack Residency**

The USER, SUPER, and EXEC stacks do not need to be resident. 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 are regarded as serious system failures, from which recovery is not possible.

If either of these faults occurs on a reference to the kernel stack, the processor aborts the current sequence and initiates kernel stack not valid abort on hardware level 1F (hex). If either of these faults occurs on a reference to the interrupt stack, the processor halts. Note that this does not mean that every possible reference is checked, but rather that the processor will not loop on these conditions.

It is not necessary that the kernel stack for processes other than the current one be resident, but it must be resident before a process is selected to run by the software's process dispatcher. Further, any mechanism that uses Translation Not Valid or Access Control Violation faults to gather process statistics, for instance, 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 on all processors, 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 on 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 Processor Status Longword (PSL) specify which of the five stack pointers is currently in use as follows:

| IS | MODE | REGISTER |
|---|---|---|
| 1 | 0 | ISP |
| 0 | 0 | KSP |
| 0 | 1 | ESP |
| 0 | 2 | SSP |
| 0 | 3 | USP |

The processor does not allow 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 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:

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

Values 10 (binary) and 11 (binary) of the vector<1:0> are used for other purposes.

**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. If the process stack pointers are implemented as registers, then these instructions are the only method for accessing the stack pointers of the current process. If the process stack pointers are kept in the PCB, the MTPR and MFPR of these registers access the 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. Note that interrupt service routines invoked at a priority higher than the lowest one used for context switching must not reference the process stack pointers. At bootstrap time, the contents of all stack pointers are UNPREDICTABLE. Figure 5-5 illustrates the process stack pointer.

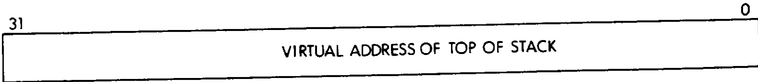



Figure 5-5 Process Stack Pointer Implemented As Read/Write Register

| | |
|---|---|
| Kernel Stack Pointer | KSP = 0 |
| Executive Stack Pointer | ESP = 1 |
| Supervisor Stack Pointer | SSP = 2 |
| User Stack Pointer | USP = 3 |
| Interrupt Stack Pointer | ISP = 4 |

## SERIALIZATION OF EXCEPTIONS AND INTERRUPTIONS

The sequence in which recognition of simultaneously occurring interrupts and exceptions takes place is indicated in the following list.

1. Machine check exception
2. Arithmetic exceptions
3. *Console halt or higher priority interrupt
4. Trace fault (only one per instruction)
5. Start instruction execution or restart suspended instruction

*the order in which console halt and interrupt recognition occur is not dictated by the VAX architecture (i.e., some future VAX machines may not take these in the same order as the VAX-11/780, which takes console halts before interrupts).

**NOTE**

> 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 PLS<T> to ensure that no additional trace faults occur when the suspended instruction is resumed.

As an example, if an instruction is started 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 initiating the new interrupt. 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 which occurs on initiation of 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. On the VAX-11/780, all UNDEFINEDs shown in the pseudo algol are HALTs.

```
if 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
{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;          !1F (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 (if vector<1:0> code is 0 or 1):

```
N ← 0;
Z ← 0;
V ← 0;
C ← 0;
```

Exceptions:

interrupt stack not valid
kernel stack not valid

Description:

The handling is determined by the contents of a longword vector in the system control block that is indexed by the exception of the interrupt being processed. If the processor is not executing on the interrupt stack, then 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 canonical state. 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 or abort or interrupt, the saved condition codes are UNPREDICTABLE. On an abort or fault or interrupt that sets FPD, the general registers except 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 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 on the kernel stack, a kernel stack not valid abort is initiated and IPL is changed to 1F (hex). The additional information, if any, associated with the original exception is lost. However, PSL and PC are pushed on the interrupt stack with the same values as would have been pushed on the kernel stack.
5. If the processor gets an access control violation or a translation not valid condition while attempting to push information on the interrupt stack, the processor is halted and only the state of ISP, PC, and PSL is ensured to be correct for subsequent analysis. The PSL and PC have the values that would have been pushed on 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 PC, PSL, and leaves SP as it was at the beginning of the instruction except for any pushes onto the kernel stack.
7. The value of PSL<TP> that is saved on the stack is as follows:

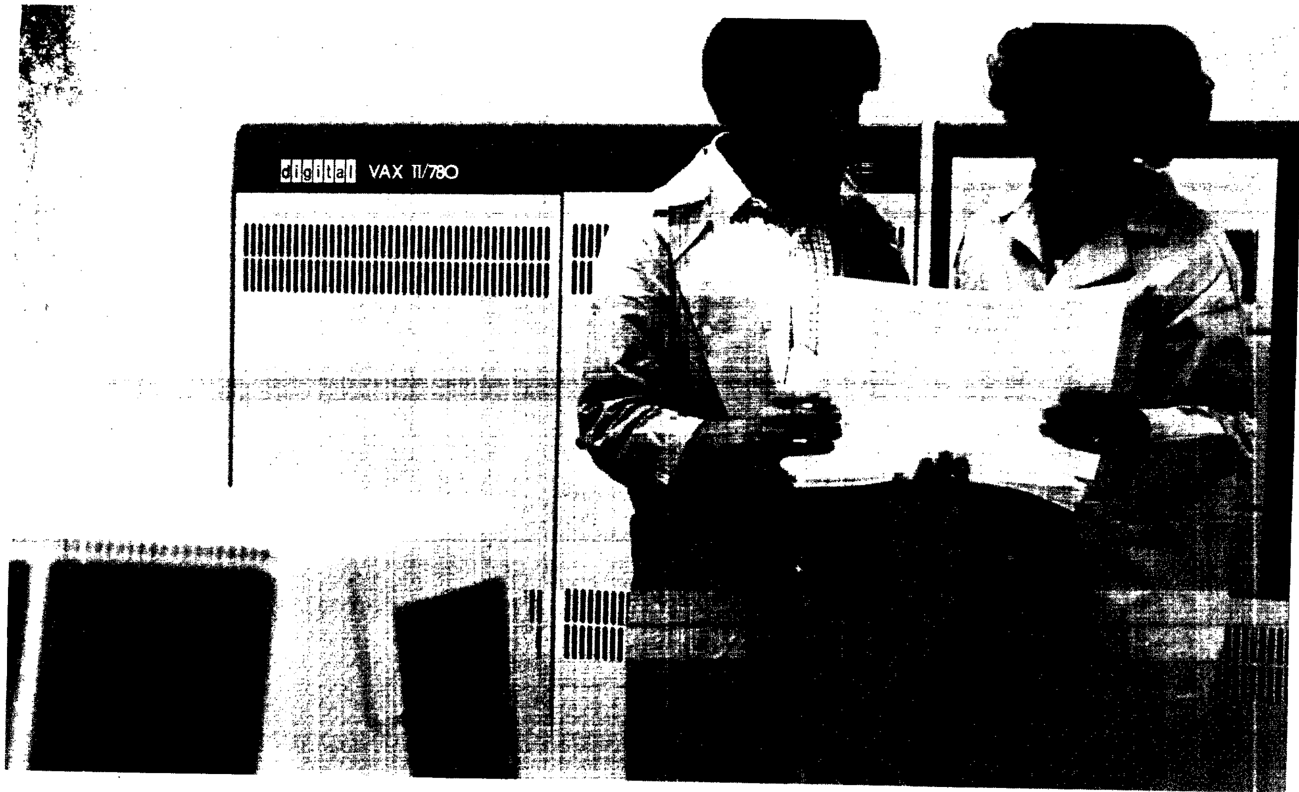
| | |
|---|---|
| fault | clear |
| trace | clear |
| interrupt | from PSL<TP> (if after traps, before trace) clear (otherwise) |
| abort | from PSL<TP> |
| trap | from PSL<TP> |
| CHMx | from PSL<TP> |

| | |
|---|---|
| BPT, XFC | clear |
| reserved instruction | clear |

8. The value of 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 non-interrupt stack pointers may be fetched and stored by hardware in either privileged registers or in their allocated slots in the PCB. Only LDPCTX and SVPCTX always fetch and store in the PCB (see Chapter 7). MFPR and MTPR always fetch and store the pointers whether in registers or the PCB.


digital VAX 11/780

## CHAPTER 6

# MEMORY MANAGEMENT

### INTRODUCTION

Memory management describes the hardware and software that control the allocation and use of physical memory. Typically, in a 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 lesser 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 utilizes this mapping information when it translates virtual addresses to physical addresses.

Therefore, memory management is the scheme that provides both the memory protection and memory mapping mechanisms of the VAX-11/780. The memory management scheme has been designed and implemented to achieve the following goals:

1. Provide a large address space for instructions and data.
2. Allow data structures up to one gigabyte.
3. Provide convenient and efficient sharing of instructions and data.
4. Contribute to software reliability.

A virtual memory system is used to provide a large address space, while allowing programs to run on small 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 4-billion-byte virtual address space.

The virtual address space is divided into two equal address spaces, the process address space and the system address space. The system address space is common for all processes and is not context switched. The operating system is located in 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 via a simple CALL. 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 4,294,967,296 bytes. This results from the fact that a virtual address is 32 bits in length. The virtual address space is broken into 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 into external storage media via the operating system.

The virtual address space is divided into two parts. The lower half, known as "per-process space," is distinct for each process running on the system. The upper half, known as "system space", is shared by all processes. Furthermore, the per-process virtual address space is divided into two equal parts, program space (P0 space) and control space (P1 space). Virtual address space is illustrated in Figure 6-1.

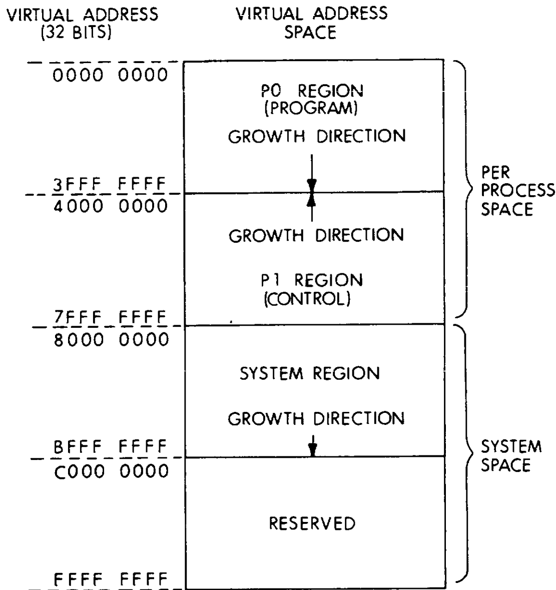



Figure 6-1 Virtual Address Space

**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 disjoint. The address map for per-process space is context-switched when the process running on the system is changed.

**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**

Independently 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 por-

tions 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 the process executes, the system translates virtual addresses to physical addresses. The virtual address format is illustrated in Figure 6-2.

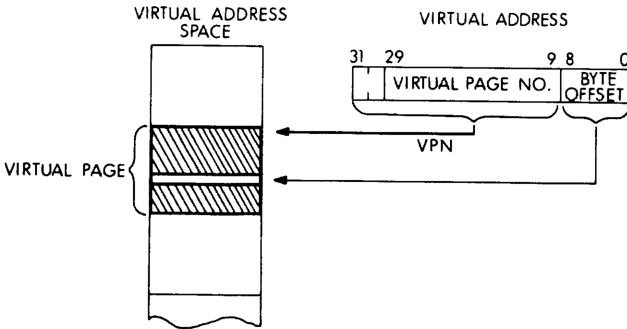



Figure 6-2 Virtual Address

**Bits<31:9> Virtual Page Number**

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 set, the address is system virtual. Bit 30 is used in conjunction with process virtual addresses to distinguish between the program and control regions. When bit 30 is set, the control region is referenced, and when it is clear, the program region is referenced.

**Bits<8:0> Byte**
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.

## 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 number of bits is implementation-dependent. This section describes the address translation process when MME is 1.

The address translation routine 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 the 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 followed 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 6-3.

Figure 6-3 Page Table Entry

**Bit<31> Valid bit**
The operating system uses the V bit to indicate whether the corre sponding page is part of the process working set (i.e., the set of physical pages currently being used by the process). If the V bit = 0 (not valid), the page is not in the working set. The hardware then issues a Translation Not Valid fault (i.e., "page fault") during address translation. The pager retrieves this page and brings it into physical memory allowing continued execution of the process image. The V bit governs the validity of the M bit and PFN field.

**Bits<30:27> Protection code**
The protection code specifies read-write access to each page. This field is explained more fully under the ACCESS CONTROL section of this chapter. This field is always valid and is used by the hardware even when V=0.

**Bit<26> Modify bit**
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 a multiprocessor system.

**Bit<25> Reserved to DIGITAL**
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.

**Bits<24,23> Reserved**
Reserved for software use 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.

**Bits<22,21> Reserved to DIGITAL**
These bits are reserved to DIGITAL and must be zero. The hardware does not necessarily test that these are zero because the PTE is established only by privileged software.

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

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

## 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.

The protection code is encoded in a 4-bit field in the Page Table Entry described in Table 6-1. Associated with each protection code is the access status for each of the access modes. During address translation, the protection code is the first field in the PTE that is checked.

**Table 6-1 Protection Codes**

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

- =no access
R=read only
RW=read/write

K=Kernel
E=Executive
S=Supervisor
U=User

Software symbols are defined using PTE$K_ as a prefix to the mnemonics listed in Table 6-1.

This code was chosen to keep the complexity of hardware access checking reasonable for implementations not using a table decoder. The access is allowed if:
(CODE NEQU 0) AND
((CODE EQLU 4) OR (CM LSSU WM) OR (READ AND (CM LEQU RM)))

CM is current mode
RM is left two bits of code
WM is 1's complement of right two bits of code

**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.

**SYSTEM SPACE ADDRESS TRANSLATION**
A virtual address with <31:30>=2 (i.e., binary 10) is an address in the system virtual address space as illustrated in Figure 6-4.

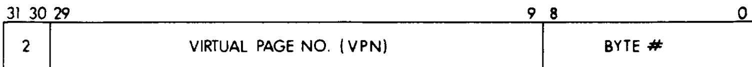



Figure 6-4 System Space Address

The system virtual address space is defined by the System Page Table (SPT), which is a vector of page table entries (PTEs). The system page table is always located in physical address space. Therefore the base address of the SPT is a physical address and is located in the System Base Register (SBR) described in Figure 6-5. The size of the SPT in longwords, i.e., the number of PTEs, is contained in the System Length Register (SLR) described in Figure 6-6. The SBR points to the first PTE in the SPT. In turn, this PTE maps the first page of system virtual space, i.e., virtual byte address 80000000 (hex).

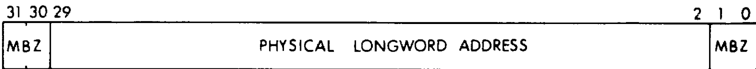



Figure 6-5 System Base Register

Figure 6-6 System Length Register

The virtual page number is contained in bits <29:9> of the virtual address. Thus, there could be as many as $2^{21}$ 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 is UNPREDICTABLE. The translation from system virtual address to physical address is illustrated in Figure 6-7.

Thus, the arithmetic necessary to generate a physical address from a system region virtual address is:

SYS_PA = (SBR+SVA<29:9>*4)<20:0>'SVA<8:0> !System Region

**NOTE**

For all occurrences within this chapter, the parentheses indicate contents of, the angle brackets indicate referenced bits, and the apostrophe indicates concatenation.

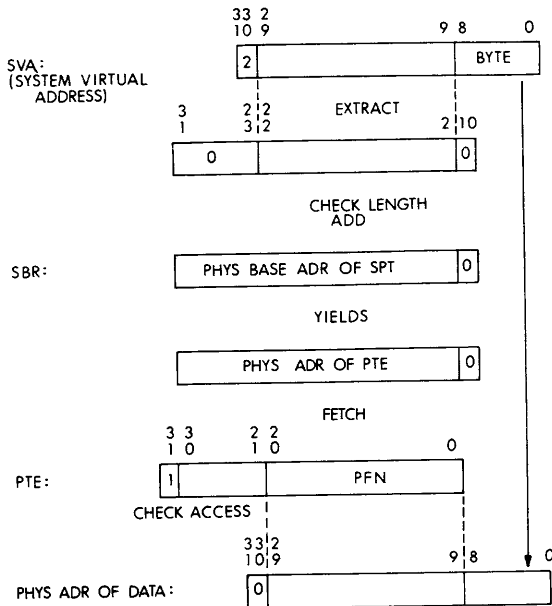



Figure 6-7 System Virtual To Physical Translation

## 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 0, the P0 region of the address space is selected, and if bit 30 is 1, the P1 region is selected.

The P0 region of the address space maps a virtually contiguous area that begins at the smallest address (0) in the process virtual space and grows in the direction of larger addresses. In contrast, the P1 region of the address space maps a virtually 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 a virtually contiguous vector of page table entries. In contrast with the system page table, which is located in physical address space, the two process page tables are located in system virtual address space. Thus, for process space, the address of a PTE is a virtual address in system space, and the fetch of a PTE is simply a fetch of a longword using a system virtual address.

There is a significant reason to address process page tables in virtual rather than physical space. A physically addressed process page table that required more than a page of PTEs (i.e., that mapped more than 64K bytes of process virtual space) would require physically contiguous pages. Such a requirement would make dynamic allocation of process page table space very awkward.

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" (i.e., 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 6-8. 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 6-9. 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 0.

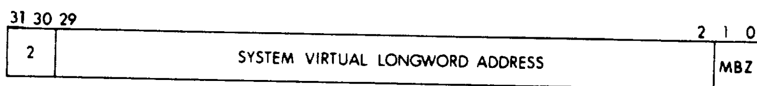



Figure 6-8 P0 Base Register

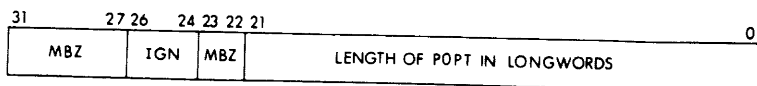



Figure 6-9 P0 Length Register

The virtual page number is contained in 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> is ignored on MTPR and read back 0 on MFPR. At bootstrap time, the content of both registers is 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 6-10.

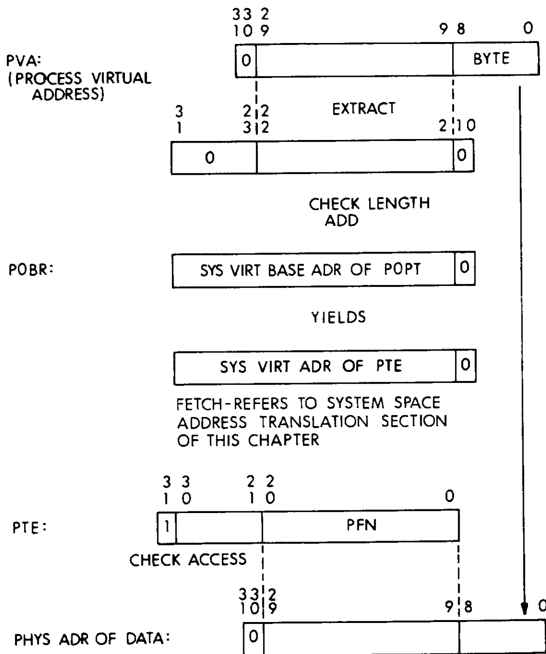



Figure 6-10 P0 Virtual To Physical Translation

Thus, the arithmetic necessary to generate a physical address from a P0 region virtual address is:

```
PVA_PTE=  (PVA<29:9>*4)+P0BR             !P0 Region
PTE_PA=   (SBR+PVA_PTE<29:9>*4)<20:0>'PVA_PTE<8:0>
PROC_PA=  (PTE_PA)<20:0>'PVA<8:0>
```

**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 backwards, 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, i.e., virtual byte address 40000000 (hex). The P1 base register is illustrated in Figure 6-11. P1LR contains the number of nonexistent PTEs. The P1 length register is illustrated in Figure 6-12.

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

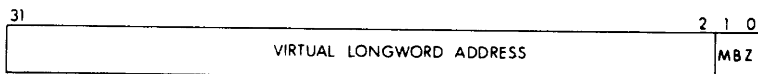



Figure 6-11 P1 Base Register (Read/Write)

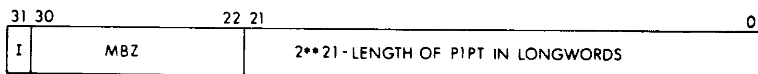



Figure 6-12 P1 Length Register (Read/Write)

P1LR<31> is ignored on MTPR and reads back 0 on MFPR. At bootstrap time, the content of both registers is UNPREDICTABLE. An attempt to load P1BR with a value less than $2^{31}—2^{23}$ (7F800000, hex) results in a reserved operand fault. The translation from P1 virtual address to physical address is illustrated in Figure 6-13.

Thus, the arithmetic necessary to generate a physical address from a P1 region virtual address is:

```
PVA_PTE=  (PVA<29:9>*4)+P0BR              !P1 Region
PTE_PA=   (SBR+PVA_PTE<29:9>*4)<20:0>'PVA_PTE<8:0>
PROC_PA=  (PTE_PA)<20:0>'PVA<8:0>
```

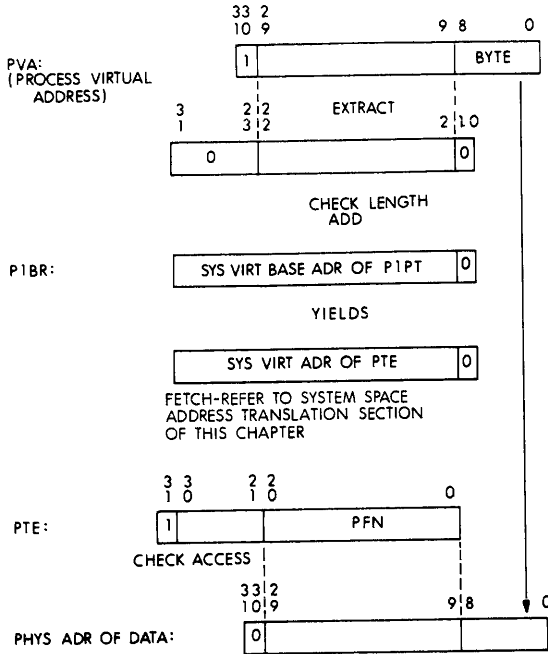



Figure 6-13 P1 Virtual To Physical Translation

## 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 0 or 1 according to whether memory management is disabled or enabled respectively. The map enable register is a privileged register and is illustrated in Figure 6-14.

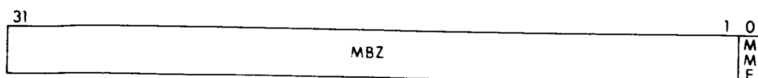



Figure 6-14 Map Enable Register (Read/Write)

At bootstrap time, this register is initialized to 0.

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, a hardware implementation may include 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 translation buffer invalidate single register (TBIS). The TBIS register is illustrated in Figure 6-15.

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). The TBIA register is illustrated in Figure 6-16.

Since the content of the translation buffer at bootstrap time is UNPREDICTABLE, the entire translation buffer must be cleared by moving 0 into TBIA before enabling memory management.

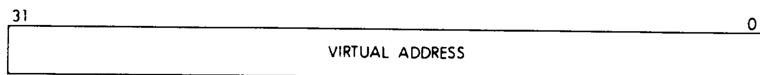



Figure 6-15 Translation Buffer Invalidate Single (write only)

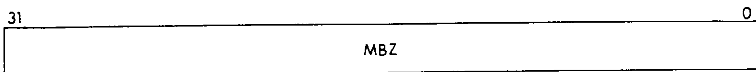



Figure 6-16 Translation Buffer Invalidate All (write only)

**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 6-17.

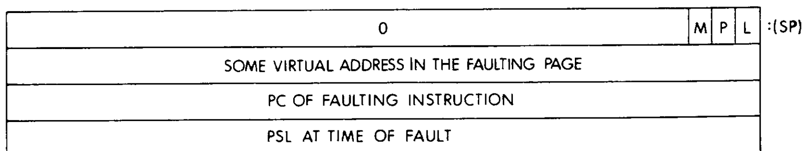



Figure 6-17 Fault Parameter Word

The same parameters are stored for both types of fault. 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 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:

| | |
|---|---|
| L<0> | Length Violation. Set to 1 to indicate that an Access Control Violation was the result of a length violation rather than a protection violation. This bit is always 0 for a Translation Not Valid fault. |
| P<1> | PTE Reference. Set to 1 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. |
| M<2> | Write or Modify Intent. Set to 1 to indicate that the program's intended access was a write or modify. This bit is 0 if the program's intended access was a read. |

## 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.

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

(Refer to the Architecture Handbook for greater detail.) These instructions provide the only mechanism for the less privileged code to call the 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.

### 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.

## ISSUES

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?

### Size Of System Page Table

To examine the size of the system page table, note first that one page of the SPT maps 64K Bytes 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 96K Bytes, 1 to 1.5 pages.
2. Memory management data for physical page management: 4 to 6 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 = 1.5M Bytes of physical memory.
3. Shared code
   - command interpreter
   - debugger
   - record manager
   - OTSes, FORTRAN, COBOL, BASIC

Allowing 16K Bytes for each of the above items, the total is 96K Bytes or 1-½ pages of system page table.

4. Process page tables. One longword of SPT maps one page of process page table which in turn maps 65K Bytes of process virtual address space. 16 longwords of SPT maps 1M Byte of process virtual address space. One page of SPT maps 8M Bytes. A very straightforward balance set management design that re-

served a fixed (SYSGEN) number of balance set slots, each with a fixed (also SYSGEN) maximum virtual address space, would use only 2 pages of SPT to allow 16 processes of up to 1M Byte each in the balance set.

It would appear from the foregoing analysis that a 6-page SPT would handle a very reasonable system, and that increasing the 1M Byte process virtual space to 4M Bytes and 16 processes in the balance set would add only 6 more pages of SPT for a total of 12. A smaller system with 256K Bytes of memory and 8 balance set processes, each 512K Bytes maximum size, would need about 3 pages of SPT.

**Sharing**

To discuss sharing, it is useful to assume 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 ORed into the shared PTE. Then the share count is decremented, and if the count is now 0, the page is released and the shared PTE is updated to reflect that. Note that the process's working set data base allows it to find its private PTE, and the physical page data base 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 seems to be 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 simply to reference such a shared section without ever having declared its intention to do that, the operating system must be prepared to do something reasonable when such a 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 could be done statically at compile or link time or dynamically at runtime.
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. Of course, the balance set manager maintains share counts on the section and discards its pages only 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 considered 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. There would be a list of the pages in use in that pool, and a fault for a new one would cause one in the pool to be 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.

**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.

The motivation for this design is the behavior the PROBE instruction would exhibit if the valid bit had to be set before it could check protection. PROBE would actually have to fault in the page to make it valid, so that it could check the protection and then indicate whether

or not the intended access was permissible. For the vast percentage of PROBE instructions, the access is permitted and faulting the page in the PROBE is reasonable. But a program could be run in user mode that would PROBE all around in the system region of the virtual address space faulting all the swappable pages of the system. Though this would not violate the integrity of the operating system, it certainly would mess up any statistics that the system might be gathering about the relative worth of the swappable pages.

**EXAMPLE**

Appendix F contains a virtual to physical address translation.

## CHAPTER 7

# 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 7-1.

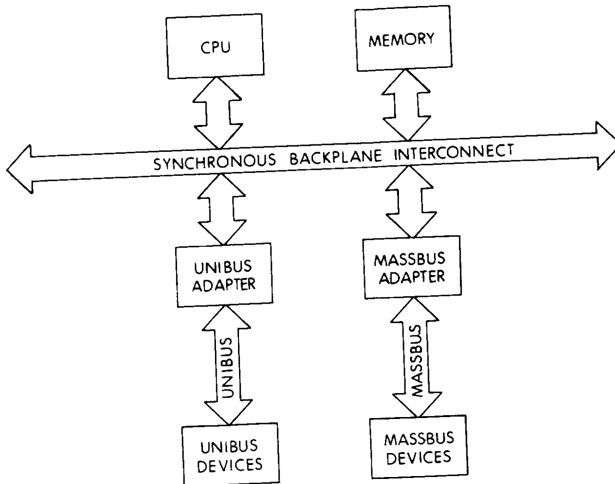



Figure 7-1 Basic Bus Configuration

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 7-2. 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

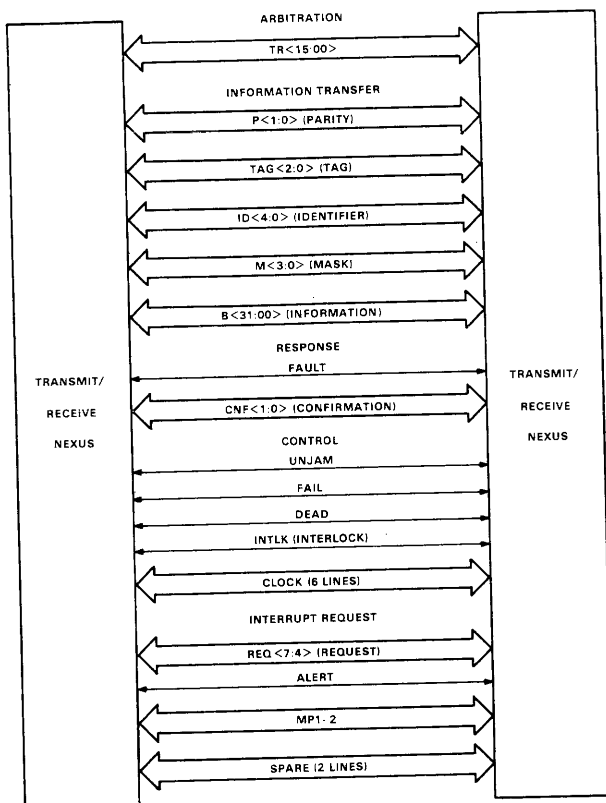



Figure 7-2 SBI Signal Description

**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 7-3). 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.

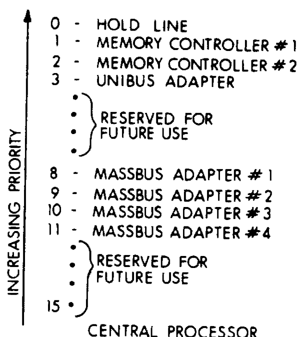



Figure 7-3 SBI Priority Access

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:

| **Field** | **Length in Bits** |
|---|---|
| Parity check | 2 (P <1:0>) |
| Information Tag | 3 (TAG <2:0>) |
| Source/Destination Identity | 5 (ID <4:0>) |
| Mask | 4 (M <3:0>) |
| Information | 32 (B <31:00>) |

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, Indentifier, and Mask fields and P1 as parity for the Information field. The parity field is illustrated in Figure 7-4.

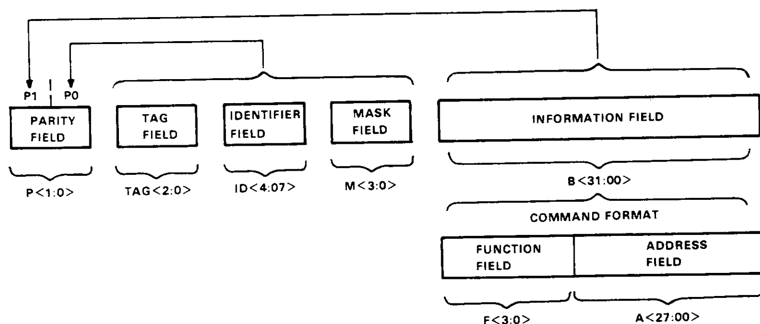



Figure 7-4 Parity Field Configuration

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.

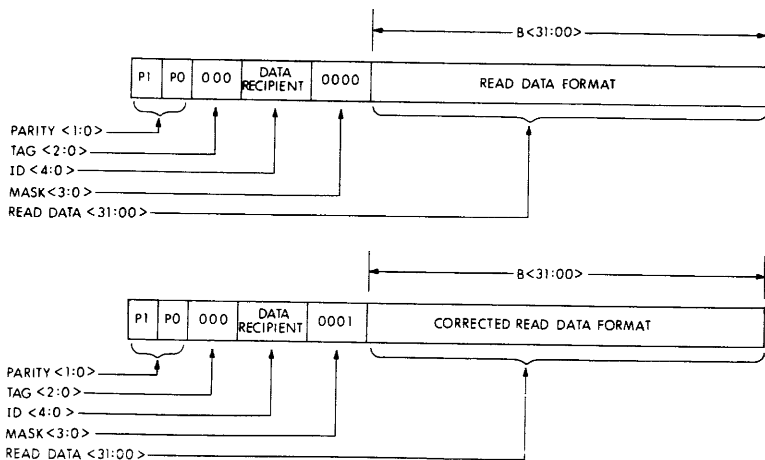
Four tag fields and four reserved fields are defined as:

| TAG <2:0> | B<31:00> contents |
|---|---|
| 000 | READ DATA |
| 011 | COMMAND ADDRESS |
| 101 | WRITE DATA |
| 110 | INTERRUPT SUMMARY READ |

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 7-5.



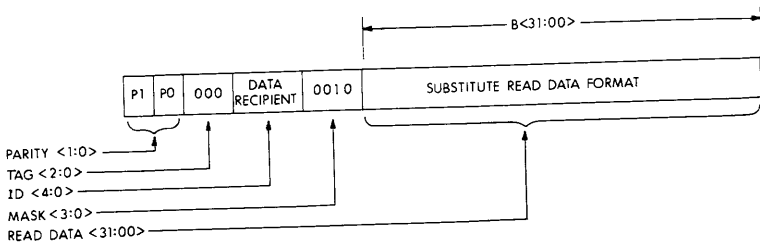



Figure 7-5 Read Data Tag Formats

- 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 7-6, B<31:00> is divided into a function field and an address field to specify the command and its associated address.

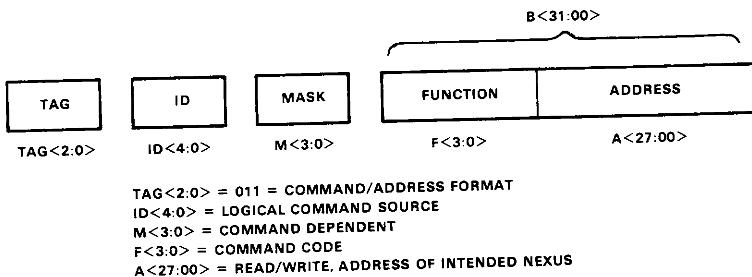



Figure 7-6 Command/Address Format

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 7-7 illustrates the VAX-11/780 physical address space. Note that both physical and SBI addresses are provided.

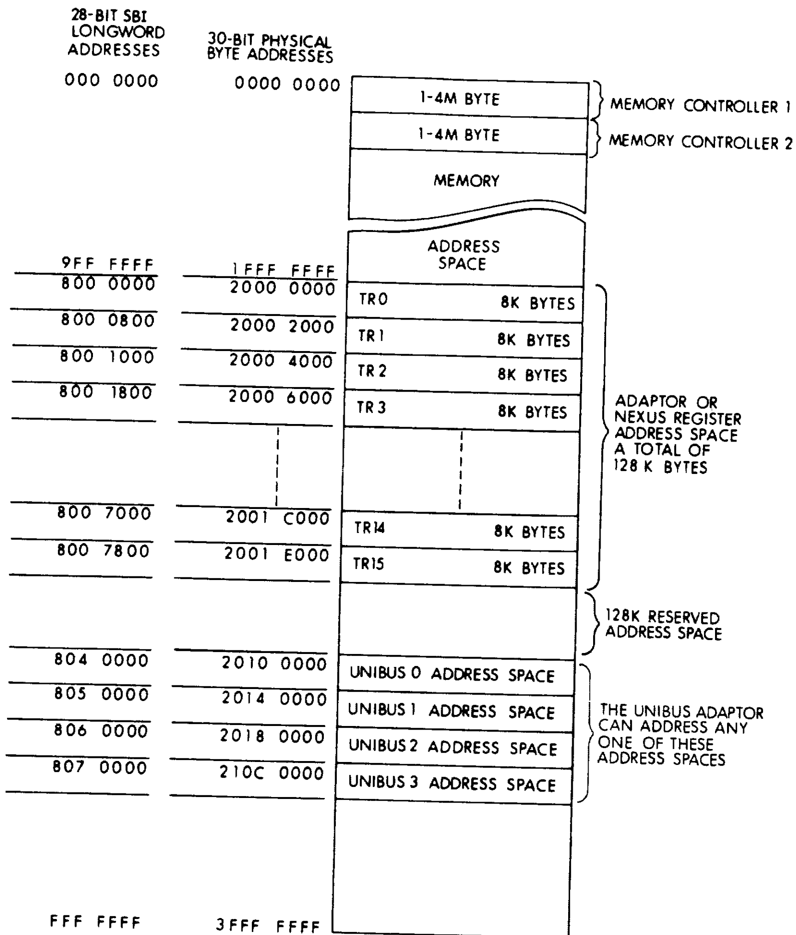



Figure 7-7 SBI Physical Address Space

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 7-8.

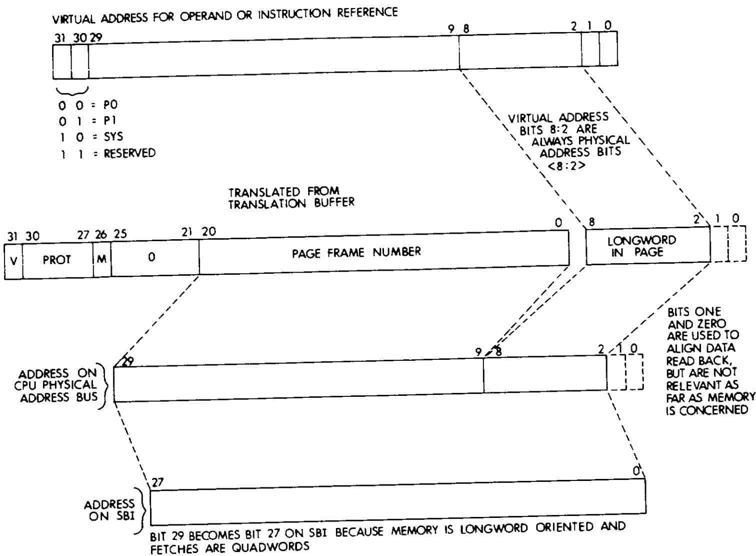



Figure 7-8 Physical To SBI Address Translation

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 7-9.

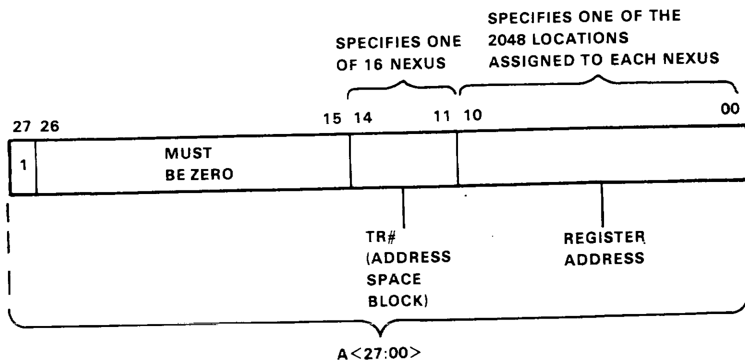
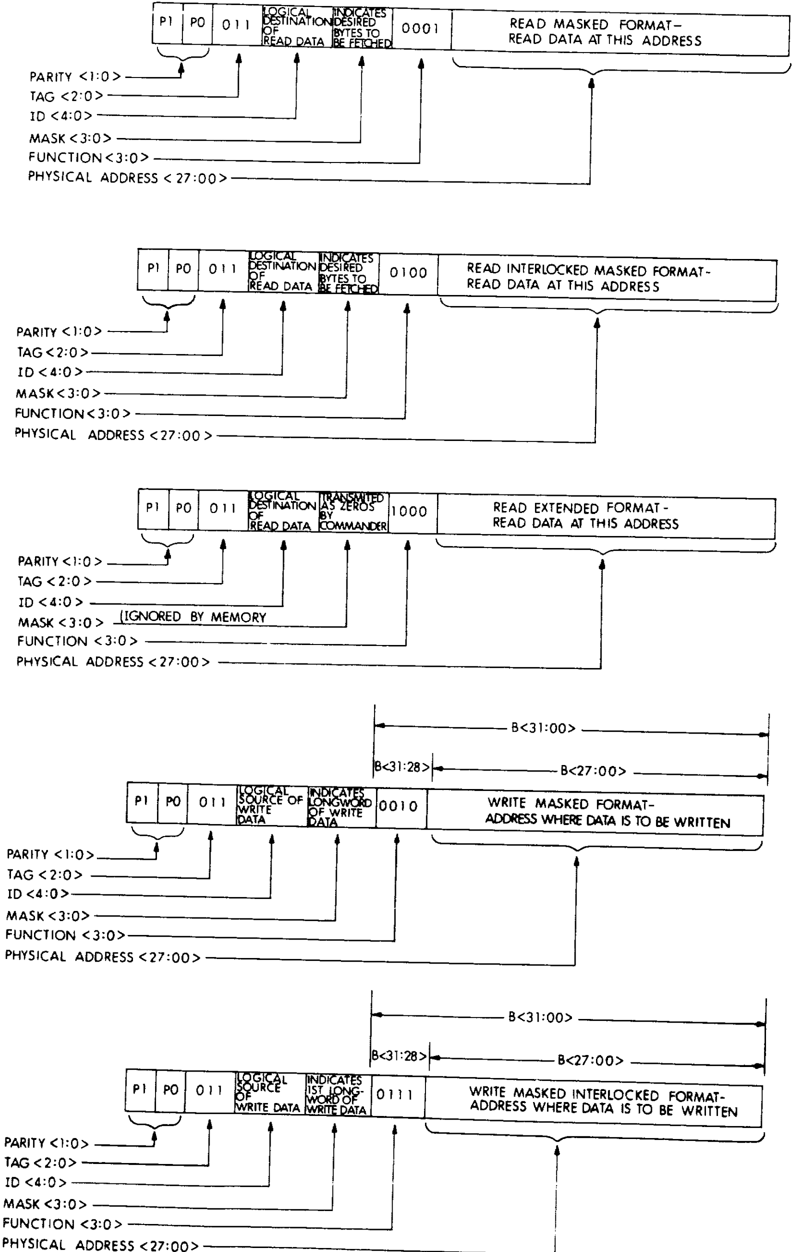



Figure 7-9 Control Address Space Assignment

The command/address tag formats are illustrated in figure 7-10.



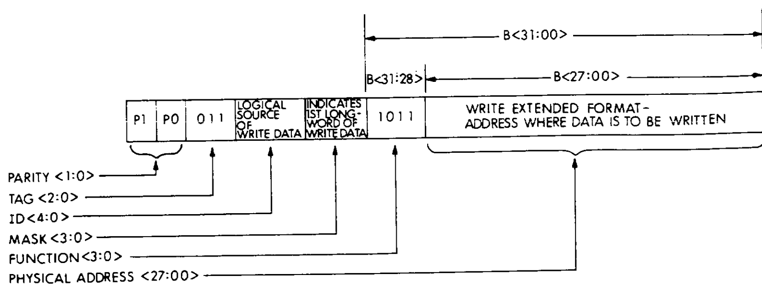



Figure 7-10 Command/Address Tag Formats

- 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 7-11 illustrates the write data tag format.

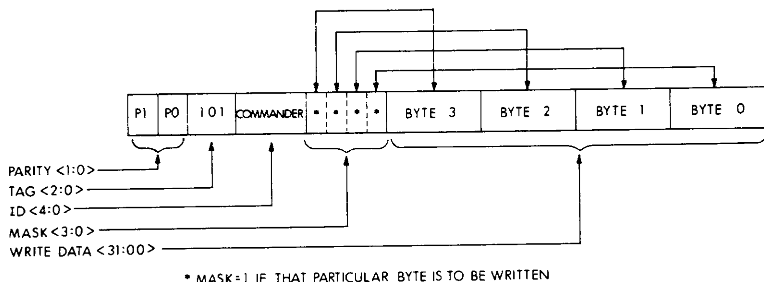



Figure 7-11 Write Data Tag Format

- 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.

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 7-12 illustrates the interrupt summary tag format.

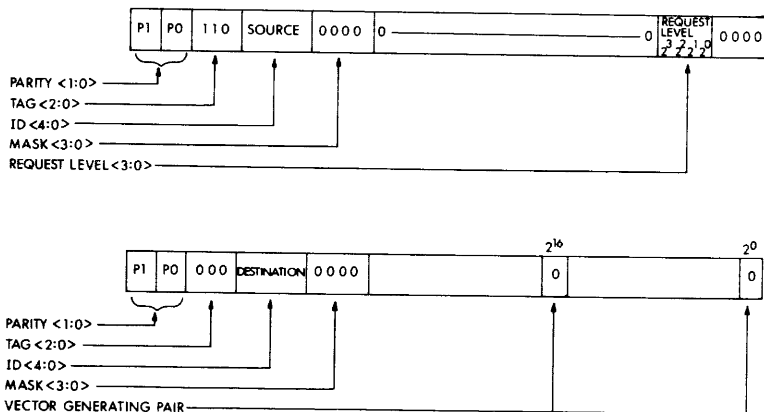



Figure 7-12 Interrupt Summary Tag Format

- Reserved Tags

Tag (<2:0>=111) is reserved for diagnostic purposes. Tag codes 001, 010, and 100 are unused and reserved for future definition.

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 7-13 each bit in the mask field corresponds to a particular byte of B<31:00>.

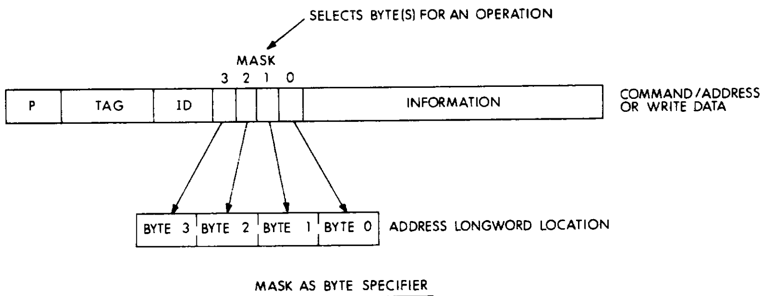



Figure 7-13 Mask Field Format

As previously mentioned, the secondary interpretation is used when Tag<2:0> = 000 (read data). Figure 7-14 illustrates the read data mask field.

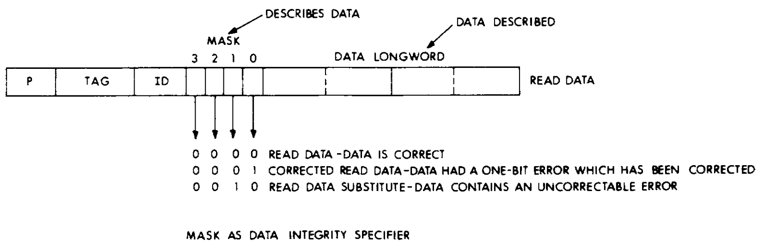



Figure 7-14 Read Data Mask Format

**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 7-1.

**Table 7-1 Confirmation Code Definitions**

| CNF Code | Definitions |
|---|---|
| 00, No Response (N/R) | The unasserted state and indicates no response to a commander's selection. |
| 01, Acknowledge (ACK) | The positive acknowledgement to any transfer. |
| 10, Busy (BSY) | The response to a command/address transfer, and indicates successful selection of a NEXUS which is presently unable to execute the command. |
| 11, Error (ERR) | The response to a command/address transfer, and indicates selection of a NEXUS which cannot execute the command. |

A BSY (10) or ERR (11) response to transfers other than command/address transfers will be considered as no response from the transmitter.

**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 μsec before the clock becomes inoperative. With power restart, the clock will be operational for at least two μsec before DC LO is negated. The negation of DC LO negates Dead.

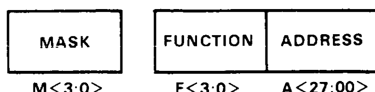
FAIL FUNCTION
A NEXUS asserts the Fail (FAIL) function asynchronously to the SBI clock when the power supply AC LO signal is asserted to that NEXUS. The assertion of Fail inhibits the CPU from initiating a power up service routine. Fail is negated asynchronous to the SBI clock when all NEXUSs that are 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 7-15. 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 8, 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.




| MASK USE | FUNCTION CODE | FUNCTION DEFINITION |
|---|---|---|
| IGNORED | 0000 | RESERVED |
| USED | 0001 | READ MASKED |
| USED | 0010 | WRITE MASKED |
| IGNORED | 0011 | RESERVED |
| USED | 0100 | INTERLOCK READ MASKED |
| IGNORED | 0101 | RESERVED |
| IGNORED | 0110 | RESERVED |
| USED | 0111 | INTERLOCK WRITE MASKED |
| IGNORED | 1000 | EXTENDED READ |
| IGNORED | 1001 | RESERVED |
| IGNORED | 1010 | RESERVED |
| USED | 1011 | EXTENDED WRITE MASKED |
| IGNORED | 1100 | RESERVED |
| IGNORED | 1101 | RESERVED |
| IGNORED | 1110 | RESERVED |
| IGNORED | 1111 | RESERVED |

Figure 7-15 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 7-16 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 7-13 and 7-14 illustrate SBI write functions. Figure 7-17 illustrates a single SBI write transaction while Figure 7-18 illustrates two SBI write transactions from devices A and B.

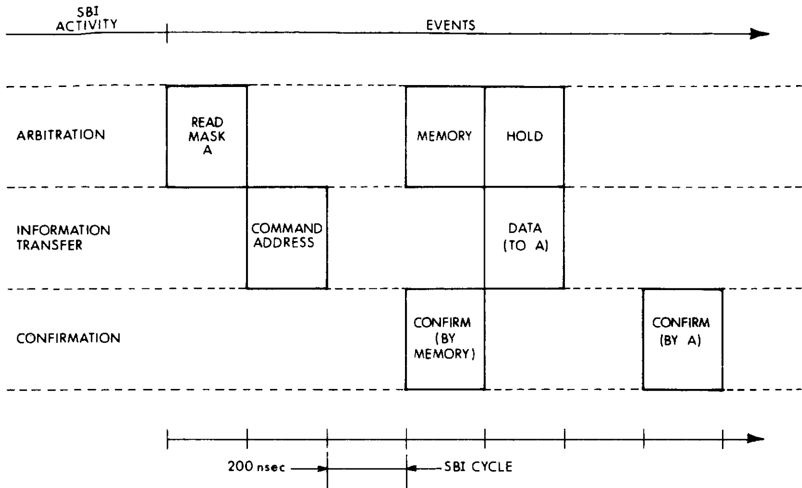



Figure 7-16 SBI Read Transaction

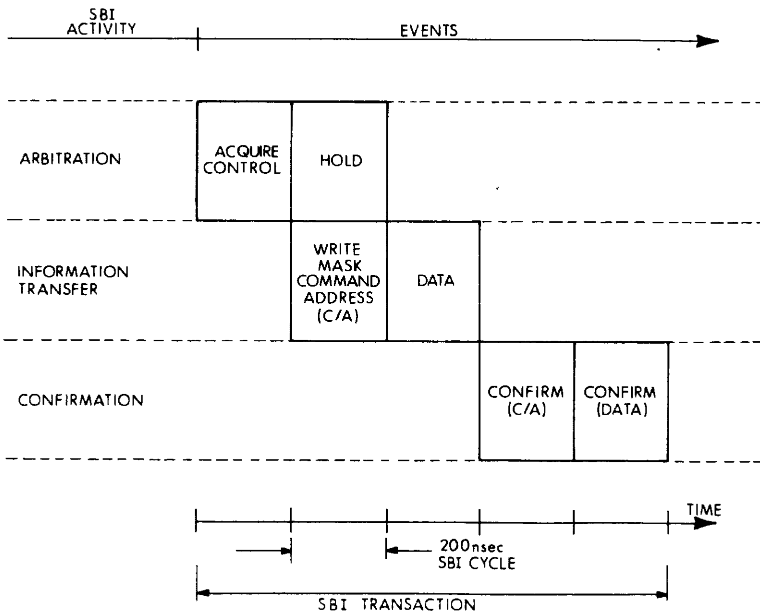



Figure 7-17 Single SBI Write Transaction

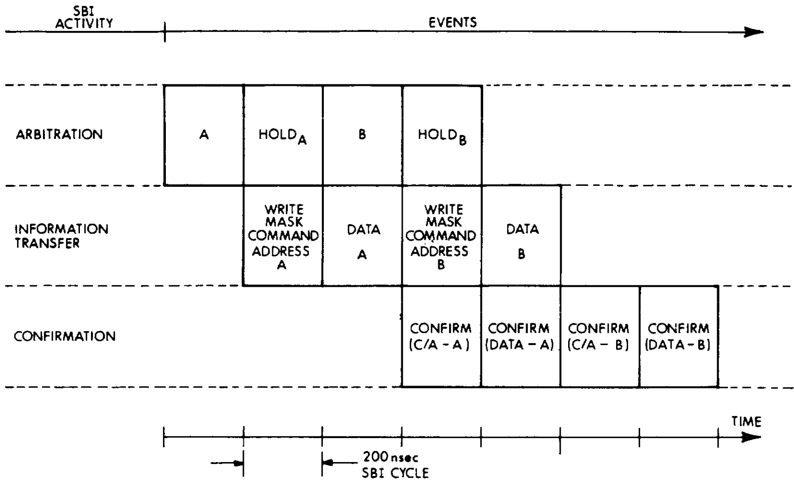



Figure 7-18 Two SBI Write Transactions

**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

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 7-19 illustrates extended read transactions by two separate devices, A and B, reading memory via a single memory controller.

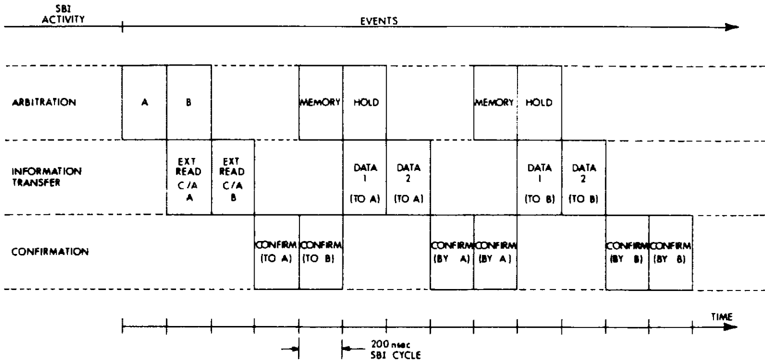



Figure 7-19 Extended Read Transactions Via Single Memory Controller

Figure 7-20 illustrates extended read transactions by two separate devices, A and B, reading memory via separate memory controllers, M1 and M2.

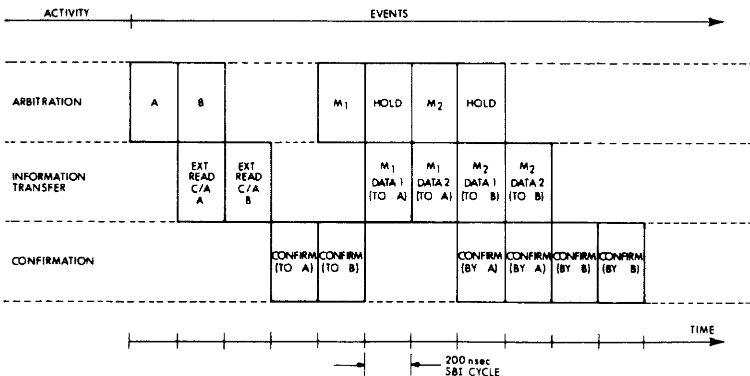



Figure 7-20 Extended Read Transactions Via Separate Memory Controllers

**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 8

# 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 either 4K or 16K N-channel MOS IC storage elements. Each array board can contain 64K or 256K bytes of memory, giving the system a capacity of either one or four megabytes, depending upon size of storage chips used.

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.

The basic memory subsystem is illustrated in Figure 8-1.

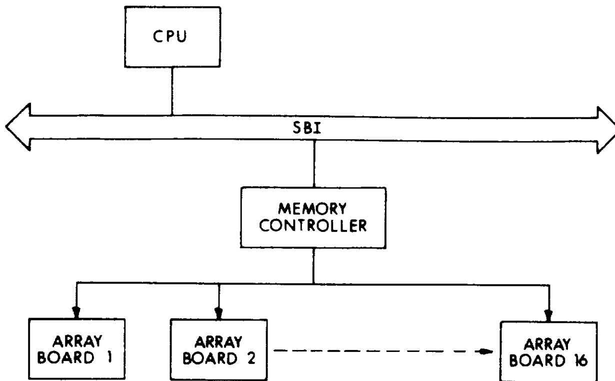



Figure 8-1 Main Memory Configuration

**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 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 7, SYNCHRONOUS BACKPLANE INTERCONNECT, the physical address space is divided into two equal areas, memory address space, and I/O address space. Figure 8-2 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 2 Mbytes of main memory utilizing the 4K chip design and 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

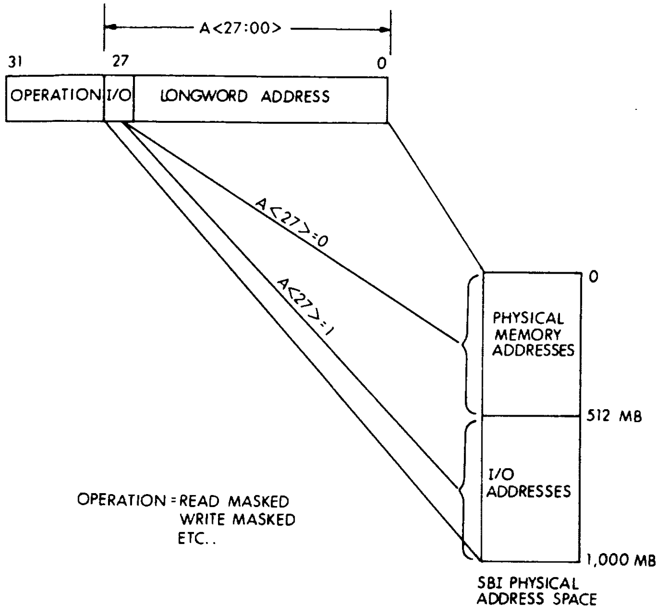



Figure 8-2 Physical Address Space

- 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 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 an 8K 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**

Figure 8-3 illustrates memory configuration register A.

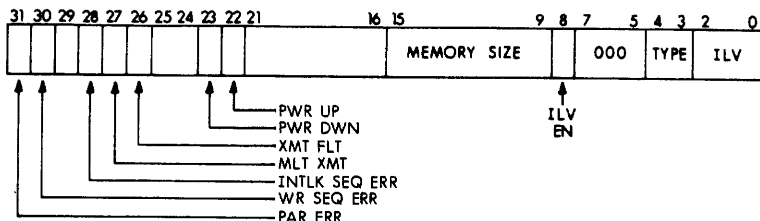



Figure 8-3 Memory Configuration Register A

Register A contains the following information:

**Bits <31:26> SBI Fault Status**

**Bits <23:22> Power Up Power Down Status**

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 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.

**Bits <15:09> Memory Size**

These bits contain the binary representation of the memory size in 64K byte increments, zero inclusive. For the 4K chip, bits <12:09> are used. For the 16K chip, bits <14:09> are used. These bits are read-only.

**Bits <04:03> Memory Type**

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.

| Bit <4> | Bit <3> | Description |
|---|---|---|
| 0 | 0 | Error condition, no array cards plugged in. |
| 0 | 1 | 4K chip |
| 1 | 0 | 16K chip |
| 1 | 1 | Error condition, both 4K and 16K chip array boards are being used. |

**Bits <02:00> Interleave**
These bits contain interleave information. If bit 00 is 0, the memory is not interleaved. If bit 00 is a 1, the system is interleaved. Bits 01 and 02 are not used at this time and should be 0. Bit 08 is the interleave write enable bit. When bit 08 is written to with a one, bit 00 will take on whatever state bit 00 in the written data is. Bit 08 will always read as a 0. If bit 08 is written to with a 0, interleave bit 00 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 8-4 illustrates memory configuration register B.

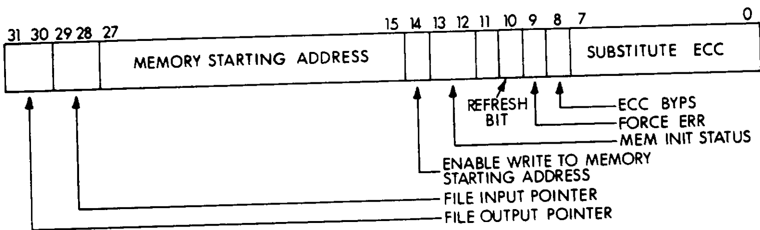



Figure 8-4 Memory Configuration Register B

Register B contains the following information:

**Bits <31:30> File Output Pointer (File Read Address Counter)**
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.

**Bits <29:28> File Input Pointer (File Write Address Counter)**
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.

**Bits <27:15> Memory Starting Address**
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 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.

| Controller No. | Starting Address Jumpers<br>SA 01 SA 00 | Starting Address |
|---|---|---|
| 1 | OPEN OPEN | zero |
| 2 | OPEN GND | 4 Megabyte |
| 3 | GND OPEN | 8 Megabyte |
| 4 | GND GND | 12 Megabyte |

Also during battery backup the contents of the starting address bits are saved.

**Bit <14> Write Enable to Memory Starting Address**
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.

**Bits <13:12> Memory Initialization Status**
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.

| Bit <12> | Bit <13> | Description |
|---|---|---|
| 0 | 0 | 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. |
| 0 | 1 | Invalid state |

| Bit <12> | Bit <13> | Description |
|---|---|---|
| 1 | 0 | This state means the memory contains valid data. This state after a CPU Power Fail implies that all memory data was saved. |
| 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> Refresh Indication**
This bit is used for diagnostic purposes only, and will verify the access time delay due to refresh collision.

**Bit <09> Force ERR**
This bit is used in conjunction with bits <07:00>. 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 <08> ECC BYPS**
This bit is set to totally bypass (BYPS) 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.

**Bits <07:00> Substitute ECC Bits**
These bits can be substituted for the eight check bits read from the memory, providing that bit <09> 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 8-5 illustrates memory configuration register C.

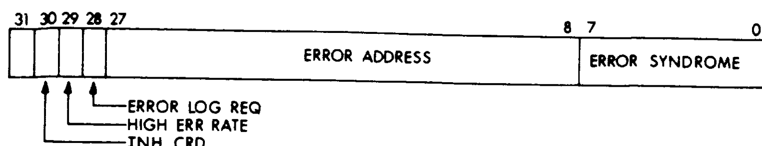



Figure Figure 8-5 Memory Configuration Register C

This register gathers all the ECC error information:

**Bit <30> Inhibit CRD**
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> HIGH ERROR RATE**
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> ERROR LOG REQUEST FLAG**
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 <27:08> ERROR ADDRESS**
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 <08> indicates the word in error.
0 = lower word
1 = upper word

Bits <20:09> 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.

**BIT <7:00> ERROR SYNDROME**
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 contiguous. 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).

# CHAPTER 9
# UNIBUS SUBSYSTEM

## INTRODUCTION

The UNIBUS Subsystem is the hardware developer's primary interface to VAX-11/780. All devices other than the high-speed disk drives and magnetic tape transports 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 9-1 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

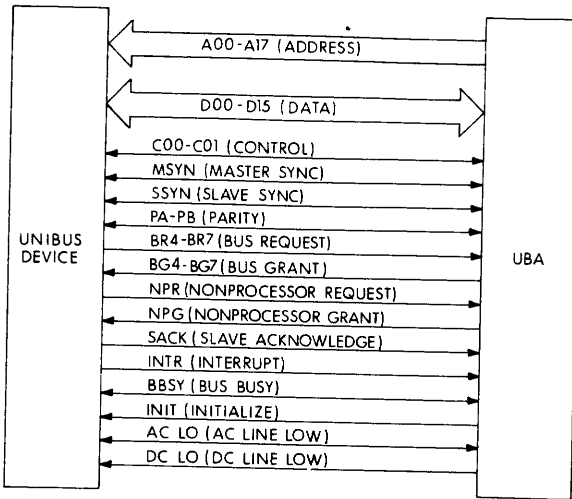



Figure 9-1 UNIBUS Configuration

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

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 9-1 describes the bus signals within each group.

**Table 9-1 UNIBUS Signal Descriptions**

| SIGNAL LINE | DESCRIPTION |
|---|---|
| **Data Transfer Group** | |
| Address Lines (A<17:00>) | These lines are used by the master device to select the slave (actually a unique memory or device register address). A <17:01> specifies a unique 16-bit word; SA00 specifies a byte within the word. |
| Data Lines (D<15:00>) | These lines transfer information between master and slave. |
| Control (C1,C0) | 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. |

**Data Transfer Designation Description**

| C1 | C0 | Description |
|---|---|---|
| 0 | 0 | Data in (DATI): a data word or byte transferred into the master from the slave. |

| | |
|---|---|
| 0 1 | Data in Pause (DATIP): similar to DATI except that it is always followed by a DATO or DATOB to the same location. The master keeps control of the UNIBUS during the entire DATIP-DATO sequence. |
| 1 0 | Data Out (DATO): a data word is transferred out of the master to the slave. |
| 1 1 | Data Out Byte (DATOB): identical to DATO except that a byte is transferred instead of a full word. Address bits A00 determine which byte will be written. A00=0, low byte (D07-00) is written. A00=1, high byte (D15-08) is written. |

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.

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
- UNIBUS power fail sequencing

The UNIBUS Subsystem is illustrated in Figure 9-2.

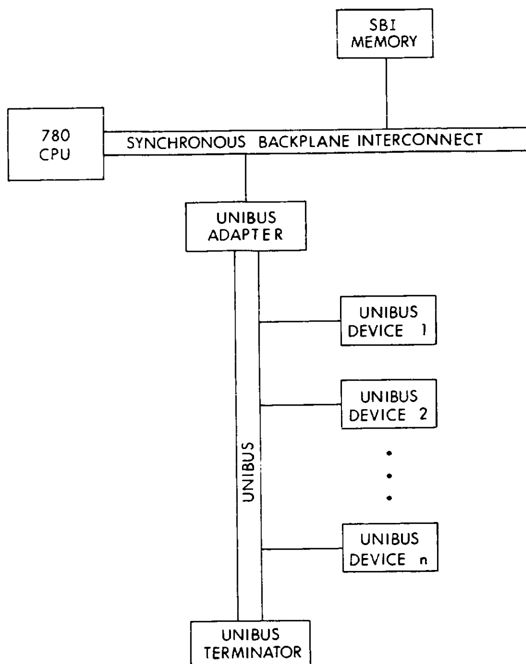



Figure 9-2 UNIBUS Subsystem

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 9-3 illustrates the SBI I/O address space.

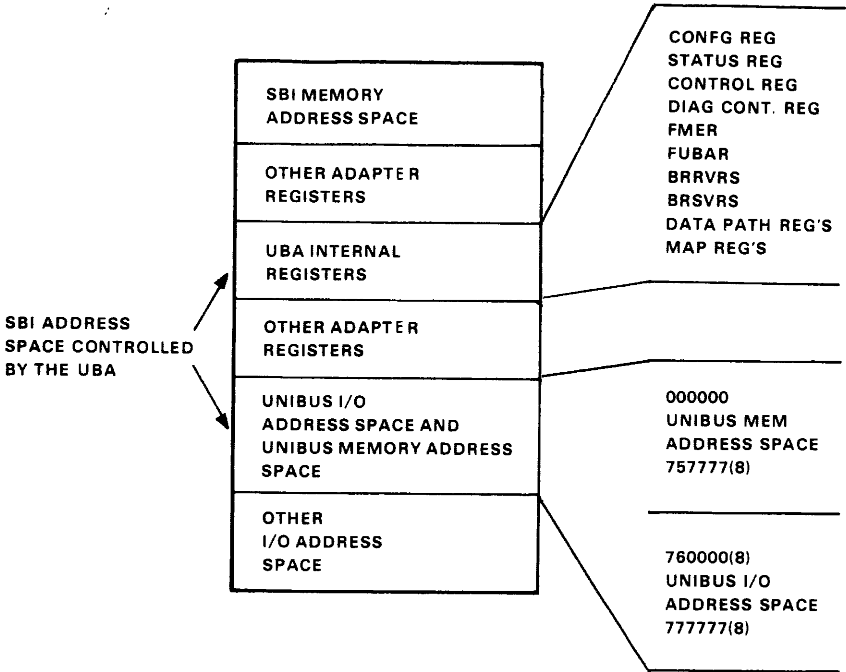



Figure 9-3 SBI I/O Address Space

## 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 $760000_8$—$777777_8$. In VAX-11/780 physical byte address terms, the device registers occupy address space $201XE000_{16}$—$201XFFFF_{16}$. The hexadecimal digit 3,7,B or $F_{16}$ 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 7-7, SBI Physical Address Space, Chapter 7). Table 9-2 illustrates the UNIBUS device register address structure.

**Table 9-2 UNIBUS Device Address Space**

| UNIBUS I/O ADDRESS SPACE | UNIBUS ADDRESS (OCTAL) | PHYSICAL BYTE LOCATIONS (HEX) |
|---|---|---|
| UNIBUS 0 Address Space | 760000-777777 | 2013E000-2013FFFF |
| UNIBUS 1 Address Space | 760000-777777 | 2017E000-2017FFFF |
| UNIBUS 2 Address Space | 760000-777777 | 201BE000-201BFFFF |
| UNIBUS 3 Address Space | 760000-777777 | 201FE000-201FFFFF |

Table 9-3 illustrates the translation of SBI to UNIBUS transfer operations involved in accessing the UNIBUS address space.

**Table 9-3 CPU-Initiated Transfer**

| SBI FUNCTION | TRANSFER DIRECTION | UNIBUS FUNCTION |
|---|---|---|
| Read-masked (word or byte) | device to UBA | DATI |
| Write-masked (word or byte) | UBA to device | DATO or DATOB |
| Interlock Read-masked then Interlocked Write-masked | device to UBA then UBA to device | DATIP then DATO or DATOB |

During such transfers, the UNIBUS Adapter becomes the highest priority UNIBUS Non-Processor request (NPR) device.

### Address And Function Translation

Figure 9-4 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 below.

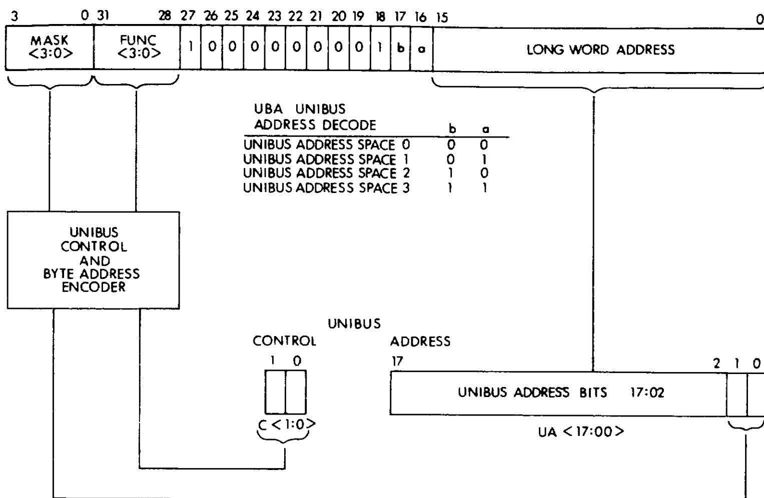



Figure 9-4 SBI To UNIBUS Control Address Translation

Table 9-4 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 9-4.

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 <17:02>. 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 9-5 shows the relationship of the UNIBUS space controlled by UBA #0 to the SBI address space.

**Table 9-4 SBI Function-Mask Translation To UNIBUS Control-Address**

| SBI | | Unibus | |
|---|---|---|---|
| Function <3:0> | Mask 3 2 1 0 | Control C<1:0> | Address UA<1:0> |
| Read Mask | 0 0 0 1 | DATI | 0 0 |
| | 0 0 1 1 | DATI | 0 0 |
| | 0 0 1 0 | DATI | 0 0 |
| | 0 1 0 0 | DATI | 1 0 |
| | 1 1 0 0 | DATI | 1 0 |
| | 1 0 0 0 | DATI | 1 0 |
| Write Mask | 0 0 0 1 | DATOB | 0 0 |
| | 0 0 1 0 | DATOB | 0 1 |
| | 0 1 0 0 | DATOB | 1 0 |
| | 1 0 0 0 | DATOB | 1 1 |
| | 0 0 1 1 | DATO | 0 0 |
| | 1 1 0 0 | DATO | 1 0 |
| Interlock Read Mask | 0 0 0 1 | DATIP | 0 0 |
| (Sets Interlock | 0 0 1 0 | DATIP | 0 0 |
| Flip Flop for | 0 1 0 0 | DATIP | 1 0 |
| DATIP-DATO | 1 0 0 0 | DATIP | 1 0 |
| Sequence) | 0 0 1 1 | DATIP | 0 0 |
| | 1 1 0 0 | DATIP | 1 0 |
| Interlock Write Mask | 0 0 0 1 | DATOB | 0 0 |
| | 0 0 1 0 | DATOB | 0 1 |
| | 0 1 0 0 | DATOB | 1 0 |
| | 1 0 0 0 | DATOB | 1 1 |
| | 0 0 1 1 | DATO | 0 0 |
| | 1 1 0 0 | DATO | 1 0 |

### 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 <17:02> 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

# Table 9-5 UNIBUS And SBI Address Space

| System Address Space (not to scale) |
|---|
| Memory Address Space |
| Other Adaptor Registers |
| Unibus Adaptor Registers |
| Other Adaptor Registers |
| Unibus I/O Address Space |
| Other I/O Address Space |

| 30 bit Physical Byte Address (Hex) | | | | 18 bit Unibus Address Space (Hex) | | | | 18 bit Unibus Address Space (Octal) | | | | Notes |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 20100002 | 20100000 | | | 00002 | 00000 | | 000002 | | 000000 | | Unibus Memory Address Space Reserved For Expansion (496 pages) 1 page 512 bytes |
| | 20100006 | 20100004 | | | 00006 | 00004 | | 000006 | | 000004 | | |
| | 2010000A | 20100008 | | | 0000A | 00008 | | 000012 | | 000010 | | |
| | 2010000E | 2010000C | | | 0000E | 0000C | | 000016 | | 000014 | | |
| | 20100012 | 20100010 | | | 00012 | 00010 | | 000022 | | 000020 | | |
| | . | . | | | . | . | | . | | . | | |
| | . | . | | | . | . | | . | | . | | |
| | . | . | | | . | . | | . | | . | | |
| | 2013DFF6 | 2013DFF4 | | | 3DFF6 | 3DFF4 | | 757766 | | 757764 | | |
| | 2013DFFA | 2013DFF8 | | | 3DFFA | 3DFF8 | | 757772 | | 757770 | | |
| | 2013DFFE | 2013DFFC | | | 3DFFE | 3DFFC | | 757776 | | 757774 | | |
| | 2013E002 | 2013E000 | | | 3E002 | 3E000 | | 760002 | | 760000 | | Unibus I/O Address Space |
| | 2013E006 | 2013E004 | | | 3E006 | 3E004 | | 760006 | | 760004 | | |
| | 2013E00A | 2013E008 | | | 3E00A | 3E008 | | 760012 | | 760010 | | |
| | . | . | | | . | . | | . | | . | | |
| | . | . | | | . | . | | . | | . | | |
| | . | . | | | . | . | | . | | . | | |
| 2013E013 | 2013E012 | 2013E011 | 2013E010 | 3E013 | 3E012 | 3E011 | 3E010 | 760023 | 760022 | 760021 | 760020 | example for DATOB |
| | . | . | | | . | . | | . | | . | | Upper 4 K (10) 16 bit words |
| | . | . | | | . | . | | . | | . | | |
| | . | . | | | . | . | | . | | . | | |
| | 2013FFF6 | 2013FFF4 | | | 3FFF6 | 3FFF4 | | 777766 | | 777764 | | |
| | 2013FFFA | 2013FFF8 | | | 3FFFA | 3FFF8 | | 777772 | | 777770 | | |
| | 2013FFFE | 2013FFFC | | | 3FFFE | 3FFFC | | 777776 | | 777774 | | |

Note: These addresses refer to UBA 0.

least 13 μsec after the command was issued and acknowledged by 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 μsec for SSYN timeout, 50 μsec 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<17:09>) to the 21 SBI page address bits (SA<27:07>).

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

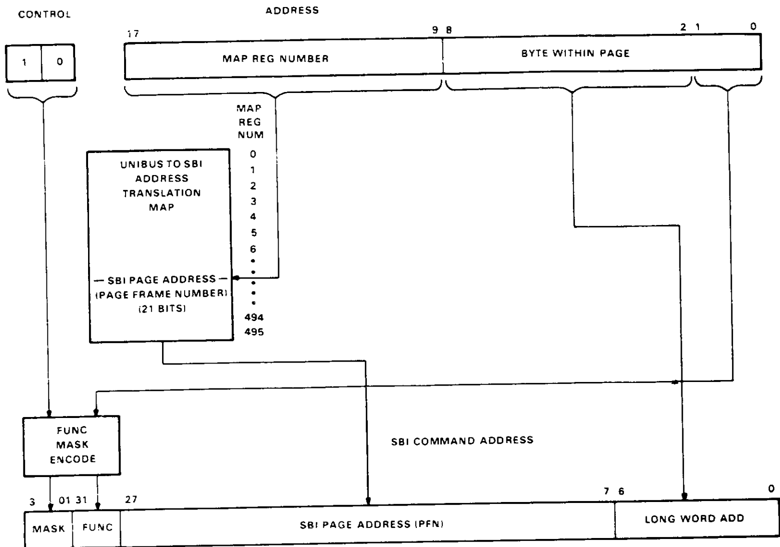



Figure 9-5 UNIBUS To SBI Address Translation

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 9-6 illustrates the translation from the UNIBUS control and byte address fields to the SBI function and mask fields.

**Table 9-6 UNIBUS Field To SBI Field Translation**

| UNIBUS CONTROL C<1:0> | UNIBUS BYTE ADDRESS A<1:0> | SBI FUNCTION FUNC<3:0> | SBI MASK M<3:0> 3210 |
|---|---|---|---|
| DATI | 0 0 | READ MASK | 0 0 1 1 |
| | 1 0 | | 1 1 0 0 |
| DATO | 0 0 | WRITE MASK | 0 0 1 1 |
| | 1 0 | | 1 1 0 0 |
| DATOB | 0 0 | WRITE MASK | 0 0 0 1 |
| | 0 1 | | 0 0 1 0 |
| | 1 0 | | 0 1 0 0 |
| | 1 1 | | 1 0 0 0 |
| DATIP | 0 0 | INTERLOCK READ MASK | 0 0 1 1 |
| | 1 0 | | 1 1 0 0 |
| followed by DATO | 0 0 | INTERLOCK WRITE MASK | 0 0 1 1 |
| | 1 0 | | 1 1 0 0 |
| OR DATOB | 0 0 | INTERLOCK WRITE MASK | 0 0 0 1 |
| | 0 1 | | 0 0 1 0 |
| | 1 0 | | 0 1 0 0 |
| | 1 1 | | 1 0 0 0 |

**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.

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.
- The maximum throughput via the DDP is approximately 400K words per second.

- 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 9-7 illustrates the translation of UNIBUS to SBI data transfer operations.

**Table 9-7 UNIBUS-Initiated Transfer Via The Direct Data Path**

| UNIBUS FUNCTION | TRANSFER DIRECTION | SBI FUNCTION |
|---|---|---|
| DATI | UBA to device | Read-masked (16 bits) |
| DATO or DATOB (byte) | device to UBA | Write-masked (8 or 16 bits) |
| DATIP then DATO or DATOB | UBA to device then device to UBA | Interlock Read-masked then Interlock Write-masked |

**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 RK06, DMC-11, etc. 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.
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 9-6 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.

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.

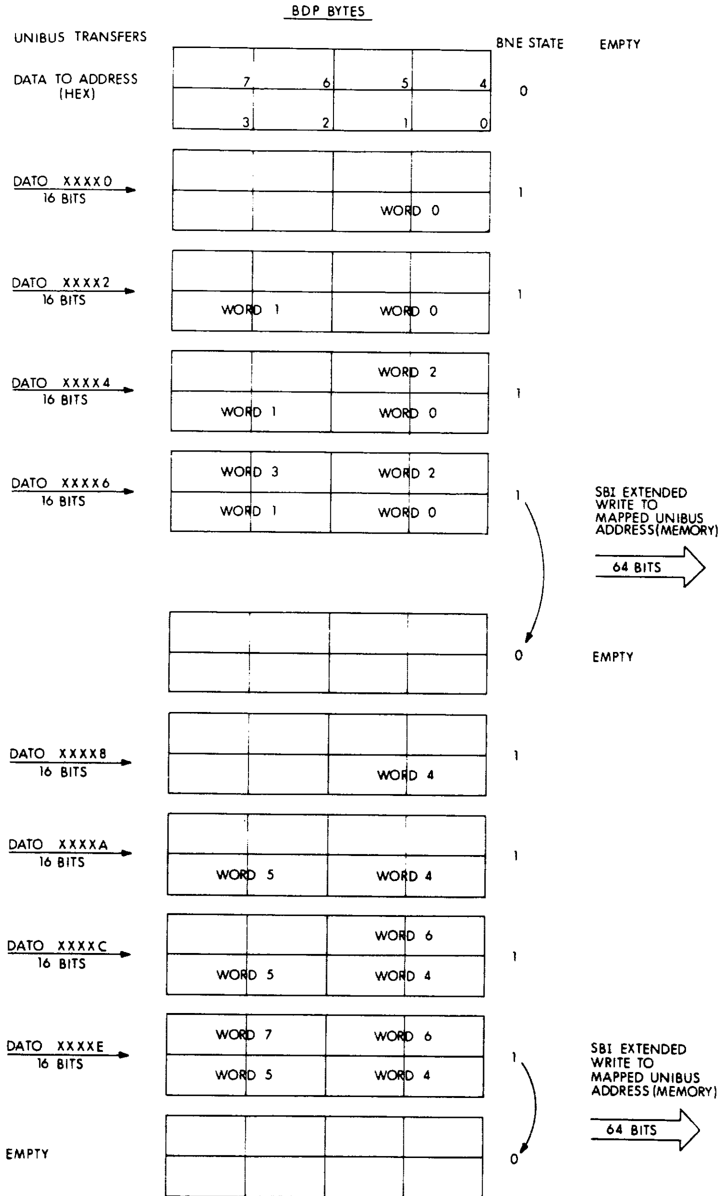



Figure 9-6 UNIBUS Transfer To Memory

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 9-7 illustrates the Buffered Data Path transfer from memory to the UNIBUS.

Software Note:
Since the prefetch allows the possibility of the UBA crossing a page boundary into nonexistent memory, resulting in a 100 μsec 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 μsec 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 the map registers involved during the devices transfer.

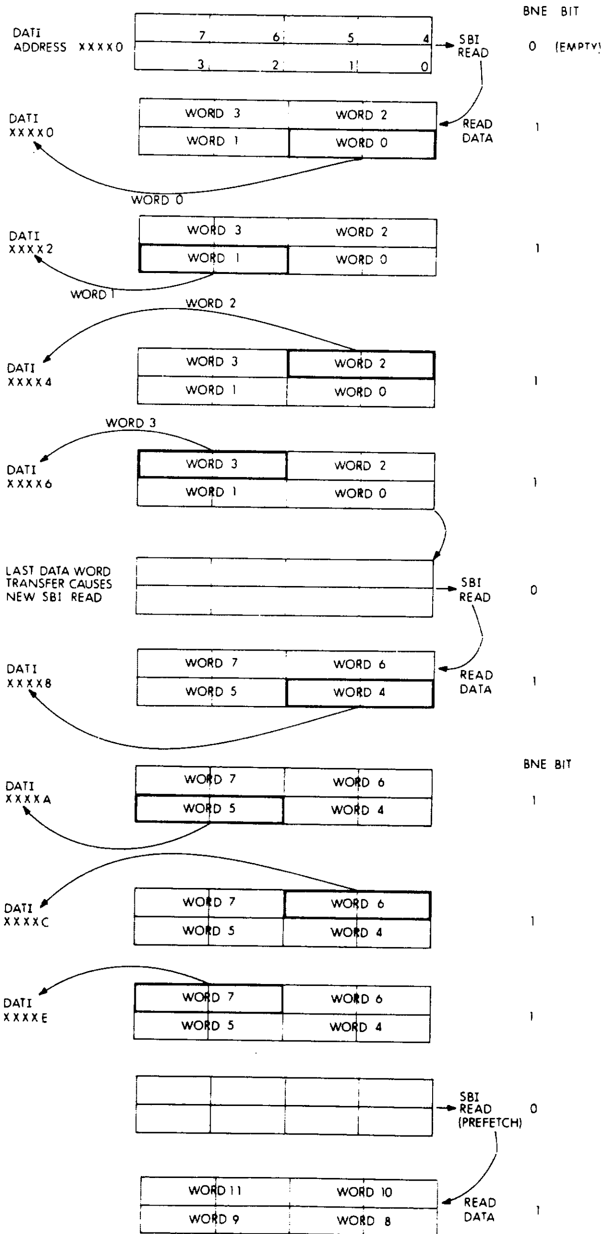



Figure 9-7 UNIBUS Transfers From Memory

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 apppear 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 9-8 shows the relative position of data being transferred between a UNIBUS device and SBI memory. Figure 9-8 top shows the relative positions without Byte Offset and Figure 9-8 bottom shows the position with Byte Offset.

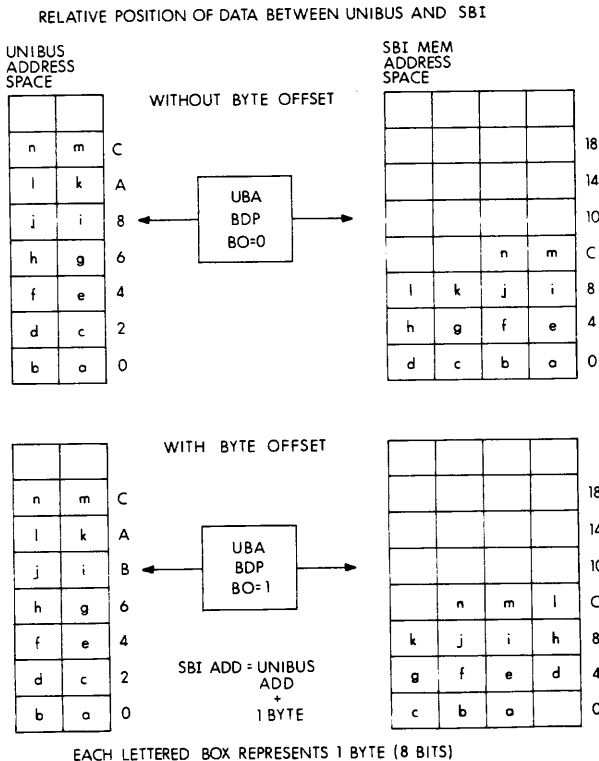



Figure 9-8 Relative Position Of Data Between UNIBUS And SBI

**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 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 9-9.

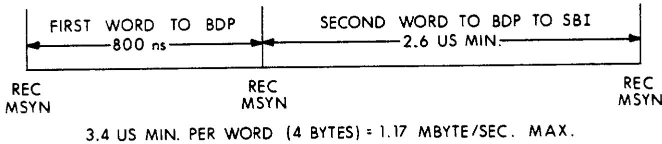



Figure 9-9 Random Access Mode Throughput

The operation of the UBA for the longword-aligned 32-bit access mode is determined by the function (DATI, DATO/DATOB) and address (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 indicate 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:

| PRESENT BNE STATE | FUNCTION | A1,A0 | | UBA OPERATIONS | NEXT BNE STATE |
|---|---|---|---|---|---|
| 0 | DATI | 0 | X | SBI READ,RETURN LOW WORD, STORE DATA | 1 |
| 0 | DATI | 1 | X | SBI READ, RETURN HIGH WORD | 0 |
| 1 | DATI | 0 | X | RETURN LOW WORD | 1 |
| 1 | DATI | 1 | X | RETURN HIGH WORD | 0 |
| 0 | DATO | 0 | X | STORE LOW WORD | 1 |
| 0 | DATO | 1 | X | STORE HIGH WORD,SBI WRITE | 0 |
| 1 | DATO | 0 | X | STORE LOW WORD | 1 |
| 1 | DATO | 1 | X | STORE HIGH WORD,SBI WRITE | 0 |
| 0 | DATOB | 0 | 0 | STORE BYTE 0 | 1 |
| 0 | DATOB | 0 | 1 | STORE BYTE 1 | 1 |
| 0 | DATOB | 1 | 0 | STORE BYTE 2 | 1 |
| 0 | DATOB | 1 | 1 | STORE BYTE 3,SBI WRITE | 0 |
| 1 | DATOB | 0 | 0 | STORE BYTE 0 | 1 |
| 1 | DATOB | 0 | 1 | STORE BYTE 1 | 1 |
| 1 | DATOB | 1 | 0 | STORE BYTE 2 | 1 |
| 1 | DATOB | 1 | 1 | STORE BYTE 3,SBI WRITE | 0 |
| 0 | DATIP | X | X | UBA DOES NOT RESPOND (NXM TO UNIBUS DEVICE) NO CHANGE | 0 |
| 1 | DATIP | X | X | UBA DOES NOT RESPOND (NXM TO UNIBUS DEVICE) NO CHANGE | 1 |

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>** | | Must be zero. |
| **BIT<26>** | 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<20:00>** | 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. |
| | 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 μsec. 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.

| Bit <31> | Bits <15:00> | |
|---|---|---|
| 0 | 0 | No service required. |
| 0 | V | UNIBUS service as indicated by vector V received from the UNIBUS device (UNIBUS device Interrupt Service Routine). |
| 1 | 0 | UNIBUS Adapter service required. Read configuration register and status register to determine the service required. |
| 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.

**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 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 9-8 illustrates the physical base address and SBI base address for a NEXUS assigned to any one of the SBI transfer request numbers.

**Table 9-8 Transfer Number Address Assignments**

| SBI TRANSFER REQUEST NUMBER | ADDRESS BASE PHYSICAL(HEX) |
|---|---|
| 0 | 20000000 |
| 1 | 20002000 |
| 2 | 20004000 |
| 3 | 20006000 |
| 4 | 20008000 |
| 5 | 2000A000 |
| 6 | 2000C000 |
| 7 | 2000E000 |
| 8 | 20010000 |
| 9 | 20012000 |
| 10 | 20014000 |
| 11 | 20016000 |
| 12 | 20018000 |
| 13 | 2001A000 |
| 14 | 2001C000 |
| 15 | 2001E000 |

Table 9-9 lists each of the UNIBUS Adapter registers and its associated physical address offset.

**Table 9-9 UNIBUS Adapter Register Address Offset**

| UNIBUS ADAPTER REGISTER | BYTE OFFSET (PHYSICAL HEX) |
|---|---|
| Configuration Register | 000 |
| UNIBUS Adapter Control Register | 004 |

| Register | Offset |
|---|---|
| UNIBUS Adapter Status Register | 008 |
| Diagnostic Control Register | 00C |
| Failed Map Entry Register | 010 |
| Failed UNIBUS Address Register | 014 |
| Failed Map Entry Register | 018 |
| Failed UNIBUS Address Register | 01C |
| Buffer Selection Verification Register 0 | 020 |
| Buffer Selection Verification Register 1 | 024 |
| Buffer Selection Verification Register 2 | 028 |
| Buffer Selection Verification Register 3 | 02C |
| Buffer Receive Vector Register 4 | 030 |
| Buffer Receive Vector Register 5 | 034 |
| Buffer Receive Vector Register 6 | 038 |
| Buffer Receive Vector Register 7 | 03C |
| Data Path Register 0 | 040 |
| Data Path Register 1 | 044 |
| . | . |
| . | . |
| . | . |
| Data Path Register 14 | 078 |
| Data Path Register 15 | 07C |
| Reserved | 080 |
| . | . |
| . | . |
| . | . |
| Reserved | 7EC |
| Map Register 0 | 800 |
| Map Register 1 | 804 |
| . | . |
| . | . |
| . | . |
| Map Register 494 | EB8 |
| Map Register 495 | EBC |
| Reserved | EC0 |
| . | . |
| . | . |
| . | . |
| Reserved | EFC |

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 9-9, 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 9-8. Therefore, the byte offset for the configuration register in Table 9-9 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 9-10 illustrates the configuration register.

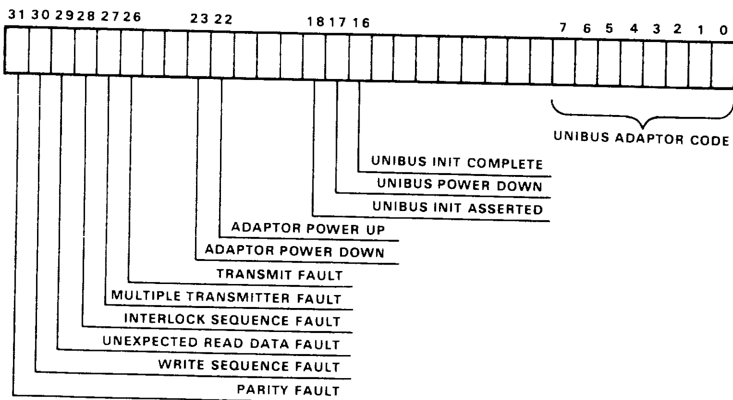



Figure 9-10 Configuration Register Bit Configuration

The contents of the Configuration Register are as follows:

**Bits <31:27> SBI faults**

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> Parity Fault (PAR FLT)**
PAR FLT is set when the UNIBUS adapter detects an SBI parity error.

**Bit <30> Write Sequence Fault (WSQ FLT)**
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> Unexpected Read Data Fault (URD FLT)**
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> Interlock Sequence Fault (ISQ FLT)**
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> Multiple Transmitter Fault (MXT FLT)**
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> Transmit Fault (XMT FLT)**
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.

**Bits <25:24> Reserved and Zero**
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> Adapter Power Down (AD PDN)**
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> Adapter Power Up (AD PUP)**
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 the setting of the Adapter Power Down bit.

**Bits <21:19> Reserved and Zero**

**Bit <18> UNIBUS INIT Asserted (UB INIT)**

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> UNIBUS Power Down (UB PDN)**

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> UNIBUS Initialization Complete (UBIC)**

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.

**Bits <15:08> Reserved**

**Bits <7:0> Adapter Code**

These bits define the code assigned to the UNIBUS adapter. Table 9-10 shows the bit assignment.

**Table 9-10 Adapter Code Bit Assignment**

| BIT NUMBER | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| UNIBUS ADDRESS SPACE | 0 | 0 | 1 | 0 | 1 | 0 | | |
| 0 | | | | | | | 0 | 0 |
| 1 | | | | | | | 0 | 1 |
| 2 | | | | | | | 1 | 0 |
| 3 | | | | | | | 1 | 1 |

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 9-11.

**Table 9-11 Selectable UNIBUS Starting Addresses**

| UNIBUS ADDRESS SPACE | STARTING ADDRESS OF THE UNIBUS ADDRESS SPACE, BASE 16 (PHYSICAL BYTE ADDRESS) |
|---|---|
| 0 | 20100000(16) |
| 1 | 20140000(16) |
| 2 | 20180000(16) |
| 3 | 201C0000(16) |

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 9-11 shows the Control Register bit configuration.

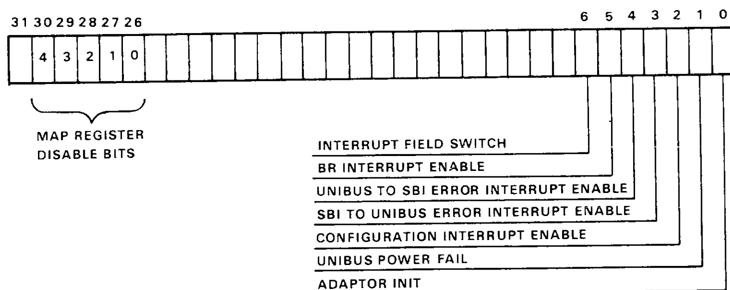



Figure 9-11 Control Register Bit Configuration

The contents of the control register are as follows:

**Bit <31> Reserved and zero.**

**Bits <30:26> Map Register Disable <4:0> (MRD)**
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 4K word units of memory attached to the UNIBUS, as shown in Table 9-12.

**Table 9-12 Map Register Disable Bit Functions**

| MRD <4:0> | AMOUNT OF UNIBUS MEMORY (WORDS) | MAP REGISTERS DISABLED |
|---|---|---|
| 00000 | 0 K | NONE |
| 00001 | 4 K | 0 TO 15 (10) |
| 00010 | 8 K | 0 TO 31 (10) |
| 00011 | 12 K | 0 TO 47 (10) |
| • | • | • |
| • | • | • |
| • | • | • |
| 11110 | 120K | 0 TO 480 (10) |
| 11111 | 124K | 0 TO 495 (10) |

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 <25:07> Reserved and Zero**

**Bit <6> Interrupt Field Switch (IFS)**

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> Bus Request Interrupt Enable (BRIE)**

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> UNIBUS to SBI Error Field Interrupt Enable (USEFIE)**

The USEFIE bit enables an interrupt request to the VAX-11 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)
CXTO (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> SBI To UNIBUS Error Field Interrupt Enable (SUEFIE)**
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> Configuration Interrupt Enable (CNFIE)**
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> UNIBUS Power Fail (UPF)**
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> Adapter INIT (ADINIT)**
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 $\mu$sec to complete, while the UNIBUS power fail sequence requires 25 msec.
Only the configuration register and the diagnostic control register can be read during the adapter initialization sequence. And only the con-

figuration 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 <10:00> 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 9-12 shows the Status Register bit configuration.

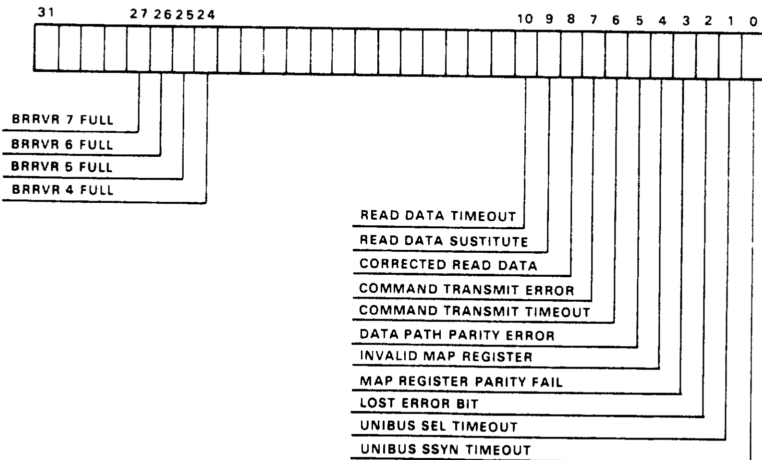



Figure 9-12 Status Register Bit Configuration

The contents of the status register are:

**Bits <31:28> Reserved and Zero**

**Bits <27:24> BR Receive Vector Register Full**

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.

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**

**Bits <23:11> Reserved and Zero**

The remaining bits identify specific data transfer errors. They are read and write-one-to clear bits.

**Bit <10> Read Data Time Out (RDTO)**

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 μsec, and when the UNIBUS device has not timed out. Note that the normal UNIBUS timeout is 10-20 μsec, and that after 10-20 μsec, the UNIBUS device will set its nonexistent memory bit. Thus, the Read Data Time Out bit will be set on the UNIBUS adapter status register only if the UNIBUS device timeout function is inoperative, or takes more than 100 μsec.

**Bit <9> Read Data Substitute (RDS)**

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> Corrected Read Data (CRD)**

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> Command Transmit Error (CXTER)**
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> Command Transmit Timeout (CXTO)**
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 $\mu$sec, which will result in the UNIBUS device setting its nonexistent memory bit and will also abort the UBA to SBI transfer. The CXTO 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 $\mu$sec timeout period if busy or no response confirmation is received, thereby setting the CXTO 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> Data Path Parity Error (DPPE)**
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> Invalid Map Register (IVMR)**
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> Map Register Parity Failure (MRPF)**
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> Lost Error Bit (LEB)**

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> UNIBUS Select Time Out (UBSTO)**

The UNIBUS adapter sets this bit if it cannot gain access to the UNIBUS within 50 μsec 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> UNIBUS Slave Sync Time Out (UBSSYNTO)**

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 μsec. UBSSYNTO indicates a transfer failure resulting when a nonexistent memory or device on the UNIBUS is addressed.

The two bits just discussed, UBSTO and UBSSYNTO, 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 UBSSYNTO 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 UBSSYNTO 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 9-13 shows the bit configuration of the DCR.

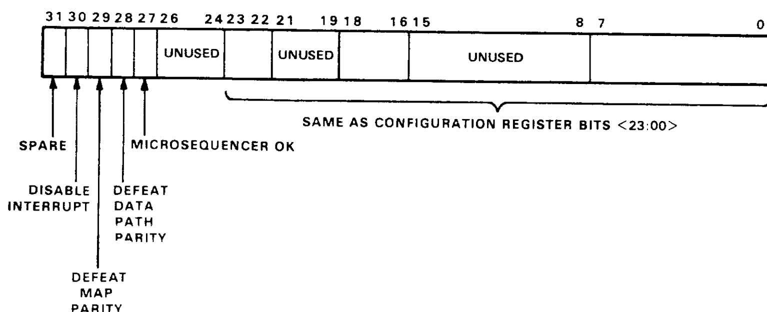



Figure 9-13 Diagnostic Control Register Bit Configuration

**Bit <31> Spare**

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> Disable Interrupt (DINTR)**

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 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> Defeat Map Parity (DMP)**

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 con-

tains 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> Defeat Data Path Parity (DDPP)**
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> Microsequencer OK (MIC OK)**
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).

**Bits <26:24> Reserved and Zero**

**Bits <23:00> Same as bits <23:00> 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 9-14 shows the bit configuration for the Failed Map Entry Register.

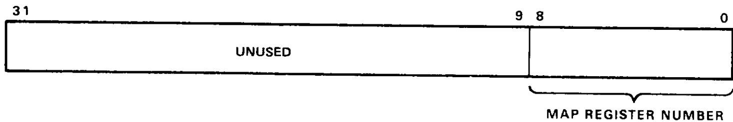



Figure 9-14 Failed Map Entry Register Bit Configuration

**Bit <31:09> Reserved and Zero**

**Bits <08:00> Map Register Number (MRN)**
These bits contain the number of the map register which was in use at the time of a failure. Bits <08:00> correspond to bits <17:09> 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 (UBSSYNTO). 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 9-15 shows the bit configuration of the FUBAR. The contents of the FUBAR are listed below.

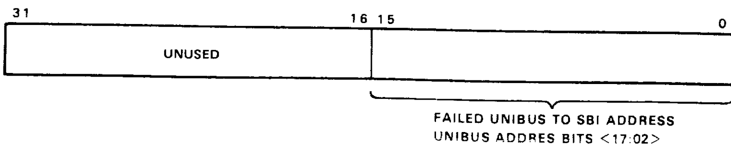



Figure 9-15 Failed UNIBUS Address Register Bit Configuration

**Bits <31:16> Reserved and Zero**

**Bits <15:00> Failed UNIBUS to SBI Address**
These bits correspond to UNIBUS Address bits <17:02>.

**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 9-16.

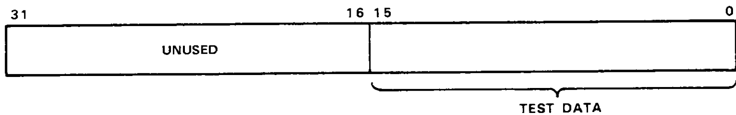



Figure 9-16 Buffer Selection Verification Register Bit Configuration

The contents of the BRSVRs are listed below.

**Bits <31:16> Always Zero**

**Bits <15:00> 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 a 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 9-17 shows the BR Receive Vector Register bit configuration.

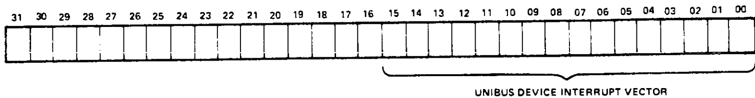



Figure 9-17 BR Receive Vector Register Bit Configuration

The contents of the four BRRVRs are as follows:

**Bit <31> Adapter Interrupt Request Indicator.**
0=No UBA interrupt pending.
1=UBA interrupt pending.

**Bits <30:16> Reserved and zero**

**Bits <15:00> Device Interrupt Vector Field**
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.

Figure 9-18 shows the Data Path Register bit configuration.

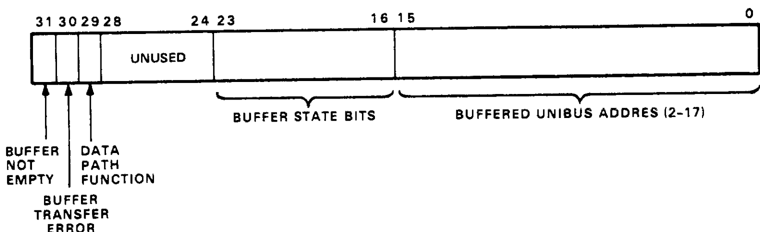



Figure 9-18 Data Path Register Bit Configuration

The DPR bit functions are as follows.
- Buffer Not Empty and the Purge Operation

**Bit <31> Buffer Not Empty (BNE)**
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, thereby 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 $\mu$sec. 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> Buffer Transfer Error (BTE)**

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.

**Bits <29:00> Read-Only Bits**

**Bit <29> Data Path Function (DPF)**

This bit indicates the function of the DMA transfer using this data path.

0=DMA Read
1=DMA Write

**Bits <28:24> Unused**

**Bits <23:16> Buffer State (BS)**
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.

**Bits <15:00> Buffered UNIBUS Address (BUBA)**
This portion of each DPR contains the upper 16 bits of the UNIBUS address, UA <17:02>, 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 <17:02>. 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).

| REG | Offset |
|---|---|
| MR0 | 800 |
| MR1 | 804 |
| MR2 | 808 |
| MR3 | 80C |
| . | . |
| . | . |
| . | . |
| MR494 | FB8 |
| MR495 | FBC |

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).
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 9-19 illustrates the map register bit configuration.

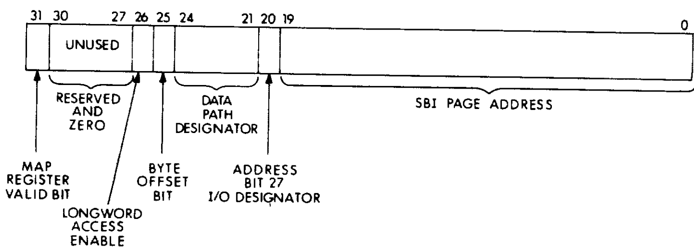



Figure 9-19 Map Register Bit Configuration

**NOTE**

In the interest of 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> Map Register Valid (MRV)**
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.

**Bits <30:27> Reserved Read/Write Bits**

**Bit <26> Longword Access Enable (LWAE)**
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> Byte Offset Bit (BO)**
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.

**Bits <24:21>=Data Path Designator Bits (DPDB)**

| | |
|---|---|
| 0000= | Direct Data Path (DDP) |
| 0001= | Buffered Data Path 1 |
| . | . |
| . | . |
| 1111= | Buffered Data Path 15 |

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.

**Bits <20:00> SBI Page Address (SPA 27:07, also known as Page Frame Number, PFN)**

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 contents of SPA<27:07> 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.

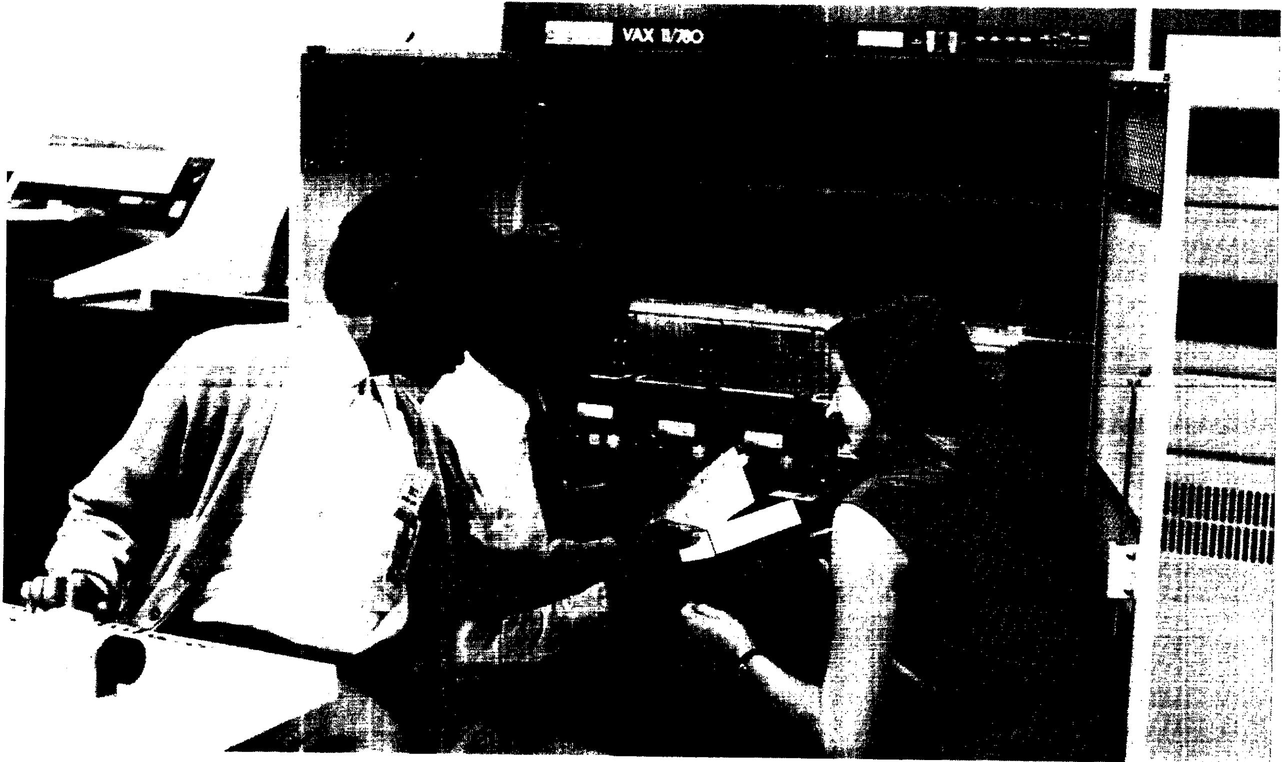
**EXAMPLE**
Presented is a program to read data from the RK06 disk subsystem into memory. The program full documented and is designed to demonstrate the loading of the UBA map registers, the use of a buffered data path (including the purge), access to UNIBUS device registers, and initialization of the UNIBUS. In order to run the progra, 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.

```
;       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 (DP5).
;               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 = ^X20006000             ; UBA AT TR = 3
        UBA_CNFR = ^X0                    ; OFFSET TO UBA CONFIGURATION REGISTER
          UBA_UBIC = ^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_DPO  = ^X40                   ; OFFSET TO UBA DATA PATH REGISTER 0
         UBA_DP_BNE = ^X80000000          ;   BUFFER NOT EMPTY BIT
                                          ;    USED TO PURGE BUFFER DATA
                                          ;    PATH AT END OF XFER.
         UBA_DP_BTE = ^X40000000          ;   BUFFER TRANSFER ERROR BIT
        UBA_MRO  = ^X800                  ; OFFSET TO UBA MAP REGISTER 0
         MAP_VALID = ^X80000000           ;   VALID BIT IN MAP REGISTER
         BYTE_OFST = ^X2000000            ;   BYTE OFFSET BIT IN MAP REGISTER

;  RK611 RELATED SYMBOLS:
        UNIBUS_BASE = ^X20100000          ; BASE ADDRESS FOR UNIBUS ADDRESS 0
        DK_BASE_ADD = ^O777440            ; UNIBUS BASE ADDRESS OF DK611
        DK_CS1 = ^O0                      ; RK611 CONTROL STATUS REGISTER 1
        DK_CS2 = ^O10                     ; RK611 CONTROL STATUS REGISTER 2
        DK_DS  = ^O12                     ; RK611 DRIVE STATUS REGISTER
        DK_DC  = ^O20                     ; RK611 DESIRED CYLINDER REGISTER
        DK_DA  = ^O6                      ; RK611 DISK ADDRESS REGISTER
        DK_WC  = ^O2                      ; RK611 WORD COUNT REGISTER
        DK_BA  = ^O4                      ; RK611 BUS ADDRESS REGISTER
        SCLR   = ^O40                     ; RK611 SUBSYSTEM CLEAR
        PACACK = ^O3                      ; RK611 PACK ACKNOWLEDGE AND GO BIT SET
        READ   = ^O21                     ; RK611 DISK READ AND GO BIT SET.
        ;  MISC SYMBOLS:
        BELL   = ^O7                      ; ASCII BELL
        TXDB   = 35                       ; CONSOLE TRAANSMIT DATA BUFFER


;  THIS SECTION LOADS THE UBA_BASE ADDRESS INTO R0 AND THEN INITIALIZES
;  THE UNIBUS.

BEGIN:  MOVL    #UBA_BASE, R0             ; LOAD UBA'S ADDRESS INTO R0
INIT:   MOVL    #UBA_ADINIT,UBA_CR(R0)    ; INIT UBA AND UNIBUS

;  THE UBA AND UNIBUS ARE BEING INITIALIZED.  THE PROGRAM CANNOT MAKE
;  ANY ACCESSES 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
                                          ;   TO FIND ITS OFFSET AND PUT INTO R1.

        ADDL    #UBA_MRO, R1              ; ADD THE BASE ADDRESS OF THE MAP REGISTERS
                                          ;  TO THE OFFSET AND PUT IN R1.
        ADDL    #UBA_BASE, R1             ; ADD IN THE UBA BASE ADDRESS AND PUT IN R1.
```

```
;  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.
        BISL    #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    CONT$                       ; IF NOT THEN CONTINUE.
        BISL    #BYTE_OFST, R2              ;   YES -- SET BYTE OFFSET BIT

CONT$:  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   #^XFFFFFE00, 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, #8, #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_DA REGISTER
```

```
;  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, #8, #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_DA REGISTER

;  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$:     BITL    #UBA_UBIC, UBA_CNFR(R0) ; IS UNIBUS INITIALIZATION COMPLETE?
        BEQL    1$                      ; NO - KEEP TESTING
                                        ; 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
;  CONTIUOUS.  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.

        MOVL    BCOUNT, R6              ; LOAD BYTE COUNT INTO R6
2$:     MOVL    R2, (R1)+               ; LOAD MAP REGISTER WITH CONTENTS
                                        ; OF R2.
        INCL    R2                      ; INCREMENT PAGE FRAME NUMBER.
                                        ;  (ASSUMES CONTIGUOUS PAGES OF
                                        ;   PHYSICAL MEMORY).
        SUBL    #^X200, R6              ; THERE ARE 200(HEX) BYTES PER PAGE.
                                        ; ARE ALL MAPS SET UP?
        BGTR    2$                      ;   NO SET UP NEXT MAP REGISTER.
        MOVL    R2, (R1)+               ; SET UP ONE MORE SINCE TRANSFER
                                        ;   TRANSFER MAY NOT BE PAGE ALIGNED.
        CLRL    (R1)                    ; INVALIDATE NEXT MAP REGISTER TO
                                        ;   STOP THE UBA SHOULD THE TRANSFER
                                        ;   GO BEYOND ITS EXPECTED LIMIT.

;  THIS SECTION WILL PERFORM THE DISK TRANSFER SEQUENCE.
;    A SUBSYSTEM CLEAR WILL BE ISSUSED TO THE RK611, THE PACK AKNOWLEDGE
;    FUNCTION WILL BE ISSUED, AND THE DISK TRANSFER WILL BE INITIATED.
; NOTE THAT ALL UNIBUS ACCESSES MUST BE OF WORD OR BYTE FORMAT..

;  WE ARE NOW FINISHED WITH R1 AND R2.
;  FIND BASE ADDRESS OF RK611 AND LOAD INTO R1.

        ADDL3   #DK_BASE_ADD, #UNIBUS_BASE, R1  ;BASE ADDRESS OF RK611 TO R1
        MOVW    #SCLR, DK_CS2(R1)       ; ISSUE A RK611 SUBSYSTEM CLEAR
        MOVW    DRIVE, DK_CS2(R1)       ; SELECT DRIVE NUMBER
        MOVW    R5, DK_DA(R1)           ; LOAD DISK ADDRESS SECTOR AND TRACK
                                        ; STORED IN R5 FROM ABOVE.
        MOVW    #PACACK, DK_CS1(R1)     ; ISSUE PACK ACKNOWLEDGE FUNCTION
3$:     TSTB    DK_CS1(R1)              ; WAIT FOR READY
        BGEQ    3$                      ;  NOT READY KEEP WAITING

        MOVW    CYLINDER, DK_DS(R1)     ; LOAD CYLINDER ADDRESS
        MNEGW   R4, DK_WC(R1)           ; LOAD 2'S COMPLIMENT OF WORD COUNT
        MOVW    R3, DK_BA(R1)           ; LOAD LOW ORDER 16 BITS OF UNIBUS
                                        ;   START ADDRESS INTO DK_BA REG
```

```
        ASHL    #-16, R3, R3              ; SHIFT UNIBUS ADDRESS RIGHT 16 BITS
        MOVL    DK_FUNC, R4               ; LOAD FUNCTION INTO R4 AND
        INSV    R3, #8, #2, R4            ; INSERT ADDRESS BITS 17 AND 16 FROM
                                          ;   R3 BITS 1:0 INTO BITS 9:8 OF
                                          ;   R4.  EXTENDED UNIBUS ADDRESS
                                          ;   BITS.
        MOVW    R4, DK_CS1(R1)            ; ISSUE FUNCTION AND GO TO RK611

4$:     TSTB    DK_CS1(R1)                ; WAIT FOR TRANSFER TO COMPLETE
        BGEQ    4$                        ;  NOT COMPLETE - KEEP WAITING.

;  THE RK611 HAS BEEN COMPLETED - THE UBA BUFFERED DATA PATH MUST NOW BE
;   PURGED.  THE PURGE 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
        ADDL    #UBA_DPO, R2              ; ADD IN OFFSET OF DATA PATH REG 0
        MOVL    #UBA_DP_BNE, UBA_BASE(R2); SET BNE BIT OF DATA PATH REGISTER
                                          ; USED BY THE TRANSFER.
        BITL    #UBA_DP_BTE, UBA_BASE(R2)
                                          ; TEST FOR ANY ERRORS THAT MAY HAVE
                                          ; OCCURRED WITHIN THE UBA BUFFER
                                          ; TRANSFER.
        BEQL    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$:     MTPR    #BELL, #TXDB              ; RING BELL ON CONSOLE IF NO ERRORS
        HALT



;  THE TRANSFER PERAMETERS 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.
DP_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
```


VAX 11/780

## CHAPTER 10

# 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 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 10-1 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 10-1 describes individual MASSBUS signal line function.

**Table 10-1 MASSBUS Line Descriptions**

| SIGNAL LINE | DESCRIPTION |
|---|---|
| CONTROL BUS | |
| Control and Status (C00-15) | Transfers 16 parallel control or status bits to or from the drive. |

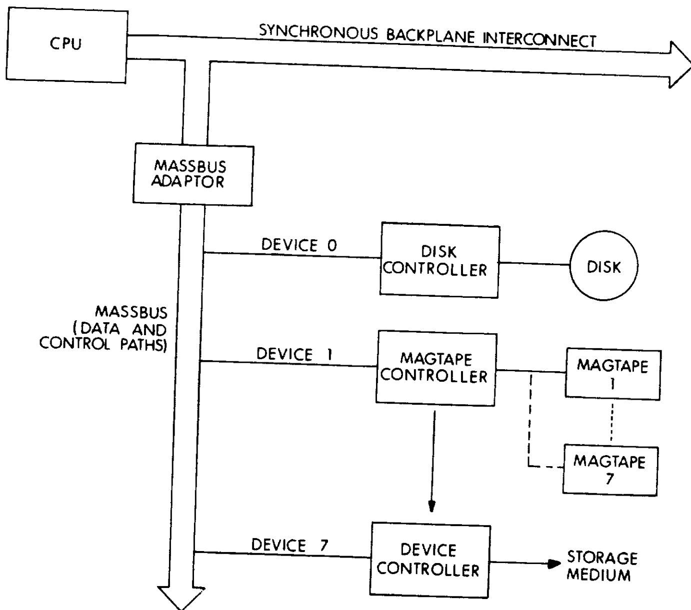



Figure 10-1 MASSBUS Subsystem Configuration

**Table 10-1 (cont.)**

| SIGNAL LINE | DESCRIPTION |
|---|---|
| CONTROL BUS | |
| Control Bus Parity (CPA) | Transfers odd control bus parity to or from the drive. Parity is simultaneously transferred with control bus data. |
| Drive Select (DS0-2) | Transfers a 3-bit binary code from the MBA to select a controller. The drive responds when the (unit) select switch in the controller corresponds to the transmitted binary code. |
| Register Select (RSO-4) | Transfers a 5-bit binary code from the MBA to select a particular drive register. |

**Table 10-1 (cont.)**

| SIGNAL LINE | DESCRIPTION |
|---|---|
| CONTROL BUS | |
| Controller to Drive (CTOD) | Indicates in which direction information is to be transferred on the control bus. For a controller-to-drive transfer, the MBA asserts CTOD; for a drive-to-controller transfer, the MBA negates CTOD. |
| Demand (DEM) | Asserted by the MBA to indicate a transfer is to take place on the control bus. For a controller-to-drive transfer, DEM is asserted by the MBA when data is present. For a drive-to-controller transfer, DEM is asserted by the MBA to request data and is negated when the data has been strobed from the control bus. In both cases, the RS, DS, and CTOD lines are asserted and allowed to settle before assertion of DEM. |
| Transfer (TRA) | 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. |
| Attention (ATTN) | 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. |
| Initialize (INIT) | Asserted by the MBA to initialize all drives on the bus. This signal is transmitted whenever the MBA receives an initialize command. |
| Fail (FAIL) | When asserted, this line indicates a power fail condition has occurred in the MBA or the MBA is in maintenance mode. |

**Table 10-1 (cont.)**

| SIGNAL LINE | DESCRIPTION |
|---|---|
| DATA BUS | |
| Data (D00-15) | These bidirectional lines transfer 16 parallel data bits between the MBA and drives. |
| Data Bus Parity (DPA) | Transfers an odd parity bit to or from the drive. Parity is simultaneously transferred with bits on the data bus. |
| Sync Clock (SCLK) | 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. |
| Write Clock (WCLK) | Asserted by the MBA to indicate when data written to the drive is to be strobed. |
| Run (RUN) | 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. |
| End-of-Block (EBL) | 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. |
| Exception (EXC) | 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. |
| Occupied (OCC) | Indicates acceptance of a valid data transfer command. |

Figure 10-2 illustrates the MASSBUS signal line configuration.

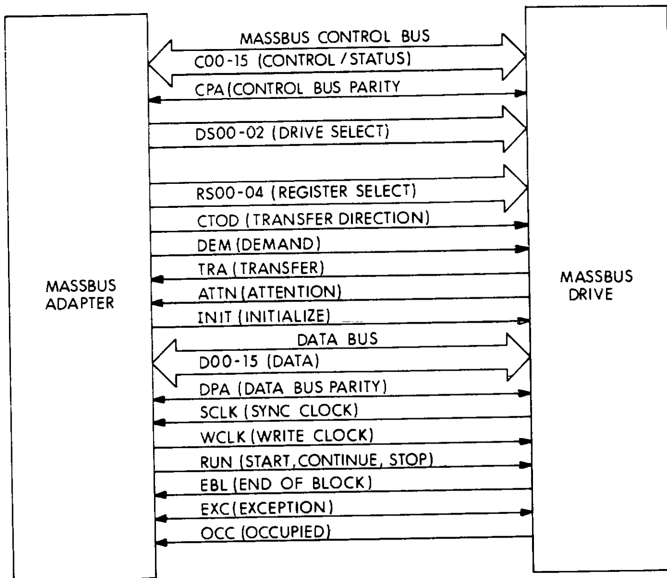



Figure 10-2 MASSBUS Signal Line Configuration

## MASSBUS ADAPTER OPERATION

The MASSBUS adapter consists of an SBI/MBA interface, internal registers, control paths and data paths. Figure 10-3 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.

The MBA is capable of handling a MASSBUS drive with a maximum data transfer rate of 16 bits per μsec via the 16-bit wide MASSBUS data path. The MASSBUS adapter controls data transfers between MASSBUS devices and physical memory. A MASSBUS adapter can

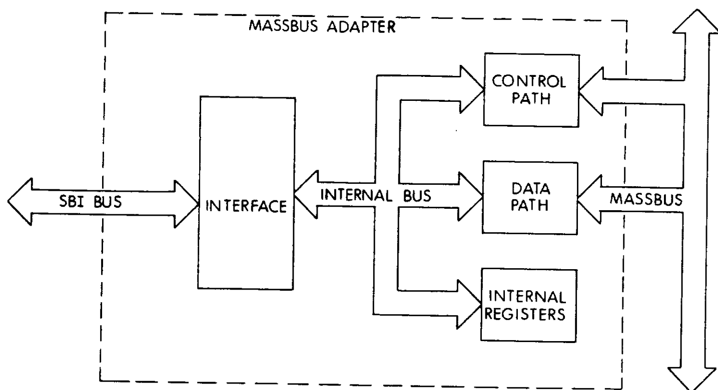



Figure 10-3 MASSBUS Adapter

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 X 32-bit RAM. The internal register is 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.

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 10-4 illustrates the mapping of a virtual address to an SBI address.

**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 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

VIRTUAL ADDRESS REGISTER
31
17 16
9 8
2 1 0
MAP POINTER
LONG WORD
BYTE
INDEX INTO MAP REGISTERS
MAP REGISTERS
31 30
21 20
0
V
RESERVED
PHYS PAGE ADDRESS
DIRECT TRANSFER
DIRECT TRANSFER
SBI ADDRESS
27
7 6
0
PHYSICAL PAGE ADDRESS

Figure 10-4 Virtual To SBI Address Translation

SBI I/O longword address space. The command/address format used to access the MBA registers is illustrated in Figure 10-5.

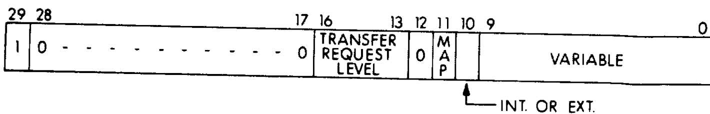
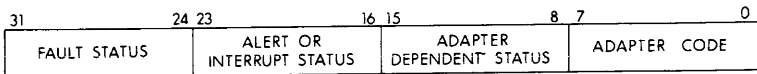



Figure 10-5 MASSBUS Adapter Addressing Format (Physical Byte Address)

| | |
|---|---|
| Bit <29>=1 | I/O Address space |
| Bits <28:17> | All zeros |
| Bits <16:13> | Transfer request number of this MBA |
| Bits <11:10> | |
| 0 0 | MBA internal register |
| | Bits<9:5>=must be zero |
| | Bits<4:0>=register select offset |
| 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> SBI parity error**
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> Write data sequence(WS)**
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> Unexpected read data (URD)**
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> This bit must be zero.**

**Bit <27> Multiple transmitter (MT)**
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 deassertion 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> XMTFLT**
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.

**Bits <25:24> Zeros**
Reserved for future use.

**Bit <23> Adapter power down (PD)**
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> Adapter power up (PU)**
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> Over temperature (OT)**
Zero

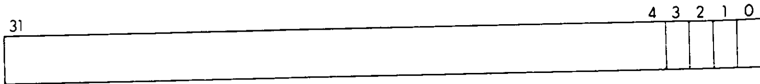
**Bits <20:8> All zeros**
Reserved for future use.

**Bits <7:0>**
Each adapter is assigned a unique code identifying it.

MBA adapter code is:

Bits <7:0>=00100000

## MBA Control Register (Byte Offset =4)




**Bits <31:4> All zeros**
Reserved for future use.

**Bit <3> MB Maintenance Mode**
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> Interrupt Enable**
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> ABORT**
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> Initialization (INIT)**
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.

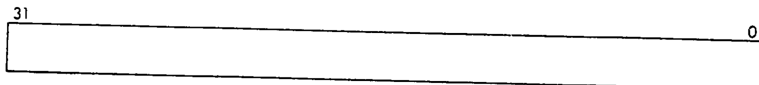
Clear control and status bits of diagnostic registers.

Cancel all pending commands except Read Data Pending.

Abort data transfer.

Assert MASSBUS INIT.

**MBA Status Register (Byte Offset=8)**




**Bit <31> DTBUSY**
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> NRCONF**
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.

**Bit <29> CRD**
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.

**Bits <28:20>**
All zeros. Reserved for future use.

**Bit <19> PGE**
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.

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> NFD**
Nonexisting drive. This bit is set when drive fails to assert TRA within 1.5 $\mu$secs 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> MCPE**
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> ATTN**
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.

**Bits <15:14>**
All zeros. Reserved for future use.

**Bit <13> DT CMP**
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> DTABT**
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> DLT**
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.

**Bit <10> WCK UP ERR**
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 <09> WCK LWR ERR**
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 <08> MXF**
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 <07> MBEXC**
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 <06> MDPE**
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 the bit or INIT. The setting of this bit will cause the data transfer operation to be aborted.

**Bit <05> MAPPE**
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 <04> INVMAP**
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 <03> ERR CONF**
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 <02> RDS**
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.

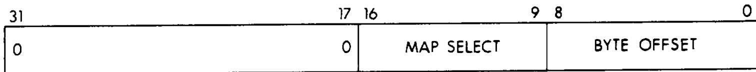
**Bit <01> IS TIMEOUT**
Interface Sequence Timeout. An interface sequence is defined as the

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.

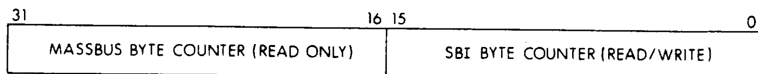
**Bit <00> RD TIMEOUT**
Read Data Timeout is defined as the time from when an interface sequence that specifies a read command is completed to the time that the specified read data is returned to the commander. The maximum time out is 102.4 μsecs. The setting of this bit will cause data transfer abort. Cleared by writing a one to this bit or INIT.

**MBA Virtual Address Register (Byte Offset=12)**




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 (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)




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)

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> IMDPG**
Invert MASSBUS Data Parity Generator.

**Bit <30> IMCPG**
Invert MASSBUS Control Parity Generator.

**Bit <29> IMAPP**
Invert Map Parity.

**Bit <28> BLKSCOM**
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> SIMSCLK**
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> SIMEBL**
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> SIMOCC**
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> SIMATTN**
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> MDIB SEL**
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.

**Bits <22:21> MAINT ONLY**
Read/write with no effect. (Used to test writability of these bits).

**Bit <20> MFAIL**
MASSBUS Fail (read-only). Fail is asserted when MMM is set.

**Bit <19> MRUN**
Maintenance MASSBUS Run (read-only).

**Bit <18> MWCLK**
Maintenance MASSBUS WCLK (read-only).

**Bit <17> MFXC**
Maintenance MASSBUS FXC (read-only).

**Bit <16> MCTOD**
Maintenance MASSBUS MCTOD (read-only).

**Bits <15:13> MDS**
Maintenance MASSBUS Device Select (read-only).

**Bits <12:8> MRS**
Maintenance MASSBUS Register Select (read-only).

**Bits <7:0> U/L MDIB**
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)**

**Bit <31>**
Valid Bit

**Bits <30:21>**
Zeros. Reserved for future use.

**Bits <20:0>**
Physical Page Frame Number.

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 subsystem 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 AT TR=8
        MBA_CSR =^X0               ;  OFFSET TO SBI STATUS REGISTER
        MBA_CR  =^X4               ;  OFFSET TO MBA CONTROL REGISTER
         MBA_INIT=^X1              ;   INITIALIZE BIT
        MBA_SR  =^X8               ;  OFFSET TO MBA STATUS REGISTER
         MBA_DT_BUSY=^X80000000    ;   DTBUSY STATUS BIT
         MBA_DT_ABORT=^X2000       ;   DTABORT STATUS BIT
        MBA_VAR =^XC               ;  OFFSET TO MBA VIRTUAL ADDRESS REGISTER
        MBA_BCR =^X10              ;  OFFSET TO MBA BYTE COUNT REGISTER
        MBA_MAP0=^X800             ;  OFFSET TO FIRST MAP REGISTER
         MAP_VALID=^X80000000      ;   VALID BIT IN MAP REGISTER
        RP1_CSR =^X480             ;  OFFSET TO CSR ON RP DRIVE 1
         RP_SET_VV_CMD=^X13        ;   SET VOLUME VALID COMMAND
         RP_READ_CMD  =^X39        ;   READ COMMAND
        RP1_DA   =^X494            ;  OFFSET TO SECTOR AND TRACK REGISTER
        RP1_OF   =^X4A4            ;  OFFSET TO TRACK OFFSET REGISTER?
         RP_16BIT_FORMAT=^X1000    ;   16BIT FORMAT ON DISC SURFACE
        RP1_DC   =^X4A8            ;  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 R0
        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 ORED IN AND MAP
;       REGISTER 0 IS LOADED.  SUCESSIVE 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    MEMSAD,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
```

```
1$:     BISL3   #MAP_VALID,R4,(R1)+ ; SET VALID BIT AND MOVE INTO MAP
        INCL    R4                  ; BUMP PAGE ADDRESS
                                    ; THERE ARE 200 (HEX) BYTES PER PAGE
        SUBL    #^X200,R2           ; SET UP ALL MAPS?
        BGTR    1$                  ; NO, SO SET UP NEXT MAP
        BISL3   #MAP_VALID,R4,(R1)+ ; ENABLE ONE MORE MAP SINCE
                                    ; TRANSFER MAY NOT BE PAGE ALIGNED
        CLRL    (R1)                ; CLEAR VALID BIT IN UNUSED MAP
                                    ; THIS WILL STOP THE MBA IF IT SHOULD
                                    ; DO SOMETHING WRONG AND TRY TO USE
                                    ; THIS MAP ENTRY
        BICL    #^XFFFFFE00,R3      ; LEAVE ONLY BYTE OFFSET INTO PAGE
        MOVL    R3,MBA_VAR(R0)      ; LOAD VIRTUAL ADDRESS REGISTER.

;       THIS SECTION LOADS THE MBA BYTE COUNT REGISTER, AND THE
;       ADDRESS REGISTERS IN THE DISC

        MNEGL   BCOUNT,MBA_BCR(R0) ; 2'S COMPLEMENT OF # BYTES TO XFER
        MOVL    SCYL,RP1_DC(R0)    ; SET DESIRED CYLINDER
        MOVL    SECTOR,R2
        INSV    STRACK,#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 SUCESSFULL THE ERROR
;       BIT WILL BE CLEAR.

        MOVL    #RP_READ_CMD,RP1_CSR(R0) ; INITIATE READ OPERATION
2$:     BITL    #MBA_DT_BUSY,MBA_SR(R0) ; DTBUSY STILL SET?
        BNEQ    2$                 ; YES -- TRANSFER STILL IN PROGRESS
        BITL    #MBA_DT_ABORT,MBA_SR(R0) ; XFER ABORTED?
        BNEQ    3$                 ; YES

        HALT    ; NORMAL COMPLETION. SUCESS
3$:     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

        .END BEGIN
$
```

# CHAPTER 11
# 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 2, 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 11-1 illustrates the system identification register.

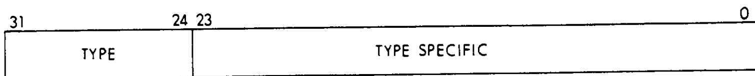



Figure 11-1 System Identification Register

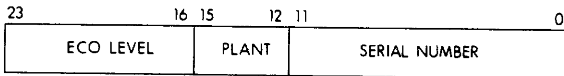
Type — A unique number assigned by engineering to identify a specific processor:

| | |
|---|---|
| 0 | Reserved to DIGITAL (error) |
| 1 | VAX-11/780 |

| | |
|---|---|
| 2 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:




## 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 11-2 illustrates the console receive control/status register.

Figure 11-2 Console Receive Control/Status Register (RXCS)

Figure 11-3 illustrates the read-only console receive data buffer register.

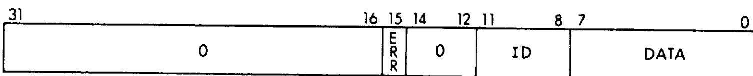



Figure 11-3 Console Receive Data Buffer Register (RXDB)

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 DON) 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 appears in DATA. When an MFPR #RXDB,dst is executed, DONe is cleared as is any interrupt request. If ID is 0 then the data is from the console terminal. If ID is non-zero, then the entire register is implementation dependent. In the case of the VAX-11/780, if ID = 1, data is from the floppy disk.

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 is sent to the floppy disk. Figure 11-4 illustrates the console transmit control/status register.

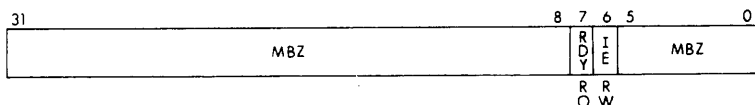



Figure 11-4 Console Transmit Control/Status Register (TXCS)

Figure 11-5 illustrates the read-only console transmit data buffer register.

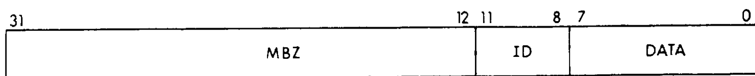



Figure 11-5 Console Transmit Data Buffer Register (TXDB)

## 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 (optional)**

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 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 11-6.

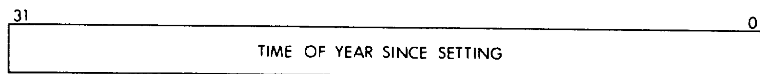



Figure 11-6 Time-of-Year Clock Register (TODR)

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 $\mu$sec 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 11-7 illustrates the interval count register.

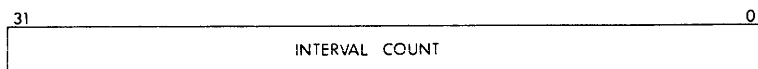



Figure 11-7 Interval Count Register (ICR)

Figure 11-8 illustrates the next interval count register.

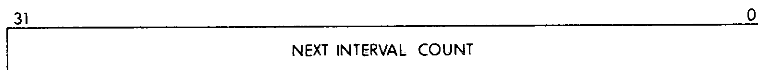



Figure 11-8 Next Interval Count Register (NICR)

Figure 11-9 illustrates the interval clock control/status register.

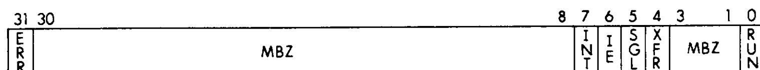



Figure 11-9 Interval Clock Control/Status Register (ICCS)

**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.

**ERR<31>**
Whenever ICR overflows, if INT is already set, then ERR is set. Thus, ERR indicates a missed clock tick. Attempt to set this bit via MTPR clears ERR.

**MBZ<30:8>**
Must Be Zero.

**INT<7>**
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).

**IE<6>**
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).

**SGL<5>**
A write-only bit. If RUN is clear, each time this bit is set, ICR is incremented by one.

**XFR<4>**
A write-only bit. Each time this bit is set, NICR is transferred to ICR.

**MBZ<3:1>**
Must Be Zero.

**RUN<0>**
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 #↑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.

**VAX-11/780 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 11-10 illustrates the accelerator control/status register.

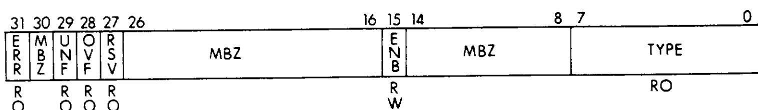



Figure 11-10 Accelerator Control/Status Register (ACCS)

**ERR<31>**
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.

**MBZ<30>**
Must Be Zero.

**UNF<29>**
Read-only bit specifying that the last operation had an underflow.

**OVF<28>**
Read-only bit specifying that the last operation had an overflow.

**RSV<27>**
Read-only bit specifying that the last operation had a reserved operand.

**MBZ<26:16>**
Must Be Zero.

**ENB<15>**
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.

**TYPE<7:0>**
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 11-11 illustrates the accelerator maintenance register.

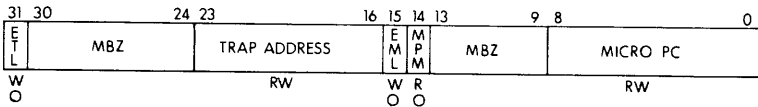



Figure 11-11 Accelerator Maintenance Register (ACCR)

**ETL<31>**
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.

**MBZ<30:24>**
Must Be Zero.

**TRAP<23:16>**
Trap Address. A read/write field used by the main processor to force the accelerator to a specified micro location.

**EML<15>**
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.

**MPM<14>**
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.

**MBZ<13:9>**
Must Be Zero.

**PC<8:0>**
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 control/status of its

microcode. Two are used for writing into any writable control store (WCS) and one is used to control micro breakpoints. Figure 11-12 illustrates the writable control store address register.

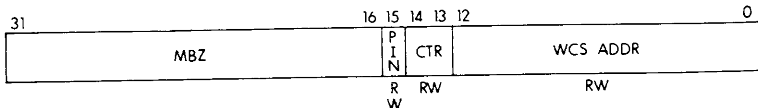



Figure 11-12 Writable Control Store Address Register (WCSA)

Figure 11-13 illustrates the writable control store data register.

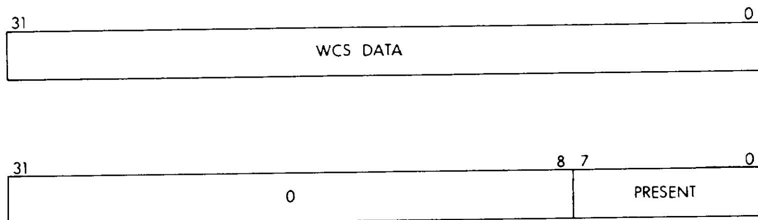



Figure 11-13 Writable Control Store Data Register (WCSD)

Reading WCSD indicates which control store addresses are writable. If WCSD<n> is set, then addresses n*1024 through n*1024+1023 are writable (i.e., that WCSA<12:10> 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 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 11-14 illustrates the microprogram breakpoint address register.

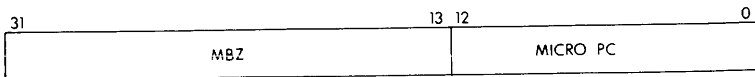
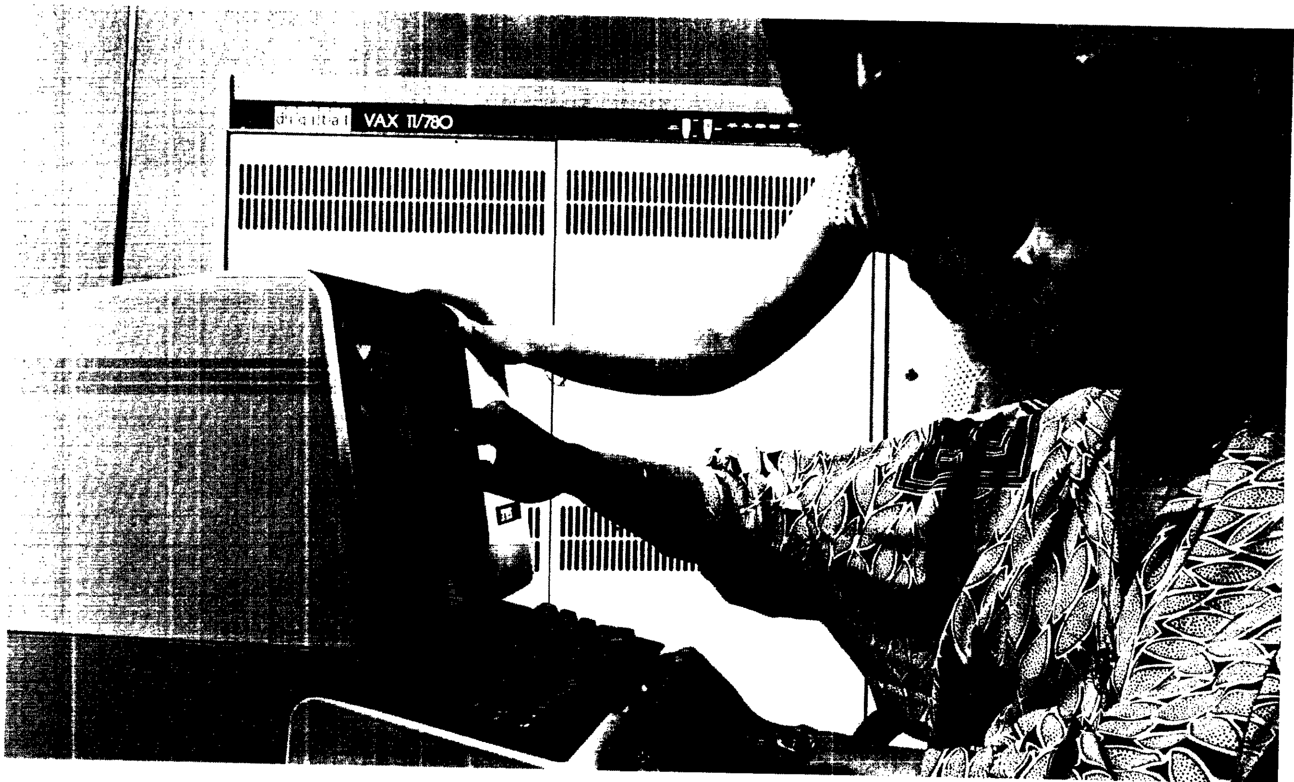



Figure 11-14 Microprogram Breakpoint Address Register (MBRK)

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.


digital VAX 11/780

# CHAPTER 12
# PRIVILEGE INSTRUCTIONS

## INTRODUCTION

The privilege instructions allow access to privilege operations within the VAX-11 system. The change mode instructions provide a controlled mechanism for unprivileged software to request services of more privileged software. In particular, the change mode instructions are the only normal way for code executing at executive, supervisor, or user access modes to change to a more privileged mode. In all cases, the change mode results in transferring control to a fixed location depending upon contents of the System Control Block.

The probe instructions allow software executing in response to a change mode to probe the accessibility of specified virtual locations by the program that changed moe. Thus, privileged software can verify that the arguments passed to it represent locations that could be accessed by its caller.

The extended function instruction provides a controlled mechanism for software to request services of non-standard 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.

The move to and from processor register instructions provide software executing in kernel mode access to the internal control registers of the processor. This allows such operations as control of the memory management system and selection of the address of the Process Control Block of the next process to execute. The load and save process context instructions allow kernel mode software to save and restore the general register and memory management status of a proccess when switching between processes.

Refer to Appendix G for a description of the symbolic notation associated with the instruction descriptions.

# CHM

## 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>) };
-(SP)    ←PSL;                          !initiate CHMx
-(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 Codes:**

```
Z ←0;
N ←0;
V ←0;
C ←0;
```

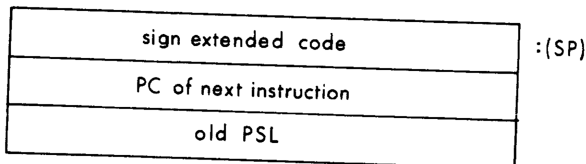
**Exceptions:** halt

**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:



# CHM

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.

**Example:**

```
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 ser-
               vice
              ;specified by customer code -2
```

# PROBE

## PROBE ACCESSIBILITY

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

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

**Operation:** probe-mode ← MAXU(mode<1:0>, PSL<previousr-mode>); condition codes ←{ {accessibility of (base) } and {accessiblity 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 it 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 privileged 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 residency. However, probing a process address may cause a page fault in the system address space on the per-process page tables.

**Notes:**

**Example:**

```
MOVL     4(AP),R0     ;copy address of first arg 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
```

# PROBE

```
PROBEW    $0,R0,R1    ;verify that the buffer described
                      ;by the second and third argu-
                        ments
                      ;could be written by the previous
                      ;access mode
                      ;Note that the argument list must
                      ;already have been probed and
                        that
                      ;the second argument must be
                        known
                      ;to be less than 514
```

# XFC

## EXTENDED FUNCTION CALL

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

**Format:** opcode

**Operation:** {XFC fault};

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

**Exceptions:**
opcode reserved to customer
customer reserved exception

**Opcodes:** FC XFC Extended Function Call

**Description:** This instruction requests services of non-standard microcode or software. If no special microcode is loaded then an exception is generated to a kernel mode software simulator (see Chapter 12). 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.

# MFPR
# MTPR

## MOVE FROM PRIVILEGED REGISTER
## MOVE TO PRIVILEGED REGISTER

**Purpose:** provide access to the internal privileged regsters

**Format:**

opcode src.rl, regnumber.rl — MTPR
opcode regnumber.rl, dst.wl — MFPR

**Operation:**

if PSL<current-mode> NEQU kernel then {reserved instruction falt};
PRS [regnumber] ← src; !MTRP
dst ←PRS[regnumber]; !MFPR

**Condition Codes:**

N ← dst LSS 0;
Z ← dst EQL 0;
V ← 0;
C ← C;

**Exceptions:**

reserved operand
reserved 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 implicaton 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 implicity 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

# MFPR
# MTPR

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-11 Series Registers**

| Register Name | Mnemonic | Number | Type | Scope | Init? |
|---|---|---|---|---|---|
| Kernel Stack Pointer | KSP | 0 | R/W | PROC | — |
| Executive Stack Pointer | ESP | 1 | R/W | PROC | — |
| Supervisor Stack Pointer | SSP | 2 | R/W | PROC | — |
| User Stack Pointer | USP | 3 | R/W | PROC | — |
| Interrupt Stack Pointer | ISP | 4 | R/W | CPU | — |
| P0 Base Register | P0BR | 8 | R/W | PROC | — |
| P0 Length Register | P0LR | 9 | R/W | RPOC | — |
| P1 Base Register | P1BR | 10 | R/W | RPOC | — |
| P1 Length Register | P1LR | 11 | R/W | PROC | — |
| System Base Register | SBR | 12 | R/W | CPU | — |
| System Length Register | SLR | 13 | R/W | CPU | — |
| Process Control Block Base | PCBB | 16 | R/W | PROC | — |
| System Control Block Base | SCBB | 17 | R/W | CPU | — |
| Interrupt Priority Level | IPL | 18 | R/W | CPU | yes |
| AST Level | ASTLVL | 19 | R/W | PROC | yes |
| Software Interrupt Request | SIRR | 20 | W | CPU | — |
| Software Interrupt Summary | SISR | 21 | R/W | CPU | yes |
| Interval Clock Control | ICCS | 24 | R/W | CPU | yes |
| Next Interval Count | NICR | 25 | W | CPU | — |
| Interval Count | ICR | 26 | R | CPU | — |
| Time of Year (optional) | TODR | 27 | R/W | CPU | no |
| Console Receiver C/S | RXCS | 32 | R/W | CPU | yes |
| Console Receiver D/B | RXDB | 33 | R | CPU | — |
| Console Transmit C/S | TXCS | 34 | R/W | CPU | yes |
| Console Transmit D/B | TXDB | 35 | W | CPU | — |
| Memory Management Enable | MAPEN | 56 | R/W | CPU | yes |
| Trans. Buf. Invalidate All | TBIA | 57 | W | CPU | — |
| Trans. Buf. Invalidate Single | TBIS | 58 | W | CPU | — |
| Performance Monitor Enable | PMR | 61 | R/W | PROC | yes |
| System Identification | SID | 62 | R | CPU | no |

# MFPR
# MTPR

**VAX-11 Series Registers**

| Register Name | Mne-monic | Num-ber | Type | Scope | Init? |
|---|---|---|---|---|---|
| Accelerator Control/ Status | ACCS | 40 | R/W | CPU | yes |
| Accelerator Maintenance | ACCR | 41 | R/W | CPU | no |
| WCS address | WCSA | 44 | R/W | CPU | no |
| WCS data | WCSD | 45 | R/W | CPU | yes |
| SBI Fault/Status | SBIFS | 48 | R/W | CPU | yes |
| SBI Silo | SBIS | 49 | R | CPU | no |
| SBI Silo Comparator | SBISC | 50 | R/W | CPU | yes |
| SBI Maintenance | SBIMT | 51 | R/W | CPU | yes |
| SBI Error Register | SBIER | 52 | R/W | CPU | yes |
| SBI Timeout Address | SBITA | 53 | R | CPU | — |
| SBI Quadword Clear | SIBQC | 54 | W | CPU | — |
| Micro Program Breakpoint | MBRK | 60 | R/W | CPU | no |

# LDPCTX
# SVPCTX

## LOAD PROCESS CONTEXT
## SAVE PROCESS CONTEXT

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

**Format:** opcode

**Operation:**

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

**Condition Codes:**

N ← N;
Z ← Z;
V ← V;
C ← C;

**Exceptions:** reserved operand
reserved instruction

**Opcodes:**

| | | |
|---|---|---|
| 06 | LDPCTX | Load Process Context |
| 07 | SVPCTX | 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.

## CHAPTER 13
# SYSTEM ARCHITECTURAL IMPLICATIONS

### INTRODUCTION

Portions of the VAX-11 architecture have implications on the hardware system structure. The areas of interaction are: data sharing and synchronization, restartability, interrupts and errors, and the I/O structure. Of these, data sharing is most visible to the programmer.

### 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-11 architecture, implicit sharing is transparent to the programmer. The memory system must be implemented so 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, and one processor executes INCB 0 and another executes INCB 1. Then, regardless of the order of execution, including effectively simultaneous, the final contents must be 6 and 7.

Access to explicitly shared data that may be written must be synchronized by the programmer or hardware designer. Before accessing shared writable data, the programmer must acquire control of the data structure. Five 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 then a 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 then a write operation to a single aligned word in an interlocked sequence to allow counters to be maintained without other interlocks. The INSQUE and REMQUE instructions use a hardware-provided primitive operation to make a series of aligned long-

word 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 that is within the synchronizing part of these instructions can occur between the synchronized reads and the writes of these instructions. Hardware designers must use these primitive operations directly when making references in an interlocked sequence. Hardware faults during an interlocked sequence must not hang any processor. On the VAX-11/780, only interlocking instructions are locked out by the interlock.

The SBI primitive operations are interlock read and interlock write.

In order to provide a functionality upon which some UNIBUS peripheral devices rely, processors must 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 if implemented using byte- or word-modifying references. This constraint also does not apply to instructions precluded from I/O space references.

The operation of the ADAWI instruction is interlocked against similiar operations on other processors in a multiprocessor system.

In a multiprocessor system, any software clearing the V bit, or changing the protection code of a page table entry for system space that it issues an MTPR xxx,#TBIS, must arrange for all other processors to issue a similar TBIS. The original processor must wait until all the other processors have completed their TBIS before it allows access to the system page.

## CACHE

A hardware implementation may include a mechanism to reduce access time by making local copies of recently used memory contents. Such a mechanism is termed a cache. A cache must be implemented in such a way that its existence is transparent to software (except for timing and error reporting/control). In particular, the following must be true:

1. Program writes to memory, followed by starting a peripheral output transfer, must output the updated value.
2. Completing a peripheral input transfer followed by the program reading of memory must read the input value.
3. A write or modify followed by a HALT on one processor followed by a read or modify on another processor must read the updated value.

4. A write or modify followed by a power failure, followed by restoration of power, followed by a read or modify, must read the updated value provided that the duration of the power failure does not exceed the maximum nonvolatile period of the main memory.
5. In multiprocessor systems, access to variables shared between processors must be interlocked by software executing BBxxI, ADAWI, or xxxQUE 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. Valid accesses to I/O registers must not be cached.

On the VAX-11/780, this is achieved by a cache that writes through to memory and watches the memory bus for all external writes to memory.

At bootstrap time, the cache must be either empty or valid.

**RESTARTABILITY**

The VAX-11 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.

In order that instructions which store intermediate results in the general registers not compromise system integrity, they must insure that any addresses stored or used are virtual addresses, subject to protection checking, and that any state information stored or used cannot result in a noninterruptable or nonterminating sequence.

Instruction operands that are peripheral device registers being accessed, may produce UNPREDICTABLE results because of instruction restarting after faults. In order that software may dependably access peripheral device registers, instructions used to access them must not permit device interrupts during their execution.

Memory modifications produced as a byproduct of instruction execution, e.g., memory access statistics, are specifically excluded from the constraint that memory not be altered until the instruction can be completed.

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

## INTERRUPTS

Underlying the VAX-11 architectural concept of an interrupt is the notion that an interrupt request is a static condition, not a transient event, which 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 (subject to implementation-dependent timing constraints) without consequence.

It is necessary that any instruction changing the processor priority (IPL), so that a pending interrupt is enabled, must allow the interrupt to occur before executing the next waiting instruction.

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

## ERRORS

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

Error notification interrupts may be delayed by the apparent completion of the instruction in execution at the time of the error, but if enabled, the interrupt must be requested before processor context is switched.

## I/O STRUCTURE

The VAX-11 I/O architecture is very similar to the PDP-11 structure, the principal difference being the method by which processor registers (such as the PSL) are accessed (reference the Architecture Handbook). 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. On the VAX-11/780 implementaton, this I/O space occupies the upper half of the physical address space and is $2^{29}$ bytes in length. Use of general instructions permits all the virtual address mapping and protection mechanisms described in Chapter 6, Memory Management, to be used when referencing I/O registers.

**NOTE**

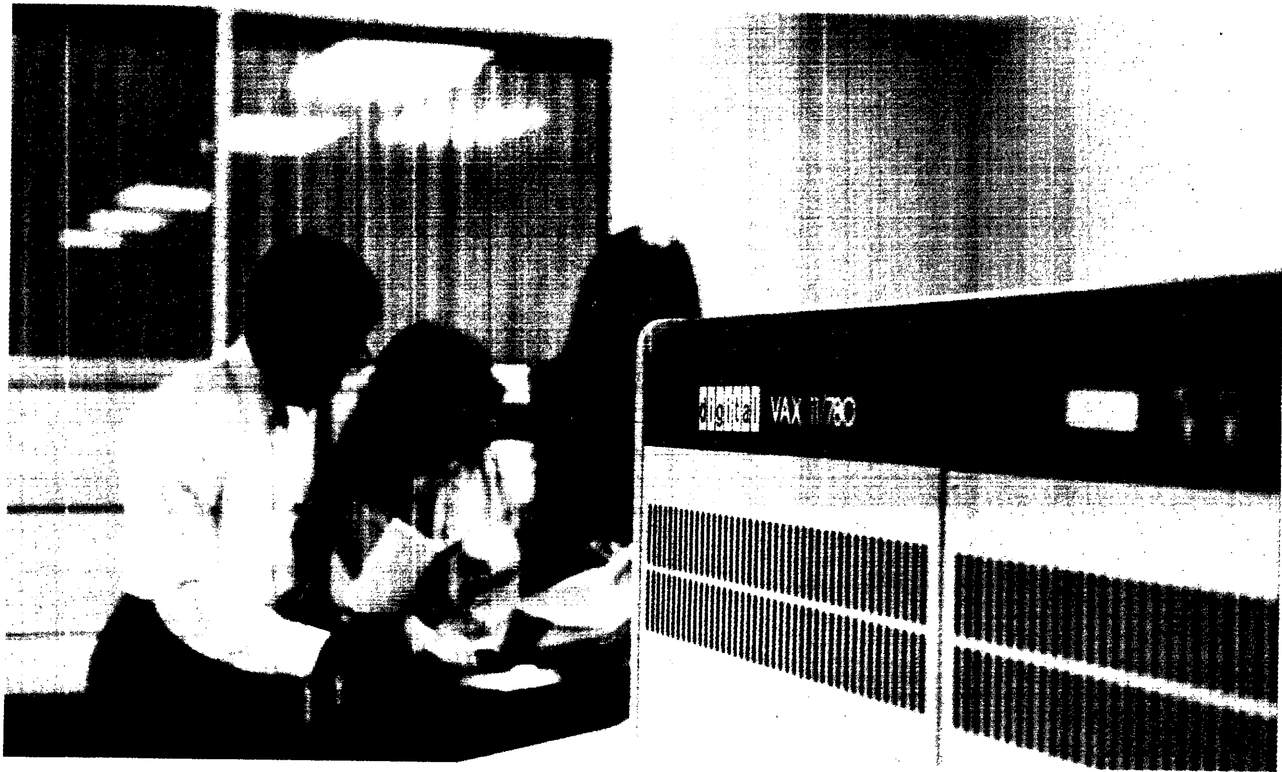
Implementations that include a cache feature must suppress caching for references in the I/O space.

For any member of the VAX-11 series implementing the UNIBUS, there will be one or more areas of the I/O physical address space, each $2^{18}$ bytes in length, which "maps through" to the UNIBUS addresses. The collection of these areas is referred to as the UNIBUS space.

**Constraints on I/O Registers**

The following is a list of both hardware and programming constraints on I/O registers. These items affect both hardware register design and programming considerations.

1. The physical address of an I/O register must be an integral multiple of the register size in bytes (which must be a power of two), i.e., all registers must be aligned on natural boundaries.
2. References using a length attribute other than the length of the register and/or 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 all peripheral devices, error and status bits that may be asynchronously set by the device must be cleared by software writing a "0". 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 (i.e., ".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, double, floating, and field references in the I/O space result in UNDEFINED behavior.


digital
VAX 11/730

CHAPTER 14

# 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 reliablity.

  *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 register save/restore and consistency checking. The use of these instructions 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 memory 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 corrects errors in a single error burst of 11 bits. Detections and corrections are noted in the error log as a preventive maintenance aid.
- A *System Identifications (SID) hardware register* maintains information 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 predetermined 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 information 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.

APPENDIX A

# 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 |

| | |
|---|---|
| **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 |
| **UETP** | User Environment Test Package |
| **UFD** | User File Directory |
| **UIC** | User Identification Code |

| | |
|---|---|
| **USP** | User mode Stack Pointer |
| **VCB** | Volume Control Block |
| **VPN** | Virtual Page Number |
| **WCB** | Window Control Block |
| **WCS** | Writeable Control Store |
| **WDCS** | Writeable Diagnostic Control Store |

# APPENDIX B

# INSTRUCTION INDEX

## B.1. MNEMONIC LISTING

| MNEMONIC | INSTRUCTION | OPCODE |
|---|---|---|
| ACBB | Add compare and branch byte | 9D |
| ACBD | Add compare and branch double | 6F |
| ACBF | Add compare and branch floating | 4F |
| ACBL | Add compare and branch long | F1 |
| ACBW | Add compare and branch word | 3D |
| ADAWI | Add aligned word interlocked | 58 |
| ADDB2 | Add byte 2 operand | 80 |
| ADDB3 | Add byte 3 operand | 81 |
| ADDD2 | Add double 2 operand | 60 |
| ADDD3 | Add double 3 operand | 61 |
| ADDF2 | Add floating 2 operand | 40 |
| ADDF3 | Add floating 3 operand | 41 |
| ADDL2 | Add long 2 operand | C0 |
| ADDL3 | Add long 3 operand | C1 |
| ADDP4 | Add packed 4 operand | 20 |
| ADDP6 | Add packed 6 operand | 21 |
| ADDW2 | Add word 2 operand | A0 |
| ADDW3 | Add word 3 operand | A1 |
| ADWC | Add with carry | D8 |
| AOBLEQ | Add one and branch on less or equal | F3 |
| AOBLSS | Add one and branch on less | F2 |
| ASHL | Arithmetic shift long | 78 |
| ASHP | Arithmetic shift and round packed | F8 |
| ASHQ | Arithmetic shift quad | 79 |
| BBC | Branch on bit clear | E1 |
| BBCC | Branch on bit clear and clear | E5 |
| BBCCI | Branch on bit clear and clear interlocked | E7 |
| BBCS | Branch on bit clear and set | E3 |
| BBS | Branch on bit set | E0 |
| BBSC | Branch on bit set and clear | E4 |
| BBSS | Branch on bit set and set | E2 |
| BBSSI | Branch on bit set and set interlocked | E6 |
| BCC | Branch on carry clear | 1E |
| BCS | Branch on carry set | 1F |
| BEQL | Branch on equal | 13 |
| BEQLU | Branch on equal unsigned | 13 |
| BGEQ | Branch on greater or equal | 18 |
| BGEQU | Branch on greater or equal unsigned | 1E |

| MNEMONIC | INSTRUCTION | OPCODE |
|---|---|---|
| BGTR | Branch on greater | 14 |
| BGTRU | Branch on greater unsigned | 1A |
| BICB2 | Bit clear byte 2 operand | 8A |
| BICB3 | Bit clear byte 3 operand | 8B |
| BICL2 | Bit clear long 2 operand | CA |
| BICL3 | Bit clear long 3 operand | CB |
| BICPSW | Bit clear program status word | B9 |
| BICW2 | Bit clear word 2 operand | AA |
| BICW3 | Bit clear word 3 operand | AB |
| BISB2 | Bit set byte 2 operand | 88 |
| BISB3 | Bit set byte 3 operand | 89 |
| BISL2 | Bit set long 2 operand | C8 |
| BISL3 | Bit set long 3 operand | C9 |
| BISPSW | Bit set program status word | B8 |
| BISW2 | Bit set word 2 operand | A8 |
| BISW3 | Bit set word 3 operand | A9 |
| BITB | Bit test byte | 93 |
| BITL | Bit test long | D3 |
| BITW | Bit test word | B3 |
| BLBC | Branch on low bit clear | E9 |
| BLBS | Branch on low bit set | E8 |
| BLEQ | Branch on less or equal | 15 |
| BLEQU | Branch on less or equal unsigned | 1B |
| BLSS | Branch on less | 19 |
| BLSSU | Branch on less unsigned | 1F |
| BNEQ | Branch on not equal | 12 |
| BNEQU | Branch on not equal unsigned | 12 |
| BPT | Break point fault | 03 |
| BRB | Branch with byte displacement | 11 |
| BRW | Branch with word displacement | 31 |
| BSBB | Branch to subroutine with byte displacement | 10 |
| BSBW | Branch to subroutine with word displacement | 30 |
| BVC | Branch on overflow clear | 1C |
| BVS | Branch on overflow set | 1D |
| CALLG | Call with general argument list | FA |
| CALLS | Call with stack | FB |
| CASEB | Case byte | 8F |
| CASEL | Case long | CF |
| CASEW | Case word | AF |
| CHME | Change mode to executive | BD |
| CHMK | Change mode to kernel | BC |
| CHMS | Change mode to supervisor | BE |
| CHMU | Change mode to user | BF |
| CLRB | Clear byte | 94 |
| CLRD | Clear double | 7C |

| MNEMONIC | INSTRUCTION | OPCODE |
| --- | --- | --- |
| CLRF | Clear float | D4 |
| CLRL | Clear long | D4 |
| CLRQ | Clear quad | 7C |
| CLRW | Clear word | B4 |
| CMPB | Compare byte | 91 |
| CMPC3 | Compare character 3 operand | 29 |
| CMPC5 | Compare character 5 operand | 2D |
| CMPD | Compare double | 71 |
| CMPF | Compare floating | 51 |
| CMPL | Compare long | D1 |
| CMPP3 | Compare packed 3 operand | 35 |
| CMPP4 | Compare packed 4 operand | 37 |
| CMPV | Compare field | EC |
| CMPW | Compare word | B1 |
| CMPZV | Compare zero-extended field | ED |
| CRC | Calculate cyclic redundancy check | 0B |
| CVTBD | Convert byte to double | 6C |
| CVTBF | Convert byte to float | 4C |
| CVTBL | Convert byte to long | 98 |
| CVTBW | Convert byte to word | 99 |
| CVTDB | Convert double to byte | 68 |
| CVTDF | Convert double to float | 76 |
| CVTDL | Convert double to long | 6A |
| CVTDW | Convert double to word | 69 |
| CVTFB | Convert float to byte | 48 |
| CVTFD | Convert float to double | 56 |
| CVTFL | Convert float to long | 4A |
| CVTFW | Convert float to word | 49 |
| CVTLB | Convert long to byte | F6 |
| CVTLD | Convert long to double | 6E |
| CVTLF | Convert long to float | 4E |
| CVTLP | Convert long to packed | F9 |
| CVTLW | Convert long to word | F7 |
| CVTPL | Convert packed to long | 36 |
| CVTTP | Convert trailing numeric to packed | 26 |
| CVTPT | Convert packed to trailing numeric | 24 |
| CVTPS | Convert packed to leading separate numeric | 08 |
| CVTRDL | Convert rounded double to long | 6B |
| CVTRFL | Convert rounded float to long | 4B |
| CVTSP | Convert leading separate numeric to packed | 09 |
| CVTWB | Convert word to byte | 33 |
| CVTWD | Convert word to double | 6D |
| CVTWF | Convert word to float | 4D |
| CVTWL | Convert word to long | 32 |
| DECB | Decrement byte | 97 |
| DECL | Decrement long | D7 |
| DECW | Decrement word | B7 |

| MNEMONIC | INSTRUCTION | OPCODE |
|---|---|---|
| DIVB2 | Divide byte 2 operand | 86 |
| DIVB3 | Divide byte 3 operand | 87 |
| DIVD2 | Divide double 2 operand | 66 |
| DIVD3 | Divide double 3 operand | 67 |
| DIVF2 | Divide floating 2 operand | 46 |
| DIVF3 | Divide floating 3 operand | 47 |
| DIVL2 | Divide long 2 operand | C6 |
| DIVL3 | Divide long 3 operand | C7 |
| DIVP | Divide packed | 27 |
| DIVW2 | Divide word 2 operand | A6 |
| DIVW3 | Divide word 3 operand | A7 |
| EDITPC | Edit packed to character | 38 |
| EDIV | Extended divide | 7B |
| EMODD | Extended modulus double | 74 |
| EMODF | Extended modulus floating | 54 |
| EMUL | Extended multiply | 7A |
| EXTV | Extract field | EE |
| EXTZV | Extract zero-extended field | EF |
| FFC | Find first clear bit | EB |
| FFS | Find first set bit | EA |
| HALT | Halt | 00 |
| INCB | Increment byte | 96 |
| INCL | Increment long | D6 |
| INCW | Increment word | B6 |
| INDEX | Compute index | 0A |
| INSQUE | Insert into queue | 0E |
| INSV | Insert field | F0 |
| JMP | Jump | 17 |
| JSB | Jump to subroutine | 16 |
| LDPCTX | Load process context | 16 |
| LOCC | Locate character | 3A |
| MATCHC | Match characters | 39 |
| MCOMB | Move complemented byte | 92 |
| MCOML | Move complemented long | D2 |
| MCOMW | Move complemented word | B2 |
| MFPR | Move from privileged register | DB |
| MNEGB | Move negated byte | 8E |
| MNEGD | Move negated double | 72 |
| MNEGF | Move negated floating | 52 |
| MNEGL | Move negated long | CE |
| MNEGW | Move negated word | AE |
| MOVAB | Move address of byte | 9E |
| MOVAD | Move address of double | 7E |
| MOVAF | Move address of float | DE |
| MOVAL | Move address of long | DE |

| MNEMONIC | INSTRUCTION | OPCODE |
|---|---|---|
| MOVAQ | Move address of quad | 7E |
| MOVAW | Move address of word | 3E |
| MOVB | Move byte | 90 |
| MOVC3 | Move character 3 operand | 28 |
| MOVC5 | Move character 5 operand | 2C |
| MOVD | Move double | 70 |
| MOVF | Move float | 50 |
| MOVL | Move long | D0 |
| MOVP | Move packed | 34 |
| MOVPSL | Move processor status longword | DC |
| MOVQ | Move quad | 7D |
| MOVTC | Move translated characters | 2E |
| MOVTUC | Move translated until character | 2F |
| MOVW | Move word | B0 |
| MOVZBL | Move zero-extended byte to long | 9A |
| MOVZBW | Move zero-extended byte to word | 9B |
| MOVZWL | Move zero-extended word to long | 3C |
| MTPR | Move to privileged register | DA |
| MULB2 | Multiply byte 2 operand | 84 |
| MULB3 | Multiply byte 3 operand | 85 |
| MULD2 | Multiply double 2 operand | 64 |
| MULD3 | Multiply double 3 operand | 65 |
| MULF2 | Multiply floating 2 operand | 44 |
| MULF3 | Multiply floating 3 operand | 45 |
| MULL2 | Multiply long 2 operand | C4 |
| MULL3 | Multiply long 3 operand | C5 |
| MULP | Multiply packed | 25 |
| MULW2 | Multiply word 2 operand | A4 |
| MULW3 | Multiply word 3 operand | A5 |
| NOP | No operation | 01 |
| POLYD | Evaluate polynomial double | 75 |
| POLYF | Evaluate polynomial floating | 55 |
| POPR | Pop registers | BA |
| PROBER | Probe read access | 0C |
| PROBEW | Probe write access | 0D |
| PUSHAB | Push address of byte | 9F |
| PUSHAD | Push address of double | 7F |
| PUSHAF | Push address of float | DF |
| PUSHAL | Push address of long | DF |
| PUSHAQ | Push address of quad | 7F |
| PUSHAW | Push address of word | 3F |
| PUSHL | Push long | DD |
| PUSHR | Push registers | BB |
| REI | Return from exception or interrupt | 02 |
| REMQUE | Remove from queue | 0F |

| MNEMONIC | INSTRUCTION | OPCODE |
|---|---|---|
| RET | Return from called procedure | 04 |
| ROTL | Rotate long | 9C |
| RSB | Return from subroutine | 05 |
| SBWC | Subtract with carry | D9 |
| SCANC | Scan for character | 2A |
| SKPC | Skip character | 3B |
| SOBGEQ | Subtract one and branch on greater or equal | F4 |
| SOBGTR | Subtract one and branch on greater | F5 |
| SPANC | Span characters | 2B |
| SUBB2 | Subtract byte 2 operand | 82 |
| SUBB3 | Subtract byte 3 operand | 83 |
| SUBD2 | Subtract double 2 operand | 62 |
| SUBD3 | Subtract double 3 operand | 63 |
| SUBF2 | Subtract floating 2 operand | 42 |
| SUBF3 | Subtract floating 3 operand | 43 |
| SUBL2 | Subtract long 2 operand | C2 |
| SUBL3 | Subtract long 3 operand | C3 |
| SUBP4 | Subtract packed 4 operand | 22 |
| SUBP6 | Subtract packed 6 operand | 23 |
| SUBW2 | Subtract word 2 operand | A2 |
| SUBW3 | Subtract word 3 operand | A3 |
| SVPCTX | Save process context | 07 |
| TSTB | Test byte | 95 |
| TSTD | Test double | 73 |
| TSTF | Test float | 53 |
| TSTL | Test long | D5 |
| TSTW | Test word | B5 |
| XFC | Extended function call | FC |
| XORB2 | Exclusive OR byte 2 operand | 8C |
| XORB3 | Exclusive OR byte 3 operand | 8D |
| XORL2 | Exclusive OR long 2 operand | CC |
| XORL3 | Exclusive OR long 3 operand | CD |
| XORW2 | Exclusive OR word 2 operand | TC |
| XORW3 | Exclusive OR word 3 operand | AD |
| | *Reserved to DEC* | 57 |
| | *Reserved to DEC* | 59 |
| | *Reserved to DEC* | 5A |
| | *Reserved to DEC* | 5B |
| | *Reserved to DEC* | 5C |
| | *Reserved to DEC* | 5D |
| | *Reserved to DEC* | 5E |
| | *Reserved to DEC* | 5F |
| | *Reserved to DEC* | 77 |
| ESCD | *Reserved to DEC* | FD |
| ESCE | *Reserved to DEC* | FE |
| ESCF | *Reserved to DEC* | FF |

## B.2. OPCODE LISTING

| OPCODE | MNEMONIC | INSTRUCTION |
|---|---|---|
| 00 | HALT | Halt |
| 01 | NOP | No operation |
| 02 | REI | Return from exception or interrupt |
| 03 | BPT | Break point fault |
| 04 | RET | Return from called procedure |
| 05 | RSB | Return from subroutine |
| 06 | LDPCTX | Load process context |
| 07 | SVPCTX | Save process context |
| 08 | CVTPS | Convert packed to leading separate numeric |
| 09 | CVTSP | Convert leading separate numeric to packed |
| 0A | INDEX | Compute index |
| 0B | CRC | Calculate cyclic redundancy check |
| 0C | PROBER | Probe read access |
| 0D | PROBEW | Probe write access |
| 0E | INSQUE | Insert into queue |
| 0F | REMQUE | Remove from queue |
| 10 | BSBB | Branch to subroutine with byte displacement |
| 11 | BRB | Branch with byte displacement |
| 12 | BNEQ, BNEQU | Branch on not equal unsigned, Branch on not equal |
| 13 | BEQL, BEQLU | Branch on equal, Branch on equal unsigned |
| 14 | BGTR | Branch on greater |
| 15 | BLEQ | Branch on less or equal |
| 16 | JSB | Jump to subroutine |
| 17 | JMP | Jump |
| 18 | BGEQ | Branch on greater or equal |
| 19 | BLSS | Branch on less |
| 1A | BGTRU | Branch on greater unsigned |
| 1B | BLEQU | Branch on less or equal unsigned |
| 1C | BVC | Branch on overflow clear |
| 1D | BVS | Branch on overflow set |
| 1E | BGEQU, BCC | Branch on greater or equal unsigned, Branch on carry clear |
| 1F | BLSSU, BCS | Branch on less unsigned, Branch on carry set |
| 20 | ADDP4 | Add packed 4 operand |
| 21 | ADDP6 | Add packed 6 operand |
| 22 | SUBP4 | Subtract packed 4 operand |
| 23 | SUBP6 | Subtract packed 6 operand |
| 24 | CVTPT | Convert packed to trailing numeric |
| 25 | MULP | Multiply packed |

| OPCODE | MNEMONIC | INSTRUCTION |
|---|---|---|
| 26 | CVTTP | Convert trailing numeric to packed |
| 27 | DIVP | Divide packed |
| 28 | MOVC3 | Move character 3 operand |
| 29 | CMPC3 | Compare character 3 operand |
| 2A | SCANC | Scan for character |
| 2B | SPANC | Span characters |
| 2C | MOVC5 | Move character 5 operand |
| 2D | CMPC5 | Compare character 5 operand |
| 2E | MOVTC | Move translated characters |
| 2F | MOVTUC | Move translated until character |
| 30 | BSBW | Branch to subroutine with word displacement |
| 31 | BRW | Branch with word displacement |
| 32 | CVTWL | Convert word to long |
| 33 | CVTWB | Convert word to byte |
| 34 | MOVP | Move packed |
| 35 | CMPP3 | Compare packed 3 operand |
| 36 | CVTPL | Convert packed to long |
| 37 | CMPP4 | Compare packed 4 operand |
| 38 | EDITPC | Edit packed to character |
| 39 | MATCHC | Match characters |
| 3A | LOCC | Locate character |
| 3B | SKPC | Skip character |
| 3C | MOVZWL | Move zero-extended word to long |
| 3D | ACBW | Add compare and branch word |
| 3E | MOVAW | Move address of word |
| 3F | PUSHAW | Push address of word |
| 40 | ADDF2 | Add floating 2 operand |
| 41 | ADDF3 | Add floating 3 operand |
| 42 | SUBF2 | Subtract floating 2 operand |
| 43 | SUBF3 | Subtract floating 3 operand |
| 44 | MULF2 | Multiply floating 2 operand |
| 45 | MULF3 | Multiply floating 3 operand |
| 46 | DIVF2 | Divide floating 2 operand |
| 47 | DIVF3 | Divide floating 3 operand |
| 48 | CVTFB | Convert float to byte |
| 49 | CVTFW | Convert float to word |
| 4A | CVTFL | Convert float to long |
| 4B | CVTRFL | Convert rounded float to long |
| 4C | CVTBF | Convert byte to float |
| 4D | CVTWF | Convert word to float |
| 4E | CVTLF | Convert long to float |
| 4F | ACBF | Add compare and branch floating |
| 50 | MOVF | Move float |
| 51 | CMPF | Compare floating |
| 52 | MNEGF | Move negated floating |

| OPCODE | MNEMONIC | INSTRUCTION |
| --- | --- | --- |
| 53 | TSTF | Test float |
| 54 | EMODF | Extended modulus floating |
| 55 | POLYF | Evaluate polynomial floating |
| 56 | CVTFD | Convert float to double |
| 57 | | RESERVED to DEC |
| 58 | ADAWI | Add aligned word interlocked |
| 59 | | RESERVED to DEC |
| 5A | | RESERVED to DEC |
| 5B | | RESERVED to DEC |
| 5C | | RESERVED to DEC |
| 5D | | RESERVED to DEC |
| 5E | | RESERVED to DEC |
| 5F | | RESERVED to DEC |
| 60 | ADDD2 | Add double 2 operand |
| 61 | ADDD3 | Add double 3 operand |
| 62 | SUBD2 | Subtract double 2 operand |
| 63 | SUBD3 | Subtract double 3 operand |
| 64 | MULD2 | Multiply double 2 operand |
| 65 | MULD3 | Multiply double 3 operand |
| 66 | DIVD2 | Divide double 2 operand |
| 67 | DIVD3 | Divide double 3 operand |
| 68 | CVTDB | Convert double to byte |
| 69 | CVTDW | Convert double to word |
| 6A | CVTDL | Convert double to long |
| 6B | CVTRDL | Convert rounded double to long |
| 6C | CVTBD | Convert byte to double |
| 6D | CVTWD | Convert word to double |
| 6E | CVTLD | Convert long to double |
| 6F | ACBD | Add compare and branch double |
| 70 | MOVD | Move double |
| 71 | CMPD | Compare double |
| 72 | MNEGD | Move negated double |
| 73 | TSTD | Test double |
| 74 | EMODD | Extended modulus double |
| 75 | POLYD | Evaluate polynomial double |
| 76 | CVTDF | Convert double to float |
| 77 | | RESERVED to DEC |
| 78 | ASHL | Arithmetic shift long |
| 79 | ASHQ | Arithmetic shift quad |
| 7A | EMUL | Extended multiply |
| 7B | EDIV | Extended divide |
| 7C | CLRQ, CLRD | Clear quad, Clear double |
| 7D | MOVQ | Move quad |
| 7E | MOVAQ, MOVAD | Move address of quad, Move address of double |
| 7F | PUSHAQ, PUSHAD | Push address of quad, Push address of double |

| OPCODE | MNEMONIC | INSTRUCTION |
|---|---|---|
| 80 | ADDB2 | Add byte 2 operand |
| 81 | ADDB3 | Add byte 3 operand |
| 82 | SUBB2 | Subtract byte 2 operand |
| 83 | SUBB3 | Subtract byte 3 operand |
| 84 | MULB2 | Multiply byte 2 operand |
| 85 | MULB3 | Multiply byte 3 operand |
| 86 | DIVB2 | Divide byte 2 operand |
| 87 | DIVB3 | Divide byte 3 operand |
| 88 | BISB2 | Bit set byte 2 operand |
| 89 | BISB3 | Bit set byte 3 operand |
| 8A | BICB2 | Bit clear byte 2 operand |
| 8B | BICB3 | Bit clear byte 3 operand |
| 8C | XORB2 | Exclusive OR byte 2 operand |
| 8D | XORB3 | Exclusive OR byte 3 operand |
| 8E | MNEGB | Move negated byte |
| 8F | CASEB | Case byte |
| 90 | MOVB | Move byte |
| 91 | CMPB | Compare byte |
| 92 | MCOMB | Move complemented byte |
| 93 | BITB | Bit test byte |
| 94 | CLRB | Clear byte |
| 95 | TSTB | Test byte |
| 96 | INCB | Increment byte |
| 97 | DECB | Decrement byte |
| 98 | CVTBL | Convert byte to long |
| 99 | CVTBW | Convert byte to word |
| 9A | MOVZBL | Move zero-extended byte to long |
| 9B | MOVZBW | Move zero-extended byte to word |
| 9C | ROTL | Rotate long |
| 9D | ACBB | Add compare and branch byte |
| 9E | MOVAB | Move address of byte |
| 9F | PUSHAB | Push address of byte |
| A0 | ADDW2 | Add word 2 operand |
| A1 | ADDW3 | Add word 3 operand |
| A2 | SUBW2 | Subtract word 2 operand |
| A3 | SUBW3 | Subtract word 3 operand |
| A4 | MULW2 | Multiply word 2 operand |
| A5 | MULW3 | Multiply word 3 operand |
| A6 | DIVW2 | Divide word 2 operand |
| A7 | DIVW3 | Divide word 3 operand |
| A8 | BISW2 | Bit set word 2 operand |
| A9 | BISW3 | Bit set word 3 operand |
| AA | BICW2 | Bit clear word 2 operand |
| AB | BICW3 | Bit clear word 3 operand |
| AC | XORW2 | Exclusive OR word 2 operand |
| AD | XORW3 | Exclusive OR word 3 operand |

| OPCODE | MNEMONIC | INSTRUCTION |
|---|---|---|
| AE | MNEGW | Move negated word |
| AF | CASEW | Case word |
| B0 | MOVW | Move word |
| B1 | CMPW | Compare word |
| B2 | MCOMW | Move complemented word |
| B3 | BITW | Bit test word |
| B4 | CLRW | Clear word |
| B5 | TSTW | Test word |
| B6 | INCW | Increment word |
| B7 | DECW | Decrement word |
| B8 | BISPSW | Bit set processor status word |
| B9 | BICPSW | Bit clear processor status word |
| BA | POPR | Pop registers |
| BB | PUSHR | Push register |
| BC | CHMK | Change mode to kernel |
| BD | CHME | Change mode to executive |
| BE | CHMS | Change mode to supervisor |
| BF | CHMU | Change mode to user |
| C0 | ADDL2 | Add long 2 operand |
| C1 | ADDL3 | Add long 3 operand |
| C2 | SUBL2 | Subtract long 2 operand |
| C3 | SUBL3 | Subtract long 3 operand |
| C4 | MULL2 | Multiply long 2 operand |
| C5 | MULL3 | Multiply long 3 operand |
| C6 | DIVL2 | Divide long 2 operand |
| C7 | DIVL3 | Divide long 3 operand |
| C8 | BISL2 | Bit set long 2 operand |
| C9 | BISL3 | Bit set long 3 operand |
| CA | BICL2 | Bit clear long 2 operand |
| CB | BICL3 | Bit clear long 3 operand |
| CC | XORL2 | Exclusive OR long 2 operand |
| CD | XORL3 | Exclusive OR long 3 operand |
| CE | MNEGL | Move negated long |
| CF | CASEL | Case long |
| D0 | MOVL | Move long |
| D1 | CMPL | Compare long |
| D2 | MCOML | Move complemented long |
| D3 | BITL | Bit test long |
| D4 | CLRL, CLRF | Clear long, Clear float |
| D5 | TSTL | Test long |
| D6 | INCL | Increment long |
| D7 | DECL | Decrement long |
| D8 | ADWC | Add with carry |
| D9 | SBWC | Subtract with carry |
| DA | MTPR | Move to processor register |
| DB | MFPR | Move from processor register |

| OPCODE | MNEMONIC | INSTRUCTION |
|---|---|---|
| DC | MOVPSL | Move processor status longword |
| DD | PUSHL | Push long |
| DE | MOVAL, MOVAF | Move address of long, Move address of float |
| DF | PUSHAL, PUSHAF | Push address of long, Push address of float |
| E0 | BBS | Branch on bit set |
| E1 | BBC | Branch on bit clear |
| E2 | BBSS | Branch on bit set and set |
| E3 | BBCS | Branch on bit clear and set |
| E4 | BBSC | Branch on bit set and clear |
| E5 | BBCC | Branch on bit clear and clear |
| E6 | BBSSI | Branch on bit set and set interlocked |
| E7 | BBCCI | Branch on bit clear and clear interlocked |
| E8 | BLBS | Branch on low bit set |
| E9 | BLBC | Branch on low bit clear |
| EA | FFS | Find first set bit |
| EB | FFC | Find first clear bit |
| EC | CMPV | Compare field |
| ED | CMPZV | Compare zero-extended field |
| EE | EXTV | Extract field |
| EF | EXTZV | Extract zero-extended field |
| F0 | INSV | Insert field |
| F1 | ACBL | Add compare and branch long |
| F2 | AOBLSS | Add one and branch on less |
| F3 | AOBLEQ | Add one and branch on less or equal |
| F4 | SOBGEQ | Subtract one and branch on greater or equal |
| F5 | SOBGTR | Subtract one and branch on greater |
| F6 | CVTLB | Convert long to byte |
| F7 | CVTLW | Convert long to word |
| F8 | ASHP | Arithmetic shift and round packed |
| F9 | CVTLP | Convert long to packed |
| FA | CALLG | Call with general argument list |
| FB | CALLS | Call with stack |
| FC | XFC | Extended function call |
| FD | ESCD to DEC | |
| FE | ESCE to DEC | |
| FF | ESCF to DEC | |

# APPENDIX C
# I/O SPACE RESTRICTIONS

A subset of native mode instructions is not used to reference I/O space. The reasons for this 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, floating, double, 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.

In any case, the programmer is responsible for ensuring that any memory reference to I/O space is in an instruction which cannot take an exception after the first I/O space reference. This includes deferred references to I/O space.

Instructions for which any explicit operands can be in I/O space are:

MOV{B,W,L}, PUSHL, CLR{B,W,L}, MNEG{B,W,L}, MCOM{B,W,L}, MOVZ{BW,BL,WL}, CVT{BW,BL,WB,WL,LB,LW}, CMP{B,W,L}, TST{B,W,L}, ADD{B,W,L}2, ADD{B,W,L}3, ADAWI, INC{B,W,L}, ADWC, SUB{B,W,L}2, SUB{B,W,L}3, DEC{B,W,L}, SBWC, BIT{B,W,L}, BIS{B,W,L}2, 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}, MOVPSL, 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 operand can be in I/O space are:

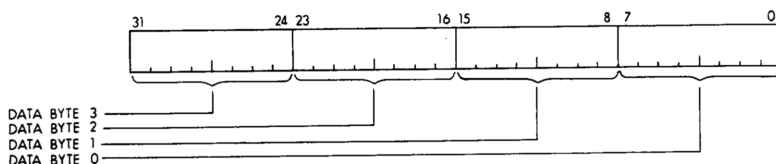
XFC (depending on implementation)
REMQUE addr (destination)

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, for example, be used as part of the bootstrap process. If this is done, then it is valid for software to transfer to this code.

# APPENDIX D
# INTERNAL DATA (ID) BUS REGISTERS

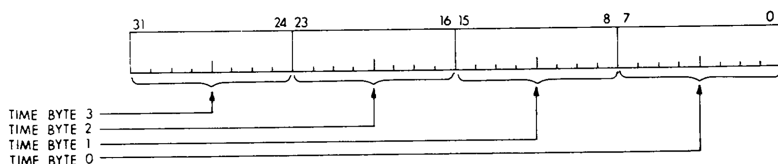
Instruction Buffer Register

ID Address 00
Processor Address —



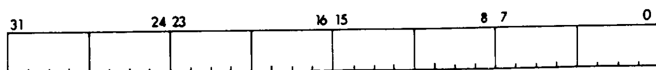

Time of Day Register

ID Address 01
Processor Address 1B



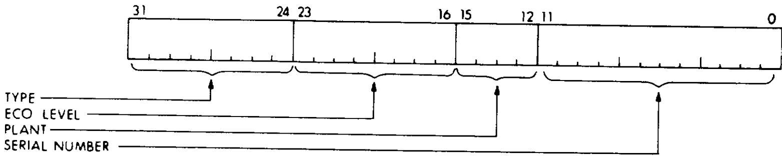

Reserved Register

ID Address 02
Processor Address —



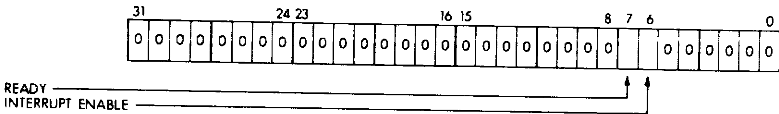
## System Identification Register

ID Address 03
Processor Address 3E



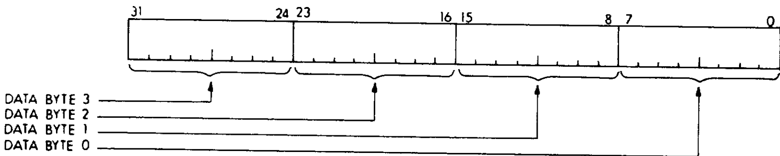

## Console Receive Control/Status Register

ID Address 04
Processor Address 20




## Console Receive Data Buffer Register

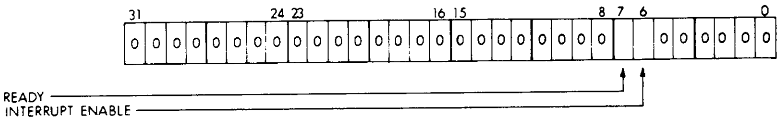
ID Address 05
Processor Address 21




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.

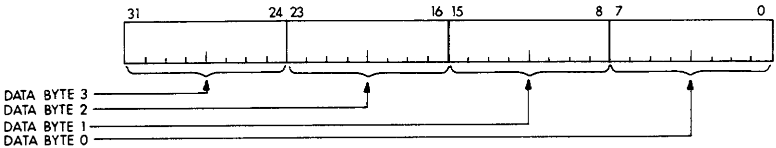
## Console Transmit Control/Status Register

ID Address 06
Processor Address 22




## Console Transmit Data Buffer Register

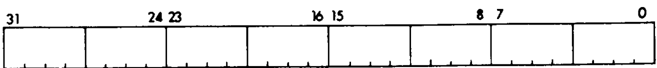
ID Address 07
Processor Address 23




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.

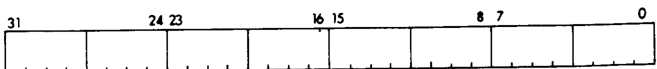
## DQ Register

ID Address 08
Processor Address —



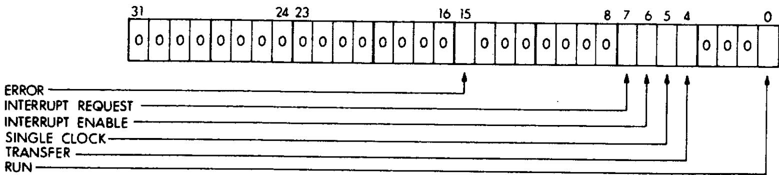

## Next Interval Register

ID Address 09
Processor Address 19



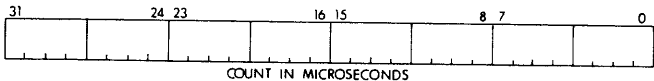
## Interval Clock Control/Status Register

### ID Address 0A
### Processor Address 18

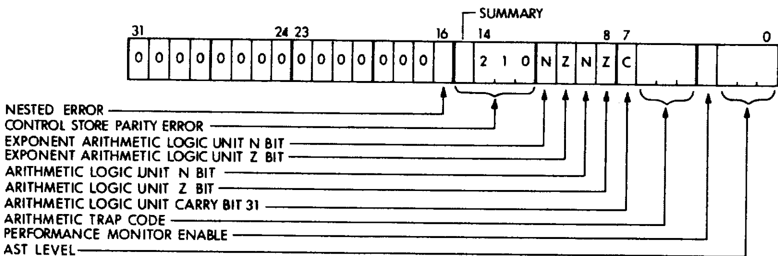



## Interval Counter Register

### ID Address 0B
### Processor Address 1A

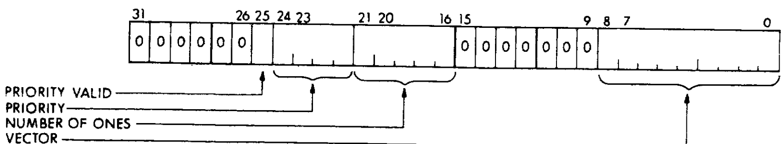



## ID Address 0C
## Processor Address 13 (Accesses AST level bits <2:0>)
## Processor Address 3D (Accesses Performance Monitor Enable bit <3>)



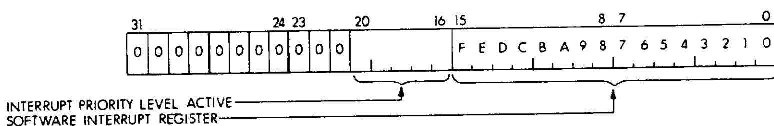

## Vector Register

### ID Address 0D
### Processor Address —



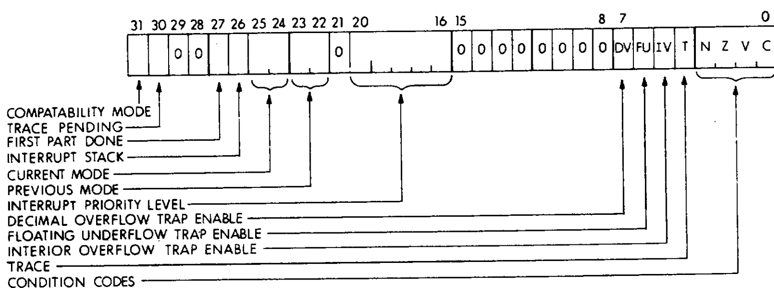
## Software Interrupt Register

### ID Address 0E
### Processor Address 15



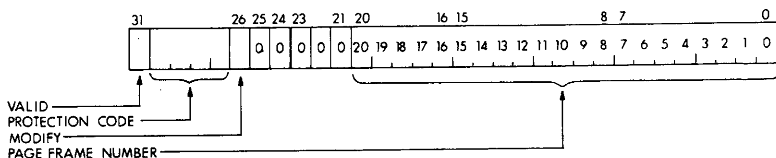

## Processor Status Longword Register

### ID Address 0F
### Processor Address 12



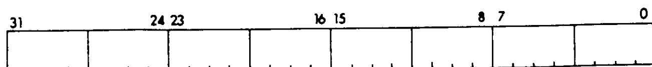

## Translation Buffer Data Register

### ID Address 10
### Processor Address —



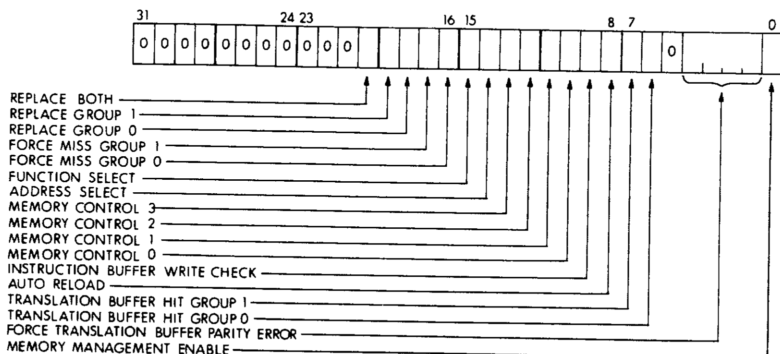

## Reserved Register

### ID Address 11
### Processor Address —



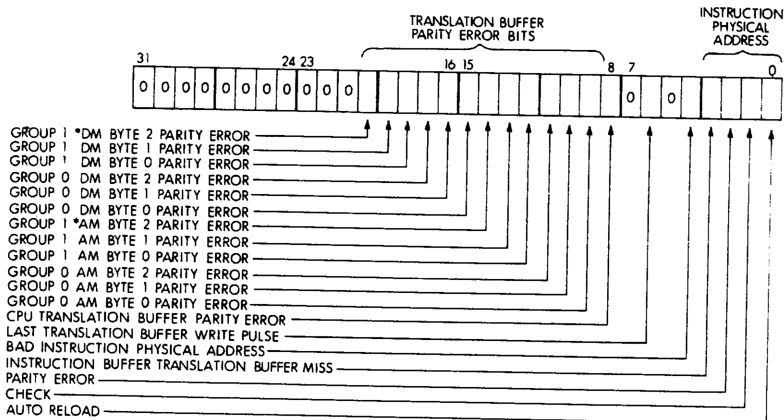
## Translation Buffer Control Register 0

### ID Address 12
### Processor Address —



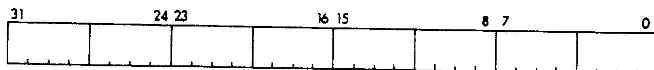

## Translation Buffer Control Register 1

### ID Address 13
### Processor Address —



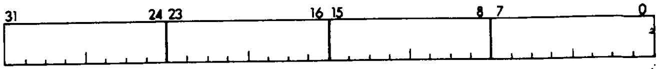

## Accelerator Control Register 0

### ID Address 14
### Processor Address —



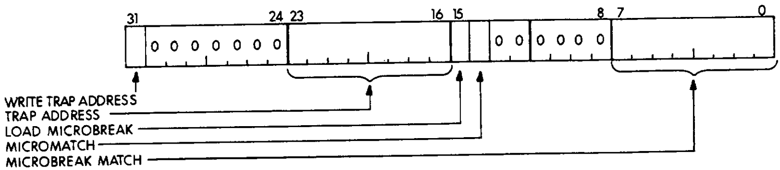
## Accelerator Control Register 1

### ID Address 15
### Processor Address —



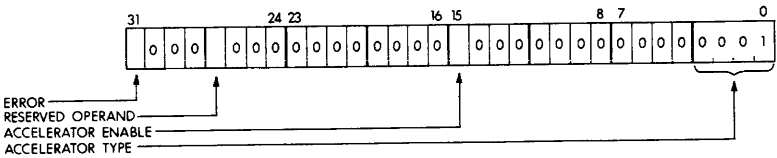

## Accelerator Maintenance Register

### ID Address 16
### Processor Address —



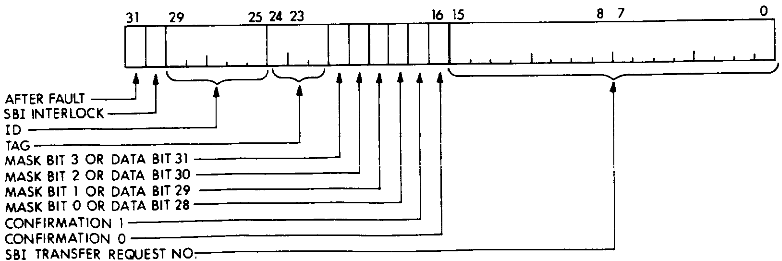

## Accelerator Control/Status Register

### ID Address 17
### Processor Address 28



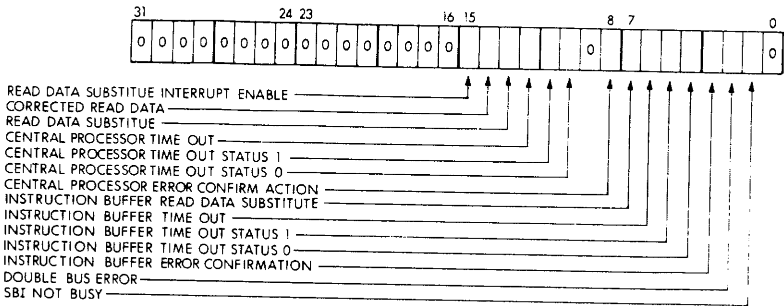

## SBI Silo Register

### ID Address 18
### Processor Address 31



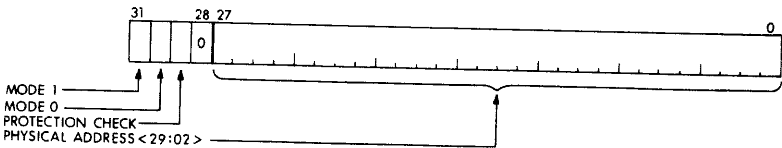
## SBI Silo Register

### ID Address 19
### Processor Address 34



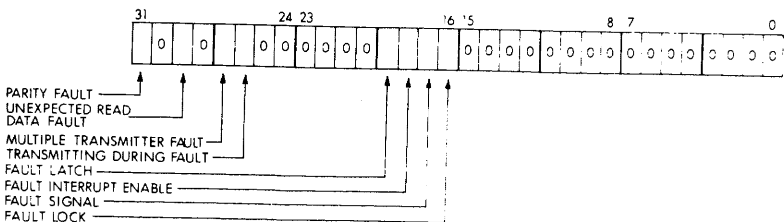

## SBI Time Out Address Register

### ID Address 1A
### Processor Address 35



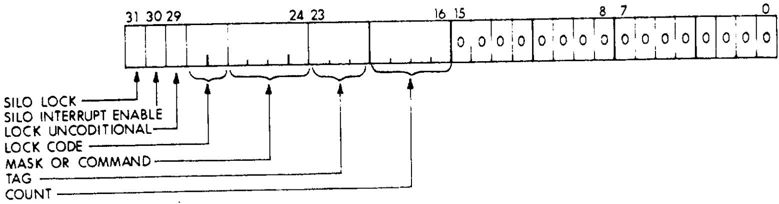

## SBI Fault Signal Register

### ID Address 1B
### Processor Address 30



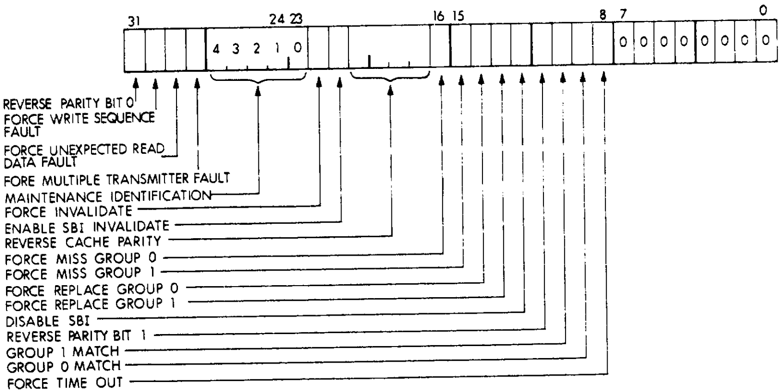
## SBI Silo Comparator Register

### ID Address 1C
### Processor Address 32



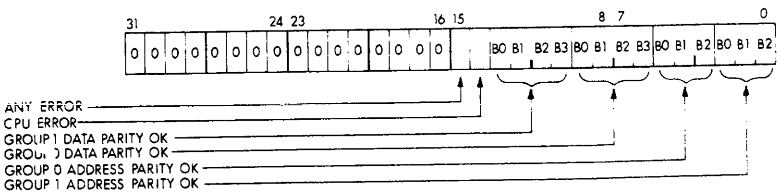

## SBI/Cache Maintenance Register

### ID Address 1D
### Processor Address 33



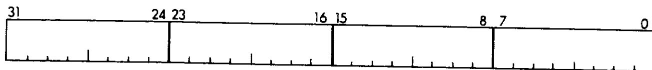

## Cache Parity Register

### ID Address 1E
### Processor Address —



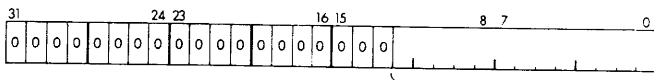
## Reserved Register

ID Address 1F
Processor Address —



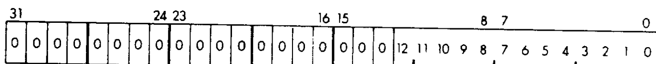

## Micro Stack Register

ID Address 20
Processor Address —



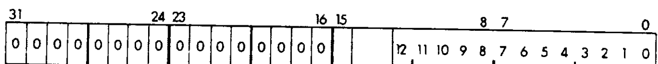

## Micro Match Register

ID Address 21
Processor Address 3C



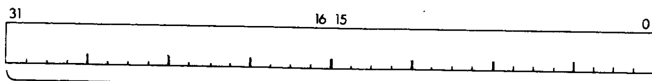

## Writable Control Store Address Register

ID Address 22
Processor Address 2C



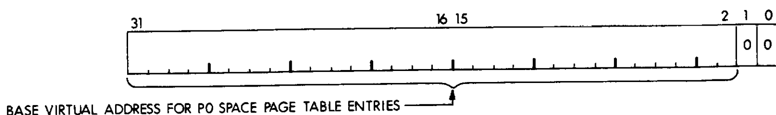

## Writable Control Store Data Register

ID Address 23
Processor Address 2D



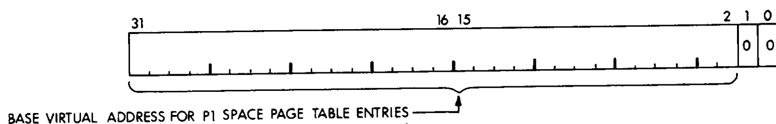
## P0 Base Register

ID Address 24
Processor Address 08



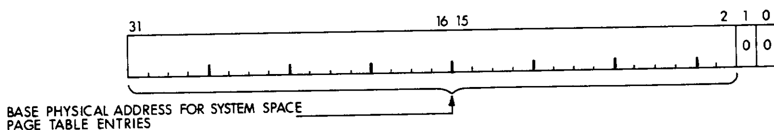

## P1 Base Register

ID Address 25
Processor Address 0A



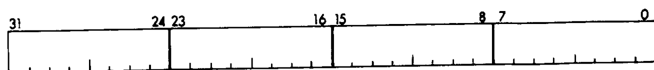

## System Base Register

ID Address 26
Processor Address 0C



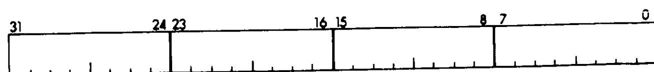

## Reserved Register

ID Address 27
Processor Address —



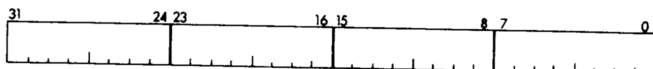

## Kernel Stack Pointer Register

ID Address 28
Processor Address 00



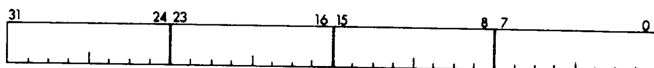
Executive Stack Pointer Regster

ID Address 29
Processor Address 01



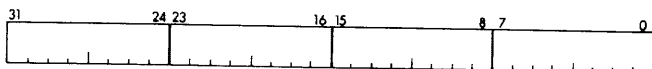

Supervisor Stack Pointer Register

ID Address 2A
Processor Address 02



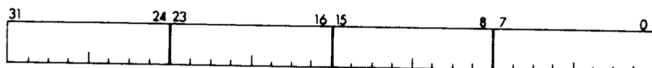

User Stack Pointer Register

ID Address 2B
Processor Address 03



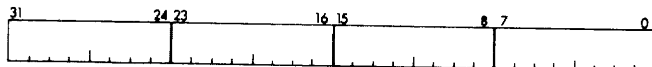

Interrupt Stack Pointer Register

ID Address 2C
Processor Address 04



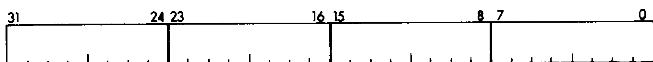

First Part Done Address Register

ID Address 2D
Processor Address —



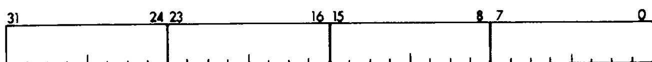
## D Save Register

ID Address 2E
Processor Address —



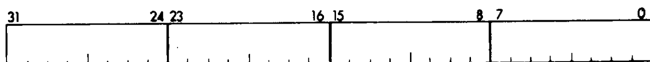

## Q Save Register

ID Address 2F
Processor Address —



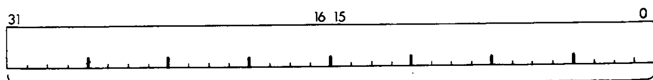

## Temp 0 to Temp 9 Registers

ID Address (30 to 39)
Processor Address —



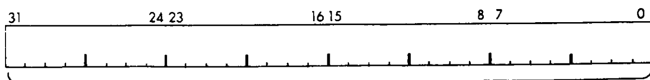

## Process Control Block Base Register

ID Address 3A
Processor Address 10



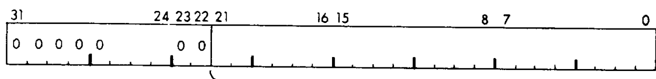

## System Control Block Base Register

ID Address 3B
Processor Address 11



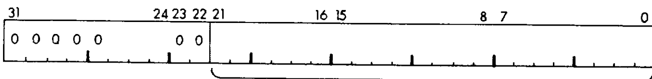
## P0 Length Register

ID Address 3C
Processor Address 09



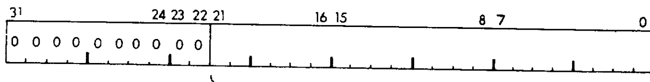

## P1 Length Register

ID Address 3D
Processor Address 0B



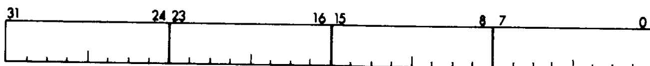

## System Length Register

ID Address 3E
Processor Address 0D




## Reserved Register

ID Address 3F
Processor Address —



APPENDIX E

# ADDRESS VALIDATION RULES

The memory management system described in Chapter 6 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.

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**

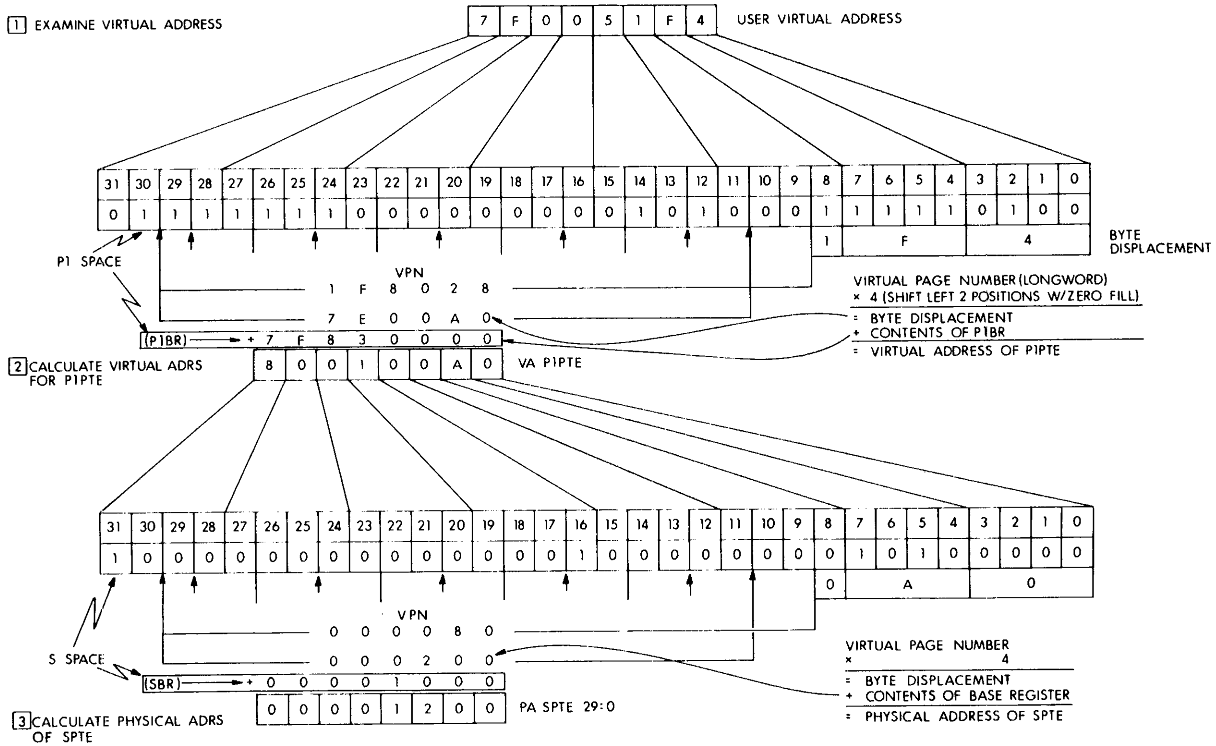
Using least privileged does not work properly when the data structure resides in pages with different protection and the first page has a lesser protection value than the others. When checking the accessibility of such a structure in the context of the serial execution of the process, the check will 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.

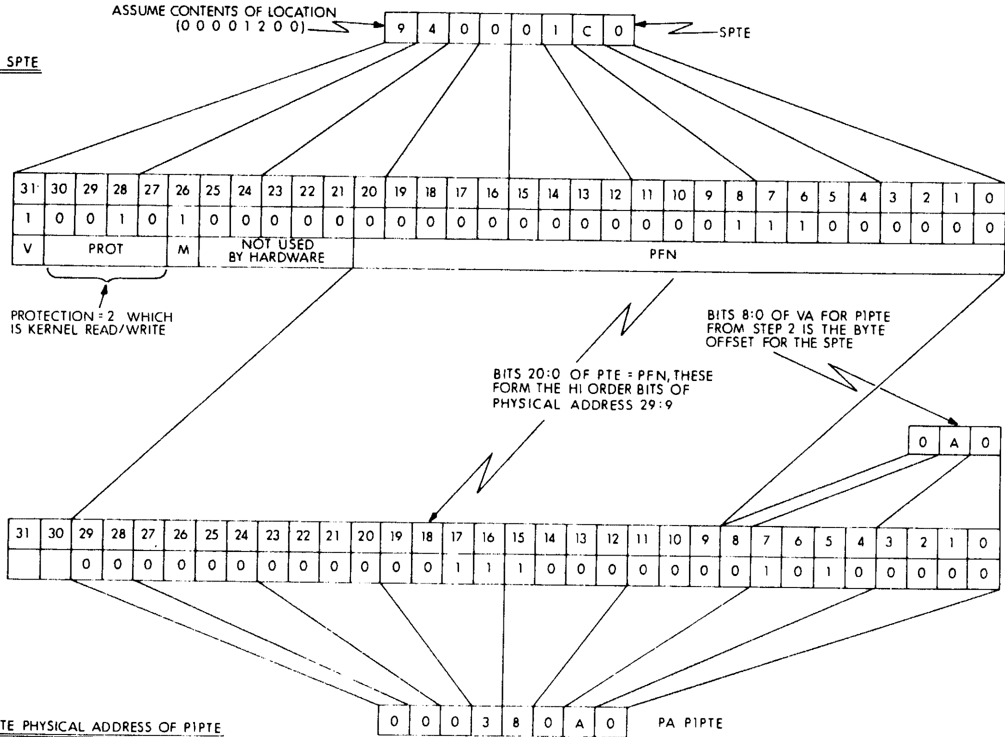
2. The synchronous validation of argument addresses (i.e., as the result of serial program execution) must be explicitly coded using Probe instructions specifying an access mode of zero (i.e., cause maximization to previous access mode).
3. The asynchronous validation of argument addresses (i.e., as 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.

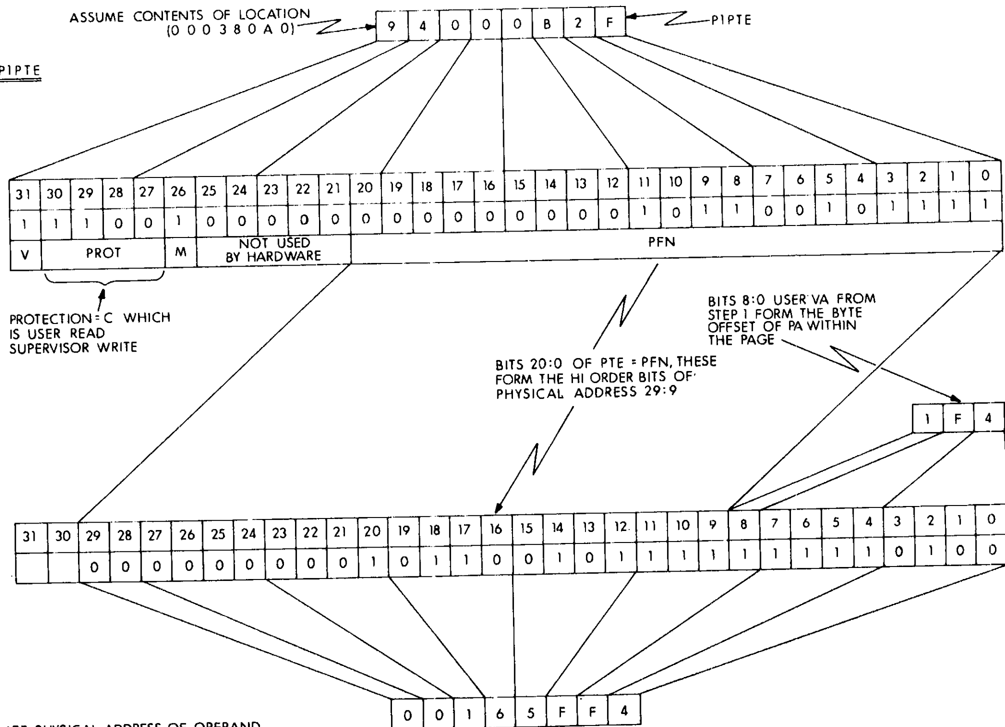
# APPENDIX F

# VIRTUAL TO PHYSICAL ADDRESS TRANSLATION





ASSUME CONTENTS OF LOCATION
(0 0 0 0 1 2 0 0)
9 4 0 0 0 1 C 0
SPTE
4 FETCH SPTE
V
PROT
M
NOT USED
BY HARDWARE
PFN
PROTECTION = 2 WHICH
IS KERNEL READ/WRITE
BITS 8:0 OF VA FOR P1PTE
FROM STEP 2 IS THE BYTE
OFFSET FOR THE SPTE
BITS 20:0 OF PTE = PFN, THESE
FORM THE HI ORDER BITS OF
PHYSICAL ADDRESS 29:9
0 A 0
5 CALCULATE PHYSICAL ADDRESS OF P1PTE
0 0 0 3 8 0 A 0
PA P1PTE



6 FETCH P1PTE
ASSUME CONTENTS OF LOCATION (0 0 0 3 8 0 A 0)
9 4 0 0 0 B 2 F
P1PTE
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0 0 1 0 1 1 1 1
V
PROT
M
NOT USED BY HARDWARE
PFN
PROTECTION = C WHICH IS USER READ SUPERVISOR WRITE
BITS 8:0 USER VA FROM STEP 1 FORM THE BYTE OFFSET OF PA WITHIN THE PAGE
BITS 20:0 OF PTE = PFN, THESE FORM THE HI ORDER BITS OF PHYSICAL ADDRESS 29:9
1 F 4
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 1 0 1 1 0 0 1 0 1 1 1 1 1 1 1 1 1 0 1 0 0
0 0 1 6 5 F F 4
7 CALCULATE PHYSICAL ADDRESS OF OPERAND

# APPENDIX G
# OPERAND SPECIFIER NOTATION

## 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 acces type:

| | |
|---|---|
| a | Calculate the effective address of the specified oprand. 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 oprands 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>.<br>If the effective addres is Rn, 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 | double floating |
| f | 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 assigment 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 |
| − | ubtraction, unary minus |
| * | multiplication |
| / | division (quotient only) |
| ** | exponentiation |
| ' | concatenation |
| ← | is replaced by |
| = | is defined as |
| Rn or R[n] | contents of register Rn |
| PC, SP, FP, or AP | the contents of register R15, R14, R13, or R12 respectively |
| PSW | the contents of the processor status word |
| PSL | the contents of the processor status long word |
| (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 |
| <x1,x2,...,xn> | a mȯdifier which enumerates bits x1,x2...,xn |

| | |
|---|---|
| x...y | x through y inclusive |
| { } | arithmetic parentheses used to indicate precedence |
| AND | logical AND |
| OR | logical OR |
| XOR | logical XOR |
| NOT | logical (ones) conplement |
| 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 signed extended to size of oprand 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 oprands 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 explicityly 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 operads is an integer byte add. Similarly, "LSS" is a floating comparison when

applied to floating operads while "LSS" is an integer byte comparison when applied to byte operands.

- 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, 2 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.

# GLOSSARY

**abort** An exception that occurs in the middle of an instruction and potentially leaves the registers and memory in an indeterminate state, so 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 not more protected than normal user programs.

**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.

**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.

**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).

**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).

**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.

**asynchronous** A mode of activity that operates without respect to a clock. For example, asynchronous hardware uses ready and done

signals to schedule operations rather than time intervals. If two activities are asynchronous, the second can begin before the first is complete.

**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 (rises 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.

**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 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.

**BBCCI** Branch on Bit Clear and Clear Interlock instruction. One can think of it as a Clear Interlock Bit instruction with a branch side-effect if the bit is already clear. This instruction permits interlocked access to a control variable.

**BBSSI** Branch on Bit Set and Set Interlock instruction. One can think of it as a Set Interlock Bit instruction with a branch side-effect if the bit is already set. This instruction permits interlocked access to a control variable.

**binary** A number system using two symbols: 0 and 1.

**bit** Binary digit. Any two-state device. A bit is said to be set (or on) when it represents the value 1, to be clear (or off) when it represents the value 0.

**bit string** See variable-length bit field.

**bits per inch** A measure of the recording density of magnetic tape, indicating the number of bits that can fit in one inch of the recording surface.

**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** A data accessing technique in which the program manipulates the blocks (physical records) that make up a file, instead of its logical records.

**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.

**buffer** A temporary data storage area in a process address space.

**Buffered Data Path (BDP)** A data path supporting block transfer devices on the UNIBUS. A block transfer device is one that transfers data to consecutive increasing addresses.

**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 two'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.

**central processor** The collection of interconnected logic modules that execute the control and arithmetic functions of a computer system. A central processor includes the logic which fetches and decodes instructions stored in main memory, an arithmetic logical unit for computation, and the primary I/O interfaces for the computer system.

**Change Mode instruction** The processor instruction that raises the access mode of the currently executing procedure by trapping to the operating system's change mode handlers. Procedures can only issue a CHM to a more protected access mode. An REI issued from within that access mode changes the mode back to a less protected access mode.

**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.

**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.

**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.

**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.

**compiler** A program that translates a program written in a high-level language (such as FORTRAN or BASIC) into an object program.

**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 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, loads another process' hardware PCB into the hardware context using the Load Process Context instruction, scheduling that process for execution.

**contiguous** Physically adjacent and/or consecutively numbered units of data.

**contiguous area** A space allocation on disk where the reserved area for all blocks in a file is physically adjacent on the recording medium.

**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 nonexistent 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.

**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.

**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, longword, and quadword integer, floating and double floating, character string, packed decimal string, and variable-length bit field.

**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 paging** One technique that enables a program to execute without having all of its pages resident in physical memory. In demand paging, a program page is not brought into physical memory until it is actually needed. If there is insufficient physical memory, the least recently used page in the system is moved out of memory to make room for the needed page. The page moved out may belong to any program in the system residing in physical memory. For the technique used in this system, see paging.

**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.

**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 Data Path (DDP)** A data path allowing 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.

**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.

**direct mapping cache** A cache organization in which only one address completion 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.

**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.

**Distributed Priority Arbitration** Each device connected to the SBI decides whether or not it has access to the bus (rather than a central arbitrator making the decision). That is, each device on the SBI maintains its own priority arbitration logic.

**double floating datum** Eight contiguous bytes (64 bits), starting on an addressable byte boundary, which are interpreted as containing a floating point number. The bits are labeled from right to left, 0 to 63. A four-word floating point number is identified by the address of the byte containing bit 0. Bit 15 contains the sign of the number. Bits 14 through 7 contain the excess 128 binary exponent. Bits 63 through 16 and 6 through 0 contain a normalized 56-bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits of decreasing significance go from 6 through 0, 31 through 16, 47 through 32, then 63 through 48. Exponent values of 1 through 255 in the 8-bit exponent field represent true binary exponents of -128 to 127. An exponent value of 0 together with a sign bit of 0 represent a floating value of 0. An exponent value of 0 with a sign bit of 1 is a reserved representation; floating point instructions processing this value return a reserved operand fault. The value of a floating datum is in the approximate range (+ or -) 0.29 x 10**-38 to 1.7 x 10**38. The precision is approximately one part in 2**55 or sixteen decimal digits.

**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.

**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.

**error correction code (ECC)** Single bit error detection and correction, double bit error detection.

**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.

**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 traps, compatibility mode faults, breakpoint instruction execution, and arithmetic traps such as 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 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.

**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.

**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 leaves the registers and memory in a consistent state, so that eliminating 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.

**floating (point) datum** Four contiguous bytes (32 bits) starting on an addressable byte boundary. The bits are labeled from right to left from 0 to 31. A two-word floating point number is identified by the address of the byte containing bit 0. Bit 15 contains the sign of the number. Bits 14 through 7 contain the excess 128 binary exponent. Bits 31 through 16 and 6 through 0 contain a normalized 24-bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits of decreasing significance go from bit 6 through 0, then 31 through 16. Exponent values of 1 through 255 in the 8-bit exponent

field represent true binary exponents of -128 to 127. An exponent value of 0 together with a sign bit of 0 represent a floating value of 0. An exponent value of 0 with a sign bit of 1 is a reserved representation; floating point instructions processing this value return a reserved operand fault. The value of a floating datum is in the approximate range (+ or -) 0.29 x 10**-38 to 1.7 x 10**38. The precision is approximately one part in $2^{23}$ or seven decimal digits.

**foreign volume** Any volume other than a Files-11 formatted volume which may or may not be file structured.

**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 comparison must take place against each block in the cache to find any particular block. Contrast with direct mapping cache.

**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.

**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.

**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.

**Hit Rate (cache-main memory)** The percentage of times the CPU requests data and that data appears in cache, therefore not requiring a main memory access.

**image** An image consists of procedures and data that have been bound together by the linker. There are three types of images: executable, sharable, and system.

**image privileges** The privileges assigned to an image when it is linked. See process privileges.

**immediate mode** In immediate mode addressing, the PC is used as the register in autoincrement mode addressing.

**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.

**index register** A register used to contain an address offset.

**instruction buffer** An 8-byte buffer in the processor used to contain bytes of the instruction currently being decoded and to prefetch instructions in the instruction stream. The control logic continously fetches data from memory to keep the 8-byte buffer full.

**interleaving** Assigning consecutive physical memory addresses alternately between two memory controllers.

**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.

**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 inter-

rupt 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 space** The region of physical address space that contains the configuration registers, and device control/status and data registers. The space is located in physical address space, but can be addressed virtually through the SCBB register using the MTPR and MFPR instructions.

**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.

**linker** A program that reads one or more object files created by language processors and produces an executable image file, a sharable image file, or a system image file.

**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, or quadword, the operand is zero extended and can express values in the range 0 through 63 (decimal). If the operand data type is floating or double 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 floating or double floating format.

**locality** See program locality.

**logical block** A block on a mass storage device identified using 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.

**longword** Four contiguous bytes (32 bits) starting on an addressable byte boundary. Bits are numbered from right to left with 0 through 31. The address of the longword is the address of the byte containing bit 0. When interpreted arithmetically, a longword is a two'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.

**main memory** See physical memory.

**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.

**MASSBUS adapter** The NEXUS connecting the MASSBUS subsystem to the SBI. The MASSBUS adapter provides virtual to physical address mapping functionality, data transfer buffering between the MASSBUS and main memory, and transfer of interrupts from the MASSBUS device to the SBI.

**memory controller** The NEXUS interfacing main memory to the SBI. The memory controller provides the necessary timing and control to complete all memory transactions.

**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.

**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 on-line (an activity accomplished by a system operator).

**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, and quadword integer, floating and double floating, 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.

**NEXUS** SBI interface logic.

**nibble** The low-order or high-order four bits of a byte.

**node** An individual computer system in a network.

**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.

**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's console** Any terminal identified as a terminal attended by a system operator.

**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 zero through nine.

**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 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.

**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.

**parity** A count maintained to check the reliability of a group of bits. Even parity refers to the use of a parity bit appended to a group of bits which is set to make the sum of all the bits an even value, where odd parity makes the sum of all the bits an odd value.

**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 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.

**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).

**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 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, perform direct I/O to a file-structured device.

**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.

**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 trap 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.

**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.

**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.

**pure code** See reentrant 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 two'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 $2^{63}-1$.

**queue** 1. n. A circular, doubly-linked list. See system queue. 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.

**read access type** An instruction or procedure operand attribute indicating that the specified operand is only read during instruction or procedure execution.

**reentrant 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 reentrant 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.

**return status code** See status code.

**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.

**sharable 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 sharable image file can be used to contain a library of routines. A sharable image can be used to create a global section by the system manager.

**slave terminal** A terminal from which it is not possible to issue commands to the command interpreter. A terminal assigned to application software.

**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.

**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.

**supervisor mode** The third most privileged processor access mode (mode 2). The operating system's command interpreter runs in supervisor mode.

**synchronous** Refers to a mode of activity that operates using an external timing mechanism. For example, synchronous data transmission hardware uses fixed time intervals to frame characters, where as asynchronous data transmission hardware uses start and stop codes. If two activities are synchronous, the second cannot take place until the first is complete.

**Synchronous Backplane Interconnect (SBI)** The part of the hardware that interconnects the processor, memory controllers, MASSBUS adapters, the UNIBUS adapter.

**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.

**System Base Register (SBR)** A processor register containing the physical address of the base of the system page table.

**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 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 queue** A queue used and maintained by operating system procedures. See also state queue.

**system region** The third quarter of virtual address space. The lowest-addressed half of system space. Virtual addresses in the system region are sharable 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 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.

**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 VT52 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.

**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).

**transfer request number (TR)** The data transfer request priority assigned to a device interfacing to the SBI.

**translation buffer** An internal processor cache containing translations for recently used virtual addresses.

**Translation Buffer Invalidate Single** A processor register used in controlling memory management address translation buffers.

**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 condition with a single instruction.

**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.

**UNIBUS adapter** The hardware interface between the UNIBUS and the synchronous backplane interconnect. The UNIBUS adapter provides the mapping function allowing access to UNIBUS address space from the SBI, and mapping of UNIBUS addresses to SBI addresses for UNIBUS DMA transfers to SBI memory. The UNIBUS adapter provides the data paths for UNIBUS device access to random SBI memory addresses, and high-speed transfers for UNIBUS devices that transfer to consecutive increasing memory addresses. The UNIBUS adapter also provides interrupt fielding, priority arbitration, and UNIBUS power fail sequencing.

**unit record device** A device such as a card reader or line printer.

**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 mode** The least privileged processor access mode (mode 3). User processes and the Run Time Library procedures run in user mode.

**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 zero 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.

**vector** 1. An 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 the purposes of 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.

**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 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.

**volume** A mass storage medium such as a disk pack or reel of magnetic tape.

**volume set** The file-structured collection of data residing on one or more mass storage media.

**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 two'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 -32768 to 32767. 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 65535.

**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.

**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 is 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 is copied in both cache and main memory. Cache and main memory data are always consistent. Contrast with write back.

# INDEX


Accelerator control registers 304, 305

Accelerator control/status register (ACCS) 250, 251, 305

Accelerator maintenance register (ACCR) 252, 305

Access Control Violation fault 109, 117, 118

Access Modes 55, 16, 101, 107, 118, 119, 275, 277
    see also CPU, access modes

Address
    see also Physical address
    see also Virtual address space
  assignments
    UNIBUS adapter 194, 195
  space
    see also Physical address space
    see also Virtual address space
  UNIBUS adapter 169
  translation
    overview 105 to 107
    physical to SBI 122, 133
    SBI to UNIBUS 171 to 173
    UNIBUS to SBI 176 to 178
    virtual to physical 59, 60, 109, to 115, 317 to 319
    virtual to SBI 230
  validation 313 to 315

Addressing modes 8, 42, 44, 47 to 49

Address translation buffer 9, 35, 36

Arbitration
  UNIBUS priority 165, 167, 168

Arbitration lines
  SBI 127, 128

Architecture 2, 269 to 273

Argument list 50

Argument pointer (AP) 47, 50

Arguments
  passing 44, 45

Arithmetic traps 277

AST Level (ASTLVL) register 71

Asynchronous System Traps (AST) 72

Bad block handling 279 280

Base register 46

Battery backup 15

Bootstrap 160

BR Receive Vector registers 4-7 (BRRVR) 209 to 211

Buffered data paths 178 to 181

Buffers 9

Bus
    see also Names of specific buses
    see also MASSBUS
    see also UNIBUS
  arbitration lines 127, 128
  configuration 125
  structure
    console 20 to 22

Bus Request (BR) 164, 165, 167

Byte offset data transfers 183, 185

Cache parity register 307

Caches 9, 35, 270, 271

Call Frame 50, 51

Calls
  nested 50

CCL (console command language) 23 to 27

Central Processing Unit
    see CPU

Change mode (CHM) instructions 57, 257 to 259
Character string data type 42, 43
Clock margining 279
Clock registers 247 to 250
Clocks 9, 36, 248 to 250
Clock signals
  SBI 139
Command Address register 240
Command codes
  SBI 140 to 145
Commander 125
Communication
    see MASSBUS
    see UNIBUS
Compatibility mode instruction set 41
Condition codes 45
Condition handlers 52
Configuration register (CNFGR) 196 to 199
Confirmation codes (CNF)
  information transfer 138
Consistency checking 276 to 278
Console command language (CCL) 23 to 27
Console command mode 18
Console interface Board 18 to 20
Console receive control/status register 300
Console receive buffer register 300
Console subsystem
  bus structure 20 to 22
  command language 23 to 27
  components 18
  data transfer 22
  error messages 27 to 30
  interaction combinations 18 to 20
  overview 10, 11, 17, 18
Console terminal registers 246, 247
Console transmit control/status register 301
Console transmit data buffer register 301
Context
  process 41, 64, 67 to 71
  switching 52, 53, 58
Control bus
  signal lines 223 to 225
Control lines
  SBI 139, 140
Control register (UACR) 199 to 202
Control space (P1 region) 46, 102, 103, 111, 114, 115
Control store 34, 35
CPU
  access modes 55, 56
    see also Access modes
  operation 37, 39 to 41
  overview 4 to 12, 33, 34
CPU hardware 34 to 38
CPU programming concepts 41 to 64
Cycle time
  memory 11, 12
  SBI 145
Data
  management 3, 4
  paths
    see also Synchronous Backplane Interconnect
    CPU 35
    UNIBUS adapter 178 to 181
  sharing 269, 270
  transfer
    see also MASSBUS
    see also UNIBUS
    memory to UNIBUS 183 to 190
    program flow 240
    SBI to UNIBUS 170 to 175
    UNIBUS to memory 181 to 183
    UNIBUS to SBI 175 to 178
  transfer rate 126, 145
  types 6, 42, 43
Data bus
  signal lines 226

Data Path registers (DPR) 211 to 213
Data transfer signal lines 166, 167
DATI operation 186, 187
DATO operation 186, 187
Dead signal SBI 139, 140
Definitions of terms 325 to 357
Device address space
UNIBUS 171
interrupts 84
registers 165, 166
Diagnostic
console 275, 277
Diagnostic Control register (DCR) 205 to 207
Diagnostic register
MASSBUS adapter 238, 239
Diagnostics
system 275 to 280
Direct data path (DDP) 178 to 180
Direct memory access (DMA) transfers 175, 176
DQ register 301
D save register 311
Error checking 276 to 278
Error Correcting Code (ECC) 11, 153, 278
Error notification interrupts 272
Error reporting 148
Errors 272
Exceptions
discussion 45, 53, 81 to 83
handling 52
initiating 96 to 99
sequence 95, 96
vectors 60, 61, 89 to 92
Executive mode 107
Executive stack pointer register 310
Extended functions call (XFC) instruction 257, 262
Extended read cycle 151
Extended write masked function 152
Failed Map Entry register (FMER) 207, 208
Failed UNIBUS Address register (FUBAR) 208, 209
Fail function
SBI 140
Fault detection 276, 278 to 280
First part done address register 310
Floating point accelerator (FPA) 15, 37
Floating point data type 42, 43
Frame Pointer (FP) 47, 50
Function codes
SBI 140 to 145
General registers 8
Glossary 325 to 351
Hardware
context 53, 64
CPU 34 to 38
memory management 9, 10
see also Memory, management
optional 15
overview 4, 5
ID (internal data) Bus 20, 21
ID (internal data) Bus register 299 to 312
Image 41
Indexed addressing modes 47
Information lines
SBI 128 to 137
Initialization signals
UNIBUS 168
Input/Output
see I/O
Instruction buffer 9, 36
Instruction buffer register 299
Instructions
index
by mnemonics 285 to 290

by opcode 291 to 296
native mode 6 to 8, 41, 42
privilege 257 to 266
restartability 271, 272

Integer data type 42, 43

Interleaving 12, 159, 160

Interlock cycles 152, 153

Interlocked access 269, 270

Interlock read masked cycle 152, 153

Interlocked write masked cycle 153

Interrupt priority level (IPL) 81, 82

Interrupt priority level (IPL) register 86

Interrupt request lines
SBI 139

Interrupts
architectural implications 272
discussion 53, 54, 81 to 87
initiating 96 to 99
priority 61, 63, 72, 73
sequence 95, 96
UNIBUS 190 to 194
vectors 60, 61, 89 to 92

Interrupt stack not valid hault 88

Interrupt stack pointer register 310

Interrupt summary exchange 129

Interval clock 9, 36, 248 to 250

Interval Clock Control/Status register (ICCS) 249, 250, 302

Interval count register 249, 302

Interval timer 279

I/O
address space
SBI 170
processing 61, 64
structure 272, 273
subsystems 12 to 14

I/O space restrictions 297

Kernel mode 107

Kernel stack not valid abort 87

Kernel stack pointer register 309

Length violation 109

Limit checking traps 277

Load Process Control (LDPCTX) instruction 74, 75, 266

Longword aligned 32-bit random access mode 186 to 189

LSI-11 microprocessor 18 to 20

Machine check exception 88

Main memory subsystem
see Memory

Maintenance
system 275 to 280

Map enable register (MAPEN) 115, 116

Map registers 213 to 216

MASSBUS
adapter 227 to 240
control path 229
data transfer
example 240 to 242
overview 14, 15, 223
signal lines 223 to 226

MBA Byte Counter 238

MBA Configuration/Status register 231, 232

MBA Control register 233, 234

MBA external registers 240

MBA map 240

MBA Status register 234 to 237

MBA Virtual Address register 237

Memory
access 101, 107 to 109, 118, 119
battery backup 15
cache 9, 35
configuration registers 153 to 159
controller 148, 149
cycle time 11, 12
data transfer
UNIBUS 181 to 190
error checking and correction 153
interlock cycles 152, 153
interleaving 159, 160

management
  see also Hardware, memory management
  address translation 105 to 107, 109 to 115
  control 115 to 117
  faults 117, 118
  overview 3, 9, 10, 58, 59, 101, 102
  privileged services 118, 119
  protection 107 to 109
  shared sections 120, 121
  system page table 119, 120
  virtual address space 102 to 105
 mapping 59, 60, 105 to 107, 117, 118
 operations 149 to 152
 overview 11, 12, 147, 148
 protection 3, 55, 56, 107 to 109
 ROM bootstrap 160
 virtual 54
  see also Virtual address space

Memory Mapping Enable (MME) bit 105

Micro control store 252 to 254

Micro match register 308

Micro stack register 308

Microprogram breakpoint address register (MBRK) 253

MME bit 105

Mnemonics
  instructions 285 to 290
  terms 281 to 283

MOS memory 11, 15

Move from privileged register (MFPR) instruction 257, 263 to 265

Move to privileged register (MTPR) instruction 257, 263 to 265

Multiprocessor systems interrupt priority level 81

Multiprogramming hardware support 52 to 54

Native instruction set 6 to 8, 41, 42, 285 to 296

Next Interval Count register 249, 301

NEXUS 125, 126

NEXUS register space 194

Non-Processor Requests (NPR) 164, 167

Notation 321 to 324

Operand
  specifier notation 321, 322

Operation description notation 322 to 324

P0 base register 309

P0 length register 312

P0 region
  see Program space

P1 base register 309

P1 length register 312

P1 region
  see Control space

Packed deimal data type 42, 43

Page
  description 59, 102
  mapping 59, 60
  protection 103, 104, 107 to 109

Page table 60

Page Table Entry (PTE) 105 to 107

Parity checks 279

PCBB (Process Control Block Base) 67

PCB (Process Control Block) 68 to 71

PDP-11 instructions 41

Performance Monitor Enable (PME) register 71

Per-process space 46, 102, 103
  see also Process space

Physical address
  description 54, 58, 59
  memory mapping 105 to 107
  translation 109 to 115

Physical address space
  memory 149, 150
  SBI 131, 132

Physical memory 11

PME register 71

Power fail
  UNIBUS 216, 217

Priority
  dispatching 53, 54
  exception service routines 82
  interrupts 61, 63, 72, 73, 81, 84
  UNIBUS 164, 165, 167, 168

Privilege
  instructions 56 to 58, 227 to 266
  memory access 56 to 58, 107, 118, 119

Privileged processor registers 245 to 254

Privileged registers 263 to 265

PROBE instruction 57, 121, 122, 257, 260, 261

Procedure 44, 45

Process
  Asynchronous System Traps 72
  context 64, 67 to 71
  definition 41, 53, 67
  interrupt priorities 72, 73
  privileged registers 71
  structure 67 to 78
  structure instruction 73 to 78

Process Control Block Base (PCBB) 67

Process Control Block Base register 311

Process Control Block (PCB) 68 to 71

Processor
  see also CPU
  errors 272
  interrupt priority levels 61, 63, 72, 73
  Status Longword (PSL) 55, 303

Processor Status Word 51, 52

Process space
  address translation 111 to 115
  description 41, 46, 102, 103
  shared sections 120, 121

Program 41

Program counter addressing modes 48

Program Counter (PC) 42, 47

Program I/O mode 18

Programmable real-time clock 9, 36, 248 to 250

Programming
  concepts 41 to 45
  envirnoment 45 to 52

Program space (P0 region) 46, 102, 103, 111 to 113

Protection
  faults 117, 118
  memory 3, 55, 56, 107 to 109, 118, 119
  page 103, 104, 107 to 109

Protocol checks 279

Purge operation 186

Q bus 21

Q save register 311

Quadword 11

RAMP 275 to 280

Random access 186 to 189

Read cycle 150, 151

Read only memory
  Console Interface Board 23

Real-time clock 9, 36, 248 to 250

Receiver 125

Record management 4

Registers
  see also names of specific registers
  definition 42
  device 165, 166
  general 8, 46, 47
  internal data (ID) bus 299 to 312

interrupt control 84 to 86
I/O
constraints 273
maintenance 278, 279
MASSBUS adapter 228 to 240
memory configuration 153 to 159
privileged 263 to 265
privileged processor 245 to 254
process privileged 71
UNIBUS adapter 194 to 216

Reliability availability maintainability program (RAMP) 275 to 280

Reserved operand traps 278

Responder 125

Response lines
SBI 138

Restartability 271, 272

Return from Exception or Interrupt (REI) instruction 57, 58

ROM Console Interface Board 23

Save Process Context (SVPCTX) instruction 73, 76 to 78, 266

SBI
see Synchronous Backplane Interconnect

SBI/cache maintenance register 307

SBI fault signal register 306

SBI silo compactor register 307

SBI silo registers 305, 306

SBI time out address register 306

SBI UNJAM 217

Sections
shared 120, 121

Selected Map register

Shared data 269, 270

Shared sections 120, 121

Signal lines
MASSBUS 223 to 226
SBI 126
UNIBUS 166 to 168

Silo 138, 278

Software
overview 2 to 4

Software-generated interrupts 84

Software Interrupt Request register (SIRR) 85, 86

Software Interrupt Summary register (SISR) 84, 85, 303

Stack frame 50, 51

Stack pointer (SP) 44, 47, 48, 50

Stacks 8, 44, 45, 92 to 95

Status register (USAR) 202 to 205

Subroutine 44

Supervisor mode 107

Supervisor stack pointer register 310

Synchronization 269, 270

Synchronous Backplane Interconnect (SBI)
address space 170, 175 to 178
command code description 140 to 145
data transfer
UNIBUS 170 to 181
overview 12 to 14, 125, 126
physical address space 131, 132
structure 126 to 140
throughput 145

System failures 87, 88

System maintenance 275 to 280

System
programming 52 to 64
services 3

System base register 309

System Control Block Base (SCBB) register 60, 62, 88, 89, 311

System Control Block (SCB) 60, 62, 88 to 92

System identification register (SID) 245, 246, 278, 300

System length register 312

System page table 119, 120

System space 46, 102, 103, 109 to 111, 121

Temp registers 311

Terminology 325 to 351

Time-of-day register 299

Time-of-year clock 9, 36, 248

Trace trap 52

Translation buffer 9, 116, 117

Translation buffer control registers 304

Translation buffer data register 303

Translation Not Valid fault 117, 118

Transmitter 125

Traps 52, 277, 278

UNIBUS
- adapter
  - description 168 to 170
  - registers 194 to 216
- address space 169 to 175
- configuration 164
- data transfer
  - memory 181 to 190
  - SBI 170 to 181
- example 217 to 211
- initialization 216
- interrupts 190 to 194
- line definitions 166 to 168
- overview 14, 163
- power failure 216, 217
- request levels 164, 165
- summary 163 to 168

UNJAM function 140, 217

Urgent interrupts 84

User mode 107

User stack pointer register 310

User writable control store (WCS) 15, 37

Variable bit field data type 42, 43

VAX-11 architectural implications 269 to 273
- series registers 264, 265

VAX/VMS
- data management 3, 4
- record management 4
- software overview 2 to 4

V bus 21

Vector register 302

Vectors 60, 61, 89 to 92

Virtual address 54

Virtual address space
- discussion 41, 58, 59, 102 to 105
- memory mapping 105 to 107
- structure 46
- translation 59, 60, 109 to 115, 317 to 319

Virtual memory 54

Virtual page 59

Watchdog timer 279

WCS (writable control store) 15, 37

WDCS (writable diagnostic control store) 9, 36

Writable control store address register (WCSA) 253, 308

Writable control store data register (WCSD) 253, 308

Writable control store (WCS) 15, 37

Writable diagnostic control store (WDCS) 9, 36

Write masked function 151, 152

Write-through 35

## READER'S COMMENTS

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 ____________________

Company ____________________ Dept. ____________________

Title ____________________

City ____________________ State ____________________ Zip ____________________

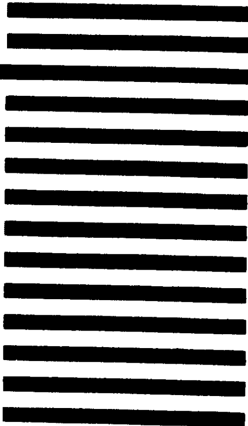
(staple here)

(please fold here)

BUSINESS REPLY MAIL

NO POSTAGE STAMP NECESSARY
IF MAILED IN THE UNITED STATES

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.



Postage will be paid by:

DIGITAL EQUIPMENT CORPORATION
SALES SUPPORT LITERATURE GROUP
PK3-2/M-88
MAYNARD, MASS. 01754

(staple here)